

April 1981

XPORT
Programmer's Guide
A Tutorial and Reference Manual
Order No. AA-J201A-TK

SUPERSESSON/UPDATE INFORMATION: This is a new document for this release.

OPERATING SYSTEM AND VERSION: VAX/VMS V2.2
TOPS-10 V7.01
TOPS-20 V4.0

SOFTWARE VERSION: XPORT V1.0

To order additional copies of this document, contact the Software Distribution
Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation • maynard, massachusetts

First Printing, April 1981

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Copyright © 1981 by Digital Equipment Corporation.
All Rights Reserved.

Printed in U.S.A.

The postage-paid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DECsystem-10	PDT
DECUS	DECSYSTEM-20	RSTS
DIGITAL	DECwriter	RSX
PDP	DIBOL	VMS
UNIBUS	Edusystem	VT
VAX	IAS	digital
DECnet	MASSBUS	

CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	APPLICABILITY OF XPORT FACILITIES	1-i
1.2	PROGRAM TRANSPORTABILITY	1-2
1.3	FILE TRANSPORTABILITY	1-3
1.4	SYMBOL NAMING CONVENTIONS	1-3
1.5	COMPILATION ERROR MESSAGES	1-4
1.6	SMALL SAMPLE PROGRAM	1-5
CHAPTER 2	TRANSPORTABLE DATA STRUCTURES	
2.1	INTRODUCTION	2-1
2.1.1	The Problem	2-1
2.1.2	The Solution	2-2
2.1.3	Simple Example	2-3
2.1.4	Terminology	2-5
2.2	\$FIELD DECLARATION AND \$FIELD_SET_SIZE	2-5
2.2.1	\$FIELD Declaration	2-5
2.2.2	Transportable Field-Types	2-6
2.2.3	Nontransportable Field-Types	2-8
2.2.4	Guidelines for Individual Field-Types	2-8
2.2.5	\$FIELD_SET_SIZE Usage	2-12
2.3	SUPPLEMENTARY FEATURES	2-12
2.3.1	Field-Positioning Features	2-12
2.3.2	Literal-Defining Features	2-14
2.3.3	Value-Display Feature	2-15
2.3.4	Subfield Referencing Feature	2-16
2.4	TRANSPORTABILITY CONCERNS	2-20
2.4.1	Field Size	2-20
2.4.2	Integer Value Range	2-21
2.4.3	Use of \$BYTES for Character Strings	2-21
2.5	EFFICIENCY CONCERNS	2-22
CHAPTER 3	INPUT/OUTPUT FACILITIES	
3.1	INTRODUCTION	3-1
3.1.1	General Characteristics	3-2
3.1.2	Specific Functions	3-3
3.2	CAPABILITIES	3-3
3.2.1	File Level Capabilities	3-3
3.2.2	Input/Output Capabilities	3-5
3.2.3	File Specification Resolution	3-8
3.3	I/O RELATED MACROS	3-9
3.3.1	General Format and Common Parameters	3-10
3.3.2	File-Level Macros	3-11

3.3.3	Input/Output Macros	3-15
3.4	INPUT/OUTPUT CONTROL BLOCKS	3-18
3.4.1	Creating and Initializing IOBs	3-18
3.4.2	Using IOB Fields and Values	3-19
3.5	STANDARD I/O DEVICES	3-22
3.6	FILE SPECIFICATION PROCESSING	3-23
3.6.1	File Specification Resolution	3-23
3.6.2	File Specification Parsing	3-26
3.7	I/O COMPLETION CODES	3-27
3.8	I/O ACTION ROUTINES	3-28
CHAPTER 4	MEMORY MANAGEMENT FACILITIES	
4.1	INTRODUCTION	4-1
4.2	CAPABILITIES	4-1
4.3	MEMORY MANAGEMENT MACROS	4-2
4.3.1	\$XPO_GET_MEM - Allocating Dynamic Memory	4-2
4.3.2	\$XPO_FREE_MEM - Releasing Dynamic Memory	4-3
4.3.3	Dynamic Memory Elements	4-4
4.4	COMPLETION CODES	4-4
4.5	ACTION ROUTINES	4-4
CHAPTER 5	OTHER SYSTEM SERVICES	
5.1	INTRODUCTION	5-1
5.2	\$XPO_PUT_MSG	5-1
5.2.1	Completion Codes	5-3
5.2.2	Action Routines	5-3
5.3	\$XPO_TERMINATE	5-3
CHAPTER 6	STRING HANDLING FACILITIES	
6.1	STRING DESCRIPTORS	6-1
6.1.1	\$STR_DESCRIPTOR -- Creating a String Descriptor	6-2
6.1.2	\$STR_DESCRIPTOR -- Compile-Time Descriptor Initialization	6-2
6.1.3	\$STR_DESC_INIT -- Run-Time String Descriptor Initialization	6-3
6.1.4	String Descriptor Formats	6-4
6.1.5	String Descriptor Usage Rules	6-6
6.1.6	Descriptor Data Types	6-8
6.2	STRING DESCRIPTOR STRUCTURE REFERENCES	6-9
6.3	STRING MODIFICATION	6-9
6.3.1	\$STR_COPY Operation	6-10
6.3.2	\$STR_APPEND Operation	6-11
6.4	STRING COMPARISON	6-12
6.5	STRING SCANNING	6-14
6.5.1	\$STR_SCAN Overview	6-14
6.5.2	\$STR_SCAN FIND - Find a Character Sequence	6-15
6.5.3	\$STR_SCAN SPAN - Match a Set of Characters	6-16

6.5.4	\$STR_SCAN STOP - Search for a Set of Characters	6-16
6.5.5	\$STR_SCAN - Returning a Substring	6-17
6.5.6	\$STR_SCAN - "Scanning Through" a BOUNDED String	6-17
6.6	STRING CONVERSION	6-18
6.6.1	\$STR_CONCAT and \$STR_FORMAT - ASCII to ASCII String Conversions	6-18
6.6.2	\$STR_ASCII - Binary-Data to ASCII String Conversion	6-19
6.6.3	Nesting \$STR_ASCII, \$STR_CONCAT, \$STR_FORMAT Pseudo-Functions	6-21
6.6.4	\$STR_BINARY - ASCII String to Binary-Data Conversion	6-22

CHAPTER 7 BINARY DATA DESCRIPTORS

7.1	INTRODUCTION	7-1
7.2	BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION	7-2
7.2.1	\$XPO_DESCRIPTOR -- Creating a Binary Data Descriptor	7-2
7.2.2	\$XPO_DESCRIPTOR -- Compile-Time Descriptor Initialization	7-2
7.2.3	\$XPO_DESC_INIT -- Run-Time Data Descriptor Initialization	7-3
7.2.4	Classes Of Descriptors	7-4

APPENDIX A MACRO DESCRIPTIONS

A.1	DESCRIPTIVE NOTATION AND CONVENTIONS	A-1
A.1.1	Syntax Notation	A-1
A.1.2	Character-String and Binary-Data Parameters	A-2
A.2	\$STR_APPEND - Append a String	A-4
A.2.1	Syntax	A-4
A.2.2	Restrictions	A-4
A.2.3	Parameter Semantics	A-4
A.2.4	Operational Semantics	A-5
A.2.5	Completion Codes	A-6
A.3	\$STR_ASCII - Binary-to-ASCII Conversion Pseudo-Function	A-7
A.3.1	Syntax	A-7
A.3.2	Restrictions	A-7
A.3.3	Parameter Semantics	A-7
A.3.4	Usage Guidelines	A-8
A.4	\$STR_BINARY - Convert ASCII to Binary	A-9
A.4.1	Syntax	A-9
A.4.2	Restrictions	A-9
A.4.3	Parameter Semantics	A-10
A.4.4	Usage Guidelines	A-10
A.4.5	Completion Codes	A-11
A.5	\$STR_COMPARE - String Comparison	A-12
A.5.1	Syntax	A-12
A.5.2	Restrictions	A-12

A.5.3	Parameter Semantics	A-12
A.5.4	Operational Semantics	A-13
A.5.5	Completion Codes	A-13
A.6	\$STR CONCAT - String Concatenation	
	Pseudo-Function	A-14
A.6.1	Syntax	A-14
A.6.2	Restrictions	A-14
A.6.3	Parameter Semantics	A-14
A.6.4	Usage Guidelines	A-15
A.7	\$STR COPY - Copy a String	A-16
A.7.1	Syntax	A-16
A.7.2	Restrictions	A-16
A.7.3	Parameter Semantics	A-16
A.7.4	Operational Semantics	A-17
A.7.5	Completion Codes	A-18
A.8	\$STR DESCRIPTOR - Declare a String Descriptor .	A-19
A.8.1	Syntax	A-19
A.8.2	Restrictions	A-19
A.8.3	Parameter Semantics	A-19
A.9	\$STR_DESC_INIT - Initialize a String Descriptor	A-21
A.9.1	Syntax	A-21
A.9.2	Restrictions	A-21
A.9.3	Parameter Semantics	A-21
A.9.4	Completion Code	A-22
A.10	\$STR_EQL - String Equality Comparison	A-23
A.10.1	Syntax	A-23
A.10.2	Restrictions	A-23
A.10.3	Parameter Semantics	A-23
A.10.4	Operational Semantics	A-24
A.10.5	Completion Codes	A-25
A.11	\$STR_FORMAT - String Formatting Pseudo-Function	A-26
A.11.1	Syntax	A-26
A.11.2	Restrictions	A-26
A.11.3	Parameter Semantics	A-27
A.11.4	Usage Guidelines	A-27
A.12	\$STR_GEQ - String Greater-Than-or-Equal	
	Comparison	A-29
A.12.1	Syntax	A-29
A.12.2	Restrictions	A-29
A.12.3	Parameter Semantics	A-29
A.12.4	Operational Semantics	A-30
A.12.5	Completion Codes	A-31
A.13	\$STR_GTR - String Greater-Than Comparison . . .	A-32
A.13.1	Syntax	A-32
A.13.2	Restrictions	A-32
A.13.3	Parameter Semantics	A-32
A.13.4	Operational Semantics	A-33
A.13.5	Completion Codes	A-34
A.14	\$STR_LEQ - String Less-Than-or-Equal Comparison	A-35
A.14.1	Syntax	A-35
A.14.2	Restrictions	A-35
A.14.3	Parameter Semantics	A-35
A.14.4	Operational Semantics	A-36

A.14.5	Completion Codes	A-37
A.15	\$STR_LSS - String Less-Than Comparison	A-38
A.15.1	Syntax	A-38
A.15.2	Restrictions	A-38
A.15.3	Parameter Semantics	A-38
A.15.4	Operational Semantics	A-39
A.15.5	Completion Codes	A-40
A.16	\$STR_NEQ - String Inequality Comparison	A-41
A.16.1	Syntax	A-41
A.16.2	Restrictions	A-41
A.16.3	Parameter Semantics	A-41
A.16.4	Operational Semantics	A-42
A.16.5	Completion Codes	A-43
A.17	\$STR_SCAN - String Scanning	A-44
A.17.1	Syntax	A-44
A.17.2	Restrictions	A-44
A.17.3	Parameter Semantics	A-45
A.17.4	Operational Semantics	A-46
A.17.5	Completion Codes	A-47
A.18	\$XPO_BACKUP - Preserve an Input File	A-48
A.18.1	Syntax	A-48
A.18.2	Parameter Semantics	A-48
A.18.3	Usage Guidelines	A-49
A.18.4	Completion Codes	A-49
A.18.5	Example	A-50
A.19	\$XPO_CLOSE - Close a File	A-51
A.19.1	Syntax	A-51
A.19.2	Parameter Semantics	A-51
A.19.3	Usage Guidelines	A-52
A.19.4	Completion Codes	A-52
A.20	\$XPO_DELETE - Delete a File	A-54
A.20.1	Syntax	A-54
A.20.2	Parameter Semantics	A-54
A.20.3	Completion Codes	A-55
A.21	\$XPO_DESCRIPTOR - Declare a Data Descriptor	A-57
A.21.1	Syntax	A-57
A.21.2	Restrictions	A-57
A.21.3	Parameter Semantics	A-57
A.22	\$XPO_DESC_INIT - Initialize a Data Descriptor	A-59
A.22.1	Syntax	A-59
A.22.2	Parameter Semantics	A-59
A.22.3	Completion Code	A-60
A.23	\$XPO_FREE_MEM - Release a Memory Element	A-61
A.23.1	Syntax	A-61
A.23.2	Restrictions	A-61
A.23.3	Parameter Semantics	A-61
A.23.4	Completion Codes	A-62
A.24	\$XPO_GET - Read From a File	A-63
A.24.1	Syntax	A-63
A.24.2	Parameter Semantics	A-63
A.24.3	Usage Guidelines	A-64
A.24.4	Completion Codes	A-65
A.25	\$XPO_GET_MEM - Allocate Dynamic Memory Element	A-67

A.25.1	Syntax	A-67
A.25.2	Restrictions	A-67
A.25.3	Parameter Semantics	A-67
A.25.4	Completion Codes	A-68
A.26	\$XPO_IOB - Declare an IOB	A-70
A.26.1	Syntax	A-70
A.26.2	Parameter Semantics	A-70
A.26.3	Examples	A-70
A.27	\$XPO_IOB_INIT - Initialize an IOB	A-71
A.27.1	Syntax	A-71
A.27.2	Restrictions	A-71
A.27.3	Parameter Semantics	A-71
A.27.4	Completion Code	A-72
A.28	\$XPO_OPEN - Open a File	A-73
A.28.1	Syntax	A-73
A.28.2	Parameter Semantics	A-74
A.28.3	Completion Codes	A-77
A.29	\$XPO_PARSE_SPEC - Parse a File Specification	A-79
A.29.1	Syntax	A-79
A.29.2	Parameter Semantics	A-79
A.29.3	Completion Codes	A-80
A.30	\$XPO_PUT - Write to a File	A-81
A.30.1	Syntax	A-81
A.30.2	Restrictions	A-81
A.30.3	Parameter Semantics	A-82
A.30.4	Usage Guidelines	A-82
A.30.5	Completion Codes	A-82
A.31	\$XPO_PUT_MSG - Send a Message	A-84
A.31.1	Syntax	A-84
A.31.2	Parameter Semantics	A-84
A.31.3	Completion Codes	A-85
A.32	\$XPO_RENAME - Rename a File	A-86
A.32.1	Syntax	A-86
A.32.2	Parameter Semantics	A-87
A.32.3	Completion Codes	A-89
A.33	\$XPO_SPEC_BLOCK - Declare a File Specification Block	A-90
A.33.1	Syntax	A-90
A.33.2	Examples	A-90
A.34	\$XPO_TERMINATE - Terminate Program Execution	A-91
A.34.1	Syntax	A-91
A.34.2	Parameter Semantics	A-91
A.34.3	Routine Value	A-91

APPENDIX B CONTROL BLOCKS

B.1	INPUT/OUTPUT BLOCK (IOB)	B-2
B.2	STRING DESCRIPTORS	B-4
B.3	BINARY DATA DESCRIPTORS	B-5
B.4	FILE SPECIFICATION PARSE BLOCK	B-6

APPENDIX C	COMPLETION CODES	
APPENDIX D	SAMPLE PROGRAM	
APPENDIX E	ACTION ROUTINES	
E.1	ACTION-ROUTINE CALLS AND RETURNS	E-1
E.1.1	Action Routine Calls	E-1
E.1.2	Action Routine Return Values	E-3
E.2	XFAIL.BLI FAILURE-ACTION ROUTINE LISTING	E-4
E.3	SFAIL.BLI FAILURE-ACTION ROUTINE LISTING	E-15
APPENDIX F	COMPILING AND LINKING	
F.1	DEFINING A TRANSPORTABLE LOGICAL DEVICE	F-1
F.2	COMPILING	F-2
F.3	LINKING	F-3
APPENDIX G	XDUMP UTILITY PROGRAM	
G.1	XDUMP - XPORT DATA STRUCTURE DISPLAY UTILITY . . .	G-1
G.1.1	Running the XDUMP Utility	G-1
G.1.2	Compiling a Structure Display Module	G-2
G.1.3	Linking a Structure Display Module	G-3
G.1.4	Displaying a User Declared Structure While Debugging	G-3
G.2	XDESC, XIOB, and XSPEC - XPORT STRUCTURE DISPLAY MODULES	G-3
G.2.1	Linking an XPORT Structure Display Module . . .	G-3
G.2.2	Displaying an XPORT Structure While Debugging .	G-4
APPENDIX Z	EASY-TO-USE I/O PACKAGE (EZIO)	
Z.1	OVERVIEW	Z-1
Z.2	LIMITATIONS	Z-1
Z.3	FUNCTIONAL DESCRIPTION	Z-2
Z.3.1	The FILOPN Routine	Z-2
Z.3.2	The FILIN Routine	Z-4
Z.3.3	The FILOUT Routine	Z-4
Z.3.4	The FLLCLS Routine	Z-5
Z.3.5	Restrictions	Z-5
Z.3.6	Example	Z-6
Z.4	LOADING EZIO WITH USER PROGRAM	Z-7
Z.4.1	EZIOFC - File Services 11 (RSX-11M)	Z-7
Z.4.2	EZIORT - RT-11	Z-8
Z.4.3	EZIO10 - TOPS-10	Z-8
Z.4.4	EZIO20 - TOPS-20	Z-8
Z.4.5	EZIOVX - VAX/VMS	Z-8

Z.5	PACKAGING	Z-8
-----	---------------------	-----

CHAPTER 1 INTRODUCTION

1.1	APPLICABILITY OF XPORT FACILITIES	1-1
1.2	PROGRAM TRANSPORTABILITY	1-2
1.3	FILE TRANSPORTABILITY	1-3
1.4	SYMBOL NAMING CONVENTIONS	1-3
1.5	COMPILATION ERROR MESSAGES	1-4
1.6	SMALL SAMPLE PROGRAM	1-5

CHAPTER 1

INTRODUCTION

This manual describes a collection of transportable, source-level programming tools for use with the BLISS language. These tools provide extensive input/output facilities, a uniform interface for obtaining operating-system services - such as dynamic memory, and aids for data structuring and string handling.

Stand-alone BLISS utility programs, such as PRETTY or BLSCRF, are not described in this manual.

The Transportable Programming Tools package is informally referred to as XPORT - for "transportable" - which indicates the design emphasis on source-level transportability across all BLISS target systems, as well as on ease-of-learning and ease-of-use. (Transportability is discussed in further detail below.)

1.1 APPLICABILITY OF XPORT FACILITIES

Except for the data-structure definition aids described in Chapter 2, the facilities described in this manual are intended for use in development of 'application' programs, that is, programs that can make use of operating-system services, as opposed to the programs that make up the operating system itself. (The BLISS language, of course, is intended for implementation of programs in both categories.)

In the system-software context, an 'application' might be a compiler, linker, or utility program. The use of XPORT for such implementations offers programming convenience, maintainability, and transportability in the system-services area, combined with the efficiency and reliability obtainable through the use of BLISS.

Introduction

APPLICABILITY OF XPORT FACILITIES

The data-structuring facilities do not involve calls on underlying target-system services (as do the other XPORT facilities), and thus can be used in any type of program development.

1.2 PROGRAM TRANSPORTABILITY

With a very few exceptions, the XPORT programming tools are fully transportable. This means that, used in accordance with the guidelines given in this manual, the same XPORT source code will produce identical or equivalent results on each BLISS target system.

XPORT provides a small number of system-specific features deemed mandatory for some programs on a particular target system. An example of this type of feature is a field type, \$SIXBIT, that relates to the SIXBIT string encoding. Some programs for the TOPS-10 environment may require its use.

XPORT also provides a few discretionary features that allow you to take advantage of the storage characteristics of a particular system, e.g., to possibly eke out a bit more storage efficiency in certain situations.

Use of the discretionary type of non-transportable feature is strongly discouraged except in very unusual situations. In the majority of cases only a very little storage will be saved at the cost of transportability, which may turn out to be a very serious cost at some point in the product's lifecycle. Also, compaction of storage beyond that provided by default may incur a significant execution-speed penalty.

Obviously the use of XPORT will not make an otherwise non-transportable BLISS program into a transportable one. And as in the case of Common BLISS constructs, features of XPORT that are in themselves transportable can be used in completely non-transportable ways. This is due to the very significant architectural differences that exist among the several target-system families.

In order to use either Common BLISS constructs or XPORT features in a transportable fashion, these crucial differences must be recognized at the outset and kept constantly in mind. (One such difference is in the range of integer values possible on the 16-bit systems versus the 32- or 36-bit systems.) Just as with BLISS itself, transportable programming using XPORT is primarily a matter of design rather than of coding.

Introduction

PROGRAM TRANSPORTABILITY

Throughout this manual we have tried to provide transportability guidelines and warnings concerning all "problematic" features of XPORT. Transportable BLISS programming in general is discussed in the "Transportability Guidelines" chapter of the several BLISS User's Guides.

1.3 FILE TRANSPORTABILITY

There are two aspects of XPORT file transportability. The first concerns the transportability of I/O operations at the source-program level. With very few exceptions (clearly noted), the XPORT I/O functions behave the same in all environments.

The second aspect of file transportability concerns the ability to move XPORT-created files themselves from one environment to another. It is not a goal of XPORT to provide this kind of file transportability. Instead, standard file-interchange utilities - provided as a part of each target system - must be used to move files from system to system.

1.4 SYMBOL NAMING CONVENTIONS

The names used in the XPORT package are formed in accordance with VAX/VMS operating-system conventions. The names so formed are somewhat unwieldy in a few cases, but they all have the virtues of consistency and clarity. After you have learned the few simple patterns involved, you can easily identify the kind of entity (macro, completion code, control-block field, etc.) represented by any given name.

VAX/VMS-compatible names are formed of facility codes, dollar signs, underscores, and alphanumeric strings. The facility codes assigned to the XPORT package are the following (more may be added in future):

- o XPO - I/O, memory, and host-system service facilities
- o IOB - I/O control block definitions
- o STR - String-handling facilities.

Names formed according to the convention begin with one of the facility codes listed above. Arbitrarily, we will use the facility code XPO in the name formats and most of the examples given below.

Introduction
SYMBOL NAMING CONVENTIONS

The name formats used in XPORT are as follows:

- o `$XPO_xxxx` Macro name
Example: `$XPO_PUT`
- o `XPO$xxxx` Global symbol, usually a routine name
Example: `XPO$FAILURE`
- o `XPO$_xxxx` Completion code (special case of a literal name)
Example: `XPO$_NORMAL`
- o `XPO$c_xxxx` Variable or field name

where c is a data-type code, as follows:

c = A Address/pointer value
B Byte value (9-bit field on -10/20 systems)
G General value (fullword integer)
H Short value (two bytes)
K Literal ("konstant")
T Text string (STR descriptor)
V Arbitrary field (usually one or more bits)
Z Other

Examples:

`IOB$T_STRING` -- a string descriptor sub-block,
which includes the fields:

`IOB$H_STRING` -- a two-byte count value
`IOB$A_STRING` -- a character-string pointer

Two other forms are used for internal names, some of which you may encounter in listings and compiler diagnostics:

`XPOxxxx` and `$XPO$$xxxx`

In Chapter 2, the macros associated with the data-structuring facility have simple names of the form "`$xxxx`", such as `$FIELD`, `$INTEGER`, `$TINY_INTEGER`, `$DESCRIPTOR`, and `$OVERLAY`. This is so because the data-structuring facility has no facility code assigned to it; the names otherwise adhere to the naming convention.

1.5 COMPILATION ERROR MESSAGES

The use of XPORT facilities usually implies the expansion of many macros during compilation. Macro expansion following the occurrence of a source-language error can result in the generation of spurious error messages.

Introduction

COMPILATION ERROR MESSAGES

Once an error-level diagnostic (prefixed %ERR) has been generated, the compilers stop performing most expression evaluations. This causes the object file to be discarded and, assuming there are macros in the source program, sometimes results in erroneous macro expansions and the consequent generation of spurious error messages.

If the cause of a compiler diagnostic is not immediately evident (especially one referring to a macro expansion) and an error-level diagnostic has already been issued, ignore the diagnostic in question, correct all known source errors, and recompile. Any spurious error messages will "go away" when the preceding actual errors are eliminated.

1.6 SMALL SAMPLE PROGRAM

As an initial "taste" of XPORT coding, we present below a minimal sample program consisting almost entirely of XPORT macro calls. This program is a simple file-copy utility, called LISTER.

Appendix D contains a much more extensive programming example which illustrates the use of many XPORT features in a completely realistic context.

SAMPLE PROGRAM

```
MODULE LISTER ( MAIN = LISTER ) =  
  
BEGIN  
!+  
! This program asks for a file name, opens the named file,  
! and copies the file to the terminal.  
!-  
  
LIBRARY 'BLI:XPORT';                ! XPORT definitions  
  
OWN  
    terminal_iob : $XPO_IOB(),  
    file_iob : $XPO_IOB();  
  
ROUTINE LISTER =  
    BEGIN  
  
! Open the user's terminal and ask for a file name.  
  
    $XPO_OPEN( IOB=terminal_iob, FILE_SPEC=$XPO_INPUT );  
  
    $XPO_GET( IOB=terminal_iob,  
              PROMPT= 'Enter name of file to be listed: ' );
```

Introduction
SMALL SAMPLE PROGRAM

! Open the file.

```
$XPO_OPEN( IOB=file_iob, FILE_SPEC=terminal_iob[IOB$T_STRING] );
```

! Process each line.

```
WHILE $XPO_GET( IOB=file_iob ) DO  
  $XPO_PUT( IOB=terminal_iob, STRING=file_iob[IOB$T_STRING] );
```

```
$XPO_CLOSE( IOB=file_iob );
```

```
$XPO_PUT( IOB=terminal_iob, STRING= '*** End of file ***' );
```

```
$XPO_CLOSE( IOB=terminal_iob )
```

```
END;
```

END

ELUDOM

Note: \$XPO_INPUT is the transportable name for the standard input device. It is, for example, equivalent to "TTY:" on some systems.

CHAPTER 2 TRANSPORTABLE DATA STRUCTURES

2.1	INTRODUCTION	2-1
2.1.1	The Problem	2-1
2.1.2	The Solution	2-2
2.1.3	Simple Example	2-3
2.1.4	Terminology	2-5
2.2	\$FIELD DECLARATION AND \$FIELD_SET_SIZE	2-5
2.2.1	\$FIELD Declaration	2-5
2.2.2	Transportable Field-Types	2-6
2.2.3	Nontransportable Field-Types	2-8
2.2.4	Guidelines for Individual Field-Types	2-8
2.2.4.1	\$ADDRESS vs. \$POINTER Usage	2-8
2.2.4.2	Coding the \$SUB_BLOCK Field-Type	2-9
2.2.4.3	Coding the \$DESCRIPTOR Field-Type	2-10
2.2.4.4	Coding the \$REF_DESCRIPTOR Field-type	2-11
2.2.4.5	Coding the \$STRING Field-Type	2-11
2.2.5	\$FIELD_SET_SIZE Usage	2-12
2.3	SUPPLEMENTARY FEATURES	2-12
2.3.1	Field-Positioning Features	2-12
2.3.1.1	\$ALIGN Usage	2-13
2.3.1.2	\$OVERLAY and \$CONTINUE Usage	2-14
2.3.2	Literal-Defining Features	2-14
2.3.2.1	\$LITERAL and \$DISTINCT Usage	2-15
2.3.3	Value-Display Feature	2-15
2.3.3.1	\$SHOW Usage	2-15
2.3.4	Subfield Referencing Feature	2-16
2.3.4.1	Subfield Referencing Using a BIND Declaration	2-17
2.3.4.2	Subfield References Using General Structure References	2-18
2.3.4.3	Subfield References Using \$SUB_FIELD	2-19
2.3.4.4	Using \$SUB_FIELD with \$OVERLAY	2-19
2.4	TRANSPORTABILITY CONCERNS	2-20
2.4.1	Field Size	2-20
2.4.2	Integer Value Range	2-21
2.4.3	Use of \$BYTES for Character Strings	2-21
2.5	EFFICIENCY CONCERNS	2-22

CHAPTER 2

TRANSPORTABLE DATA STRUCTURES

2.1 INTRODUCTION

This chapter describes a high-level aid for defining BLOCK-type data structures in a convenient and transportable manner.

2.1.1 The Problem

The specification of reasonably compact, non-uniform data structures -- typically control blocks -- for use across target systems poses a difficult transportability problem. The difficulty is caused by the basic architectural differences found among the BLISS target systems, as discussed in Chapters I and III of the BLISS Language Guide.

Briefly, the differences are found in the following system characteristics:

- o The size of the fullword (or BLISS value): 16, 32, or 36 bits
- o The size of the addressable unit: 8 bits vs. 36 bits
- o The ability to access fields that cross fullword boundaries: VAX-11 only

These differences affect the transportable design of all types of data segments, scalar or structured, to some extent. But in particular, the difficulty of defining BLOCK and BLOCKVECTOR structures transportably, using standard BLISS declarations, is such that the programmer often ducks the issue altogether and resorts to separate sets of FIELD and data declarations for each target system.

Transportable Data Structures

INTRODUCTION

The numeric notation ordinarily used in the standard FIELD declaration -- e.g., [1,0,16,1] -- while powerful, is inconvenient to code and modify and is quite error-prone, in addition to being nontransportable.

2.1.2 The Solution

The XPORT structure-definition facility is a collection of macros that provides a solution to these problems. It allows you to define efficient BLOCK structures in a way that is both convenient and, for the most part, system independent. The facility consists principally of a replacement for the standard BLISS FIELD declaration, using the keyword \$FIELD.

The facility also includes several groups of supporting features, namely:

\$<field-types>	}	
\$FIELD_SET_SIZE	}	Block-defining features
\$ALIGN	}	
\$OVERLAY	}	Field-positioning features
\$CONTINUE	}	
\$LITERAL	}	
\$DISTINCT	}	Literal-defining features
\$SHOW	}	Value-display feature
\$SUB_FIELD	}	Subfield-referencing feature

The overall strategy of the structure-definition facility is to allow you to name the kind of field required for each block component, as opposed to specifying its (necessarily machine dependent) position and size, and to have the compiler calculate the appropriate structure-reference values and the aggregate block size for each target system. A "kind of field" or field-type (e.g., \$SHORT_INTEGER) generally implies not only the size but also the alignment and sign-extension mode required.

In addition to transportability, this facility offers a degree of coding convenience (and consequent ease of modification) that alone justifies its use.

Transportable Data Structures INTRODUCTION

CAVEAT

The XPORT structure-definition facility cannot solve all transportability problems. Particularly, it cannot of itself solve the kind of transportability problem that is often encountered when transporting a program between a 32- or 36-bit system on one hand and a 16-bit system on the other, caused by the large discrepancy in BLISS-value size between those two system groups. Section 2.4 discusses such transportability problem areas in some detail.

2.1.3 Simple Example

The following example of a fairly simple block structure called XCB (for Xample Control Block), gives the general flavor of the XPORT structure-definition facility:

```
$FIELD
    XCB_FIELDS =
        SET
            XCB_1 = [$ADDRESS],
            XCB_2 = [$TINY_INTEGER],
            XCB_3 = [$BYTE],
            XCB_4 = [$SHORT_INTEGER],
            XCB_5 = [$POINTER]
        TES;
LITERAL
    XCB_SIZE = $FIELD_SET_SIZE ;

. . .

OWN
    XCB_ALPHA : BLOCK[XCB_SIZE] FIELD(XCB_FIELDS) ;
```

If the XPORT data-structure macros were not used, the following sets of equivalent, system-specific declarations would be required for each BLISS dialect:

Transportable Data Structures
INTRODUCTION

```
FOR BLISS-16 =>
  FIELD
    XCB_FIELDS =
      SET
        XCB_1 = [0,0,16,0],
        XCB_2 = [1,0,8,1],
        XCB_3 = [1,8,8,0],
        XCB_4 = [2,0,16,1],
        XCB_5 = [3,0,16,0]
      TES;
    . . .

  OWN
    XCB_ALPHA : BLOCK[4] FIELD(XCB_FIELDS) ;

FOR BLISS-32 =>
  FIELD
    XCB_FIELDS =
      SET
        XCB_1 = [0,0,32,0],
        XCB_2 = [1,0,8,1],
        XCB_3 = [1,8,8,0],
        XCB_4 = [1,16,16,1],
        XCB_5 = [2,0,32,0]
      TES;
    . . .

  OWN
    XCB_ALPHA : BLOCK[3] FIELD(XCB_FIELDS) ;

FOR BLISS-36 =>
  FIELD
    XCB_FIELDS =
      SET
        XCB_1 = [0,0,18,0],
        XCB_2 = [0,18,9,1],
        XCB_3 = [0,27,9,0],
        XCB_4 = [1,0,18,1],
        XCB_5 = [2,0,36,0]
      TES;
    . . .

  OWN
    XCB_ALPHA : BLOCK[3] FIELD(XCB_FIELDS) ;
```

Transportable Data Structures

INTRODUCTION

2.1.4 Terminology

The coined term "fullword" stands for "word" on the PDP-11, for "longword" on the VAX-11, and for "word" on the DEC-10/20 systems, and corresponds to the size of a BLISS value on all target systems.

The BLISS predefined literal name %BPVAL is used in subsequent descriptions to denote the number of bits in a fullword (i.e., bits per BLISS value) for any target system. That is, %BPVAL implies the value 16, 32, or 36 for BLISS-16, BLISS-32, or BLISS-36 respectively.

The BLISS predefined literal name %UPVAL is used in subsequent descriptions to denote the number of addressable units in a fullword (i.e., units per BLISS value) for any target system. That is, %UPVAL implies the value 2, 4, or 1 for BLISS-16, BLISS-32, or BLISS-36 respectively.

Also, the abbreviation "DEC-10/20" is used to stand for "DECsystem-10 or DECSYSTEM-20". It denotes the entire 36-bit family of target systems.

2.2 \$FIELD DECLARATION AND \$FIELD_SET_SIZE

The XPORT \$FIELD declaration is used in place of the BLISS FIELD declaration to define the structure-reference actuals for fields of a BLOCK structure. The \$FIELD_SET_SIZE feature is used to calculate the size of a block defined via \$FIELD.

2.2.1 \$FIELD Declaration

The general form of a \$FIELD declaration is:

```
$FIELD field-set-name =  
    SET  
    field-name-1 = [field-type],  
    field-name-2 = [field-type],  
    .  
    .  
    field-name-n = [field-type]  
    TES;
```

Transportable Data Structures
\$FIELD DECLARATION AND \$FIELD_SET_SIZE

where field-set-name and field-name-i are user-chosen names, as in a FIELD declaration

field-type is one of the (macro) names defined in the following two subsections.

The XPORT \$FIELD declaration is used in much the same way as the standard FIELD declaration. The major differences are summarized below.

- o A field-type name must be used instead of the conventional list of numeric values in each field-definition. The field-type names are actually macro calls (as is the 'keyword' \$FIELD itself).
- o The \$FIELD keyword indicates the beginning of a new block-structure description (the macro call generates counter-initializing code as well as a "FIELD" lexeme). This implies that, in normal usage, each \$FIELD declaration will contain exactly one field-set-definition.
- o The ordering of the individual field-definitions within the declaration is significant. That is, the order in which the fields are specified determines the physical ordering of the corresponding block components. Fields are packed as densely as possible, and thus several fields may be allocated within one fullword.
- o The \$ALIGN feature can be used preceding a field-definition to force a non-default boundary alignment for that field. The use of \$ALIGN is described in Section 2.3.1.
- o The \$OVERLAY and \$CONTINUE features can be used within the declaration to create overlapping field-definitions; the use of these features is described in Section 2.3.1.

2.2.2 Transportable Field-Types

The following field-types can be used with any BLISS compiler:

<u>Field Type</u>	<u>Definition</u>
\$BIT	Single bit field
\$BITS(n)	Specified number of bits - transportability limited if n>16 (see Section 2.4.1)

Transportable Data Structures
\$FIELD DECLARATION AND \$FIELD_SET_SIZE

\$BYTE	Unsigned field: eight bits on a PDP-11 or VAX-11; nine bits on a DEC-10/20
\$BYTES(n)	Unsigned field: specified number of bytes - transportability limited if n>2 (see Section 2.4.1). Not to be used for character strings; see Section 2.4.3.
\$INTEGER	Signed fullword integer (two bytes on a PDP-11, four bytes on a VAX-11 or a DEC-10/20)
\$TINY_INTEGER	Signed one-byte integer
\$SHORT_INTEGER	Signed two-byte integer
\$LONG_INTEGER	Signed four-byte integer - transportability limited for PDP-11 (see Section 2.4.1)
\$ADDRESS	Address of a memory location (fullword on a PDP-11 or VAX-11, two bytes on a DEC-10/20)
\$POINTER	Pointer to a character position in memory (a fullword on all target systems; to be used with CH\$PTR)
\$SUB_BLOCK(len)	Fullword-aligned beginning of a substructure within the current structure, where the length <u>len</u> is specified in fullwords. (The resulting field description may have a zero length.) See Section 2.2.4.2 for usage guidelines.
\$DESCRIPTOR(class)	Fullword aligned, standard character-string or binary-data descriptor sub-block, the length of which varies according to the class of descriptor requested. The class may be FIXED, BOUNDED, DYNAMIC, DYNAMIC_BOUNDED, or UNDEFINED. See Section 2.2.4.3 for usage guidelines. Descriptors in general are discussed in Chapter 6.
\$REF_DESCRIPTOR	Address of a character-string or binary-data descriptor (see Section 2.2.4.4)
\$STRING(n)	Specified number of ASCII character positions, i.e., a character-position sequence (eight bits per character for PDP-11 or VAX-11, seven bits per character for DEC-10/20), addressable-unit aligned. See Section 2.2.4.5 for usage guidelines.

Transportable Data Structures
\$FIELD DECLARATION AND \$FIELD_SET_SIZE

2.2.3 Nontransportable Field-Types

An additional nontransportable field-type, BLISS-36 only, is provided for the sake of completeness:

<u>Field Type</u>	<u>Definition</u>
\$SIXBIT(n)	Specified number of 6-bit characters, where n must be a multiple of three (DEC-10/20 only)

2.2.4 Guidelines for Individual Field-Types

The use of the bit, byte, and integer form of XPORT field-type is straightforward and does not require further discussion, except under the heading of "Transportability Concerns" (Section 2.4). The proper use of \$ADDRESS, \$POINTER, \$SUB_BLOCK, \$DESCRIPTOR, \$REF_DESCRIPTOR, and \$STRING, however, may not be completely obvious. These field-types are discussed individually below.

2.2.4.1 \$ADDRESS vs. \$POINTER Usage

On the PDP-11 and VAX-11, the \$ADDRESS and \$POINTER field-types produce the same size field, a fullword in each case. In the DEC-10/20 environment (without extended addressing), however, these two field-types produce quite different-sized fields: 18 bits vs. 36 bits respectively. It is important to understand the distinction between these two related types of field, and their intended usage.

An address is simply the storage address of a binary data item or structure. A pointer is a 'character address' that designates a position in a BLISS character-position sequence, as interpreted by the BLISS character-handling (CH\$xxxx) functions. For example, a pointer to the nth character of a character string beginning at a known address can be calculated using the BLISS CH\$PTR function. Thus, "pointer" denotes a specialized, transportable kind of address value for string data.

In a 36-bit environment without extended addressing, an address can be represented by only 18 bits (half a BLISS fullword), but a pointer requires 36 bits (an entire BLISS fullword). In a 16-bit or 32-bit environment, an address and a pointer have the same internal representation, each requiring an entire BLISS fullword.

Transportable Data Structures
\$FIELD DECLARATION AND \$FIELD_SET_SIZE

2.2.4.2 Coding the \$SUB_BLOCK Field-Type

The intended purpose of the \$SUB_BLOCK field-type is to signify the beginning of a related group of fields within a block structure. This field-type can be used in one of two ways:

1. As a "placeholder" only. In the form \$SUB_BLOCK() or \$SUB_BLOCK(0), it identifies a specific point in a block but reserves no space. When used in this form, the next field-definition defines a field beginning at the same point as that identified by the "name = [\$SUB_BLOCK()]" definition. (Fullword alignment is implicit for this field-type.) For example, consider the following \$FIELD-declaration fragment:

```
SET
. . .

STATUS_CELLS = [$SUB_BLOCK()],
COMP_CODE   = [$BYTE],
RETRY_STATUS = [$BITS(3)],
UPDATE_STATUS = [$BIT],
. . .

TES;
```

In the block structure defined by this declaration, the field COMPL_CODE begins at the fullword also identified by the field-name STATUS_CELLS. The implicit fullword alignment for field-name STATUS_CELLS ensures that that the positions identified by these two names will in fact coincide, on any system and no matter what definitions precede the ones shown.

A field description generated by \$SUB_BLOCK() always has a size-value of zero, and thus can only be used in a context that does not specify or imply indirection, that is, in a non-fetch/non-store context.

2. As a "spaceholder". In the form \$SUB_BLOCK(len), where len is the length of the sub-block in addressable units, it identifies a specific point in a block and reserves a specified amount of space. When used in this form, the next field-definition defines a field that begins immediately following the sub-block being defined. Consider the following coding fragment:

Transportable Data Structures
\$FIELD DECLARATION AND \$FIELD_SET_SIZE

```
SET
. . .

INPUT_IOB = [$SUB_BLOCK(IOB$K_LENGTH)],
OUTPUT_IOB = [$SUB_BLOCK(IOB$K_LENGTH)],
TEMP_IOB = [$SUB_BLOCK(IOB$K_LENGTH)],
SERVICE_CODE = [$BYTE],
. . .
```

TES;

In this example three separate sub-blocks, each 'IOB\$K_LENGTH' units long, are defined in sequence. The SERVICE_CODE field follows the sub-block TEMP_IOB.

A field description generated by \$SUB_BLOCK(n) where n is either 0 or greater than %UPVAL has a size-value of zero, and can only be used in an address (i.e., non-fetch, non-store) context.

The field-positioning features \$OVERLAY and \$CONTINUE are often used in combination with this field-type, to define individual fields within the sub-block. The use of these features is shown in a later example (Section 2.3.1.2).

2.2.4.3 Coding the \$DESCRIPTOR Field-Type

The purpose of the specialized \$DESCRIPTOR(class) field-type is to define the beginning and extent of a standard XPORT sub-block, called a descriptor, within a larger block structure. The size of this sub-block varies according to the class of descriptor specified; the class keywords are FIXED, BOUNDED, DYNAMIC, DYNAMIC_BOUNDED, and UNDEFINED. Section 6.1 of this manual describes the properties of the different classes, as well as descriptor usage in general.

The \$DESCRIPTOR field-type is actually equivalent to the "spaceholder" form of \$SUB_BLOCK, that is, to \$SUB_BLOCK(len) where the value of len is implied by a class keyword. As for \$SUB_BLOCK, fullword alignment is implicit. Also, the field description generated by \$DESCRIPTOR always has a size-value of zero, and can only be used in an address context. The default class is FIXED for a field definition of the form \$DESCRIPTOR().

The field-positioning features \$OVERLAY and \$CONTINUE can be used in combination with this field-type, to define individual fields within the descriptor. The use of these positioning features is shown in a later example (Section 2.3.1.2).

Transportable Data Structures \$FIELD DECLARATION AND \$FIELD_SET_SIZE

The various string and binary-data descriptors included within the XPORT I/O control block (IOB), described in Section 3.4 and Appendix B, provide an actual instance of descriptors used as sub-structures (as opposed to independent block structures).

2.2.4.4 Coding the \$REF_DESCRIPTOR Field-type

The purpose of the specialized \$REF_DESCRIPTOR field-type is to define a field that is the address of either a character-string or binary-data descriptor. (See Chapters 6 and 7 for general information on descriptors.) For the purposes of field declaration, the \$REF_DESCRIPTOR field-type is equivalent to \$ADDRESS.

Use of the \$REF_DESCRIPTOR field-type has two benefits, however: (1) the field declaration in which it is used is more self-documenting, and (2) the XDUMP utility can provide a more symbolic field display during program debugging. (See Appendix G for a description of XDUMP.)

2.2.4.5 Coding the \$STRING Field-Type

The \$STRING(n) field-type reserves space for an ASCII character-position sequence of length n. Each character position occupies eight bits on the PDP-11 or seven bits on the DEC-10/20. The BLISS character-handling (CH\$xxxx) functions, described in Chapter 20 of the BLISS Language Guide, are specifically designed to access and manipulate such sequences in a completely transportable fashion. For example, a pointer to the beginning (first character position of) a transportable string field defined as, e.g.,

```
STRING_BUFFER = [$STRING(80)]
```

should be constructed with the function CH\$PTR(address), for example, CH\$PTR(IOBLOCK[STRING_BUFFER]).

The field description generated by a \$STRING(n) definition has a zero size value if n specifies a character-position sequence whose length exceeds %BPVAL bits (or if n is 0). In usual practice, however, this is of no consequence because the field-name is only used to generate a character-position pointer as described above. A \$STRING field is aligned to the nearest (higher order) addressable-unit boundary: A byte boundary for the PDP-11 or VAX-11, or a word boundary for the DEC-10/20. (This ensures that the field-reference implied by the field name is usable as a primary expression, i.e., as an address value.)

Transportable Data Structures
\$FIELD DECLARATION AND \$FIELD_SET_SIZE

2.2.5 \$FIELD_SET_SIZE Usage

The \$FIELD_SET_SIZE feature calculates the length, in fullwords, of a block defined with the \$FIELD declaration. You would ordinarily use this feature in a LITERAL declaration following a \$FIELD declaration, to obtain the number of fullwords needed to accomodate the fields defined in that declaration. (The usual placement is immediately following the \$FIELD declaration, since \$FIELD_SET_SIZE implicitly refers to the last such declaration.) For example:

```
$FIELD
    field-set-name =
    SET
        .
        .
        .
    TES;

LITERAL literal-name = $FIELD_SET_SIZE;
```

The example given at the beginning of this chapter amply illustrates the use of this feature and its transportability aspects. As demonstrated by that example, \$FIELD_SET_SIZE produces the exact number of fullwords needed to allocate a BLOCK structure.

2.3 SUPPLEMENTARY FEATURES

The supplementary features of the XPORT structure-definition facility are described below under several different functional categories. Like the rest of the structure-definition facilities, these features are implemented as macro calls.

2.3.1 Field-Positioning Features

The field-positioning features -- \$ALIGN, \$OVERLAY, and \$CONTINUE -- are used within the \$FIELD declaration and allow you to alter the default positioning of a field within a structure.

By default, the only boundary alignment performed by the \$FIELD declaration is the minimum required by the respective machine architectures. For example, the PDP-11 and DEC-10/20 systems cannot fetch from or store into a field that spans a fullword boundary. Therefore, for those systems \$FIELD provides fullword alignment for such fields that would otherwise cross that boundary. On the VAX-11, however, no such restriction exists and no boundary alignment is performed for a fetchable field (i.e., any field of up to %BPVAL bits in length).

Transportable Data Structures SUPPLEMENTARY FEATURES

2.3.1.1 \$ALIGN Usage

The \$ALIGN(mode) feature forces a specified mode of alignment for the immediately following field-definition. The alignment modes are BYTE, WORD, UNIT, and FULLWORD. \$ALIGN(xxx) causes the subsequently defined field to begin at the next xxx-type boundary point assuming that it is not already at such a boundary by default. The precise meaning of the mode keywords are as follows:

- o BYTE indicates alignment to the next (8-bit) byte boundary for a PDP-11 or VAX-11; or to the next 9-bit byte position on a DEC-10/20 system, that is, to the next bit that is a multiple of 9.
- o WORD indicates alignment to the next even-numbered byte boundary for a PDP-11 or VAX-11; or to the next fullword or halfword boundary for a DEC-10/20 system, whichever is encountered first.
- o UNIT indicates alignment to the next addressable-unit: to a byte boundary on a PDP-11 or VAX-11, or to a word boundary on a DEC-10/20.
- o FULLWORD indicates alignment to the next fullword boundary on any system.

For example:

```
$FIELD
  FIELDSET_A =
  SET
  FIELD_A1 = [$SHORT_INTEGER],
  FIELD_A2 = [$BYTE],
    $ALIGN(FULLWORD)
  FIELD_A3 = [$ADDRESS],
    . . .
  TES;
```

The use of \$ALIGN here ensures that field FIELD_A3 will fall on a fullword boundary on any system. It also ensures that this alignment is insensitive to any changes that may be made to preceding field definitions. (See Section 2.4 on efficiency considerations.)

Transportable Data Structures SUPPLEMENTARY FEATURES

2.3.1.2 \$OVERLAY and \$CONTINUE Usage

The \$OVERLAY(field) feature causes the next field to begin at the same point in the structure as the named field, which itself must be defined by a preceding field-definition. Essentially it allows you to create alternate definitions for a portion of a structure.

The \$CONTINUE feature is used after a \$OVERLAY and, as its name implies, allows you to end a sequence of overlapping definitions and continue defining fields at the "end" of the structure. That is, it causes the field defined subsequent to it to start at a point following the highest-order position already occupied by any previously defined field.

Consider the following coding fragment:

```
. . .  
STATUS = [$BITS(16)],  
    $OVERLAY(STATUS)  
INITIAL = [$BIT],  
OPEN    = [$BIT],  
ERROR   = [$BIT],  
EOF     = [$BIT],  
    $CONTINUE  
NEXTFIELD = [$INTEGER],  
. . .
```

In this example, the INITIAL, OPEN, ERROR, and EOF bit fields all overlap the 16-bit field named STATUS, while NEXTFIELD starts beyond the STATUS field, or indeed beyond the furthestmost field yet defined. The STATUS field has 12 high-order bits that, presumably, are reserved for future use.

2.3.2 Literal-Defining Features

The literal-defining features \$LITERAL and \$DISTINCT are used in conjunction with (but outside of) a \$FIELD declaration. They allow you to define a sequence of literal values in a convenient and easily maintainable fashion.

Transportable Data Structures SUPPLEMENTARY FEATURES

2.3.2.1 \$LITERAL and \$DISTINCT Usage

These two features are used together to generate members of the integer set 1,2,3,4,... and assign them in ascending order to literal names. This is useful, for example, for defining an arbitrary but dense set of status-code values and assigning names to those values.

The \$LITERAL 'keyword' essentially provides a modified form of the BLISS LITERAL declaration, in which the \$DISTINCT feature may be used. The general form of this declaration is:

```
$LITERAL
    literal-name-1 = $DISTINCT,
    literal-name-2 = $DISTINCT,
    literal-name-3 = $DISTINCT,
    literal-name-4 = $DISTINCT,
    . . .
    literal-name-n = $DISTINCT;
```

\$LITERAL initializes an internal "value counter" to 0; \$DISTINCT bumps that counter by 1. Thus the literal names are given the values 1 through n in the order of their declaration.

Although this feature has no specific transportability aspect, it is a coding convenience that particularly facilitates the modification or maintenance task when such a set of literals needs to be altered.

2.3.3 Value-Display Feature

The \$SHOW feature allows the numeric values generated by the XPORT \$<field-type>, \$FIELD_SET_SIZE, and \$DISTINCT macros to be displayed in the program immediately following each macro call. It also allows the suppression of any informational-level messages (prefixed %INFORM) concerning \$FIELD usage.

2.3.3.1 \$SHOW Usage

\$SHOW is used anywhere in a module to enable or disable the display of subsequent XPORT-generated field definitions, literal values, and informational messages related to \$FIELD usage. The general form of the \$SHOW macro is

```
$SHOW ( display-keyword ,... )
```

Transportable Data Structures SUPPLEMENTARY FEATURES

where the display-keywords are:

FIELDS or NOFIELDS	- Display/don't display field definitions
LITERALS or NOLITERALS	- Display/don't display literal values
INFO or NOINFO	- Give/suppress informational messages concerning field definitions
ALL	- Display both definitions and values, and give informational messages
NONE	- Don't display any definitions, and suppress informational messages.

The initial (default) \$SHOW options are NOFIELDS, NOLITERALS, and INFO.

An example of the \$SHOW macro is

```
$SHOW( FIELDS, NOLITERALS )
```

which would cause the actual field-definition values generated by any subsequent \$FIELD declaration to appear in the source listing. The values are displayed immediately following the source line to which they correspond. (Any \$FIELD-related informational messages would be reported by default, assuming no prior setting of NOINFO.)

2.3.4 Subfield Referencing Feature

(Note to readers: If this manual were organized according to levels of difficulty, the following material would be presented under "Advanced Features". Therefore, if the material in this section seems obscure to you, it is likely that you have not yet experienced the need for a subfield-referencing capability.)

The \$SUB_FIELD feature provides a convenient and clear means of referring to a subfield; that is, to a field within a substructure. This feature can be used instead of, or in addition to, methods of subfield referencing involving BIND declarations or the use of general-structure-references.

The need for a subfield-referencing mechanism arises when a data structure is defined in a composite fashion, using separate "building blocks", as shown in the subsequent example. The building blocks can be either explicit \$FIELD declarations or can be implicit (predefined) XPORT descriptor structures; see the \$DESCRIPTOR field-type and Section 6.1.

Transportable Data Structures SUPPLEMENTARY FEATURES

The following declarations provide a context for discussion and examples of subfield referencing. The XCB structure definition given in Section 2.1.3 is referred to in the first declaration, and forms a part of the context.

```
$FIELD
  COMPOSITE_BLOCK =
    SET
      X_CHAIN    = [$ADDRESS],
      X_CONTROL  = [$SUB_BLOCK(XCB_SIZE)],
      X_NAME     = [$DESCRIPTOR(FIXED)],
      X_BUFFER   = [$DESCRIPTOR(BOUNDED)]
    TES;

LITERAL COMP_BLK_SIZE = $FIELD_SET_SIZE;

OWN
  X_PROCESS : BLOCK[COMP_BLK_SIZE] FIELD(COMPOSITE_BLOCK);
GLOBAL
  X_REPORT : BLOCK[COMP_BLK_SIZE] FIELD(COMPOSITE_BLOCK);
```

The structure described by the COMPOSITE BLOCK field-set consists of a single address field and three sub-blocks. The format and size of each of the sub-blocks are defined at some point prior to this code fragment (see Section 2.1.3 and Appendix B.2).

Note carefully at this point that, given the preceding declarations, the ordinary-structure-references

```
X_PROCESS[XCB_1]
```

and

```
X_PROCESS[STR$H_LENGTH]
```

would be incorrect because the subfield names are not defined relative to the origin of the entire block.

2.3.4.1 Subfield Referencing Using a BIND Declaration

To facilitate references to X_CONTROL subfields within the X_PROCESS block, for example, you could write the declaration

```
BIND
  PROCESS_CTL = X_PROCESS[XCONTROL]: BLOCK[] FIELD(XCB_FIELDS);
```

Transportable Data Structures SUPPLEMENTARY FEATURES

Then, within the scope of this declaration, you would be able to use the following fetch expression:

```
.PROCESS_CTL[XCB_1]
```

(The subfields XCB_2, XCB_3, etc., could also be referred to in the same way, of course.)

Similar references to X_CONTROL subfields of the X_REPORT block would, however, require an additional BIND declaration, as would references to X_NAME or X_BUFFER subfields of either block. For example, ordinary-structure-references to X_BUFFER subfields of the X_REPORT block would require a governing declaration of the form

```
BIND
  REPORT_BUF = X_REPORT[X_BUFFER]: $STR_DESCRIPTOR();
```

where \$STR_DESCRIPTOR() is an XPORT macro, described in Chapter 6, that provides a structure-attribute and field-attribute for standard XPORT string descriptors. Within the scope of this BIND declaration, you would be able to use expressions such as

```
.REPORT_BUF[STR$H_LENGTH]
```

2.3.4.2 Subfield References Using General Structure References

Sub-field references equivalent to those shown above can be achieved using a general-structure-reference, as shown in the following parallel examples of fetch expressions:

```
.BLOCK[X_PROCESS[X_CONTROL],XCB_1]
.BLOCK[X_REPORT[X_BUFFER],STR$H_LENGTH]
```

These expressions are more verbose than their equivalents in the preceding subsection, but have the advantage of not requiring the BIND declarations when only one or two references to each substructure are to be made. The syntax of a general-structure-reference does, however, leave something to be desired in the way of clarity and simplicity.

Transportable Data Structures SUPPLEMENTARY FEATURES

2.3.4.3 Subfield References Using \$SUB_FIELD

The \$SUB_FIELD feature offers the same advantage as the general-structure-reference for subfield references, and in addition provides clarity of intent and a simpler syntax. The general form of a \$SUB_FIELD reference is

```
$SUB_FIELD( substructure-name, field-name )
```

where the definition of <field-name> and the definition of <substructure-name> occur in different \$FIELD declarations.

The following fetch expressions, using \$SUB_FIELD, are equivalent to the fetch expressions given in the preceding two subsections:

```
.X_PROCESS[$SUB_FIELD(X_CONTROL,XCB_1)]  
.X_REPORT[$SUB_FIELD(X_BUFFER,STR$H_LENGTH)]
```

2.3.4.4 Using \$SUB_FIELD with \$OVERLAY

When a particular subfield of a structure will be referred to frequently, it is common practice to provide an explicit definition of that subfield within the overall structure definition. The following alternative definition of the COMPOSITE_BLOCK structure demonstrates the use of \$SUB_FIELD with \$OVERLAY in order to declare an "explicit subfield":

```
$FIELD  
  COMPOSITE_BLOCK =  
    SET  
      X_CHAIN    = [$ADDRESS],  
      X_CONTROL  = [$SUB_BLOCK(XCB_SIZE)],  
      X_NAME     = [$DESCRIPTOR(FIXED)],  
      X_BUFFER   = [$DESCRIPTOR(BOUNDED)],  
      $OVERLAY( $SUB_FIELD(X_BUFFER,STR$H_MAXLEN) )  
      X_MAX_BUFF_SIZE = [$BYTES(2)]  
      $CONTINUE  
    TES;
```

Thus the following two references to the subfield in question would be equivalent:

```
.X_PROCESS[X_MAX_BUFF_SIZE]  
.X_PROCESS[$SUB_FIELD(X_BUFFER,STR$H_MAXLEN)]
```

Transportable Data Structures TRANSPORTABILITY CONCERNS

2.4 TRANSPORTABILITY CONCERNS

Two related problem areas cause most of the transportability concerns that arise in connection with XPORT data structures: field size, and integer value range. Another, lesser problem is the possible confusion of \$BYTES(n) with \$STRING(n) as a means of transportably defining a character-string field. These potential problem areas are discussed individually below.

2.4.1 Field Size

Several of the XPORT field-types can cause a transportability problem with respect to field size (assuming that the field being defined is to be fetched from or stored into). The problem stems from the fact that a fetchable/storable field cannot exceed a fullword (%BPVAL bits) in length on any system. The problematic field-types are:

1. \$LONG_INTEGER. This field-type is not fully transportable to the PDP-11, since no more than two bytes can be fetched/stored on that system. If compiled for that system, it will reserve four bytes in the data structure but the corresponding field description will have a zero size value and will be useful in an address context only. (Note here that the field-type \$INTEGER will produce the exact same effect as \$LONG_INTEGER on the VAX-11 and DEC-10/20, and is fully transportable to the PDP-11 -- albeit with a radical reduction in integer value range, as discussed below.)
2. \$BITS(n). If the value of n exceeds 16, this field type is not fully transportable across all target systems, for the reasons given under item (1) above. (This field-type, however, is often used as a "spaceholder" for a collection of bits that is not typically fetched or stored as a whole.)
3. \$BYTES(n). If the value of n exceeds 2, this field-type is not fully transportable across all target systems, for the reasons given under item (1) above.

Observe that we have not included either \$SUB BLOCK or \$STRING in the above list, on the assumption that fields defined with these field-types are typically longer than a fullword and are 'pointed to' in some fashion rather than accessed directly. If, however, \$STRING were used (e.g.) to define a character-string field that was to be fetched directly, anything over two character positions would not be fully transportable.

Transportable Data Structures

TRANSPORTABILITY CONCERNS

In all cases, if a longer-than-fullword field is generated by a field-type other than \$SUB BLOCK or \$STRING, it is the user's responsibility to align it to an addressable boundary (with \$ALIGN(UNIT)) if its address is to be used.

2.4.2 Integer Value Range

Transportation of programs between the two larger BLISS target-system families (i.e., VAX-11 and DEC-10/20) does not usually present a problem with regard to range of integer values, since each system accommodates a fullword value of the same order of magnitude. However, unless considerable care is taken at the design stage, the potential problems involved in moving a program from a system with 32 or 36 bits per value to a system with only 16 bits can be very troublesome indeed. The discrepancy between the maximum values that can be handled is enormous. For example, \$INTEGER defines a field on the VAX-11 or the DEC-10/20 that can accommodate a value in excess of two billion or 34 billion respectively. On the PDP-11, however, the maximum (positive) value that a \$INTEGER field can hold is 32,767.

Obviously, this problem can only be resolved by a design that ensures that no calculated value need exceed the value limit of the smallest system to which the program is to be transported.

2.4.3 Use of \$BYTES for Character Strings

The \$BYTES field-type is not intended for the transportable definition of character-string fields. Although it will serve the purpose on the PDP-11 and VAX-11, where each ASCII character position occupies an 8-bit byte, on the DEC-10/20 it will define a sequence of 9-bit bytes whereas a sequence of 7-bit character positions is required.

The BLISS transportable character-handling functions assume a 7-bit character size on the DEC-10/20. The \$STRING field-type is intended expressly for this purpose. It provides the correct space allocation in each case, and also provides automatic alignment of the field to an addressable-unit boundary, which is required in order that the field's address may be passed (or otherwise used as an address).

Transportable Data Structures EFFICIENCY CONCERNS

2.5 EFFICIENCY CONCERNS

On some systems, a certain amount of speed advantage can be gained from having fields begin on a major addressable boundary. The \$ALIGN feature allows you to request this alignment. By default XPORT produces maximally packed fields, on the assumption that space efficiency will more often be desired.

As a case in point, the VAX-11 system allow fetchable/storable fields to cross fullword boundaries, and therefore XPORT will begin most fields (addresses, integers, byte sequences) at any point in a VAX longword. As an extreme example, an address field may begin at bit 31 of a given word and extend through bit 30 of the following word. A considerable amount of efficiency would be gained (assuming, of course, that the address value is used with some frequency) if the field began on a fullword boundary. In general it can be said that, on the byte-oriented architectures, a field that corresponds to a standard allocation unit (BYTE, WORD, LONG) should, for maximum efficiency, start on an address boundary that is 'natural' for its size. On a PDP-11 system this alignment of fields tends to occur "automatically" in XPORT data structures as a result of the PDP-11 field-reference restrictions (discussed above). On the VAX-11 however this alignment is the user's responsibility.

Another potential candidate for alignment consideration is a two-byte (18-bit) field on a DEC-10/20, which can be very efficiently accessed from either a word or half-word boundary. By default XPORT will align such a field to a fullword boundary only if it would otherwise span a fullword boundary. Judicious use of the BYTE option of \$ALIGN can achieve the desired positioning in this case.

On the whole, however, the real probability of significant speed gains can only be determined by a study of the target-system architectures in relation to the program's use of the fields in question.

CHAPTER 3 INPUT/OUTPUT FACILITIES

3.1	INTRODUCTION	3-1
3.1.1	General Characteristics	3-2
3.1.2	Specific Functions	3-3
3.2	CAPABILITIES	3-3
3.2.1	File Level Capabilities	3-3
3.2.1.1	OPEN	3-4
3.2.1.2	CLOSE	3-4
3.2.1.3	BACKUP	3-4
3.2.2	Input/Output Capabilities	3-5
3.2.2.1	Opening Concatenated Input Files	3-6
3.2.2.2	Output File Opening Options	3-7
3.2.2.3	Devices Openable for Both Input and Output	3-8
3.2.3	File Specification Resolution	3-8
3.3	I/O RELATED MACROS	3-9
3.3.1	General Format and Common Parameters	3-10
3.3.2	File-Level Macros	3-11
3.3.3	Input/Output Macros	3-15
3.3.3.1	Use of Pointers Vs. Addresses in Macro Calls	3-17
3.4	INPUT/OUTPUT CONTROL BLOCKS	3-18
3.4.1	Creating and Initializing IOBs	3-18
3.4.2	Using IOB Fields and Values	3-19
3.5	STANDARD I/O DEVICES	3-22
3.6	FILE SPECIFICATION PROCESSING	3-23
3.6.1	File Specification Resolution	3-23
3.6.1.1	Rules for File Specification Resolution	3-24
3.6.2	File Specification Parsing	3-26
3.7	I/O COMPLETION CODES	3-27
3.8	I/O ACTION ROUTINES	3-28

CHAPTER 3

INPUT/OUTPUT FACILITIES

This chapter describes the input/output-related portions of the XPORT Programming Tools Facility. The description given here is intended to present concepts rather than give complete details in all cases. Appendix A contains complete, detailed descriptions of all XPORT macro calls in a form designed for concise reference.

3.1 INTRODUCTION

The BLISS language does not provide built-in I/O capabilities. Among other services, the XPORT facility provides easy-to-use, transportable input/output support, via macro calls, for application programs that do not have particularly stringent or sophisticated I/O requirements.

XPORT I/O is a system-independent service package that supports sequential I/O operations in record, character-stream, and binary mode, and provides basic file-level functions. The file-level functions include, for example, file deletion and renaming as well as opening and closing.

The XPORT I/O facility actually consists of a separate source-and-object package for each target operating system and file system. However, the source-program interface to each of these packages is identical, and the results are equivalent. Thus the I/O support provided is fully transportable except where clearly noted.

Input/Output Facilities

INTRODUCTION

3.1.1 General Characteristics

The program interface is implemented as a set of BLISS keyword-macros, of the form `$XPO_xxxx(...)`. For example, `$XPO_PUT` is the name of the write operation; a typical call for writing out the content of a character-string buffer might look as follows:

```
$XPO_PUT( IOB = output_file ,
          STRING = ( .char_count, .line_pointer ) ) ;
```

where the lowercase names are user-defined.

Central to the operation of the I/O macros is the I/O control block, or IOB. The IOB is a standard BLISS block structure, for which an extensive set of field names is defined. As will be shown, the IOB fields can be accessed either by standard structure-references or, in many cases, by means of macro keywords. The IOB field names are of the form `IOB$x xxxx`; for example, `IOB$T_STRING` names a string descriptor sub-block of the IOB. (See Section 2.2.3 concerning sub-blocks.) The corresponding macro keywords are similar to the `xxxx` portion of the field names, e.g., `STRING`.

Each system-dependent implementation of XPORT consists essentially of two parts: a set of BLISS declarations (macro, literal, and field), and a set of object-time service routines. The source declarations, many of which define routine calls, are obtained from the XPORT library file for the target system, which must be named in a `LIBRARY` declaration in your source program. An example of such a declaration is

```
LIBRARY 'BLI:XPORT'
```

The relevant service routines are linked with your object program. An example of an appropriate command to perform such linking in the VAX/VMS environment is

```
$ LINK user-module-name, SYS$LIBRARY:XPORT/LIBRARY
```

(Appendix F gives additional examples for other target systems.)

The names of the required source-and object-time files for each target system are given in Appendix F, along with suggested methods of referring to those files in a transportable manner.

Input/Output Facilities

INTRODUCTION

3.1.2 Specific Functions

XPORT I/O comprises the following I/O and file-manipulation functions:

- OPEN prepares an existing file for reading (input), or prepares either a new or existing file for writing (output).
- CLOSE terminates the processing of an input or output file, flushing any I/O buffers as necessary.
- GET returns the address and length of character or binary data read from an opened input file. Logical concatenation of several input files can be automatically performed when an intermediate end-of-file is reached.
- PUT writes character or binary data to an opened output file.
- DELETE deletes an existing file.
- RENAME changes the name of an existing file.
- BACKUP provides a mechanism for preserving a copy of an input file when a program creates a new version of that file (typically used by editor-type applications).
- PARSE parses a file specification into its component parts.

Each of these functions is represented by a macro name; for example \$XPO_RENAME. There are also several other related macros, for creating and initializing various XPORT control blocks.

3.2 CAPABILITIES

XPORT I/O provides sequential input and output for standard I/O devices on the user's target system. In addition, a number of file-level operations are available for named files. The specific capabilities are discussed below.

3.2.1 File Level Capabilities

XPORT allows the user to delete, rename, and 'backup' named files, as well as to open and close files (and devices) for input/output. While the general result of a file delete or rename operation should be reasonably familiar and not in need of further elaboration, the XPORT open, close, and backup operations require some further discussion.

Input/Output Facilities CAPABILITIES

3.2.1.1 OPEN

The open operation incorporates a feature called file-specification resolution which is somewhat unusual in a high-level language facility. This feature is essentially a file-specification defaulting mechanism. It allows the end user, for example, to give an incomplete file specification which, at open time, is automatically combined with program- and system-supplied default components to make up a full file specification. File-specification resolution is discussed in detail in Section 3.6.1. (DELETE and RENAME also perform file-specification resolution when necessary.)

3.2.1.2 CLOSE

The close operation provides a REMEMBER option that preserves the file-attribute information in the corresponding IOB for subsequent operations: reopening, deletion, renaming, or backup. When the REMEMBER option is used, any subsequent operations which would ordinarily perform file-specification resolution do not do so, but take the already resolved (and 'remembered') file specification from the IOB. When a file is closed without the REMEMBER option, all fields of the corresponding IOB are reinitialized.

3.2.1.3 BACKUP

The backup operation is intended for applications that produce an output file that is, in some sense, an "upgraded" version of the input file; typically an editor-type application. The presumption upon which the operation is based is that the output, or new, file is to take the name of the input, or old, file, but also that the new file is to be "backed up" by the old file for safety's sake (e.g., for recovery purposes). For example, the BLISS Language Formatter, PRETTY, produces a reformatted version of the input source file as its output file, and the output file name defaults to the input file name.

On each host system, the new file is renamed by the backup operation with the old file name. The old file is "backed up," or preserved, in either of two ways depending upon the file-system capabilities of the host system. On host systems such as VAX/VMS which support multiple file-version numbers, the new file is simply given the old-file name with the next higher version number and the old file is not renamed. On systems that do not support multiple version numbers, however, such as TOPS-10, the new file takes the old-file name and the old file is given a different file type (or extension). The default file type for backup operations is ".BAK".

Input/Output Facilities CAPABILITIES

Note that the new output file involved in a backup operation is typically an XPORT temporary file prior to its renaming; see \$XPO_TEMPORARY in Section 3.5.

The backup operation requires that, prior to its invocation, both of the files involved have been opened, and then closed with the REMEMBER option (see above). (Presumably the input and output files are processed to completion in the interval.)

3.2.2 Input/Output Capabilities

XPORT provides a get and a put operation, which perform sequential input and output respectively. The get operation works in "locate" -- as opposed to "move" -- mode; that is, it reports the location (and length) of the data read in, rather than reading the data into a user-specified location. (The put operation takes its output data from a user-specified location, per usual practice.)

The data read or written by these operations can be in one of three basic forms:

1. A logical record, consisting of a variable (or sometimes fixed) amount of character-string data (RECORD mode).
2. A stream of characters of user-specified length (STREAM mode). The character stream is not differentiated by XPORT as to control characters vs. non-control characters.
3. A user-specified number of of binary values (BINARY mode). The number of binary values to be read or written can be specified in terms of either BLISS values (FULLWORDS) or addressable units (UNITS), although specification of the latter normally limits program transportability.

The mode keywords RECORD, STREAM, and BINARY are file-level attributes that must be specified when opening a given file. Another such keyword, SEQUENCED, implies RECORD and further specifies that an output file is to be sequence numbered; i.e., that a sequence number is to be associated with each logical record written. This type of file is produced by the SOS text editors, for example. (For a sequenced input file, the SEQUENCED indicator in the IOB is set by the open operation if the file is opened in RECORD mode.)

Input/Output Facilities CAPABILITIES

For character-encoded (e.g., ASCII) files, RECORD mode is normally used. In this mode, terminal-control-character (TCC) sequences are not passed to the program on input, and are automatically supplied by XPORT on output as appropriate for the host system involved. Note, also, that the program does not specify the number of characters to be read in this mode; the size of an input record is determined from information contained in the file.

Character-string files can also be processed in STREAM mode, in which all of the data read (plus, in some cases, the TCC sequences implied by the file format) is passed to the program as an undifferentiated stream of characters, and on output, no TCC sequences are supplied by XPORT. In other words, the user is "in full control" in this mode, and must have a detailed knowledge of the file formats and conventions of each host system for which the program is intended. The amount of data to be read, as well as the amount to be written, must be specified in each I/O call. STREAM mode is provided on a caveat programmer basis for the relatively few kinds of applications needing this capability.

BINARY mode is used to read and write files of binary data. The amount of data to be read, as well as the amount to be written, must be specified in each I/O call. The specification of an input or output datum size in terms of FULLWORDS allows source program transportability, assuming that CPU word size is not a limiting factor. The use of UNITS, on the other hand, allows transportability between the 16-bit and 32-bit environments with identical results, but drastically reduces the likelihood of transportability between the 16/32-bit environments and the 36-environment.

3.2.2.1 Opening Concatenated Input Files

A sequence of input files can be treated as a single file through use of the XPORT "concatenated input" feature. Concatenation is indicated to XPORT by giving a file specification of the form

file-spec-1 + file-spec-2 + ... + file-spec-n

in an OPEN call. This requests that when the end of each file is reached, the next file in the specified sequence of input files is to be opened and read (in a way that is completely transparent to the reading program).

Input/Output Facilities CAPABILITIES

The only file in the sequence that is explicitly opened by the program is the first one. All GET operations on the entire sequence are requested by reference to the same IOB. The file attributes and other parameters specified in the OPEN call are assumed for all of the files in the sequence. Actually, when a concatenated file needs to be opened, the following actions take place:

- o The currently opened file (e.g., the initial file) is closed,
- o The file-specification information for the next file is resolved, using default and related-file information, if any, from the IOB, and
- o The new resultant file specification is placed in the IOB, replacing the previous one. (File specification resolution is described in detail in Section 3.6.1.)
- o The concatenated file is then opened and read. Thus the associated GET call refers to the same IOB throughout the processing of an entire input-file sequence.

Note that the concatenated input-file capability can be completely automatic and transparent to the user program. That is to say, the logic and structure of a program need not be affected by the fact that multiple input files may be processed. Detailed \$XPO_GET completion codes do, however, allow the program to monitor concatenated-input processing (see Section A.8.3).

3.2.2.2 Output File Opening Options

In addition to the OUTPUT file-processing option itself, the options applicable to output files are APPEND and OVERWRITE.

APPEND means that, if the specified file already exists, the initial write operation is to begin at the current end-of-file, that is, the file is to be extended. OVERWRITE means that, if the specified file already exists, the initial write operation is to begin at the current beginning-of-file, overwriting the existing file content. (Neither of these options have significance if the named file does not already exist.) Both of these options imply the OUTPUT option.

If the OUTPUT option alone is specified, XPORT will attempt to create the file even if the indicated file already exists. If this cannot be done (i.e., the host system doesn't support file versions, the user has specified a version number matching that of the existing file, etc.), the open operation will fail. In all cases, if the indicated file does not already exist, it will be created if possible.

Input/Output Facilities CAPABILITIES

In RECORD mode, a "RECORD SIZE = n" parameter may be specified for a file that is to contain fixed-length records. Records 'put' to such a file that are shorter or longer than n are padded (with ASCII spaces) or truncated respectively. An error indication is returned in the truncation case. RECORD_SIZE = VARIABLE is the default.

A fixed physical-block size may also be specified for appropriate devices, such as magnetic tape, or for file formats to which it is applicable. (For RMS ISAM files, for example, this parameter is converted into an RMS "bucket size".)

If no file-processing option is specified for a file by file-opening time, INPUT is assumed. The INPUT and OUTPUT (as well as APPEND and OVERWRITE) options are mutually exclusive except for non-file-structured devices; see the following subsection.

3.2.2.3 Devices Openable for Both Input and Output

A communications (as opposed to file-storage) device that can be both read and written is opened for input and output by default. (The user can specify that the device be opened either for input or output only.) Such devices include all standard interactive-user terminals and DECnet communication links. Note that all devices, including interactive terminals, must be explicitly opened in order to be used.

3.2.3 File Specification Resolution

XPORT file-specification resolution, which is performed by the open, delete, and rename operations, provides a general-purpose mechanism for defaulting various components of a file specification. Specifically, it is the process of augmenting a partial file specification (possibly only a filename) with default information provided by both the program and the host file system, in order to arrive at a complete file specification for the system in question.

For an input file, the program typically provides a default file type, and the host file system provides a default file-version number, if any. For an output file, the program may provide both a default file name and file type, with the system providing the version number if any. For both input and output files, the host file system (through XPORT) provides a default for the network-node, logical- or physical-device, and directory (or PPN) components, as applicable for the system in question.

Input/Output Facilities CAPABILITIES

The "initial" file specification with which the process begins is typically one given by the end user (e.g., through direct interaction with the program), and is called the primary file specification. The program can also specify a

- o Default file specification, usually only a file type, used for both input and output files, and a
- o Related file specification (a 'related' input-file spec, for example), normally used only for output files.

There is a descriptor sub-block in the IOB for each of these specifications, and each has a corresponding keyword parameter (e.g., FILE_SPEC, DEFAULT) in the appropriate macro calls. The result of the resolution process is called the resultant file specification, for which there is also a descriptor in the IOB (IOB\$T_RESULTANT).

The order of application, or precedence, of file-specification components is as follows:

1. The primary file specification
2. The default file specification
3. The related file specification
4. System-provided defaults.

In addition, for output files there is a special convention concerning the related file specification (involving the use of an asterisk in either the primary or default specification) which "forces" file-specification components to be copied from the related file specification. The actual coding details and precise rules for using file-specification resolution are given in Section 3.6. That section also contains a discussion of the 'parse' operation and its macro call, \$XPO_PARSE_SPEC.

3.3 I/O RELATED MACROS

This section discusses the file-level and I/O macro calls, mostly by way of typical usage examples. Other macros are discussed in later sections and chapters: IOB creation and initialization macros are covered in Section 3.4.1; the macros concerned with file-specification parsing in Section 3.6.2. String- and data-descriptor creation macros are described in Chapter 6.

Input/Output Facilities

I/O RELATED MACROS

Four parameters are common to most of the macros calls; we will describe these first, along with the general macro format, to avoid subsequent repetition.

3.3.1 General Format and Common Parameters

The general format of an XPORT I/O macro call is given in the following syntax diagram. (The notational conventions of the BLISS Language Guide apply here as well: lowercase indicates syntactic variables, uppercase indicates syntactic literals, and so on.)

i/o-macro-call	\$XPO_function (parameter ,...)
function	{ BACKUP CLOSE DELETE } { GET OPEN PUT RENAME }
parameter	{ keyword = user-defined-string-or-value } { keyword = (user-defined-string-or-value ,...) } { keyword = secondary-keyword } { keyword = (secondary-keyword ,...) }

Examples of parameter keywords are FILE_SPEC, STRING, PROMPT, and OPTIONS. Examples of secondary keywords are INPUT, APPEND, RECORD, and SEQUENCED.

The four commonly occurring parameters are the following:

- o IOB = address of iob
This required parameter specifies the address of the IOB upon which or through which the operation is to be performed. (Note: BACKUP requires two IOB addresses.)
- o FAILURE = address of a failure-action routine
This optional parameter specifies the address of a routine to which XPORT is to pass control in the event of a failure condition (i.e., an exception or error). If this parameter is not specified, a default failure-action routine is supplied by XPORT, as discussed in Section 3.8.

Input/Output Facilities

I/O RELATED MACROS

- o **SUCCESS** = address of a success-action routine
This optional parameter specifies the address of a routine to which XPORT is to pass control following successful completion of the requested operation. If this parameter is not specified, no success-action routine is called, and control returns directly to the call site. (This parameter is rarely used.)
- o **USER** = user-defined fullword value
This optional parameter specifies a value to be placed in the IOB\$Z_USER field of the IOB involved. This value is not interpreted or used by XPORT in any way; the parameter is provided simply for the user's convenience. It might be used, for example, to maintain a count of a given operation, or to hold an address or value to be passed to the user's failure-action routine.

The next two subsections discuss specific examples of the file-level and I/O macro calls.

3.3.2 File-Level Macros

Typical examples of the \$XPO_OPEN macro call follow. In these examples, user-defined names are shown in lower case for the sake of clarity. (Case is irrelevant, of course, outside of quoted strings.)

```
$XPO_OPEN ( IOB = user_terminal ,  
            FILE_SPEC = $XPO_INPUT ) ;
```

This call opens an interactive end-user's terminal for both input and output (by default), through an IOB named "user_terminal" with the symbolic file specification "\$XPO_INPUT". This symbolic specification (actually a macro call) indicates the standard -- or system default -- input device; see Section 3.5. (For a batch job, of course, this is simply the job's input stream, which would be opened for input only.)

A second example of the OPEN call,

```
$XPO_OPEN ( IOB = change_file ,  
            FILE_SPEC = user_terminal[IOB$T_STRING] ,  
            DEFAULT = '.UPD' ) ;
```

opens an input file through the IOB named "change_file". (Recall that OPTIONS=INPUT is the default for files.) The file specification is given indirectly, via the IOB\$T_STRING descriptor in the user-terminal IOB. This string descriptor describes the most recent record read from the user's terminal, presumably a file specification. A default file specification consisting of the file type ".UPD" is specified.

Input/Output Facilities

I/O RELATED MACROS

This will be combined with the primary file specification if it lacks a file type.

A further OPEN example:

```
$XPO_OPEN ( IOB = output_file ,  
            FILE_SPEC = user_terminal[IOB$T_STRING] ,  
            DEFAULT = '*.MAS' ,  
            RELATED = change_file[IOB$T_RESULTANT] ,  
            OPTION = APPEND ,  
            ATTRIBUTE = SEQUENCED ) ;
```

In this case, the file is opened for output and is to be extended if it already exists (APPEND). Like the input file, the primary file specification is to be read from the user-terminal IOB, but if the terminal response is null (or contains one or more asterisks), the missing components are obtained from the related file specification, which is the resultant file specification for the input (change) file. Prior to application of components from the related file specification, however, the default file type ".MAS" is applied if needed, since a default file specification has precedence over a related file specification. Finally, each logical record written to this file is to have a record sequence number. (This number must be provided by the user as we will see in a subsequent PUT example). Note that, because the file is being opened for APPENDING, if the file already exists and is not sequence numbered, the SEQUENCED option is ignored.

Several examples of the \$XPO_CLOSE macro call follow.

```
$XPO_CLOSE ( IOB = user_terminal ) ;
```

This example is self-explanatory; the user's terminal is closed and all fields in the associated IOB are reinitialized.

A further example:

```
$XPO_CLOSE ( IOB = change_file ,  
            OPTIONS = REMEMBER ) ;
```

In this case, the input file controlled by the change_file IOB is closed with the REMEMBER option, possibly because it is going to be renamed or deleted, or is to be reopened for reprocessing. The REMEMBER option causes all file-attribute information in the IOB to be retained. All file-processing options, on the other hand, are 'forgotten' in any case, including the REMEMBER option.

Input/Output Facilities
I/O RELATED MACROS

An example of the DELETE call:

```
$XPO_DELETE ( IOB = arbitrary_file ,  
              FILE_SPEC = terminal_input[IOB$T_STRING] ) ;
```

Here we assume that the file has never been opened (or has been closed without the REMEMBER option); i.e., that the resultant file-specification field of the IOB is empty. The call relies on end-user input to supply a complete file specification (except for system supplied defaults), since no default or related-file parameters are specified. In contrast, the call

```
$XPO_DELETE ( IOB = change_file ) ;
```

relies on the fact that a resultant file specification has been retained in the IOB for a file previously closed with the REMEMBER option. Thus no file-specification information is required.

An example of the RENAME call (which assumes that the immediately preceding DELETE example has not been executed):

```
$XPO_RENAME ( IOB = change_file ,  
              NEW_SPEC = ( .spec_length , .spec_ptr ) ) ;
```

Here the call relies, as before, on the fact that a resultant file specification is still in the IOB (identifying the "current" name of the closed file); thus no FILE_SPEC parameter is given. For the new name, the call supplies a length value and pointer to a character-position sequence, which presumably contains a file specification determined by the program.

As the basis for another example of RENAME, assume that the program elicits from the end-user's terminal (1) a 'current' file specification and (2) a 'new' file specification. When response (1) is read, the program moves the string to an area pointed to by, say, .old_name_ptr, then reads response (2) and executes the following call:

```
$XPO_RENAME ( IOB = arbitrary_file ,  
              FILE_SPEC = ( .oldname_length, .oldname_ptr ) ,  
              DEFAULT = '.DAT' ,  
              NEW_SPEC = terminal_input[IOB$T_STRING] ,  
              NEW_DEFAULT = '*.NEW' ) ;
```

Here the default file type is ".DAT" for the current file specification and "*.NEW" for the new file specification. Also, because no NEW_RELATED parameter is given, XPORT assumes the current file specification (described in IOB\$T_RESULTANT) as the value of the new-related file specification.

Input/Output Facilities

I/O RELATED MACROS

Thus if the new-file-spec response from the user's terminal is null for any reason, the rename operation will use the current file specification but with the default type ".NEW".

Note that the program cannot specify a NEW_RELATED parameter that has the same effect as the default assumption made by XPORT. That is, a RENAME parameter of the form

```
NEW_RELATED = the-same-iob[IOB$T_RESULTANT]
```

will cause execution errors, since the description of the new-related file specification is determined before the 'old' resultant file specification is actually created.

The following example of a BACKUP call is shown in-context, in order to illustrate the proper usage of this operation. For this example, assume that the content of the input file is modified in some way (processing not shown) to produce the output file.

```
$XPO_OPEN ( IOB = input_file ,  
            FILE_SPEC = input-spec-descriptor ,... ) ;  
$XPO_OPEN ( IOB = output_file ,  
            FILE_SPEC = $XPO_TEMPORARY ,... ) ;
```

```
!           The input and output files are  
!           now processed to completion.
```

```
$XPO_CLOSE ( IOB = input_file , OPTIONS = REMEMBER ) ;  
$XPO_CLOSE ( IOB = output_file , OPTIONS = REMEMBER ) ;
```

```
$XPO_BACKUP ( OLD_IOB = input_file , NEW_IOB = output_file ) ;
```

Note that both files are closed with the REMEMBER option. The purpose of this is to preserve the resultant file specifications in the respective IOBs. (Recall that BACKUP does not perform file-specification resolution.)

Following the backup operation, if the host file system supports file version numbers, the new file has the same file specification as the old file but with a higher version number.

If the host file system does not support multiple file versions, the new file will have the old file specification and the old file will be renamed with the file type ".BAK" (by default). Some other file type may be specified; the macro keyword is, naturally enough, FILE_TYPE. In either case, all fields of the two IOBs involved in the operation are reinitialized -- set to zeros or other initial value -- following the backup procedure.

Input/Output Facilities

I/O RELATED MACROS

In order to do any further processing through these IOBs, they must be reopened as if they had just been initialized (see \$XPO_IOB_INIT, Section 3.4.1).

3.3.3 Input/Output Macros

Examples of the \$XPO_GET and \$XPO_PUT macro calls follow. Note that the particular IOB names used have no special significance other than to suggest or reflect some contextual aspect of the example. A typical call for input from an interactive terminal, including a prompt message, would look like:

```
$XPO_GET ( IOB = user_terminal ,  
          PROMPT = 'INPUT FILE NAME: ' ) ;
```

This call results in a 'prompted read' operation; that is, the specified prompt string is written to the terminal opened through the IOB named user_terminal, and then the terminal is placed in input status (without intervening carriage/cursor movement). The same call without the prompt parameter results in a simple read operation. The length and location of the input string read from the terminal are available in the IOB\$T_STRING sub-block of the IOB, the input character-string descriptor, described further in Section 3.4.2.

A call to read a logical record from a file opened with the RECORD file-processing attribute is simply the following:

```
$XPO_GET ( IOB = input_file ) ;
```

Again, the length and location of the record read from the file are available in the descriptor IOB\$T_STRING. The pertinent fields of this sub-block are IOB\$H_STRING, which contains the number of characters in the record (in binary), and IOB\$A_STRING, which contains a pointer to the record.

A call to read a stream of characters from a file opened with the STREAM attribute is not much more complicated:

```
$XPO_GET ( IOB = stream_file ,  
          CHARACTERS = 80 ) ;
```

The result of this call is to read the next 80 characters of data, including any terminal-control-character (TCC) sequences, from the indicated file (assuming that many characters are left in the file). The number-of-characters value can, of course, be any BLISS primary expression, as well as a numeric literal as shown. As before, the string that is read in is described by the IOB\$T_STRING descriptor.

Input/Output Facilities

I/O RELATED MACROS

A call to read a given number of fullwords of data from a file opened with the BINARY attribute is as follows:

```
$XPO_GET ( IOB = binary_file ,  
           FULLWORDS = .how_much ) ;
```

This call results in reading the requested number of fullwords, i.e., BLISS values, from the indicated file. The data is read into an XPORT internal buffer; the size and location of the data read are available in the IOB descriptor IOB\$T_DATA, the pertinent fields of which are IOB\$H_UNITS (size in addressable-units) and IOB\$A_DATA (address of first data element). The field immediately following the IOB\$T_DATA descriptor, and closely associated with it, is IOB\$H_FULLWORDS, which contains the datum size in fullwords.

A PUT call to write a character string to a terminal is as follows:

```
$XPO_PUT ( IOB = user_terminal ,  
           STRING = 'Beginning 3rd-phase processing.' ) ;
```

Note that, if the user_terminal IOB were opened with no file-processing attribute (presumed to be the case here), RECORD mode is assumed. XPORT automatically provides carriage/cursor control, i.e., appropriate TCC sequences, for RECORD-mode operations. Effectively, XPORT appends a carriage-return/line-feed sequence to each string (logical record) written. Thus the user need not supply control characters for anything other than special cases (e.g., multiple spacing and formfeeds).

A call to write a logical record or stream of characters to a file (in RECORD or STREAM mode respectively) might look like this:

```
$XPO_PUT ( IOB = output_file ,  
           STRING = ( .out_rec_size, .out_rec_ptr ) ;  
  
or  
  
$XPO_PUT ( IOB = output_file ,  
           STRING = out_rec_descr ) ;
```

The first version directly specifies a string-length and string-pointer value, which together identify the data to be written. The second version specifies the address of an XPORT string descriptor (discussed in Chapter 6), which in turn contains the needed length and pointer values, as well as other information.

Input/Output Facilities

I/O RELATED MACROS

A call to write a logical record to a sequenced (SOS format) file is as follows:

```
$XPO_PUT ( IOB = sequenced_file ,  
           STRING = out_rec_descr ,  
           SEQUENCE_NUMBER = .current_line) ;
```

An alternative version of the sequence-number parameter is

```
SEQUENCE_NUMBER = .sequenced_file[IOB$G_SEQ_NUMB] + 1
```

A simple method of making output sequence numbering conditional upon whether the related input file is sequenced numbered, e.g., for file copying purposes, is shown in a later section.

A call to write, say, twenty fullwords of data to a file opened with the BINARY attribute is as follows:

```
$XPO_PUT ( IOB = binary_file ,  
           BINARY_DATA = ( 20, outbuffer, FULLWORDS ) ) ;
```

Observe here that an address value, rather than a pointer, is used to give the location of binary data, as suggested by the name "outbuffer" used without a fetch operator. Also, the keyword FULLWORDS (alternative to UNITS) is shown for the sake of completeness, although it is the default. An alternative form of the binary-data parameter is

```
BINARY_DATA = out_buf_descr
```

which specifies the address of an XPORT data descriptor containing the data size and address information.

3.3.3.1 Use of Pointers Vs. Addresses in Macro Calls

In all of the foregoing examples of character-oriented operations (those implying the RECORD or STREAM attribute), a data-location subparameter is shown as a fetched value, e.g., ".out_rec_ptr". This is meant to indicate that a character-position pointer, rather than an address, is required to identify the initial position of a character string. Such a pointer is created by the CH\$PTR function, and is manipulated by other function of the BLISS character-handling package.

String-location information returned by XPORT, such as in the IOB field IOB\$A_STRING for a GET operation, is also a pointer value. See the discussion of pointers and addresses in Section 2.2.3.1 for transportability considerations.

Input/Output Facilities

I/O RELATED MACROS

As indicated in the final PUT example, the location of binary data is given (and returned) simply as an address value.

3.4 INPUT/OUTPUT CONTROL BLOCKS

Each file or concatenated file sequence processed by XPORT must be described by an XPORT input/output block (IOB). This control block, created by means of the \$XPO_IOB macro, contains the following information by the time the file is opened:

- o The file specifications associated with the file -- the primary file specification, a default file specification, if any, possibly a related file specification, and the resolved (i.e., resultant) file specification.
- o File attributes, such as the physical block size.
- o Record attributes, such as the record format (e.g., sequence numbered), the maximum record length, and -- after an I/O operation -- the current record length and address and the sequence number, if any, of the current record.
- o File status information that indicates whether the file is open or closed, whether it has ever been opened, etc.
- o Internal information such as the current function code, the completion code of the last operation, and length of the IOB.

It is important to understand that the information in an IOB reflects the current state of processing of a file rather than the state of the file itself. This distinction is significant when two open IOBs refer to the same file. For example, if the same file is opened for input using two IOBs, input operations on the two IOBs proceed independently, just as if two different files were being read.

3.4.1 Creating and Initializing IOBs

An IOB is created by use of the \$XPO_IOB macro as an attribute in a data declaration, e.g., in an OWN or LOCAL declaration. If created in permanent storage, the IOB is automatically initialized. For example:

```
OWN
    grundoon : $XPO_IOB() ;
```

(Currently the \$XPO_IOB macro does not take any parameters.) The effect of this declaration is to declare the data segment "grundoon" as a block structure of appropriate size, and to associate a set of field names with that structure.

Input/Output Facilities

INPUT/OUTPUT CONTROL BLOCKS

The effect is just as if an appropriate structure- and field-attribute had been used in the data declaration. The field names associated with the IOB, of the form IOB\$x_abc, identify the various sub-blocks and individual fields of the structure.

In this example (permanent-storage allocation), the IOB is also initialized by the \$XPO_IOB macro to a "new IOB" state. That is, the many fields of the OWN data segment allocated for the IOB are preset, mostly to zero values. This automatic initialization is also performed for IOBs allocated in GLOBAL storage.

If an IOB is created in either temporary (e.g., LOCAL) storage or dynamically-acquired storage, however, it must be explicitly initialized by means of the \$XPO_IOB_INIT macro before it is used, that is, before any other reference to it. This macro results in executable code that dynamically initializes all fields of the IOB. For example:

```
LOCAL
    temp_iob : $XPO_IOB() ;
...

$XPO_IOB_INIT ( IOB = temp_iob ) ;
$XPO_OPEN ( IOB = temp_iob , FILE_SPEC = ... ,... ) ;
```

This macro also accepts most OPEN, GET, PUT, CLOSE, DELETE, and RENAME parameters, for optional 'presetting' of IOBs as described above for the \$XPO_IOB macro.

3.4.2 Using IOB Fields and Values

An IOB is used (1) to pass information from the user program to an XPORT I/O routine and (2) to return information to the program upon completion of the I/O operation.

As previously stated, when an IOB is allocated a set of predefined BLISS field-names is implicitly associated with it. Appendix B describes the format of the IOB and each of the IOB fields, plus some related literal values that may be of interest. You have already seen several IOB field names, however, in preceding macro examples. These field names can, of course, be used in ordinary BLISS structure references. (Similarly, the IOB-related literal names can be used in other types of BLISS expressions.)

Input/Output Facilities

INPUT/OUTPUT CONTROL BLOCKS

For example, consider the use of the field name IOB\$H_STRING in the following program fragment. The IOB\$H_STRING field contains the length of the character string last read by a GET operation:

```
OWN
    grundoon : $XPO_IOB(),
    pogo : $XPO_IOB();
    .
    .
    $XPO_GET( IOB = grundoon );

! Test for a non-null input string.

IF .grundoon[IOB$H_STRING] NEQ 0
THEN
    BEGIN
        .
        .      (filled in below)
        .
    END;
```

The I/O macros provide keyword parameters that set up appropriate IOB fields before calling the I/O function. One related group of such fields is the output-string descriptor IOB\$T_OUTPUT. This descriptor contains the fields IOB\$H_OUTPUT (length) and IOB\$A_OUTPUT (pointer), which are set up by the STRING parameter of \$XPO_PUT.

As shown in previous examples, the STRING parameter takes several forms of string description, one being simply the address of a descriptor. Thus one descriptor, i.e., set of fields, can be set up by means of the contents of another. IOB\$T_STRING is another such descriptor, comprising the subfields IOB\$H_STRING (length) and IOB\$A_STRING (pointer), among others. This is the input-string descriptor set by \$XPO_GET. Different forms of reference to these two sets of fields are shown and discussed below.

(Standard string and data descriptors can also be created by the user; see \$STR_DESCRIPTOR and \$STR_DESC_INIT in Chapter 6.)

In the following sample code, the data read from the "grundoon" input file (see above) is written twice to the "pogo" output file, first using explicit BLISS expressions to set up the pertinent IOB fields, then using keyword parameters in the \$XPO_PUT macro call to do the same thing. (The second setup of the IOB is actually unnecessary since the relevant IOB fields are not modified by the first write operation.)

Input/Output Facilities INPUT/OUTPUT CONTROL BLOCKS

```

BEGIN
pogo[IOB$A_OUTPUT] = .grundoon[IOB$A_STRING] ;
pogo[IOB$H_OUTPUT] = .grundoon[IOB$H_STRING] ;
pogo[IOB$H_PAGE_NUMB] = .grundoon[IOB$H_PAGE_NUMB] ;
pogo[IOB$G_SEQ_NUMB] = .grundoon[IOB$G_SEQ_NUMB] ;
$XPO_PUT( IOB = pogo ) ;

$XPO_PUT( IOB = pogo ,
          STRING = grundoon[IOB$T_STRING] ,
          PAGE_NUMBER = .grundoon[IOB$H_PAGE_NUMB] ,
          SEQUENCE_NUMBER = .grundoon[IOB$G_SEQ_NUMB] )
END ;

```

Note that when used as a STRING parameter value (second PUT call), just the address of the input-string descriptor is sufficient to set up the subfields of the output descriptor in the "pogo" IOB. When direct structure references are used, however, the individual subfield values themselves must be fetched and stored. (The expression .grundoon[IOB\$T_STRING] is not valid, since this field name represents only the beginning of a multifield sub-block of the IOB; see Section 2.2.3, under \$SUB_BLOCK.)

Clearly, implicit referencing of IOB fields via keyword parameters is the more economical method, when information is to be stored in the IOB. When information has to be retrieved, however (an input-string description, for instance), explicit structure references must be used.

As a final example of much that has been discussed up to this point, consider the following simple but realistic file-copy loop. This example includes a "preview" of the use of completion codes, discussed further in Section 3.7. The details of file-specification handling are omitted, since these are quite application dependent.

```

OWN
    input_file : $XPO_IOB() ,
    output_file : $XPO_IOB() ;

    $XPO_IOB_INIT( IOB = input_file ) ;
    $XPO_IOB_INIT( IOB = output_file ) ;
    .
    .
    $XPO_OPEN( IOB = input_file ,
              FILE_SPEC = "... ) ;

! Make output file SEQUENCED only if the input file is.

output_file[IOB$V_SEQUENCED] = .input_file[IOB$V_SEQUENCED] ;

```

Input/Output Facilities INPUT/OUTPUT CONTROL BLOCKS

```
$XPO_OPEN( IOB = output_file ,  
           FILE_SPEC = ... ,  
           OPTIONS = OUTPUT ) ;  
  
WHILE $XPO_GET( IOB = input_file ) DO  
    $XPO_PUT( IOB = output_file ,  
             STRING = input_file[IOB$T_STRING] ,  
             PAGE_NUMBER = input_file[IOB$H_PAGE_NUMB] ,  
             SEQUENCE_NUMBER = input_file[IOB$G_SEQ_NUMB] ) ;  
  
$XPO_CLOSE( IOB = input_file ) ;  
$XPO_CLOSE( IOB = output_file ) ;
```

Since one cannot conditionalize the SEQUENCED attribute keyword within the OPEN call, the relevant IOB field, IOB\$V_SEQUENCED, is set outside the call with the value of the same field in the input IOB, after the latter has been opened. This makes output-file record sequencing conditional upon the input file format, in a handy, 'automatic' fashion.

The DO loop controlling the GET and PUT operations depends upon the fact that all XPORT routines return a completion code that can be tested for 'true', successful completion, or for 'false', failure, with a standard low-bit test. Input end-of-file is considered a failure, but not an error, condition. (This distinction becomes important when default failure-action routines are described in Section 3.8.)

3.5 STANDARD I/O DEVICES

All operating systems support the concept of default, or standard, input and output devices (e.g., the user's terminal, the system-output line printer). However, the way in which these standard devices are named varies from system to system (e.g., TTY:, TI:, SYS\$INPUT). The XPORT I/O facility provides four macros which represent the names of these standard devices.

```
$XPO_INPUT - standard input device  
$XPO_OUTPUT - standard output device  
$XPO_ERROR - standard error message device  
$XPO_TEMPORARY - a unique temporary work file
```

Note that using a standard XPORT device does not eliminate the need to open the file; i.e., all files except concatenated input files must be explicitly opened.

Input/Output Facilities

STANDARD I/O DEVICES

The use of `$XPO_TEMPORARY` results in a uniquely named disk file having a file type (or extension) that indicates it is temporary, e.g., `".TMP"`. (Some target file systems delete these files at end of session.)

3.6 FILE SPECIFICATION PROCESSING

File specification processing commonly includes the use of two services provided by XPORT:

- o File-specification resolution, performed during open, delete, and rename operations, and
- o File-specification parsing, provided via the `$XPO_PARSE_SPEC` macro.

This section discusses the reasons for, and use of, these services.

3.6.1 File Specification Resolution

Applications that process files are usually written so that the end user can in some way specify the names of the files to be read, written, renamed, etc. In order to make such an application easier to use by allowing the user to give partial (or null) specifications, defaults for certain file-specification components are generally provided. XPORT supports a defaulting technique that provides the most frequently required capabilities.

A complete file specification consists of a network node name, a device name, a directory name or project/programmer number, a file name, a file type (or extension), and sometimes a file version. During file opening, renaming, or deletion, the XPORT facility, working in conjunction with the host file system, automatically constructs a complete file specification, using information contained in the IOB and information provided by the host system. It does this by selecting the necessary components from the following sources:

1. The primary file specification string, usually provided by the end user; described in the IOB. This string might consist only of a file name, or might even be null.
2. A default file specification (typically a file type), provided by the program; described in the IOB.

Input/Output Facilities
FILE SPECIFICATION PROCESSING

3. A related file specification (normally used only for output files), provided by the program; described in the IOB.
4. System and user defaults (node, device, directory, and file version) supplied by the host file system.

Generally speaking, default components are taken from these sources in the order listed, although the rules for selection of certain components from the related file specification are a bit more complicated (see below).

Input file specifications are typically constructed from a user-specified file name, a default file type, and system defaults. Likewise, output file specifications are frequently constructed from a related file name (taken from the resultant file specification of an associated input file), a default file type, and system defaults.

The program controls the resolution process essentially by using the primary, default, and related file-specification parameters of the file-level macros in a way that, according with the selection rules given below, achieves the desired effect.

3.6.1.1 Rules for File Specification Resolution

Although there are minor differences from system to system, all current target systems support a subset or variant of the following file-specification format:

node::device:<directory>filename.filetype.version

When a file is to be opened, renamed, or deleted, a resultant file specification is constructed by selecting tokens from the file specification(s) described in the IOB and from system defaults, according to the rules summarized in Table 3.1. If a token exists in more than one source for a given file-specification component, the first occurring token is used. "First occurrence" is determined by traversing the decision table from top to bottom in accordance with the selection rules.

Input/Output Facilities
FILE SPECIFICATION PROCESSING

Table 3.1
File-Specification Resolution Semantics

DECISION TABLE							
Source	node name	device name	directory spec	file name	file type	file version	
Primary file-spec	X	X	X	X	X	X	
Default file-spec	(X)	(X)	(X)	X	X	(X)	
Related file-spec							
Open for input	X	X	X	X	X	-	
Open for output	-	-	-	U	U	U	
Delete	X	X	X	X	X	-	
Rename - old name	X	X	X	X	X	-	
Rename - new name	-	-	-	U	U	U'	
System/user defaults							
Open for input	A	A	A	-	-	H	
Open for output	A	A	A	-	-	H+1	
Rename	A	A	A	-	-	H	
Delete	A	A	A	-	-	H	

TOKEN SELECTION RULES							
A token can be an actual file-specification component or an or an asterisk (*).							
X = This token is always used if it exists.							
(X) = This token is seldom specified but is used if it exists							
U = This token is used if the corresponding primary or default, token is an asterisk (*) or missing.							
U' = This token is used only if the corresponding primary or default token is an asterisk (*).							
A = This token is always defined and is used if necessary.							
H = Highest existing version is used.							
H+1 = Highest existing version plus 1 is used.							
- = Not used.							

Input/Output Facilities FILE SPECIFICATION PROCESSING

3.6.2 File Specification Parsing

XPORT provides file-specification parsing, i.e., the division of a file specification into its component parts together with a syntax check, via the `$XPO_PARSE_SPEC` macro. In order to use `$XPO_PARSE_SPEC`, you must also use the macro `$XPO_SPEC_BLOCK`. This supporting macro, used as an attribute of a data declaration, creates a file specification block whose address is a parameter of the `$XPO_PARSE_SPEC` macro along with a file-spec-string description. (The format of a file specification block is given in Appendix B.)

A `$XPO_PARSE_SPEC` call might look as follows:

```
$XPO_PARSE_SPEC( FILE_SPEC = input_file[IOB$T_RESULTANT] ,  
                SPEC_BLOCK = input_fs_block ) ;
```

This example assumes that a file specification was previously resolved in the input file IOB, and that the `$XPO_SPEC_BLOCK` macro was used to allocate the data segment `input_fs_block`.

The `$XPO_PARSE_SPEC` macro essentially does two things. Firstly, it checks all non-null components of the indicated file-specification string for valid syntax (for the host system) and, secondly, it places those components in the appropriate fields of the specified file-specification block, together with an indication of a 'wild card' character (*) as one of the components. (The asterisk is the only character recognized by XPORT as a 'wild card' character and must appear by itself, i.e., must represent an entire file-specification component.) If one or more of the components is syntactically invalid, the macro returns an error completion code that identifies the faulty component (the first one encountered if more than one).

The `$XPO_PARSE_SPEC` macro accepts resolved or unresolved file specifications, and recognizes null and 'wild card' components as valid along with actual components. The components described by the file specification block include the attendant punctuation characters, e.g., angle brackets or square brackets in the case of a directory, UIC, or PPN component.

TRANSPORTABILITY WARNING

It must be observed here that, as with a few other XPORT features, use of the file-specification parsing feature is not likely to result in a fully transportable program unless it is used with a considerable amount of care and forethought.

Input/Output Facilities

FILE SPECIFICATION PROCESSING

This is so not only because of the system-dependent aspects of the file-specification format, e.g., the "punctuation" characters, but also because of the substantive differences that exist from system to system, for example the conventional file type ".LST" employed on some systems versus ".LIS" on others, both used for the same purpose.

The file specification block contains, along with descriptors for each of the components, a set of bits which indicate, individually:

- o Whether a directory-name or a PPN/UIC component was specified,
- o Whether or not a 'wild card' occurred anywhere in the file specification, and
- o Which component(s) consist of a 'wild card' character, if any.

PPN/UIC component values are contained within the file specification block (as binary integers); all other components are described by standard XPORT string descriptor sub-blocks within the file specification block.

3.7 I/O COMPLETION CODES

The BLISS value of a macro that calls an XPORT I/O routine is the completion code returned by that routine. Note that in most cases there are several failure completion codes for a given routine and, in some cases, more than one success completion code.

The completion code of an XPORT routine can simply be tested for success/failure status (i.e., standard BLISS low-bit test), or can be compared with expected values for more detailed testing. XPORT provides a complete, transportable set of completion-code literals (described with each macro call in Appendix A) expressly for this purpose. Thus the programmer need not be aware of the actual (possibly system-dependent) numeric completion-code value in any case. The 'unqualified success' completion code literal, indicating successful completion with no exception condition, is XPOS_NORMAL.

Note that a warning code, as is returned for input end-of-file, is considered a failure completion code, and as such invokes a failure-action routine. The default failure-action routine, however, treats a warning code differently than failure codes having an error or fatal severity.

In addition to a primary completion code being returned as the value of the routine call, this completion code is also returned in the IOB field IOB\$G_COMP_CODE.

Input/Output Facilities

I/O COMPLETION CODES

Some completion codes (e.g., XPOS_BAD_IOB) have an associated secondary completion code which is returned in the IOB field IOBSG_2ND_CODE. This secondary completion code field is zeroed by XPORT upon return from a call if it is not used.

3.8 I/O ACTION ROUTINES

Each macro that results in a call to an XPORT I/O routine allows the programmer to specify the address of another routine to be called upon completion of the requested function. Separate routines can be specified for successful and abnormal completion. As seen previously, the macro parameter keywords are SUCCESS and FAILURE respectively. These optional user-provided routines, called action routines, are typically used to intercept and possibly correct error conditions.

An action routine must be declared in each XPORT routine call to which it applies; that is, action-routine addresses cannot be preset in the IOB as can most other I/O parameters. (No corresponding IOB fields exist, in fact.) Such presetting would in any case tend to obscure control flow within the program.

A success or failure action routine, if any, is called by XPORT just before returning to the caller of the I/O operation. The action routine is passed, as one of its calling parameters, the address of the IOB specified in the original call. The action routine may examine and/or change IOB fields and may perform any appropriate I/O operation using the passed IOB or another IOB. For example, a failed operation could be retried after possible IOB modification. The action routine may modify the completion code returned to the caller.

A listing of the default I/O failure-action routine provided by XPORT, XPOS_FAILURE, appears in Appendix E. This routine is the default for the FAILURE parameter in I/O macro calls. It issues a multiple-line error message and then terminates program execution for all I/O failures except input end-of-file. (The input-EOF condition has only a 'warning' severity.)

An optional I/O failure-action routine, XPOSIO_FAILURE, is provided to allow standard XPORT error message processing without terminating program execution. Instead, control -- and the initial completion code -- is returned to the caller. (This routine is itself called by XPOS_FAILURE and is included in the same module; see Appendix E.) There is no default success-action routine; such routines are rarely used. See Appendix E for information about writing action routines.

CHAPTER 4	MEMORY MANAGEMENT FACILITIES	
4.1	INTRODUCTION	4-1
4.2	CAPABILITIES	4-1
4.3	MEMORY MANAGEMENT MACROS	4-2
4.3.1	\$XPO_GET_MEM - Allocating Dynamic Memory	4-2
4.3.2	\$XPO_FREE_MEM - Releasing Dynamic Memory	4-3
4.3.3	Dynamic Memory Elements	4-4
4.4	COMPLETION CODES	4-4
4.5	ACTION ROUTINES	4-4

CHAPTER 4

MEMORY MANAGEMENT FACILITIES

This chapter describes the memory-management portion of the XPORT Programming Tools Facility. The description given here is intended to present concepts rather than give complete details in all cases. Appendix A contains complete, detailed descriptions of all XPORT macro calls in a form designed for concise reference.

4.1 INTRODUCTION

The XPORT memory management facilities provide the ability to acquire and release memory space during program execution, as the program's needs expand and contract, in a simple and fully transportable manner. Dynamically acquired main-storage space, often referred to simply as "dynamic memory", is typically used for data buffers and control blocks.

Dynamic and dynamic-bounded string/data descriptors may be used in conjunction with the memory management facilities; they are described in Chapters 6 and 7.

4.2 CAPABILITIES

XPORT currently provides the following system-independent memory management functions:

- o Allocation of a specified amount of dynamic memory
- o Release of a dynamically-allocated memory element.

The size of a dynamically allocated element of memory can be specified in terms of characters, fullwords, or addressable units, though use of the latter will probably limit program transportability.

Memory Management Facilities
MEMORY MANAGEMENT MACROS

4.3 MEMORY MANAGEMENT MACROS

Dynamic memory allocation and release is requested by means of the `$XPO_GET_MEM` and `$XPO_FREE_MEM` macros respectively. Both result in a call to an XPORT MEMORY function.

4.3.1 `$XPO_GET_MEM` - Allocating Dynamic Memory

The get-memory function stores a character-position pointer or address value into a user-specified location (or descriptor field) if the requested allocation is successful. Typical usage examples are given below.

The following code fragment illustrates the acquisition of a large character-string buffer:

```
OWN
    bigbuff_ptr ;
    .
    .
    .

    $XPO_GET_MEM( CHARACTERS = 1028,
                  RESULT = bigbuff_ptr,
                  FILL = %C' ' ) ;
```

Upon successful execution of the get-memory call, a memory element large enough to contain 1028 character positions is allocated, and a pointer to the first character position is stored in location `bigbuff_ptr`. The optional `FILL` parameter causes the element to be initially blank-filled (any ASCII character code may be specified). Note that the standard BLISS character size is assumed for all target systems, i.e., 8-bit characters in the 16/32-bit environments and 7-bit characters in the 36-bit environments.

Alternatively, a dynamic or dynamic-bounded descriptor can be used in conjunction with the `$XPO_GET_MEM` macro to achieve the same result, as shown in the following code fragment:

```
LOCAL
    bigbuff_desc : $STR_DESCRIPTOR( CLASS = DYNAMIC );
    ...

    $STR_DESC_INIT( DESCRIPTOR = bigbuff_desc,
                   CLASS = DYNAMIC );
    ...

    $XPO_GET_MEM( CHARACTERS = 1028,
                  DESCRIPTOR = bigbuff_desc,
                  FILL = %C' ' ) ;
```

Memory Management Facilities

MEMORY MANAGEMENT MACROS

In this case, on the successful completion of the get-memory function, the string descriptor is updated to describe the allocated memory element (See Section 6.1 for a discussion of descriptor usage.)

To illustrate acquisition of dynamic memory in terms of fullwords, assume that you want to get space in which to construct a 'dynamic' IOB -- for reading a REQUIRE-type file, for example. The XPORT literal IOB\$K_LENGTH defines the length (in BLISS values) of a standard IOB structure. The following declaration and macro call would be appropriate:

```
LOCAL
    dynamic_iob : REF $XPO_IOB() ;
    .
    .
    .

    $XPO_GET_MEM( FULLWORDS = IOB$K_LENGTH,
                  RESULT = dynamic_iob ) ;
```

Upon successful allocation, the location dynamic_iob will contain the address of the requested memory element.

Note that for each of the macro calls shown above, the default failure-action routine XPO\$FAILURE is assumed, in lieu of a FAILURE= parameter.

4.3.2 \$XPO_FREE_MEM - Releasing Dynamic Memory

In order to release the memory elements acquired in the first and third examples above, the following \$XPO_FREE_MEM calls would be used:

```
$XPO_FREE_MEM( STRING = ( 1028, .bigbuff_ptr ) ) ;

$XPO_FREE_MEM( BINARY_DATA = ( IOB$K_LENGTH, .dynamic_iob ) ) ;
```

For either a string or binary-data element, if the element is described by an XPORT descriptor at the time of its release, only the descriptor address need be specified, as in the following example:

```
$XPO_FREE_MEM( STRING = bigbuff_desc ) ;
```

Optionally, a FILL parameter may be used in any form of \$XPO_FREE_MEM call to request memory clearing at release time. Also, FAILURE and SUCCESS action-routine parameters may be specified in either the \$XPO_GET_MEM or \$XPO_FREE_MEM macros, as described in Section 4.5.

Memory Management Facilities

MEMORY MANAGEMENT MACROS

4.3.3 Dynamic Memory Elements

The `$XPO_GET_MEM` and `$XPO_FREE_MEM` macros result in allocation or release of space in multiples of fullwords, regardless of the terms in which the element is described. (Any necessary "rounding up" is done by both functions, and thus can be ignored by the user.)

Dynamically acquired memory must be released in entire elements; that is, you may not release a portion of an allocated element. The result of releasing a partial element on any given target system is undefined.

4.4 COMPLETION CODES

The `$XPO_GET_MEM` and `$XPO_FREE_MEM` macros generate a call to an XPORT memory management routine that returns a value via the standard BLISS routine-value mechanism. Like the XPORT I/O routines, the get-memory and free-memory routines return a success or failure completion code as their routine value, as well as passing that code to any action routine called by the operation. (For the memory functions, however, there is no analogue of the IOB in which the completion code can be stored.)

As with all other XPORT completion codes, the memory-management completion codes can be tested for simple success or failure status with a low-bit test (set for success, cleared for failure). The specific failure completion codes are given, in terms of transportable XPORT literals, in Appendix A as part of the description of the respective macros.

4.5 ACTION ROUTINES

The XPORT memory-management macros allow the programmer to specify the address of a routine to be called at the completion of the requested function. Separate routines can be specified for successful and abnormal completion. These optional user-provided routines, called action routines, are typically used to intercept and possibly correct error conditions.

A success or failure action routine, if any, is called just before returning to the caller of the requested operation. The action routine is passed an XPORT function code -- in this case a code identifying the specific memory-management function (`XPO$K_GET_MEM` or `XPO$K_FREE_MEM`), the completion code of the current operation, and a function-specific parameter: the address of the XPORT descriptor associated with the particular request.

Memory Management Facilities

ACTION ROUTINES

An understanding of the usage of these parameters is best gained from inspection of an actual action routine. A listing of the default XPORT failure action routine, XPO\$FAILURE, appears in Appendix E. (The memory-management specific routines called by XPO\$FAILURE are XPO\$GM_FAILURE and XPO\$FM_FAILURE, which appear in the same module.) This routine is assumed if no FAILURE parameter is specified in your \$XPO_GET_MEM or \$XPO_FREE_MEM call. This routine issues an error message for all memory-management failures. Program execution is then terminated.

Optionally, the failure action routine XPO\$GM_FAILURE (for a get-memory failure) or XPO\$FM_FAILURE (for a free-memory failure) can be specified directly (in your macro call) to allow standard XPORT error reporting without terminating program execution. Instead, control is returned to the caller. See Appendix E for information about writing action routines.

CHAPTER 5 OTHER SYSTEM SERVICES

5.1	INTRODUCTION	5-1
5.2	\$XPO_PUT_MSG	5-1
5.2.1	Completion Codes	5-2
5.2.2	Action Routines	5-3
5.3	\$XPO_TERMINATE	5-3

CHAPTER 5

OTHER SYSTEM SERVICES

This chapter describes the "miscellaneous services" portion of the XPORT Programming Tools Facility. The description given here is intended to present concepts rather than give complete details in all cases. Appendix A contains complete, detailed descriptions of all XPORT macro calls in a form designed for concise reference.

5.1 INTRODUCTION

The services in the "miscellaneous" category perform commonly needed, but not necessarily related, functions such as error/exception message generation, and orderly program termination. The services are provided in a simple and uniform manner across all target operating systems.

It is expected that this collection of services will grow with time as the need for additional transportable functions is perceived.

5.2 \$XPO_PUT_MSG

The \$XPO_PUT_MSG macro allows you to issue single-line or multiple-line messages without naming or having opened an output device. A standard message text is automatically generated for any completion codes specified in the call.

Required 'input' to the macro is either a completion code (CODE parameter), e.g., one returned by another XPORT operation, or a message-string description (STRING parameter). Each of these parameters represents a single "line" of the message, and either can be given any number of times in one call, in any combination. Thus, "mixed" multi-line messages can be produced. Consider the following example:

Other System Services
\$XPO_PUT_MSG

```
$XPO_PUT_MSG(CODE = XPO$_END_FILE ,  
             STRING = 'No End-of-Data Sentinel Encountered' );
```

The message that would be produced for this call under TOPS-10, for example, would look as follows:

```
? end-of-file has been reached  
-   No End-of-Data Sentinel Encountered
```

Another possible input to the macro is a severity-level keyword (SEVERITY parameter). The range of severity levels is SUCCESS, WARNING, ERROR, and FATAL. If no SEVERITY parameter is given, the severity level defaults to the severity associated with the condition code specified, if the first parameter is CODE, or to ERROR if the first parameter is STRING.

To illustrate, the severity level associated with the XPO\$_END_FILE code is only WARNING. One might, for example, want to raise the severity level of the sample message given above, as follows:

```
$XPO_PUT_MSG(CODE = XPO$_END_FILE ,  
             SEVERITY = _ERROR ,  
             STRING = 'No End-of-Data Sentinel Encountered' );
```

The destination device(s) are automatically determined by the severity level, assuming that differing assignments for the standard output devices (\$XPO_OUTPUT and \$XPO_ERROR) are in effect for a given program application. All messages are sent to the standard output device (\$XPO_OUTPUT), whatever their severity level. Messages having a severity level other than SUCCESS are also sent to the standard error-logging device (\$XPO_ERROR).

Furthermore, issuance of a message sequence with an implied or explicit FATAL severity level causes automatic program termination, immediately following the message processing.

The standard XPORT messages are listed in Appendix C.

Other System Services

\$XPO_PUT_MSG

5.2.1 Completion Codes

The put-message operation returns a completion code both directly (i.e., by routine value) and to any action routine called by the operation (see below). Just as for I/O and message-management return values, the completion code can be tested for success/failure status with a simple low-bit test, or can be compared with completion code literals for more detailed testing. Specific completion codes are given in Appendix A together with the macro description.

Depending upon the target system, an \$XPO_PUT_MSG call may implicitly invoke \$XPO_PUT to perform message I/O. In this case, a failure completion code returned by the latter is "passed back" as the completion code of \$XPO_PUT_MSG. Any detailed error testing must take this into account.

5.2.2 Action Routines

Optionally, success and failure action routines may be specified for a put-message operation, just as for I/O and message-management macro calls (see Chapter 3 or 4). XPORT provides the default failure-action routine XPO\$FAILURE, which issues a message (if possible) for any message processing failure and terminates program execution.

Alternatively, you may specify the failure-action routine XPO\$PM_FAILURE (itself called by XPO\$FAILURE), which returns control to the call site after issuing a message unless the failure completion-code severity is FATAL.

See Appendix E for a listing of the default failure-action routines and for information about writing action routines.

5.3 \$XPO_TERMINATE

The \$XPO_TERMINATE macro causes program termination immediately following the issuance of a termination message to the end user. In its simpler form, e.g.:

```
$XPO_TERMINATE() ;
```

the assumed program-termination code is XPO\$TERMINATE. The message text for this code is "program terminated due to program request".

Other System Services
\$XPO_TERMINATE

You can, however, specify a program-termination code with a call of the form:

```
$XPO_TERMINATE( CODE = completion-code ) ;
```

The standard XPORT message text for the completion code is issued instead of the standard termination message.

Completion codes and corresponding message texts are listed in Appendix C.

CHAPTER 6 STRING HANDLING FACILITIES

6.1	STRING DESCRIPTORS	6-1
6.1.1	\$STR_DESCRIPTOR -- Creating a String Descriptor	6-2
6.1.2	\$STR_DESCRIPTOR -- Compile-Time Descriptor Initialization	6-2
6.1.3	\$STR_DESC_INIT -- Run-Time String Descriptor Initialization	6-3
6.1.4	String Descriptor Formats	6-4
6.1.5	String Descriptor Usage Rules	6-6
6.1.6	Descriptor Data Types	6-8
6.1.6.1	Character String Data Type (STR\$K_DTYPE_T)	6-8
6.1.6.2	Binary Data Type (STR\$K_DTYPE_Z)	6-9
6.2	STRING DESCRIPTOR STRUCTURE REFERENCES	6-9
6.3	STRING MODIFICATION	6-9
6.3.1	\$STR_COPY Operation	6-10
6.3.2	\$STR_APPEND Operation	6-11
6.4	STRING COMPARISON	6-12
6.5	STRING SCANNING	6-14
6.5.1	\$STR_SCAN Overview	6-14
6.5.2	\$STR_SCAN FIND - Find a Character Sequence	6-15
6.5.3	\$STR_SCAN SPAN - Match a Set of Characters	6-16
6.5.4	\$STR_SCAN STOP - Search for a Set of Characters	6-16
6.5.5	\$STR_SCAN - Returning a Substring	6-17
6.5.6	\$STR_SCAN - "Scanning Through" a BOUNDED String	6-17
6.6	STRING CONVERSION	6-18
6.6.1	\$STR_CONCAT and \$STR_FORMAT - ASCII to ASCII String Conversions	6-18
6.6.1.1	\$STR_CONCAT	6-18
6.6.1.2	\$STR_FORMAT	6-19
6.6.2	\$STR_ASCII - Binary-Data to ASCII String Conversion	6-19
6.6.3	Nesting \$STR_ASCII, \$STR_CONCAT, \$STR_FORMAT Pseudo-Functions	6-21
6.6.4	\$STR_BINARY - ASCII String to Binary-Data Conversion	6-22

CHAPTER 6

STRING HANDLING FACILITIES

This chapter describes tools related to the manipulation of character strings. See also Appendix A for reference descriptions of the macros discussed here.

6.1 STRING DESCRIPTORS

A string descriptor is a small control structure that facilitates the interchange of character data between two procedures (e.g., between two BLISS routines). The XPORT descriptors are closely modelled on the VAX/VMS descriptor convention. The XPORT I/O and memory-management facilities make extensive use of such descriptors internally, and fully support their use by the programmer -- a descriptor address is always a valid form of string parameter, for example.

Two macros are provided for string-descriptor creation and initialization: `$STR_DESCRIPTOR` and `$STR_DESC_INIT`. (See Chapter 7 concerning binary-data descriptors.)

Terminology: The term "length" always means "number of ASCII character positions", in the transportable-BLISS sense.

Descriptors essentially provide a mechanism for communicating nonscalar data between independent procedures in an indirect, uniform, and controlled fashion. They not only describe the location and extent of an item, but also describe its "class". By convention the class of an item reflects (1) the nature of its allocation and (2) a discipline to be observed when 'writing' such an item, i.e., when modifying the item and updating the descriptor accordingly. This discipline mainly applies to the recipient of a descriptor, that is, to a routine to which a descriptor address is passed.

String Handling Facilities

STRING DESCRIPTORS

The descriptor classes are: FIXED, BOUNDED, DYNAMIC, DYNAMIC_BOUNDED, and undefined. The specific usage conventions implied by each of these classes are described in Section 6.1.5. It should be noted here, however, that for the purposes of reading an item (i.e., fetching via the descriptor), all classes of descriptors are equivalent to FIXED.

6.1.1 \$STR_DESCRIPTOR -- Creating a String Descriptor

A string descriptor can be created using the \$STR_DESCRIPTOR macro as an attribute of a data declaration (of an OWN or LOCAL declaration, for example). The macro expands to an appropriate structure-attribute and field-attribute for the class of descriptor desired. A class may either be specified or may be defaulted to FIXED. For example, the declaration

```
OWN
    my_desc : $STR_DESCRIPTOR();
```

declares a FIXED descriptor, by default. Alternatively, the declaration

```
LOCAL
    output_string : $STR_DESCRIPTOR( CLASS = DYNAMIC );
```

declares a DYNAMIC string descriptor, for use in conjunction with dynamically allocated storage.

6.1.2 \$STR_DESCRIPTOR -- Compile-Time Descriptor Initialization

A descriptor must be initialized prior to its first use. A descriptor created in permanent storage (OWN or GLOBAL) can be statically initialized by means of parameters of the \$STR_DESCRIPTOR macro. Several examples follow:

```
OWN
    eof_text : $STR_DESCRIPTOR( STRING = 'End-of-file reached' ),
    composite_string : $STR_DESCRIPTOR( CLASS = DYNAMIC,
                                         STRING = (0,0) );
```

Note that a DYNAMIC descriptor in permanent storage can be initialized to point to 'no storage' (only) at compile time.

String Handling Facilities

STRING DESCRIPTORS

Further examples of static initialization:

```
OWN
    message-buffer : VECTOR[ CH$ALLOCATION(132) ],
    message : $STR_DESCRIPTOR( CLASS = BOUNDED,
                               STRING = (132, CH$PTR(message_buffer)) );
```

NOTE: A descriptor created in temporary (LOCAL) or dynamic storage must be dynamically initialized via the \$STR_DESC_INIT macro.

6.1.3 \$STR_DESC_INIT -- Run-Time String Descriptor Initialization

The executable \$STR_DESC_INIT macro dynamically initializes all fields of a descriptor. This macro must be used for descriptors created in temporary or dynamic storage.

Following are several examples of initializing the first descriptor 'declared' in the previous section:

```
$STR_DESC_INIT( DESCRIPTOR = my_desc );
```

The named descriptor is initialized as class FIXED by default, and the string length and pointer fields are set to indicate a null string value (i.e., a string of zero length).

A string value can be preset with a STRING parameter, as follows:

```
$STR_DESC_INIT( DESCRIPTOR = my_desc,
                 STRING = ( length-exp, pointer-exp ) );
```

Alternatively, a string literal may be given as the STRING parameter value:

```
$STR_DESC_INIT( DESCRIPTOR = my_desc,
                 STRING = 'Who struck John?' );
```

The dynamic descriptor 'declared' in the previous section might be initialized as follows:

```
$STR_DESC_INIT( DESCRIPTOR = output_string,
                 CLASS = DYNAMIC );
```

The named descriptor is initialized as the descriptor of a string created in dynamic memory during a subsequent operation (e.g., \$XPO_GET MEM, \$STR_COPY). Without a STRING parameter, the length and address fields are set to indicate a null string value.

String Handling Facilities

STRING DESCRIPTORS

Note particularly that the class that is specified (or defaulted) in the `$STR_DESC_INIT` macro must match the class specified or defaulted in the `$STR_DESCRIPTOR` macro. (The `CLASS=` parameter may be omitted in either or both macros only in the case of a `FIXED` descriptor.) Also, the class of a descriptor cannot be changed "in mid-stream", that is, without reinitialization.

6.1.4 String Descriptor Formats

All string descriptors contain the following common fields:

- o A length field, named `STR$H_LENGTH`,
- o A pointer field, named `STR$A_POINTER`,
- o A class field, named `STR$B_CLASS`, and
- o A data-type field, named `STR$B_DTYPE`.

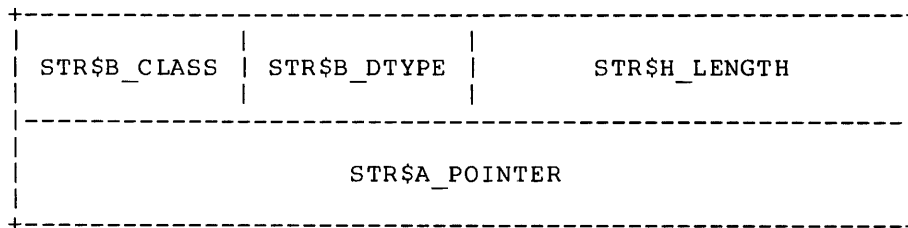
In addition to these fields, `BOUNDED` and `DYNAMIC_BOUNDED` descriptors also contain the following:

- o A prefix-length field, named `STR$H_PFXLEN`, and
- o A maximum-length field, named `STR$H_MAXLEN`.

Figure 6.1 shows the format a `FIXED` or `DYNAMIC` descriptor; figure 6.3 shows the format of a `BOUNDED` or `DYNAMIC_BOUNDED` descriptor. Note that a `BOUNDED` descriptor is simply an extension of a `FIXED` descriptor.

Figures 6.2 and 6.4 show the format of the strings described by these descriptors.

Figure 6.1
Format of a `FIXED` or `DYNAMIC` Descriptor



String Handling Facilities

STRING DESCRIPTORS

Figure 6.2
Format of a FIXED or DYNAMIC String

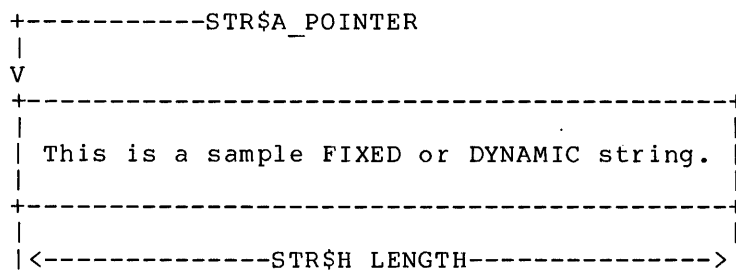


Figure 6.3
Format of a BOUNDED or DYNAMIC_BOUNDED Descriptor

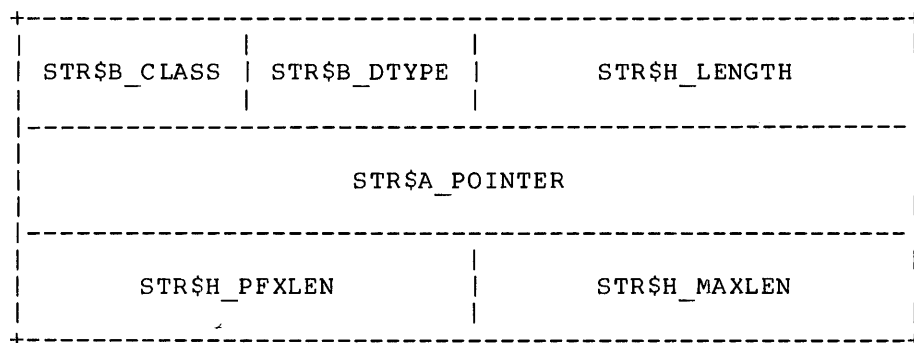
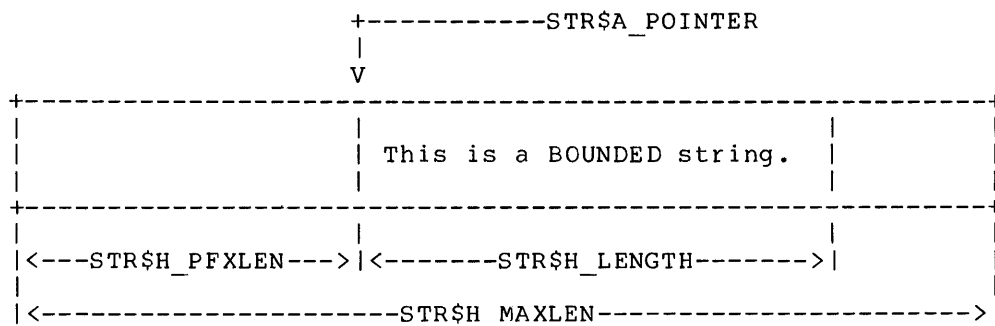


Figure 6.4
Format of a BOUNDED or DYNAMIC_BOUNDED String



String Handling Facilities

STRING DESCRIPTORS

6.1.5 String Descriptor Usage Rules

For each class of descriptor, the usage rules specify which fields may be modified, and which fields may not, by a procedure that is not the "owner" of the descriptor.

Such a procedure is typically a routine that is passed a descriptor address as one of its calling parameters. Moreover, the rules only apply where the descriptor is passed in order that the called routine can either modify information already described by the descriptor or can return new information to the caller via the descriptor. (Obviously, when a descriptor is used simply to pass information for reading only, no descriptor fields need be modified by the recipient. All descriptor classes are, in fact, equivalent to FIXED for purposes of reading, i.e., fetching data via the descriptor.)

The UNDEFINED class -- which is actually sort of a 'non-class' -- has no rules associated with it and is for intra-program use only.

Table 6.1 gives the usage rules for each class of descriptor.

Table 6.1
String Descriptor Usage Rules

Descriptor Class	Can recipient modify descriptor field?			
	Current Length STR\$H_LENGTH	String Pointer STR\$A_POINTER	Prefix Length STR\$H_PFXLEN	Maximum Length STR\$H_MAXLEN
FIXED	NO	NO	(n.a.)	(n.a.)
BOUNDED	YES	YES	YES	NO
DYNAMIC	YES*	YES*	(n.a.)	(n.a.)
DYNAMIC_BOUNDED	YES	YES*	YES	YES*
UNDEFINED	---	---	---	---
NOTE: It is always valid (with respect to these rules) to modify content, that is, to change the data described by the descriptor, within the limits set by these rules.				

* Requires interaction with \$XPO_FREE_MEM and \$XPO_GET_MEM.

String Handling Facilities

STRING DESCRIPTORS

The rules in Table 6.1 give rise to the following generalizations about descriptor classes (keep in mind that they do not necessarily apply to the "owner" of the descriptor):

- o A FIXED descriptor describes a string, whose location and extent may not be changed.

This class of descriptor is typically used to pass a string to another routine, and is seldom (almost never) used to describe space.

- o A BOUNDED descriptor describes a fixed-length buffer that contains a string of varying length. The string may begin at any point in the buffer and may extend to the end of the buffer. The buffer location and length may not be changed (presumably it is allocated in OWN, GLOBAL, or LOCAL storage); the string, however, may be moved, shortened and/or lengthened within the limits of the fixed-length buffer.

This class of descriptor is typically used to describe a space in which to construct or update a string which, by its nature, will not exceed a specific maximum length.

In addition to describing a bounded string, a BOUNDED descriptor implicitly describes three additional strings - the "container string", a "prefix string", and a "remainder string". The container string is the entire buffer and thus includes the bounded string. The prefix string is the portion of the container string (possibly null) that precedes the bounded string. The remainder string is the portion of the container string (possibly null) that follows the bounded string.

- o A DYNAMIC descriptor describes a moveable string whose length may vary from 0 to 64K characters. It is generally used for a string to be returned by a called routine.

Note that this class implies the use of dynamic memory, since the storage described is always assumed to be releasable. (See Chapter 4 concerning dynamic-memory facilities.)

- o A DYNAMIC_BOUNDED descriptor describes a moveable buffer containing a string of varying length. The string may begin at any point in the buffer and may extend to the end of the buffer.

As its name implies, this class is a generalization of both the BOUNDED and DYNAMIC classes. It can be used to avoid the maximum-length restriction imposed by BOUNDED; it can also be used for a very volatile string, to avoid much of the memory-management overhead implied by DYNAMIC (i.e., storage need be reallocated only when the buffer becomes too small).

String Handling Facilities

STRING DESCRIPTORS

Note that this class also implies the use of dynamic memory, since the storage described is always assumed to be releasable. (See Chapter 4 concerning dynamic-memory facilities.)

The term "moveable" implies that the allocated storage (if any) described by a passed descriptor can be released and newly-acquired string storage be described in its place.

Following are some typical uses for the four string descriptor classes:

- FIXED - To describe a standard message-text string.
- BOUNDED - To describe a LOCAL print-line buffer with a maximum length of 132 characters; or to describe a substring of a FIXED or DYNAMIC string.
- DYNAMIC - To describe a dynamically-created character string. For example, XPORT I/O uses a DYNAMIC descriptor (located in the IOB) to describe the resultant file specification, i.e., IOB\$T_RESULTANT.
- DYNAMIC_BOUNDED - To describe a character string whose length is continually changing, e.g., an unlimited print-line buffer. XPORT I/O uses this class of descriptor to describe its input-string buffer, i.e., IOB\$T_STRING.

6.1.6 Descriptor Data Types

The data type field in a descriptor identifies the type of item described by the descriptor. Although VAX/VMS defines a multitude of possible data types, only two of these are relevant to transportable descriptor usage: STR\$K_DTYPE_T and STR\$K_DTYPE_Z.

6.1.6.1 Character String Data Type (STR\$K_DTYPE_T)

This data type identifies XPORT-compatible ASCII character strings. The length field (STR\$H_LENGTH) of a character string descriptor specifies the number of characters in the string. The pointer field (STR\$A_POINTER) contains a BLISS character pointer which points to the first character of the string. A character string descriptor may be used in conjunction with all XPORT I/O, dynamic memory management and string processing functions.

String Handling Facilities

STRING DESCRIPTORS

6.1.6.2 Binary Data Type (STR\$K_DTYPE_Z)

This data type identifies XPORT-compatible binary data. It is described in Chapter 7, "Binary Data Descriptors".

6.2 STRING DESCRIPTOR STRUCTURE REFERENCES

An XPORT string descriptor is actually a transportable XPORT data structure. A reference description of this structure appears in the Appendix section B.2.

The individual fields of a string descriptor may be referred to using conventional BLISS field references. As an example of such references, consider the following routine which returns the first character of a string or a null (indicating a null string).

```
ROUTINE first_character( string ) =
  BEGIN
    MAP string : REF $STR_DESCRIPTOR();

    IF .string[STR$H_LENGTH] EQL 0
    THEN
      RETURN %CHAR(0)
    ELSE
      RETURN CH$RCHAR( .string[STR$A_POINTER] )
    END;
```

This sample routine (which does not distinguish between the null string and a NUL first character) will work for all classes of character strings in all target environments.

6.3 STRING MODIFICATION

The \$STR_COPY and \$STR_APPEND functions permit convenient modification of a string value regardless of the class of the string.

The \$STR_COPY function replaces a current string value with a new value. The \$STR_APPEND function adds characters to the end of an existing string. The following sample routine illustrates the use of these string modifications functions.

String Handling Facilities

STRING MODIFICATION

```
ROUTINE echo =
  BEGIN
    OWN
      terminal : $XPO_IOB(),
      message : $STR_DESCRIPTOR( CLASS = DYNAMIC_BOUNDED,
                                STRING = (0,0) );

    $XPO_OPEN( IOB = terminal, FILE_SPEC = $XPO_INPUT );

    WHILE $XPO_GET( IOB = terminal,
                    PROMPT = 'Enter a string: ' ) DO
      BEGIN
        $STR_COPY( STRING = 'ECHO: "', TARGET = message );
        $STR_APPEND( STRING = terminal[IOB$T_STRING],
                     TARGET = message );
        $STR_APPEND( STRING = '"', TARGET = message );
        $XPO_PUT( IOB = terminal, STRING = message )
      END
    END;
```

The actual processing performed by the `$STR_COPY` and `$STR_APPEND` functions depend on the class of the target string descriptor, as described in the following two sections.

6.3.1 `$STR_COPY` Operation

The string copy operation acts on the various classes of target strings as follows:

- o **FIXED target string:** The source string is copied to the target string, replacing any previous content. If the source string is shorter than the target string, the target is padded with trailing blanks.

If the source string is longer than the target string, it is either truncated (if requested by the user) or the operation fails. Optional truncation is specified by the parameter `OPTION=TRUNCATE`.

- o **DYNAMIC target string:** If the source and target strings are the same length, the source string is copied to the target string, replacing any previous content.

If the source string is either shorter or longer than the target string, the dynamic memory occupied by the target string is freed, a new dynamic-memory element is allocated, and the source string is copied to it. The target string descriptor is updated to reflect the new length and storage location.

String Handling Facilities

STRING MODIFICATION

- o BOUNDED target string: If the source string will fit within the bounded-plus-remainder portion of the target container string, the source string is copied to the target string and the length field of the target string descriptor is updated to reflect the new bounded-string length.

If the source string is longer than the bounded-plus-remainder portion of the target container string, it is either truncated (if requested by the user) or the operation fails. Optional truncation is specified by the parameter `OPTION=TRUNCATE`.

The prefix portion, if any, of the target container string is never modified by the copy operation.

- o DYNAMIC_BOUNDED target string: If the source string will fit within the bounded-plus-remainder portion of the target container string, the source string is copied to the target string and the length field of the target string descriptor is updated to reflect the new bounded-string length.

If the source string is longer than the bounded-plus-remainder portion of the target container string, a new dynamic-memory element is allocated, the prefix portion of the original target string plus the new source string is copied to it, and the original dynamic-memory element is freed. The target string descriptor is updated to reflect the new bounded-string length, storage location, and maximum (i.e., container string) length.

The prefix portion, if any, of the original target container string is never modified by the copy operation.

6.3.2 \$STR_APPEND Operation

The string append operation acts on the various classes of target strings as follows:

- o FIXED target string: An append operation is not permitted for a FIXED target string since a FIXED string cannot be extended.
- o DYNAMIC target string: The source string is logically concatenated with the target string and the result is copied into a newly allocated dynamic-memory element. The dynamic memory allocated for the original target string (if any) is then freed, and the target string descriptor is updated to reflect the new length and storage location.

String Handling Facilities

STRING MODIFICATION

- o **BOUNDED target string:** If the source string will fit within the remainder portion of the target container string, the source string is added to the end of the target string and the length field of the target string descriptor is updated to reflect the new bounded-string length.

If the source string is longer than the remainder portion of the target container string, it is either truncated (if requested by the user) or the operation fails. Optional truncation is specified by the parameter `OPTION=TRUNCATE`.

The prefix portion, if any, of the target container string is never modified by the append operation.

- o **DYNAMIC_BOUNDED target string:** If the source string will fit within the remainder portion of the target container string, the source string is added to the end of the target string and the length field of the target string descriptor is updated to reflect the new bounded-string length.

If the source string is longer than the remainder portion of the target container string, a new dynamic-memory element is allocated, the source string is logically concatenated with the prefix-plus-bounded portion of the original target string, and the result is copied to the new memory element. The original dynamic-memory element is then freed. The target string descriptor is updated to reflect the new bounded-string length, storage location, and maximum (i.e., container string) length.

The prefix portion, if any, of the original target container string is never modified by the append operation.

6.4 STRING COMPARISON

The BLISS language provides a collection of built-in character comparison functions (`CH$xxx`) which can be used to compare two character string values. Each of these functions uses pairs of length and pointer values to describe the characters strings. The XPORT string comparison functions provide a similar set of functions which are based on implicit or explicit string descriptors.

The basic form of all of the XPORT string comparison functions is as follows:

```
$STR_relation( STRING1 = string-info,  
               STRING2 = string-info  
               {, FILL = char-value-expression } )
```

String Handling Facilities

STRING COMPARISON

The "string-info" value can be the address of a string descriptor, a literal ASCII string, a string length/pointer pair, or an XPORT string pseudo-function (see Section 6.6). The "char-value-expression" can be any expression that produces a value between 0 and 127 (decimal) inclusive.

The following string comparison functions are provided:

<u>Function</u>	<u>Meaning</u>
\$STR_EQL	String-1 and string-2 are equal, i.e., both strings are the same length and have the same value.
\$STR_NEQ	String-1 and string-2 are not equal.
\$STR_LSS	The value of string-1 is less than the value of string-2.
\$STR_LEQ	The value of string-1 is less than or equal to the value of string-2.
\$STR_GEQ	The value of string-1 is greater than or equal to the value of string-2.
\$STR_GTR	The value of string-1 is greater than the value of string-2.
\$STR_COMPARE	Compare string-1 for a less-than, equal-to, or greater-than relationship to string-2.

With the exception of \$STR_COMPARE, these functions return a value of 1 if the comparison is satisfied, a value of 0 (zero) if the comparison is not satisfied, or an error completion code if either of the string descriptors is invalid. \$STR_COMPARE returns values of -1, 0, or +1 for less-than, equal-to, or greater-than, respectively.

In order to compare as equal, two strings must be equal in length as well as content. If requested with the FILL= parameter, the shorter of the two character strings is extended by adding "fill characters" to make the two strings the same length.

The relationship between two string values (e.g., less than) depends on the ordering of the characters within the two strings. That ordering is determined by the ASCII collating sequence.

String Handling Facilities

STRING COMPARISON

The following sample routine, which determines whether a keyword begins with a capital letter, illustrates the use of the string comparison functions.

```
ROUTINE keyword_test( keyword_desc ) =  
  BEGIN  
    IF $STR_LSS( STRING1 = .keyword_desc, STRING2 = 'A' ) OR  
      $STR_GTR( STRING1 = .keyword_desc, STRING2 = 'Z',  
                FILL = 'Z' )  
    THEN  
      RETURN 0  
    ELSE  
      RETURN 1  
    END;
```

6.5 STRING SCANNING

6.5.1 \$STR_SCAN Overview

The \$STR_SCAN function performs three different types of string scanning:

- o It can locate a specific sequence of characters within a string (FIND mode).
- o It can match a stream of specific characters (SPAN mode).
- o It can search for one of a set of specific characters (STOP mode).

Each of these scanning operations determines a unique substring of the string being scanned.

There are several possible results of these string scanning operations:

- o A completion code is returned which indicates whether the operation was successful.
- o A substring descriptor (FIXED or BOUNDED only) can be filled in.
- o A copy of the resulting substring can be returned.

String Handling Facilities
STRING SCANNING

- o The character that terminated the search can be returned.

For the purpose of presenting meaningful string scanning examples further on, imagine a simple application which reads and scans data records which have the following syntax:

```
keyword {=value} ,...
```

Following are sample data records which conform to this syntax:

```
BEGIN
COPIES=5
SUB_TOTALS, TOTALS, GRAND_TOTAL
END
```

Each of the string-scanning code fragments presented later on is assumed to appear in the following routine:

```
ROUTINE data_scan( data_line ) =
BEGIN
BIND
    line = .data_line : $STR_DESCRIPTOR();

OWN
    keyword : $STR_DESCRIPTOR( STRING=(0,0) ),
    keyword_copy : $STR_DESCRIPTOR( CLASS = DYNAMIC,
                                    STRING = (0,0) ),
    keyword_delim;

$STR_DESC_INIT( DESCRIPTOR = line, CLASS = BOUNDED,
                STRING = .data_line );
. . .
```

6.5.2 \$STR_SCAN FIND - Find a Character Sequence

The following code fragments illustrate the use of the \$STR_SCAN FIND operation to locate a specific sequence of characters within a string, and to modify a descriptor to isolate that sequence (if found).

String Handling Facilities

STRING SCANNING

```
IF $STR_SCAN( STRING = line, FIND = ':' )
THEN
  BEGIN
    $XPO_PUT_MSG( STRING = 'Obsolete syntax in following line',
                  STRING = line );
    RETURN 0
  END;
```

6.5.3 \$STR_SCAN SPAN - Match a Set of Characters

The \$STR_SCAN SPAN operation determines the longest substring which (1) begins at the start of a string and (2) consists solely of a specified set of characters. A successful SPAN operation may result in a null substring, i.e., zero characters spanned.

The following code fragment illustrates the creation of a substring descriptor for the first keyword of a data record.

```
$STR_SCAN( STRING = line,
           SPAN = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_',
           SUBSTRING = keyword,
           DELIMITER = keyword_delim );
```

This scan call will isolate any upper-case symbol occurring at the beginning of the data line, will modify the descriptor named "keyword" to describe that symbol, and will also return the character immediately following the symbol in the data-segment named "keyword_delim".

6.5.4 \$STR_SCAN STOP - Search for a Set of Characters

The \$STR_SCAN STOP operation determines the longest substring which (1) begins at the start of a string and (2) does not contain any of a specified set of characters. A STOP operation may result in a null substring.

The following code fragment illustrates an alternate method for the creation of a substring descriptor. It is designed to pick up any substring delimited by a comma or an equal sign.

```
$STR_SCAN( STRING = line,
           STOP = ',=',
           SUBSTRING = keyword,
           DELIMITER = keyword_delim );
```

String Handling Facilities

STRING SCANNING

6.5.5 \$STR_SCAN - Returning a Substring

A copy of a substring determined by any type of scan operation (i.e., FIND, SPAN, or STOP) can be created automatically through use of the TARGET= parameter. The target descriptor must be FIXED or DYNAMIC class only.

The following code fragment illustrates the creation of a target descriptor for a copy of the first keyword of a data record.

```
$STR_SCAN( STRING = line,  
           SPAN = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_',  
           TARGET = keyword_copy );
```

This scan call will isolate any upper-case symbol occurring at the beginning of the data line, will make a copy of that symbol, and will modify the descriptor named "keyword_copy" to describe the copy.

6.5.6 \$STR_SCAN - "Scanning Through" a BOUNDED String

In the general case, the \$STR_SCAN functions make conventional use of the source string descriptor (specified with the STRING= parameter). That is, the STR\$H_LENGTH and STR\$A_POINTER fields of the descriptor determine the string to be scanned.

However, if a BOUNDED or DYNAMIC_BOUNDED descriptor is specified in a REMAINDER= parameter, the remainder string (rather than the bounded string) is scanned. In order to use this form of string scanning, a bounded descriptor must be declared and initialized:

```
LOCAL  
  scan_line : $STR_DESCRIPTOR( CLASS = BOUNDED);  
  
$STR_DESC_INIT( DESCRIPTOR = scan_line,  
               CLASS = BOUNDED,  
               STRING = .data_line );
```

Moreover, if the same BOUNDED or DYNAMIC_BOUNDED descriptor is specified for both the REMAINDER= and SUBSTRING= parameters, the special operation of "scanning through a string", one field at a time, is achieved in a greatly simplified fashion. For example, the following two code fragments each skip past the sequence of commas and spaces which can precede the next keyword.

```
$STR_SCAN( REMAINDER = scan_line,  
           SPAN = ',',  
           SUBSTRING = scan_line );
```

String Handling Facilities

STRING SCANNING

This fragment is equivalent to:

```
BEGIN
LOCAL temp_desc :  $STR_DESCRIPTOR();
$STR_DESC_INIT( DESCRIPTOR = temp_desc, CLASS = FIXED );

$STR_SCAN( STRING = (.scan_line[STR$H_MAXLEN] -
                    .scan_line[STR$H_LENGTH] -
                    .scan_line[STR$H_PFXLEN],
                    CH$PLUS(.scan_line[STR$A_POINTER],
                    .scan_line[STR$H_LENGTH])),
          SPAN = ' ', ' ',
          SUBSTRING = temp_desc );

scan_line[STR$H_PFXLEN] = .scan_line[STR$H_PFXLEN] +
                        .scan_line[STR$H_LENGTH];
scan_line[STR$H_LENGTH] = .temp_desc[STR$H_LENGTH];
scan_line[STR$A_POINTER] = .temp_desc[STR$A_POINTER];
END;
```

6.6 STRING CONVERSION

6.6.1 \$STR_CONCAT and \$STR_FORMAT - ASCII to ASCII String Conversions

The simplest form of string conversion is to (logically) modify the form of an ASCII string. The pseudo-functions which perform this type of conversion are \$STR_CONCAT and \$STR_FORMAT. These pseudo-functions may only be used in conjunction with the STRING= parameter and similar parameters of XPORT I/O and String Handling functions.

Wherever possible, the examples in this section are built upon (modifications of) examples presented earlier.

6.6.1.1 \$STR_CONCAT

The \$STR_CONCAT pseudo-function allows two or more strings to be logically concatenated to form a single logical string.

The following code fragment illustrates the use of the \$STR_CONCAT pseudo-function to specify a single logical string which consists of two literal strings and a variable string (that is, an IOB string-descriptor address; cf. Chapter 3).

String Handling Facilities

STRING CONVERSION

```
$XPO_PUT( IOB = terminal,  
          STRING = $STR_CONCAT( 'ECHO: "',  
                                terminal[IOB$T_STRING],  
                                '"' ) );
```

6.6.1.2 \$STR_FORMAT

The \$STR_FORMAT pseudo-function provides a means of specifying non-standard string characteristics or processing. For example, a string can be converted to upper-case as part of an append operation.

```
$STR_APPEND( STRING = $STR_FORMAT( terminal[IOB$T_STRING], UP_CASE ),  
            TARGET = message );
```

Likewise, the case of a string could be 'ignored' in a string comparison or scanning operation, by means of the same logical transformation.

```
IF $STR_EQL( STRING1 = $STR_FORMAT( keyword, UP_CASE ),  
            STRING2 = 'YES' )  
THEN  
    . . .
```

The length and alignment of a string can also be specified using \$STR_FORMAT:

```
$STR_COPY( STRING = $STR_FORMAT( keyword, LENGTH=20, RIGHT_JUSTIFY ),  
          TARGET = message );  
  
$XPO_PUT( IOB = terminal,  
          STRING = $STR_FORMAT( 'Hello User!', LENGTH=80, CENTER ) );
```

6.6.2 \$STR_ASCII - Binary-Data to ASCII String Conversion

The \$STR_ASCII pseudo-function provides a means of easily creating an ASCII string representation of a single binary value. This pseudo-function, like the \$STR_CONCAT and \$STR_FORMAT pseudo-functions, may only be used in conjunction with the STRING= parameter and similar parameters on XPORT I/O and String Handling functions.

The following sample routine illustrates the use of the \$STR_ASCII pseudo-function to output an ASCII number to the user's terminal:

String Handling Facilities STRING CONVERSION

```
ROUTINE put_ascii : NOVALUE =
  BEGIN
  OWN
    number : INITIAL( 1234 ),
    negative : INITIAL( -44 ),
    terminal : $XPO_IOB();

  IF NOT .terminal[ IOB$V_OPEN ]
  THEN
    $XPO_OPEN( IOB = terminal, FILE_SPEC = $XPO_OUTPUT );

    $XPO_PUT( IOB = terminal,
      STRING = $STR_ASCII( .number ) );

  RETURN;
  END;
```

A variety of \$STR_ASCII conversion options permit the user to specify the exact form of binary to ASCII conversion desired. For example, the following list shows the terminal output which would result on VAX/VMS from the substitution of various \$STR_ASCII calls in the previous code fragment.

<u>STRING=</u>	<u>Terminal Output</u>
\$STR_ASCII(.number)	1234
\$STR_ASCII(.number,BASE10)	1234
\$STR_ASCII(.number,BASE8)	00000002322
\$STR_ASCII(.number,BASE8,LENGTH=4)	2322
\$STR_ASCII(.number,BASE16)	000004D2
\$STR_ASCII(.number,BASE2,LENGTH=12)	010011010010
\$STR_ASCII(.number,BASE8,LEADING_BLANK)	2322
\$STR_ASCII(.number,BASE10,LEADING_ZERO)	0000001234
\$STR_ASCII(.number,BASE10,LENGTH=8)	1234
\$STR_ASCII(.negative)	-44
\$STR_ASCII(.negative,BASE10)	-44
\$STR_ASCII(.negative,BASE8)	37777777724
\$STR_ASCII(.negative,BASE16)	FFFFFFD4
\$STR_ASCII(.negative,BASE8,SIGNED)	-00000000054
\$STR_ASCII(.negative,BASE10,UNSIGNED)	4294967252

Table 6.2 shows the possible \$STR_ASCII integer conversion options and indicates the option defaults. Note that the defaults for BASE10 conversion are opposite from the defaults for BASE2, BASE8 and BASE16.

String Handling Facilities
STRING CONVERSION

Table 6.2
\$STR_ASCII Integer Conversion Options

Option Type	Option	Default
Result radix	BASE2, BASE8 BASE10, BASE16	BASE10
Integer type	SIGNED, UNSIGNED	If BASE10 then SIGNED, else UNSIGNED
Nonsignificant digit	LEADING_BLANK, LEADING_ZERO	If BASE10 then LEADING_BLANK, else LEADING_ZERO
String length	LENGTH=constant	If LEADING BLANK then LENGTH=minimum, else LENGTH=large enough for maximum value

6.6.3 Nesting \$STR_ASCII, \$STR_CONCAT, \$STR_FORMAT Pseudo-Functions

The \$STR_ASCII, \$STR_CONCAT, and \$STR_FORMAT pseudo-functions may be nested as desired; that is, a \$STR_ASCII, \$STR_CONCAT, or \$STR_FORMAT pseudo-function may be the string argument of either a \$STR_CONCAT or \$STR_FORMAT pseudo-function.

The following sample routine illustrates nesting of string conversion pseudo-functions. This routine formats and outputs a single line of sequence-numbered text (created by an SOS editor).

```
ROUTINE put_sos( number, page, text ) =
  BEGIN
    $XPO_PUT( IOB = terminal,
      STRING = $STR_CONCAT( $STR_ASCII(.number,LENGTH=5,LEADING_ZERO),
        '/',
        $STR_ASCII(.page,LENGTH=3,LEFT_JUSTIFY),
        $STR_FORMAT(.text,LENGTH=123,TRUNCATE) ) )
  END;
```

String Handling Facilities

STRING CONVERSION

The output of this routine will have the following format:

00150/2 a single line of text

where "00150" is a line sequence number (always 5 digits with leading zeros) and "2" is a page number (left justified with no leading zeros).

6.6.4 \$STR_BINARY - ASCII String to Binary-Data Conversion

The \$STR_BINARY function provides a means of easily converting an ASCII string value into a corresponding binary value.

The following sample routine illustrates the use of \$STR_BINARY to convert a string value into a binary value.

```
ROUTINE echo_test : NOVALUE =
  BEGIN
    LOCAL echo_count;
    OWN terminal : $XPO_IOB();

    $XPO_OPEN( IOB = terminal, FILE_SPEC = $XPO_INPUT);

    $XPO_GET( IOB = terminal,
      PROMPT = 'Enter number of times to echo each string ' );

    $STR_BINARY( STRING = terminal[IOB$T_STRING],
      OPTION = BASE10,
      RANGE = ( 1,999 ),
      RESULT = echocount );

    WHILE $XPO_GET( IOB = terminal,
      PROMPT = 'Enter string: ' ) DO
      INCR counter FROM 1 TO .echo_count DO
        $XPO_PUT( IOB = terminal,
          STRING = terminal[IOB$T_STRING] );

    END;
```

CHAPTER 7 BINARY DATA DESCRIPTORS

7.1	INTRODUCTION	7-1
7.2	BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION	7-2
7.2.1	\$XPO_DESCRIPTOR -- Creating a Binary Data Descriptor	7-2
7.2.2	\$XPO_DESCRIPTOR -- Compile-Time Descriptor Initialization	7-2
7.2.3	\$XPO_DESC_INIT -- Run-Time Data Descriptor Initialization	7-3
7.2.4	Classes Of Descriptors	7-4

CHAPTER 7

BINARY DATA DESCRIPTORS

This chapter describes macros related to the manipulation of binary data (i.e., non-character-string data). They are used in conjunction with the XPORT I/O and memory-management facilities described in Chapters 3 and 4 respectively. See also Appendix A for reference descriptions of the macros discussed here.

7.1 INTRODUCTION

A binary-data descriptor is a small block structure used to describe a binary data item. (These descriptors are modelled on a VAX/VMS convention. The string descriptor concept, as described in Chapter 6, is extended here to include binary data as well.) The XPORT I/O and memory-management facilities make extensive use of such descriptors internally, and fully support their use by the programmer -- a descriptor address is always a valid form of binary data parameter, for example.

Two macros are provided for data-descriptor creation and initialization: `$XPO_DESCRIPTOR` and `$XPO_DESC_INIT`.

Descriptors essentially provide a mechanism for communicating nonscalar data between independent procedures in an indirect, uniform, and controlled fashion. They not only describe the location and extent of an item, but also describe its "class". By convention the class of an item reflects (1) the nature of its allocation and (2) a discipline to be observed when 'writing' such an item, i.e., when modifying the item and updating the descriptor accordingly. This discipline mainly applies to the recipient of a descriptor, that is, to a routine to which a descriptor address is passed.

The descriptor classes are: `FIXED`, `BOUNDED`, `DYNAMIC`, `DYNAMIC BOUNDED`, and undefined. The specific usage conventions implied by each of these classes are described in Section 7.2.3. It should be noted here, however, that for the purposes of reading an item (i.e., fetching via the descriptor), all classes of descriptors are equivalent to `FIXED`.

Binary Data Descriptors
BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

7.2 BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

7.2.1 \$XPO_DESCRIPTOR -- Creating a Binary Data Descriptor

A binary data descriptor can be created using the \$XPO_DESCRIPTOR macro as an attribute of a data declaration (of an OWN or LOCAL declaration, for example). The macro expands to an appropriate structure-attribute and field-attribute for the class of descriptor desired. For example, the declaration

```
OWN
    data_buffer : $XPO_DESCRIPTOR();
```

declares a FIXED descriptor, by default. In contrast, the following declaration

```
GLOBAL
    integer_array : $XPO_DESCRIPTOR( CLASS = BOUNDED );
```

explicitly declares a BOUNDED data descriptor.

As a further example, the declaration

```
LOCAL
    temp_work_area : $XPO_DESCRIPTOR( CLASS = DYNAMIC );
```

declares a DYNAMIC data descriptor for use in conjunction with dynamically allocated storage.

7.2.2 \$XPO_DESCRIPTOR -- Compile-Time Descriptor Initialization

A binary data descriptor must be initialized prior to its first use. A descriptor created in permanent storage (OWN or GLOBAL) can be statically initialized by means of parameters of the \$XPO_DESCRIPTOR macro. Several examples follow:

```
OWN
    dyna_data : $XPO_DESCRIPTOR( CLASS = DYNAMIC,
                                BINARY_DATA = (0,0) );
```

The descriptor is initialized as class DYNAMIC and the data-item size and address fields of the descriptor are set to indicate a null item (zero size and address). Note that a DYNAMIC descriptor created in permanent storage can only be initialized to point to 'no storage' at compile time; that is, the only dynamic-storage initialization value acceptable at compile time is BINARY_DATA = (0,0).

Binary Data Descriptors

BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

A FIXED (or BOUNDED) descriptor created in permanent storage can be preset with a non-null data-item value, as shown in the following example:

```
OWN
  set_of_data : VECTOR[100],
  set_desc : $XPO_DESCRIPTOR( BINARY_DATA =
                               (100,set_of_data) );
```

By default, the size expression (100) is interpreted as being expressed in BLISS fullwords (as opposed to addressable units). Alternatively, the size expression could be indicated as being expressed in addressable units, as shown in the following example (BLISS-16/32 only):

```
OWN
  many_bytes : VECTOR[1000,BYTE],
  byte_desc : $XPO_DESCRIPTOR( BINARY_DATA =
                               ( 1000, many_bytes, UNITS ) );
```

Note that, in the interests of program clarity (and for purposes of reinitialization), the alternative keyword FULLWORDS can also be specified explicitly, although it is assumed by default.

A descriptor created in temporary (LOCAL) or dynamic storage must be dynamically initialized via the \$XPO_DESC_INIT macro.

7.2.3 \$XPO_DESC_INIT -- Run-Time Data Descriptor Initialization

The executable \$XPO_DESC_INIT macro dynamically initializes all fields of a binary data descriptor. This macro must be used for descriptors created in temporary or dynamic storage.

The dynamic descriptor 'declared' in Section 7.2.1 might be initialized as follows:

```
$XPO_DESC_INIT( DESCRIPTOR = temp_work_area ,
                 CLASS = DYNAMIC );
```

The named descriptor is initialized as a descriptor of dynamic binary data. In the absence of a BINARY_DATA parameter, the size and address fields are set to indicate a null value. A descriptor created in temporary storage could be initialized to describe an area of local storage, as follows:

Binary Data Descriptors BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

```

LOCAL
    result_values : BLOCK[ result_size ],
    result_desc : $XPO_DESCRIPTOR();

$XPO_DESC_INIT( DESCRIPTOR = result_desc,
                BINARY_DATA = ( result_size, result_values ) );

```

Note particularly that the class that is specified (or defaulted) in the \$XPO_DESC_INIT macro must match the class that is either specified or defaulted in the \$XPO_DESCRIPTOR macro. (That is, the CLASS parameter may be omitted in either or both macros only in the case of a FIXED descriptor.) Also, the class of a descriptor cannot be changed "in mid-stream", that is, without reinitialization.

7.2.4 Classes Of Descriptors

All data descriptors contain the following common fields:

- o A size field, named XPO\$H_LENGTH,
- o A address field, named XPO\$A_ADDRESS,
- o A class field, named XPO\$B_CLASS, and
- o A data-type field, named XPO\$B_DTYPE.

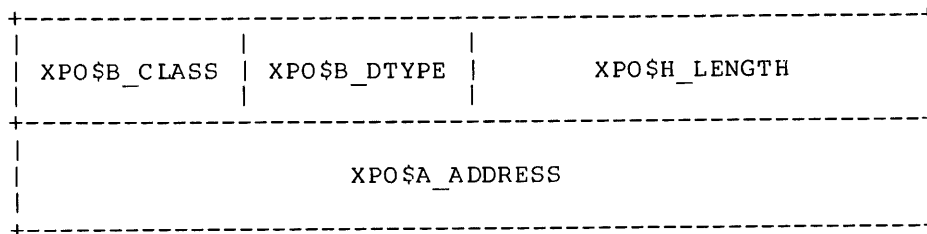
In addition to these fields, BOUNDED and DYNAMIC_BOUNDED descriptors also contain the following:

- o A prefix-size field, named XPO\$H_PFXLEN, and
- o A maximum-size field, named XPO\$H_MAXLEN.

Figure 7.1 shows the format a FIXED or DYNAMIC descriptor; figure 7.3 shows the format of a BOUNDED or DYNAMIC_BOUNDED descriptor. Note that a BOUNDED descriptor is simply an extension of a FIXED descriptor.

Figures 7.2 and 7.4 show the format of the data items described by these descriptors.

Figure 7.1
Format of a FIXED or DYNAMIC Descriptor



Binary Data Descriptors
BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

Figure 7.2
Format of a FIXED or DYNAMIC Data Item

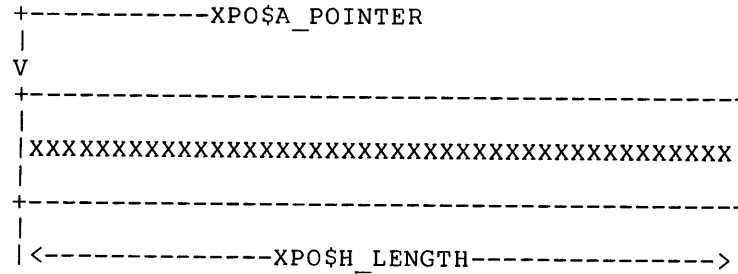


Figure 7.3
Format of a BOUNDED or DYNAMIC_BOUNDED Descriptor

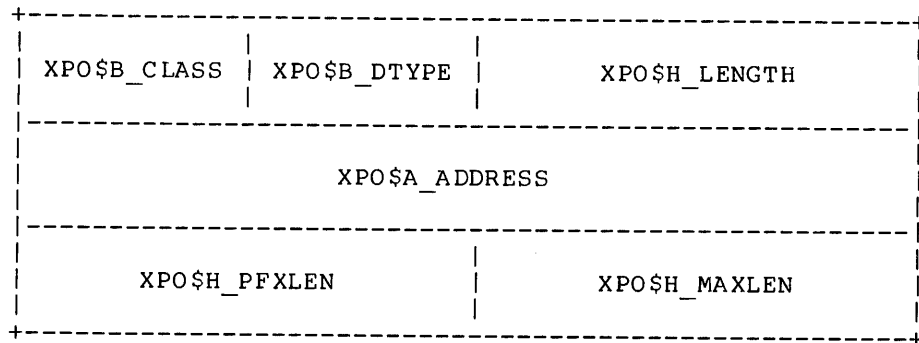
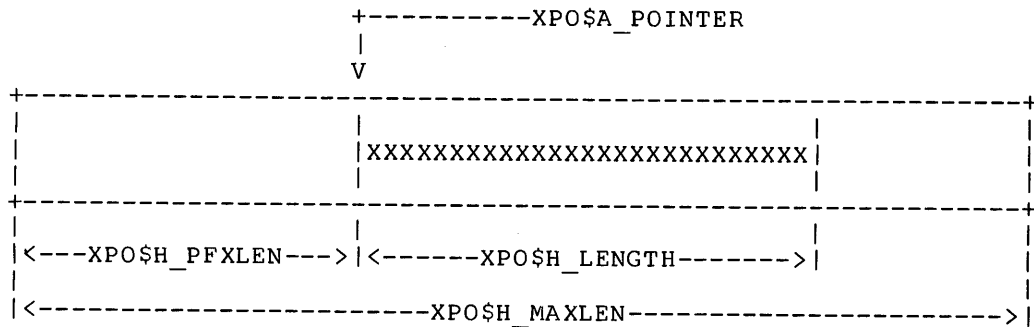


Figure 7.4
Format of a BOUNDED or DYNAMIC_BOUNDED Data Item



Binary Data Descriptors BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

For each class of descriptor, the usage rules specify which fields may be modified and which fields may not, by a procedure that is not the "owner" of the descriptor.

Such a procedure is typically a routine that is passed a descriptor address as one of its calling parameters. Moreover, the rules only apply where the descriptor is passed in order that the called routine can either modify information already described by the descriptor or can return new information to the caller via the descriptor. (Obviously, when a descriptor is used simply to pass information for reading only, no descriptor fields need be modified by the recipient. All descriptor classes are, in fact, equivalent to FIXED for purposes of reading, i.e., fetching data via the descriptor.)

The UNDEFINED class -- which is actually a 'non-class' -- has no rules associated with it and is for intra-program use only.

Table 7.1 gives the usage rules for each class of descriptor.

Table 7.1
Descriptor Usage Rules

Descriptor Class	Can recipient modify descriptor field?			
	Current Length XPO\$H_LENGTH	ITEM Address XPO\$A_ADDRESS	Prefix Length XPO\$H_PFXLEN	Maximum Length XPO\$H_MAXLEN
FIXED	NO	NO	(n.a.)	(n.a.)
BOUNDED	YES	NO	YES	NO
DYNAMIC	YES*	YES*	(n.a.)	(n.a.)
DYNAMIC_BOUNDED	YES	YES*	YES	YES*
UNDEFINED	---	---	---	---
NOTE: It is always valid (with respect to these rules) to modify content, that is, to change the information described by the descriptor, within the limits set by these rules.				
* Requires interaction with \$XPO_FREE_MEM and \$XPO_GET_MEM.				

Binary Data Descriptors
BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

The rules in Table 7.1 give rise to the following generalizations about descriptor classes (keep in mind that they do not necessarily apply to the "owner" of the descriptor):

- o A FIXED descriptor describes a data item, whose location and extent may not be changed.

This class of descriptor is typically used to pass an item to another routine, and is seldom (almost never) used to describe space.

- o A BOUNDED descriptor describes a fixed-length data buffer that contains an item of varying length. The item may begin at any addressable point in the buffer and may extend to the end of the buffer. The buffer location and size may not be changed (presumably it is allocated in OWN, GLOBAL, or LOCAL storage); the item, however, may be moved, shortened and/or lengthened within the limits of the fixed-size buffer.

This class of descriptor is typically used to describe a space in which to construct or update an item which, by its nature, will not exceed a given maximum size.

In addition to describing a bounded item, a BOUNDED descriptor implicitly describes three additional data items - the "container item", a "prefix item", and a "remainder item". The container item is the entire buffer and thus includes the bounded item. The prefix item is the portion of the container item (possibly null) that precedes the bounded item. The remainder item is the portion of the container item (possibly null) that follows the bounded item.

- o A DYNAMIC descriptor describes a moveable data item whose size may vary from 0 to 64K addressable units. It is generally used for an item to be returned by a called routine.

Note that this class implies the use of dynamic memory, since the storage described is always assumed to be releasable. (See Chapter 4 concerning dynamic-memory facilities.)

- o A DYNAMIC_BOUNDED descriptor describes a moveable data buffer containing an item of varying size. The item may begin at any addressable point in the buffer and may extend to the end of the buffer.

As its name implies, this class is a generalization of both the BOUNDED and DYNAMIC classes. It can be used to avoid the maximum-size restriction imposed by BOUNDED; it can also be used for a very volatile item, to avoid much of the memory-management overhead implied by DYNAMIC (i.e., storage need be reallocated only when the buffer becomes too small).

Binary Data Descriptors
BINARY DATA DESCRIPTOR CREATION AND INITIALIZATION

Note that this class also implies the use of dynamic memory, since the storage described is always assumed to be releasable. (See Chapter 4 concerning dynamic-memory facilities.)

The term "moveable" implies that the allocated storage (if any) described by a passed descriptor can be released and newly-acquired storage be described in its place.

APPENDIX A

MACRO DESCRIPTIONS

A.1	DESCRIPTIVE NOTATION AND CONVENTIONS	A-1
A.1.1	Syntax Notation	A-1
A.1.2	Character-String and Binary-Data Parameters	A-2
A.1.2.1	String and Data Descriptors	A-3
A.2	\$STR_APPEND - Append a String	A-4
A.2.1	Syntax	A-4
A.2.2	Restrictions	A-4
A.2.3	Parameter Semantics	A-4
A.2.4	Operational Semantics	A-5
A.2.5	Completion Codes	A-6
A.3	\$STR_ASCII - Binary-to-ASCII Conversion Pseudo-Function	A-7
A.3.1	Syntax	A-7
A.3.2	Restrictions	A-7
A.3.3	Parameter Semantics	A-7
A.3.4	Usage Guidelines	A-8
A.4	\$STR_BINARY - Convert ASCII to Binary	A-9
A.4.1	Syntax	A-9
A.4.2	Restrictions	A-9
A.4.3	Parameter Semantics	A-10
A.4.4	Usage Guidelines	A-10
A.4.5	Completion Codes	A-11
A.5	\$STR_COMPARE - String Comparison	A-12
A.5.1	Syntax	A-12
A.5.2	Restrictions	A-12
A.5.3	Parameter Semantics	A-12
A.5.4	Operational Semantics	A-13
A.5.5	Completion Codes	A-13
A.6	\$STR_CONCAT - String Concatenation Pseudo-Function	A-14
A.6.1	Syntax	A-14
A.6.2	Restrictions	A-14
A.6.3	Parameter Semantics	A-14
A.6.4	Usage Guidelines	A-15
A.7	\$STR_COPY - Copy a String	A-16
A.7.1	Syntax	A-16
A.7.2	Restrictions	A-16
A.7.3	Parameter Semantics	A-16
A.7.4	Operational Semantics	A-17
A.7.5	Completion Codes	A-18
A.8	\$STR_DESCRIPTOR - Declare a String Descriptor	A-19
A.8.1	Syntax	A-19
A.8.2	Restrictions	A-19
A.8.3	Parameter Semantics	A-19
A.9	\$STR_DESC_INIT - Initialize a String Descriptor	A-21
A.9.1	Syntax	A-21
A.9.2	Restrictions	A-21
A.9.3	Parameter Semantics	A-21
A.9.4	Completion Code	A-22
A.10	\$STR_EQ - String Equality Comparison	A-23
A.10.1	Syntax	A-23
A.10.2	Restrictions	A-23
A.10.3	Parameter Semantics	A-23
A.10.4	Operational Semantics	A-24

A.10.5	Completion Codes	A-25
A.11	\$STR_FORMAT - String Formatting Pseudo-Function	A-26
A.11.1	Syntax	A-26
A.11.2	Restrictions	A-26
A.11.3	Parameter Semantics	A-27
A.11.4	Usage Guidelines	A-27
A.12	\$STR_GEQ - String Greater-Than-or-Equal Comparison	A-29
A.12.1	Syntax	A-29
A.12.2	Restrictions	A-29
A.12.3	Parameter Semantics	A-29
A.12.4	Operational Semantics	A-30
A.12.5	Completion Codes	A-31
A.13	\$STR_GTR - String Greater-Than Comparison	A-32
A.13.1	Syntax	A-32
A.13.2	Restrictions	A-32
A.13.3	Parameter Semantics	A-32
A.13.4	Operational Semantics	A-33
A.13.5	Completion Codes	A-34
A.14	\$STR_LEQ - String Less-Than-or-Equal Comparison	A-35
A.14.1	Syntax	A-35
A.14.2	Restrictions	A-35
A.14.3	Parameter Semantics	A-35
A.14.4	Operational Semantics	A-36
A.14.5	Completion Codes	A-37
A.15	\$STR_LSS - String Less-Than Comparison	A-38
A.15.1	Syntax	A-38
A.15.2	Restrictions	A-38
A.15.3	Parameter Semantics	A-38
A.15.4	Operational Semantics	A-39
A.15.5	Completion Codes	A-40
A.16	\$STR_NEQ - String Inequality Comparison	A-41
A.16.1	Syntax	A-41
A.16.2	Restrictions	A-41
A.16.3	Parameter Semantics	A-41
A.16.4	Operational Semantics	A-42
A.16.5	Completion Codes	A-43
A.17	\$STR_SCAN - String Scanning	A-44
A.17.1	Syntax	A-44
A.17.2	Restrictions	A-44
A.17.3	Parameter Semantics	A-45
A.17.4	Operational Semantics	A-46
A.17.5	Completion Codes	A-47
A.18	\$XPO_BACKUP - Preserve an Input File	A-48
A.18.1	Syntax	A-48
A.18.2	Parameter Semantics	A-48
A.18.3	Usage Guidelines	A-49
A.18.4	Completion Codes	A-49
A.18.5	Example	A-50
A.19	\$XPO_CLOSE - Close a File	A-51
A.19.1	Syntax	A-51
A.19.2	Parameter Semantics	A-51
A.19.3	Usage Guidelines	A-52
A.19.4	Completion Codes	A-52
A.20	\$XPO_DELETE - Delete a File	A-54
A.20.1	Syntax	A-54

A.20.2	Parameter Semantics	A-54
A.20.3	Completion Codes	A-55
A.21	\$XPO_DESCRIPTOR - Declare a Data Descriptor . .	A-57
A.21.1	Syntax	A-57
A.21.2	Restrictions	A-57
A.21.3	Parameter Semantics	A-57
A.22	\$XPO_DESC_INIT - Initialize a Data Descriptor .	A-59
A.22.1	Syntax	A-59
A.22.2	Parameter Semantics	A-59
A.22.3	Completion Code	A-60
A.23	\$XPO_FREE_MEM - Release a Memory Element . . .	A-61
A.23.1	Syntax	A-61
A.23.2	Restrictions	A-61
A.23.3	Parameter Semantics	A-61
A.23.4	Completion Codes	A-62
A.24	\$XPO_GET - Read From a File	A-63
A.24.1	Syntax	A-63
A.24.2	Parameter Semantics	A-63
A.24.3	Usage Guidelines	A-64
A.24.4	Completion Codes	A-65
A.25	\$XPO_GET_MEM - Allocate Dynamic Memory Element .	A-67
A.25.1	Syntax	A-67
A.25.2	Restrictions	A-67
A.25.3	Parameter Semantics	A-67
A.25.4	Completion Codes	A-68
A.26	\$XPO_IOB - Declare an IOB	A-70
A.26.1	Syntax	A-70
A.26.2	Parameter Semantics	A-70
A.26.3	Examples	A-70
A.27	\$XPO_IOB_INIT - Initialize an IOB	A-71
A.27.1	Syntax	A-71
A.27.2	Restrictions	A-71
A.27.3	Parameter Semantics	A-71
A.27.4	Completion Code	A-72
A.28	\$XPO_OPEN - Open a File	A-73
A.28.1	Syntax	A-73
A.28.2	Parameter Semantics	A-74
A.28.3	Completion Codes	A-77
A.29	\$XPO_PARSE_SPEC - Parse a File Specification . .	A-79
A.29.1	Syntax	A-79
A.29.2	Parameter Semantics	A-79
A.29.3	Completion Codes	A-80
A.30	\$XPO_PUT - Write to a File	A-81
A.30.1	Syntax	A-81
A.30.2	Restrictions	A-81
A.30.3	Parameter Semantics	A-82
A.30.4	Usage Guidelines	A-82
A.30.5	Completion Codes	A-82
A.31	\$XPO_PUT_MSG - Send a Message	A-84
A.31.1	Syntax	A-84
A.31.2	Parameter Semantics	A-84
A.31.3	Completion Codes	A-85
A.32	\$XPO_RENAME - Rename a File	A-86
A.32.1	Syntax	A-86
A.32.2	Parameter Semantics	A-87
A.32.3	Completion Codes	A-89

A.33	\$XPO_SPEC_BLOCK - Declare a File Specification	
	Block	A-90
A.33.1	Syntax	A-90
A.33.2	Examples	A-90
A.34	\$XPO_TERMINATE - Terminate Program Execution . .	A-91
A.34.1	Syntax	A-91
A.34.2	Parameter Semantics	A-91
A.34.3	Routine Value	A-91

APPENDIX A

MACRO DESCRIPTIONS

This appendix presents a detailed description of the macros related to input/output, memory management, target-system services, and string handling. It is intended for reference purposes. A tutorial discussion of these macros and their use is given in Chapters 3 through 6 of this manual.

A.1 DESCRIPTIVE NOTATION AND CONVENTIONS

A.1.1 Syntax Notation

The notational conventions used in the subsequent macro syntax definitions are the following:

<u>Notation</u>	<u>Meaning</u>
UPPERCASE TEXT	must be coded exactly as shown.
lowercase text	must be replaced by an appropriate user-chosen value.
required-parameter	denotes a macro parameter that must be specified or a compilation error will result.
primary-parameter	denotes a parameter that affects the nature of the requested operation, and which should be specified unless the corresponding XPORT IOB field has already been set up or the user is certain that the default (if any) is acceptable.

Macro Descriptions
DESCRIPTIVE NOTATION AND CONVENTIONS

optional-parameter	denotes a parameter that need not be specified in many cases. These parameters can be used (1) to provide optional information in a request or (2) to set up an XPORT IOB field for use in a later I/O operation.
nothing	indicates the null alternative, i.e., shows that the programmer may omit the associated item.
{ alternative1 } { alternative2 }	braces enclosing a vertical list of items indicate syntactic alternatives; one of the items must be selected. The braces are not coded.
{ ABC <u>DEF</u> value }	vertical bars within braces separate mutually exclusive alternatives; that is, only one of the items displayed in this manner can be selected for any given use of the macro in question. A default, if any, is underlined. The braces and the vertical bar are not coded.
{ parameter }	braces enclosing a single syntactic element indicate that the element is optional. The braces are not coded.
,...	indicates that the preceding item may be repeated. Commas must appear between the items. The periods are not coded.

Any other punctuation must be coded exactly as shown.

A.1.2 Character-String and Binary-Data Parameters

Many of the macros described in this appendix require character-string or binary-data information as parameters. Such parameters are indicated in the syntax diagrams as "char-string-info" or "binary-data-info".

Macro Descriptions DESCRIPTIVE NOTATION AND CONVENTIONS

The following list describes the forms in which character-string information can be specified:

<u>Syntax</u>	<u>Meaning</u>
address	The address of a string descriptor (see below)
'ascii text'	A literal ASCII text string, enclosed by apostrophes, i.e., a BLISS quoted-string
(count , pointer)	The number of characters in a string and a pointer to the string, enclosed in parentheses and separated by a comma.

The following list describes the forms in which binary-data information can be specified:

<u>Syntax</u>	<u>Meaning</u>
address	The address of a data descriptor (see below)
(size , address { , <u>FULLWORDS</u> , <u>UNITS</u> nothing })	A specification of the size of a collection of binary data, the address of the data, and an optional keyword that indicates the terms in which the size is expressed.

A.1.2.1 String and Data Descriptors

Many XPORT functions use a standard block structure called a descriptor to describe input or output character strings or binary data. (These transportable descriptors are patterned after corresponding VAX/VMS standard descriptors.)

Such descriptors can be created and initialized by the programmer through the use of the \$STR_DESCRIPTOR and \$STR_DESC_INIT macros, respectively, described in this appendix. A detailed discussion of descriptor usage appears in Section 6.1.

Macro Descriptions

\$STR_APPEND - Append a String

A.2 \$STR_APPEND - Append a String

The \$STR_APPEND macro calls the XPORT STRING facility to append a copy of a specified source string to a target string. The precise behavior of the operation depends to some extent on the class of the target string, which must be described by a descriptor. See Section A.5.4 for further detail.

A.2.1 Syntax

string-append	\$STR_APPEND(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING = char-string-info } { TARGET = address of a string descriptor }
optional-parameter	{ OPTION = TRUNCATE } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.2.2 Restrictions

A string descriptor, where specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

The target string must not be a FIXED string.

A.2.3 Parameter Semantics

STRING = char-string-info
describes the source character string to be appended to the target string. The source string may be of any class. This parameter must be specified.

Macro Descriptions
\$STR_APPEND - Append a String

TARGET = address of a descriptor
specifies the address of the target string descriptor. This parameter must be specified.

OPTION = TRUNCATE
indicates that (the copy of) the source string is to be truncated as necessary if the entire string cannot be accommodated within the target container string, in the case of a BOUNDED target string only. (In any other case this parameter is ignored.) If this parameter is not specified and the target string is BOUNDED, the append operation will fail and return a warning completion code (see below) if the source string is longer than the target remainder sub-string.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the requested operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the requested operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

A.2.4 Operational Semantics

The \$STR_APPEND operation extends an existing (possibly null) target string by adding a copy of the input string. The manner in which the target string is extended varies according to the class of the target string, as follows:

- o FIXED target string

The string-append operation is invalid for FIXED target strings. (By definition, FIXED strings cannot be extended.)

- o BOUNDED target string

The target prefix sub-string is preserved and the remainder sub-string is overwritten as necessary. If the source string is longer than the target remainder sub-string, the append operation either fails or truncates the source string to fit within the container string, as requested by the user.

Macro Descriptions
\$STR_APPEND - Append a String

o DYNAMIC target string

Dynamic memory is reallocated in order to accomodate the extended target string. If the source string is the null string, however, memory is not reallocated.

o DYNAMIC_BOUNDED target string

The target prefix sub-string is preserved and the remainder sub-string is overwritten as necessary. If the source string is longer than the target remainder sub-string, dynamic memory is reallocated in order to accomodate the extended target string.

In no case is the input "source string" or its descriptor modified in any way.

A detailed discussion of the descriptor/string classes (FIXED, DYNAMIC, BOUNDED, and DYNAMIC-BOUNDED) appears in Section 6.1.

A.2.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

STR\$_NORMAL	The string-append operation was successful.
--------------	---

Error Codes:

STR\$_BAD_SOURCE +	The source string is invalid.
STR\$_BAD_TARGET +	The target string is invalid.
XPO\$_FREE_MEM +	Dynamic memory deallocation error occurred.
XPO\$_GET_MEM +	Dynamic memory allocation error occurred.

Macro Descriptions

\$STR_ASCII - Binary-to-ASCII Conversion Pseudo-Function

A.3 \$STR_ASCII - Binary-to-ASCII Conversion Pseudo-Function

The \$STR_ASCII pseudo-function produces an ASCII string representation of a binary field value (fullword or smaller). This pseudo-function interprets its value argument as a signed or unsigned integer, as requested, and produces an ASCII string that expresses the value in a user-specified radix and format.

The "value" of a \$STR_ASCII pseudo-function is the address of a temporary string descriptor and can only be used as a string argument within other XPORT macro calls.

A.3.1 Syntax

binary-to-ASCII-conversion	\$STR_ASCII(value { ,parameter ,... })
parameter	{ integer-keyword { string-length-parameter } }
integer-keyword	{ BASE2 BASE8 BASE10 BASE16 } { SIGNED UNSIGNED } { LEADING_ZERO LEADING_BLANK }
string-length-parameter	LENGTH = length-expression

A.3.2 Restrictions

The keyword alternatives shown on a single line in the syntax diagram are mutually exclusive.

A.3.3 Parameter Semantics

value

is any primary expression that, after evaluation, gives a value to be converted and returned in string form. This parameter is required.

Macro Descriptions
\$STR_ASCII - Binary-to-ASCII Conversion Pseudo-Function

BASE2, BASE8, BASE10, or BASE16

indicates that the binary value is to be expressed as a binary, octal, decimal, or hexadecimal number, respectively, in the string representation. If this parameter is not specified, BASE10 is assumed.

SIGNED or UNSIGNED

indicates that the fullword result of the value-expression evaluation is to be interpreted as a signed or unsigned integer respectively. (Note that this parameter does not affect the mode of extension of any field value involved in the evaluation.) If this parameter is not specified, SIGNED is assumed if the result radix is BASE10; otherwise UNSIGNED is assumed.

LEADING_BLANKS or LEADING_ZEROS

indicates whether non-significant digits are to be represented by blanks or by zeros in the result string. If this parameter is not specified, LEADING_BLANKS is assumed if the result radix is BASE10; otherwise LEADING_ZEROS is assumed.

LENGTH = length-expression

specifies the length of the resulting string, i.e., number of character positions. If the specified string length is less than the number of significant characters in the converted value, the result string is set to all asterisks (à la FORTRAN). If this parameter is not specified, the minimum number of characters required to represent the significant portion of the binary value is assumed if the string format is LEADING_BLANKS; otherwise the number of characters required to represent all digits of the converted value is assumed.

A.3.4 Usage Guidelines

The \$STR_ASCII pseudo-function may be used in the following contexts:

- o As the value of a STRING= parameter or other similar parameter of most XPORT I/O, message, and string-processing macros
- o As the string argument of a \$STR_CONCAT or \$STR_FORMAT pseudo-function.

That is, it can be used as char-string-info except where specifically restricted. In general, \$STR_ASCII cannot be used in XPORT structure declaration or initialization macros.

Macro Descriptions

\$STR_BINARY - Convert ASCII to Binary

A.4 \$STR_BINARY - Convert ASCII to Binary

The \$STR_BINARY macro calls the XPORT STRING facility to convert an ASCII string value into a binary value.

The macro accepts a string argument and one or more string-interpretation options, and stores the converted result at a user-specified location.

A.4.1 Syntax

ascii-to-binary	\$STR_BINARY(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING = char-string-info } { RESULT = address-expression }
optional-parameter	{ OPTION = integer-radix-keyword } { RANGE = (min-value-exp, max-value-exp) } { SUCCESS = address of action routine } { FAILURE = address of action routine }
integer-radix-keyword	{ BASE2 BASE8 BASE10 BASE16 }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.4.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

Macro Descriptions
\$STR_BINARY - Convert ASCII to Binary

A.4.3 Parameter Semantics

STRING = char-string-info
describes the numeric character string to be converted to a binary integer value. This parameter must be specified.

RESULT = address-expression
specifies the address of the location that is to receive the converted value. The expression must denote a data segment that is large enough to contain any value within the specified or default value range for the conversion operation. This parameter must be specified.

OPTION = integer-radix-keyword
indicates that the character string is to be interpreted as a binary (BASE2), octal (BASE8), decimal (BASE10), or hexadecimal (BASE16) number. If this parameter is not specified, OPTION=BASE10 is assumed.

RANGE = (min-value-exp, max-value-exp)
specifies two inclusive limiting values for the result of an integer conversion operation. The converted value is tested against the values yielded by the minimum and maximum value expressions. If the result is out of range, the conversion operation fails. If this parameter is not specified, the numeric value represented by the ASCII string must lie within the value range of a fullword on the target system. That is, the value, i , must lie in the range

$$-(2^{**}(\%BPVAL-1)) \leq i \leq (2^{**}\%BPVAL)-1$$

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the requested operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the requested operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

A.4.4 Usage Guidelines

It is possible, in the RESULT parameter, to specify a result field that is not large enough to accomodate some of the values that fall within the range of values indicated by the RANGE parameter. In that case, the too-large value will be truncated when stored (i.e., high-order significance will be lost).

Macro Descriptions
\$STR_BINARY - Convert ASCII to Binary

It is the user's responsibility to specify a result field large enough to contain the largest value expected.

A.4.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

STR\$_NORMAL The ASCII-to-binary conversion was successful.

Error Codes:

STR\$_BAD_REQ + The XPORT request was invalid.

STR\$_BAD_SOURCE + The source string is invalid.

Macro Descriptions

\$STR_COMPARE - String Comparison

A.5 \$STR_COMPARE - String Comparison

The \$STR_COMPARE macro calls the XPORT STRING facility to compare two character strings; that is, to report whether the "value" of a given string is less than, equal to, or greater than the "value" of another given string, in the sense of their relative rank in the ASCII collating sequence.

A.5.1 Syntax

compare-strings	\$STR_COMPARE(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.5.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.5.3 Parameter Semantics

STRING1 = char-string-info

STRING2 = char-string-info

describe the character strings to be compared. The value of the string described by the STRING1 parameter is compared for a less-than, equal-to, or greater-than relationship to the value of the string described by the STRING2 parameter. These parameters must be specified.

Macro Descriptions
\$STR_COMPARE - String Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the requested operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the requested operation is not successful, due to an abnormal condition (see A.7.5). Note that, by its nature, an \$STR_COMPARE comparison cannot fail. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

A.5.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character.

No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

A.5.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Codes:

-1	String-1 compared less than string-2.
0	String-1 compared equal to string-2.
+1	String-1 compared greater than string-2.

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions

\$STR_CONCAT - String Concatenation Pseudo-Function

A.6 \$STR_CONCAT - String Concatenation Pseudo-Function

The \$STR_CONCAT pseudo-function accepts as arguments any combination of strings, and produces a single logical string consisting of the concatenation of these elements. \$STR_ASCII and \$STR_FORMAT pseudo-functions are also accepted as \$STR_CONCAT string arguments.

The "value" of a \$STR_CONCAT pseudo-function is the address of a temporary string descriptor and can only be used as a string argument within other XPORT macro calls.

A.6.1 Syntax

string- concatenation	\$STR_CONCAT(parameter , parameter ,... })
parameter	char-string-info
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.6.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.6.3 Parameter Semantics

Each parameter specifies an element of the logical concatenation to be produced as the "output string" of the pseudo-function. The elements are concatenated in the order of their specification.

Macro Descriptions
\$STR_CONCAT - String Concatenation Pseudo-Function

A.6.4 Usage Guidelines

The \$STR_CONCAT pseudo-function may be used in the following contexts:

- o As the value of a STRING= parameter or other similar parameter of most XPORT I/O, message, and string-processing macros
- o As the string argument of a \$STR_CONCAT or \$STR_FORMAT pseudo-function.

That is, it can be used as char-string-info except where specifically restricted. In general, \$STR_CONCAT cannot be used in XPORT structure declaration or initialization macros.

Macro Descriptions

\$STR_COPY - Copy a String

A.7 \$STR_COPY - Copy a String

The \$STR_COPY macro calls the XPORT STRING facility to copy a specified source string to a target string; that is, to replace the contents of the target string by that of the source string. The target string (possibly null) must be described by a descriptor.

A.7.1 Syntax

string-copy	\$STR_COPY(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING = char-string-info } { TARGET = target-string-info }
optional-parameter	{ OPTION = TRUNCATE } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }
target-string-info	{ address of a string descriptor } { (length , pointer) }

A.7.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.7.3 Parameter Semantics

STRING = char-string-info
describes the source character string to be copied to the target string. (The source string may be of any class.) This parameter must be specified.

Macro Descriptions
\$STR_COPY - Copy a String

TARGET = target-string-info
describes the target string. If the length/pointer notation is used, the target string is assumed to be FIXED. This parameter must be specified.

OPTION = TRUNCATE
indicates that (the copy of) the source string is to be truncated as necessary if the entire string cannot be accommodated within the target string, in the case of a FIXED target string, or if the entire string cannot be accommodated within the target container string, in the case of a BOUNDED target string. (In any other case this parameter is ignored.) If this parameter is not specified and the target string is either FIXED or BOUNDED, the copy operation will fail if the source string cannot be accommodated by the target string as described above.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the requested operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the requested operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

A.7.4 Operational Semantics

The semantics of the \$STR_COPY operation is as follows:

- o FIXED target string

Replace the contents of the target string with the source string. If the source string is smaller than the target string, fill the remainder of the target string with blanks. If the source string is longer than the target string, the copy operation either fails or truncates the source string to fit, as requested by the user.

- o DYNAMIC target string

If the source string is the same size as the target string, simply replace the target string value with the source string value. Otherwise, free the dynamic memory occupied by the target string, create a copy of the source string in dynamic memory, and update the STR\$H_LENGTH and STR\$A_POINTER fields of the target string descriptor.

Macro Descriptions
\$STR_COPY - Copy a String

o BOUNDED target string

If there is sufficient space in the target bounded and remainder strings, replace the bounded string value with the source string value and adjust the STR\$H_LENGTH field of the target string descriptor. If there is insufficient space in the target bounded and remainder strings, the copy operation either truncates the source string to fit the container string or returns an error completion code, as requested by the user. The prefix portion of the target container string is not modified by the copy operation.

o DYNAMIC_BOUNDED target string

If there is sufficient space in the target bounded and remainder strings, replace the bounded string value with the source string value and adjust the STR\$H_LENGTH field of the target string descriptor. If there is insufficient space in the target bounded and remainder strings, create a larger container string in dynamic memory, copy the prefix portion from the original container string and the source string into the new container string, free the original container string, and update the STR\$H_LENGTH, STR\$A_POINTER and STR\$H_MAXLEN fields of the target string descriptor.

A detailed discussion of the descriptor/string classes (FIXED, DYNAMIC, BOUNDED, and DYNAMIC-BOUNDED) appears in Section 6.xxxxx.

A.7.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

XPO\$_NORMAL	The string-copy operation was successful.
--------------	---

Error Codes:

STR\$_BAD_SOURCE +	The source string is invalid.
STR\$_BAD_TARGET +	The target string is invalid.
XPO\$_FREE_MEM +	Dynamic memory deallocation error occurred.
XPO\$_GET_MEM +	Dynamic memory allocation error occurred.

Macro Descriptions

\$STR_DESCRIPTOR - Declare a String Descriptor

A.8 \$STR_DESCRIPTOR - Declare a String Descriptor

The \$STR_DESCRIPTOR macro generates an attribute list for a string descriptor within an OWN, GLOBAL, LOCAL, MAP or BIND declaration. These attributes (1) indicate that the descriptor is a BLOCK structure of a given length, (2) specify the set of field-names that can be used to reference portions of the descriptor, and (3) specify initial descriptor-field values.

A detailed description of the four types of descriptors (fixed, dynamic, bounded, and dynamic-bounded) appears in Section 6.1.

A descriptor must be initialized before it can be used. If the descriptor is declared within an OWN or GLOBAL declaration, the descriptor can be statically initialized. Otherwise, it must be initialized by use of the \$STR_DESC_INIT macro.

A.8.1 Syntax

declare-a-string-descriptor	\$STR_DESCRIPTOR({optional-parameter ,...})
optional-parameter	{ CLASS = class-keyword } { STRING = char-string-info }
class-keyword	{ FIXED DYNAMIC } { BOUNDED DYNAMIC_BOUNDED }
char-string-info	{ 'literal ascii string' } { (count , pointer) }

A.8.2 Restrictions

The STRING parameter may only be specified within an OWN or GLOBAL declaration.

A.8.3 Parameter Semantics

CLASS = class-keyword
indicates the class of descriptor being declared. See Section 6.1 for a complete description of the descriptor classes. If this parameter is not specified, CLASS=FIXED is assumed.

Macro Descriptions
\$STR_DESCRIPTOR - Declare a String Descriptor

STRING = char-string-info
describes the character string to be described by the descriptor.
If this parameter is not specified, the descriptor is not
statically initialized.

Macro Descriptions

\$STR_DESC_INIT - Initialize a String Descriptor

A.9 \$STR_DESC_INIT - Initialize a String Descriptor

The \$STR_DESC_INIT macro dynamically initializes a string descriptor; that is, this macro generates the executable code necessary to initialize all of the fields of a given descriptor.

A.9.1 Syntax

setup-a-descriptor	\$STR_DESC_INIT(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	DESCRIPTOR = address of descriptor
optional parameter	{ CLASS = class-keyword } { STRING = char-string-info }
class-keyword	{ FIXED DYNAMIC } { BOUNDED DYNAMIC_BOUNDED }
char-string-info	{ address of a descriptor } { 'literal ascii string' } { (count , pointer) }

A.9.2 Restrictions

The STRING parameter may not be used when initializing a DYNAMIC or DYNAMIC_BOUNDED descriptor.

A.9.3 Parameter Semantics

DESCRIPTOR = address of descriptor
specifies the address of the descriptor to be initialized. This parameter must be specified.

CLASS = class-keyword
indicates the class of descriptor being initialized. See Section 6.1 for a complete description of the descriptor classes. If this parameter is not specified, CLASS=FIXED is assumed.

Macro Descriptions
\$STR_DESC_INIT - Initialize a String Descriptor

STRING = char-string-info
describes the character string to be described by the descriptor.
If this parameter is not specified, the corresponding descriptor
fields are not initialized.

A.9.4 Completion Code

Success Code:
XPOS_NORMAL The descriptor was successfully initialized.

Macro Descriptions

\$STR_EQL - String Equality Comparison

A.10 \$STR_EQL - String Equality Comparison

The \$STR_EQL macro calls the XPORT STRING facility to compare two character strings for equality; that is, to report whether or not two specified strings are identical in "value". Unequal length strings are not automatically adjusted for length.

A.10.1 Syntax

compare-for-equal	\$STR_EQL(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.10.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.10.3 Parameter Semantics

STRING1 = char-string-info

STRING2 = char-string-info

describe the character strings to be compared for equality.
These parameters must be specified.

Macro Descriptions
\$STR_EQL - String Equality Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the comparison operation. If this parameter is not specified, no success action routine is assumed.

NOTE: A comparison failure is considered to be a successful operation.

FAILURE = address of action routine
specifies the address of an action routine to be called if the comparison operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

NOTE: A \$STR_EQL operation will fail only if an input string or FILL value is invalid. An unsuccessful comparison does not constitute an operation failure.

A.10.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character.

No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

The comparison operation returns a value of 1 if the requested comparison is satisfied, and a value of 0 if the comparison is unsatisfied.

If the comparison operation itself fails (e.g., due to an invalid string descriptor), the operation returns an error completion code (which always has a low-bit value of 0), unless program execution is automatically terminated by a failure-action routine.

Macro Descriptions
\$STR_EQL - String Equality Comparison

A.10.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

1	The comparison was successful.
---	--------------------------------

Warning Code:

0	The comparison was not successful.
---	------------------------------------

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions

\$STR_FORMAT - String Formatting Pseudo-Function

A.11 \$STR_FORMAT - String Formatting Pseudo-Function

The \$STR_FORMAT pseudo-function accepts a string argument and produces a reformatted or otherwise transformed logical string as a result. \$STR_ASCII and \$STR_CONCAT pseudo-functions are also accepted as \$STR_FORMAT string arguments.

The "value" of a \$STR_FORMAT pseudo-function is the address of a temporary string descriptor and can only be used as a string argument within other XPORT macro calls.

A.11.1 Syntax

string-formatting	\$STR_FORMAT(string-param, format-param,...)
string-param	char-string-info
format-param	{ editing-keyword { string-length-parameter }
editing-keyword	{ LEFT_JUSTIFY CENTER RIGHT_JUSTIFY } { UP_CASE }
string-length-parameter	LENGTH = length-expression
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.11.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

The keyword alternatives shown on a single line in the syntax diagram are mutually exclusive.

Macro Descriptions
\$STR_FORMAT - String Formatting Pseudo-Function

A.11.3 Parameter Semantics

string-param
specifies the string that is the pseudo-function's "input string".

LEFT_JUSTIFY, CENTER, or RIGHT_JUSTIFY
indicates a string positioning option, as follows:

- o LEFT_JUSTIFY indicates that the first non-blank character of the "input string" is to begin in the first character position of the logical output string. Any unused trailing character positions are blank-filled.
- o CENTER indicates that the non-blank portion of the "input string" is to be centered in the logical output string, with unused leading and trailing character positions blank-filled.
- o RIGHT_JUSTIFY indicates that the rightmost non-blank character of the "input string" is to appear in the last character position of the logical output string. Any unused leading character positions are blank-filled.

If a positioning option is to be meaningful, the result string must be longer than the input string (see LENGTH=). If a positioning option is not specified, LEFT_JUSTIFY is assumed.

UP_CASE
indicates that all alphabetic characters in the "input string" are to appear as uppercase in the logical output string. If this option is not specified, no case conversion is performed.

LENGTH = length-expression
specifies the length of the logical output string, i.e., the number of character positions to be produced in the result. If the specified output-string length is less than the input-string length, the output string is set to all asterisks (à la FORTRAN). If this parameter is not specified, the output string is assumed to be the same length as the input string.

A.11.4 Usage Guidelines

The \$STR_FORMAT pseudo-function may be used in the following contexts:

- o As the value of a STRING= parameter or other similar parameter of most XPORT I/O, message, and string-processing macros

Macro Descriptions
\$STR_FORMAT - String Formatting Pseudo-Function

- o As the string argument of a \$STR_CONCAT pseudo-function.

That is, it can be used as char-string-info except where specifically restricted. In general, \$STR_FORMAT cannot be used in XPORT structure declaration or initialization macros.

Macro Descriptions

\$STR_GEQ - String Greater-Than-or-Equal Comparison

A.12 \$STR_GEQ - String Greater-Than-or-Equal Comparison

The \$STR_GEQ macro calls the XPORT STRING facility to compare two character strings for a greater-than-or-equal relationship; that is, to report whether or not the "value" of a given string is either greater than or equal to the "value" of another given string, in the sense of their relative rank in the ASCII collating sequence. Unequal length strings are not automatically adjusted for length.

A.12.1 Syntax

compare-for-greater-or-equal	\$STR_GEQ(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.12.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.12.3 Parameter Semantics

STRING1 = char-string-info

STRING2 = char-string-info

describe the character strings to be compared. If the value of the string described by the STRING1 parameter is greater than or equal to the value of string described by the STRING2 parameter, the comparison is satisfied; otherwise the comparison is not satisfied. These parameters must be specified.

Macro Descriptions
\$STR_GEQ - String Greater-Than-or-Equal Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the comparison operation. If this parameter is not specified, no success action routine is assumed.

NOTE: A comparison failure is considered to be a successful operation.

FAILURE = address of action routine
specifies the address of an action routine to be called if the comparison operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

NOTE: A \$STR_GEQ operation will fail only if an input string or FILL value is invalid. An unsuccessful comparison does not constitute an operation failure.

A.12.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character.

No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

The comparison operation returns a value of 1 if the requested comparison is satisfied, and a value of 0 if the comparison is unsatisfied.

If the comparison operation itself fails (e.g., due to an invalid string descriptor), the operation returns an error completion code (which always has a low-bit value of 0), unless program execution is automatically terminated by a failure-action routine.

Macro Descriptions
\$STR_GEQ - String Greater-Than-or-Equal Comparison

A.12.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

1	The comparison was successful.
---	--------------------------------

Warning Code:

0	The comparison was not successful.
---	------------------------------------

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions

\$STR_GTR - String Greater-Than Comparison

A.13 \$STR_GTR - String Greater-Than Comparison

The \$STR_GTR macro calls the XPORT STRING facility to compare two character strings for a greater-than relationship; that is, to report whether or not the "value" of a given string is greater than the "value" of another given string, in the sense of their relative rank in the ASCII collating sequence. Unequal length strings are not automatically adjusted for length.

A.13.1 Syntax

compare-for-greater-than	\$STR_GTR(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.13.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.13.3 Parameter Semantics

STRING1 = char-string-info

STRING2 = char-string-info

describe the character strings to be compared. If the value of the string described by the STRING1 parameter is greater than the value of string described by the STRING2 parameter, the comparison is satisfied; otherwise the comparison is unsatisfied. These parameters must be specified.

Macro Descriptions
\$STR_GTR - String Greater-Than Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the comparison operation. If this parameter is not specified, no success action routine is assumed.

NOTE: A comparison failure is considered to be a successful operation.

FAILURE = address of action routine
specifies the address of an action routine to be called if the comparison operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

NOTE: A \$STR_GTR operation will fail only if an input string or FILL value is invalid. An unsuccessful comparison does not constitute an operation failure.

A.13.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character.

No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

The comparison operation returns a value of 1 if the requested comparison is satisfied, and a value of 0 if the comparison is unsatisfied.

If the comparison operation itself fails (e.g., due to an invalid string descriptor), the operation returns an error completion code (which always has a low-bit value of 0), unless program execution is automatically terminated by a failure-action routine.

Macro Descriptions
\$STR_GTR - String Greater-Than Comparison

A.13.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

1	The comparison was successful.
---	--------------------------------

Warning Code:

0	The comparison was not successful.
---	------------------------------------

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions

\$STR_LEQ - String Less-Than-or-Equal Comparison

A.14 \$STR_LEQ - String Less-Than-or-Equal Comparison

The \$STR_LEQ macro calls the XPORT STRING facility to compare two character strings for a less-than-or-equal relationship; that is, to report whether or not the "value" of a given string is less than or equal to the "value" of another given string, in the sense of their relative rank in the ASCII collating sequence. Unequal length strings are not automatically adjusted for length.

A.14.1 Syntax

compare-for-less-than-or-equal	\$STR_LEQ(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.14.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.14.3 Parameter Semantics

STRING1 = char-string-info

STRING2 = char-string-info

describe the character strings to be compared. If the value of the string described by the STRING1 parameter is less than or equal to the value of string described by the STRING2 parameter, the comparison is satisfied; otherwise the comparison is unsatisfied. These parameters must be specified.

Macro Descriptions
\$STR_LEQ - String Less-Than-or-Equal Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the comparison operation. If this parameter is not specified, no success action routine is assumed.

NOTE: A comparison failure is considered to be a successful operation.

FAILURE = address of action routine
specifies the address of an action routine to be called if the comparison operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

NOTE: A \$STR_LEQ operation will fail only if an input string or FILL value is invalid. An unsuccessful comparison does not constitute an operation failure.

A.14.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character. No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

The comparison operation returns a value of 1 if the requested comparison is satisfied, and a value of 0 if the comparison is unsatisfied.

If the comparison operation itself fails (e.g., due to an invalid string descriptor), the operation returns an error completion code (which always has a low-bit value of 0), unless program execution is automatically terminated by a failure-action routine.

Macro Descriptions
\$STR_LEQ - String Less-Than-or-Equal Comparison

A.14.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

1	The comparison was successful.
---	--------------------------------

Warning Code:

0	The comparison was not successful.
---	------------------------------------

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions

\$STR_LSS - String Less-Than Comparison

A.15 \$STR_LSS - String Less-Than Comparison

The \$STR_LSS macro calls the XPORT STRING facility to compare two character strings for a less-than relationship; that is, to report whether or not the "value" of a given string is less than the "value" of another given string, in the sense of their relative rank in the ASCII collating sequence. Unequal length strings are not automatically adjusted for length.

A.15.1 Syntax

compare-for-less-than	\$STR_LSS(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.15.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.15.3 Parameter Semantics

STRING1 = char-string-info
STRING2 = char-string-info

describe the character strings to be compared. If the value of the string described by the STRING1 parameter is less than the value of string described by the STRING2 parameter, the comparison is satisfied; otherwise the comparison is unsatisfied. These parameters must be specified.

Macro Descriptions
\$STR_LSS - String Less-Than Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the comparison operation. If this parameter is not specified, no success action routine is assumed.

NOTE: A comparison failure is considered to be a successful operation.

FAILURE = address of action routine
specifies the address of an action routine to be called if the comparison operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

NOTE: A \$STR_LSS operation will fail only if an input string or FILL value is invalid. An unsuccessful comparison does not constitute an operation failure.

A.15.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character. No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

The comparison operation returns a value of 1 if the requested comparison is satisfied, and a value of 0 if the comparison is unsatisfied.

If the comparison operation itself fails (e.g., due to an invalid string descriptor), the operation returns an error completion code (which always has a low-bit value of 0), unless program execution is automatically terminated by a failure-action routine.

Macro Descriptions
\$STR_LSS - String Less-Than Comparison

A.15.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

1	The comparison was successful.
---	--------------------------------

Warning Code:

0	The comparison was not successful.
---	------------------------------------

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions

\$STR_NEQ - String Inequality Comparison

A.16 \$STR_NEQ - String Inequality Comparison

The \$STR_NEQ macro calls the XPORT STRING facility to compare two character strings for a not-equal relationship; that is, to report whether or not the "values" of two given string are not equal to each other. Unequal length strings are not automatically adjusted for length.

A.16.1 Syntax

compare-for-not-equal	\$STR_NEQ(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING1 = char-string-info } { STRING2 = char-string-info }
optional-parameter	{ FILL = char-value-expression } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.16.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

A.16.3 Parameter Semantics

STRING1 = char-string-info

STRING2 = char-string-info

describe the character strings to be compared for inequality. If the value of the two strings are not equal, the comparison is satisfied; otherwise the comparison is unsatisfied. These parameters must be specified.

Macro Descriptions
\$STR_NEQ - String Inequality Comparison

FILL = char-value-expression
specifies an ASCII character-code value to be used to (logically) fill out the shorter of the two strings if they are of unequal length, prior to the comparison. The expression value must be between 0 and 127 (decimal), inclusive. If this parameter is not specified, no string extension is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the comparison operation. If this parameter is not specified, no success action routine is assumed.

NOTE: A comparison failure is considered to be a successful operation.

FAILURE = address of action routine
specifies the address of an action routine to be called if the comparison operation is not successful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

NOTE: A \$STR_NEQ operation will fail only if an input string or FILL value is invalid. An unsuccessful comparison does not constitute an operation failure.

A.16.4 Operational Semantics

The notion of "value" in the string-handling context includes both the length and the content, if any, of a string. The value of the null string, for example, having a length of zero, is not the same as the value of a string consisting of a NUL character. No automatic adjustment for strings of unequal length is performed. Optionally, however, the shorter of two unequal-length strings can be logically padded "on the right" with a specified fill character.

The comparison operation returns a value of 1 if the requested comparison is satisfied, and a value of 0 if the comparison is unsatisfied.

If the comparison operation itself fails (e.g., due to an invalid string descriptor), the operation returns an error completion code (which always has a low-bit value of 0), unless program execution is automatically terminated by a failure-action routine.

Macro Descriptions
\$STR_NEQ - String Inequality Comparison

A.16.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

1	The comparison was successful.
---	--------------------------------

Warning Code:

0	The comparison was not successful.
---	------------------------------------

Error Codes:

STR\$_BAD_STRNG1 +	The primary string is invalid.
STR\$_BAD_STRNG2 +	The secondary string is invalid.

Macro Descriptions
\$STR_SCAN - String Scanning

A.17 \$STR_SCAN - String Scanning

The \$STR_SCAN macro calls the XPORT STRING facility to scan a string in order to determine a given portion, or substring, of that string. The substring to be determined can be specified by different types of pattern strings.

A.17.1 Syntax

scan-string	\$STR_SCAN(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ source-string } { pattern-string }
source-string	{ STRING = char-string-info } { REMAINDER = address-of-a-descriptor }
pattern-string	{ FIND = char-string-info } { SPAN = char-string-info } { STOP = char-string-info }
optional-parameter	{ result-parameter } { DELIMITER = address } { SUCCESS = address of action routine } { FAILURE = address of action routine }
result-parameter	{ SUBSTRING = descriptor address } { TARGET = descriptor address }
char-string-info	{ address of a string descriptor } { 'literal ascii string' } { (length , pointer) } { string conversion pseudo-function }

A.17.2 Restrictions

A string descriptor, if specified, must describe a standard XPORT string; that is, the data type must be STR\$K_DTYPE_T, and the descriptor class must be STR\$K_CLASS_F, _D, _B, or _DB.

The char-string-info of the STRING= parameter must not be a string conversion pseudo-function if the SUBSTRING= parameter is specified.

Macro Descriptions
\$STR_SCAN - String Scanning

If REMAINDER is specified, the source-string descriptor class must be BOUNDED or DYNAMIC_BOUNDED only.

The STRING= and SUBSTRING= parameters must not point to the same descriptor.

The STRING= and TARGET= parameters must not point to the same descriptor.

If SUBSTRING= is specified, the substring descriptor class must be FIXED or BOUNDED only.

The pattern string must not be the null string (i.e., a zero-length string.)

A.17.3 Parameter Semantics

STRING = char-string-info
describes the character string to be scanned. The precise semantics of the scan operation is determined by the pattern-string parameter employed. Either this parameter or the REMAINDER parameter must be specified.

REMAINDER = descriptor address
specifies the address of the descriptor of a bounded or dynamic-bounded string of which the remainder portion is to be scanned. The precise semantics of the scan operation is determined by the type of pattern-string parameter employed. Either this parameter or the STRING parameter must be specified.

FIND = char-string-info
specifies a particular sequence of characters to be located within the source string. If the specified character sequence is found anywhere in the string the scan operation is successful. Otherwise, the operation is not successful. Either a FIND, SPAN, or STOP parameter must be specified.

SPAN = char-string-info
specifies the list of characters that may occur in the substring to be located within the source string. The semantics of the SPAN-type scan operation is given in Section A.17.4. Either a SPAN, FIND, or STOP parameter must be specified.

STOP = char-string-info
specifies the list of characters that may not occur in the substring to be located within the source string. The semantics of the STOP-type scan operation is given in Section A.17.4. Either a STOP, FIND, or SPAN parameter must be specified.

Macro Descriptions
\$STR_SCAN - String Scanning

DELIMITER = address

specifies a location in which to store the character that "stopped" the scan operation; that is, the character immediately following the substring determined by the operation. If the string scan operation fails, the delimiter location is unchanged. If the substring extends to the end of the source string, a NUL character is returned. If this parameter is not specified, the delimiter character is not reported.

SUBSTRING = descriptor address

specifies a descriptor that is to be modified to describe the substring determined by the scan operation. If the string scan operation fails, the substring descriptor is not changed. If this parameter is not specified, substring information is not reported. The use of the SUBSTRING parameter is equivalent to the use of the \$STR_DESC_INIT macro.

TARGET = descriptor address

specifies the descriptor that is to describe a copy of the substring obtained by the scan operation. If the string scan operation fails, the target descriptor is not changed. If this parameter is not specified, no copy of the substring is provided. The use of the TARGET parameter is equivalent to the use of the \$STR_COPY macro.

SUCCESS = address of action routine

specifies the address of an action routine to be called upon successful completion of the scan operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine

specifies the address of an action routine to be called if the scan operation is not successful due to an abnormal condition (see Section A.17.5). Note that an unsuccessful scan does not trigger a failure-action routine. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=STR\$FAILURE is assumed (see Appendix E).

A.17.4 Operational Semantics

If the pattern-string parameter keyword is FIND, the scan operation attempts to find a match for the specified pattern anywhere in the source string, beginning with the first character position of the string. If such a match is found, the scan operation returns a low-bit value of 1 (STR\$_NORMAL, or STR\$_END_STRING if the scan ends at end of line). If no match is found, the operation returns a value of 0.

Macro Descriptions

\$STR_SCAN - String Scanning

If the pattern-string parameter keyword is SPAN, the scan operation determines the longest substring that (1) begins at the first character position of the string and (2) consists of any one or more of the characters specified in the pattern string and only of those characters. The SPAN-type scan operation may validly determine a substring of zero length, that is, the null string. This scan operation always returns a low-bit value of 1 (STR\$ NORMAL, or STR\$ END_STRING if the scan ends at end of line), unless the operation itself fails due to an abnormal condition (see Section A.17.5).

If the pattern-string parameter keyword is STOP, the scan operation determines the longest substring that (1) begins at the first character position of the string and (2) does not contain any of the characters specified in the pattern string. The STOP-type scan operation may validly determine a substring of zero length, that is, the null string. This scan operation always returns a low-bit value of 1 (STR\$ NORMAL, or STR\$ END_STRING if the scan ends at end of line), unless the operation itself fails due to an abnormal condition (see Section A.17.5).

If the same BOUNDED or DYNAMIC_BOUNDED descriptor is specified as both a REMAINDER= and a SUBSTRING= parameter, the typical operation of "scan thru a string" can be achieved.

A.17.5 Completion Codes

A primary completion code is returned as the routine-call value. A secondary completion code (if any) is available only in a failure-action routine associated with the function. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the corresponding primary code in the listing below.

Success Code:

STR\$ NORMAL	The scan operation was successful.
STR\$ END_STRING	The scan operation was successful, and the SPAN-type operation ended at end-of-line.

Warning Code:

STR\$ FAILURE	The scan operation was unsuccessful.
---------------	--------------------------------------

Error Codes:

STR\$ BAD_PATTRN +	The pattern string is invalid.
STR\$ BAD_SOURCE +	The source string is invalid.
STR\$ BAD_TARGET +	The target string is invalid.

Macro Descriptions

\$XPO_BACKUP - Preserve an Input File

A.18 \$XPO_BACKUP - Preserve an Input File

The \$XPO_BACKUP macro calls the XPORT I/O facility to perform a "file backup" operation in a transportable manner. This capability is intended for applications which create an output file with the same name as the input file from which it was created (e.g., an editor).

If the host operating-system supports multiple versions of a file (e.g., VAX/VMS, RSX-11M), the output file is renamed to be the same as the input file except that its version number is one greater than the highest existing version. The input file is not renamed.

If the host system does not support multiple versions of a file (e.g., TOPS-10, TOPS-20, RT-11), first, a previous backup file, if any, is deleted. The input file is then renamed by changing the file extension/type. Finally, the output file is then renamed to be the same as the original name of the input file.

A.18.1 Syntax

preserve a file	\$XPO_BACKUP(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ OLD_IOB = address of iob } { NEW_IOB = address of iob }
optional-parameter	{ FILE_TYPE = char-string-info } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character-string descriptor } { 'literal ascii string' } { (count , pointer) }

A.18.2 Parameter Semantics

OLD_IOB = address of IOB
 specifies the address of the IOB that describes the input file. This file must be in a closed state, and must have been closed with the REMEMBER option. The resultant-file-specification field of this IOB is cleared on completion of the backup operation. This parameter must be specified.

Macro Descriptions
\$XPO_BACKUP - Preserve an Input File

NEW_IOB = address of IOB
specifies the address of the IOB that describes the output file. This file must be in a closed state, and must have been closed with the REMEMBER option. The resultant-file-specification field of this IOB is cleared on completion of the backup operation. This parameter must be specified.

FILE_TYPE = character-string-info
describes a file type, or extension, to be used for input-file renaming if the input file cannot be preserved using version numbers. The file-type string must include the 'dot' (.) and may not contain any other parts of a file specification. If this parameter is not specified, FILE_TYPE='.BAK' is assumed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the backup operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the backup operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.18.3 Usage Guidelines

The backup operation requires a valid resultant-file-specification field in both IOBs. This requirement can be satisfied by using the REMEMBER option of the CLOSE operation (see A.19). Note that REMEMBER is implied for a file specified as \$XPO_TEMPORARY.

The backup operation is invalid for files that have never been opened.

A.18.4 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G_COMP_CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G_2ND_CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Macro Descriptions
\$XPO_BACKUP - Preserve an Input File

Success Code:

XPO\$_NORMAL The file backup was successful.

Error Codes:

XPO\$_BACKUP + File could not be backed up.
XPO\$_BAD_RSLT + The old or new file-spec is invalid.
XPO\$_BAD_TYPE The backup file-type is invalid.
XPO\$_DELETE + Error deleting a previous backup copy.
XPO\$_NOT_CLOSED One of the IOBs was not closed.
XPO\$_RENAME_NEW + Error renaming the output file.
XPO\$_RENAME_OLD + Error renaming the input file.

Fatal Error Codes:

XPO\$_BAD_IOB + One of the IOBs is invalid.
XPO\$_BAD_LOGIC An XPORT logic error was detected.

A.18.5 Example

The following BLISS coding example illustrates the sequence of operations which should be followed in using \$XPO_BACKUP.

```
$XPO_OPEN( IOB = input-iob, FILE_SPEC = input-spec, ... );  
$XPO_OPEN( IOB = output-iob, FILE_SPEC = $XPO_TEMPORARY, ... );  
  
!            Process information from the input file, and  
!            write it to the (temporary) output file.  
  
$XPO_CLOSE( IOB = input-iob, OPTIONS = REMEMBER );  
$XPO_CLOSE( IOB = output-iob );  
  
$XPO_BACKUP( OLD_IOB = input-iob, NEW_IOB = output-iob );
```

Macro Descriptions

\$XPO_CLOSE - Close a File

A.19 \$XPO_CLOSE - Close a File

The \$XPO_CLOSE macro calls the XPORT I/O facility to terminate the processing of an input or output file and flush any internal XPORT I/O buffers. If a program terminates without closing a file, the resulting state of the file is unpredictable.

A.19.1 Syntax

close a file	\$XPO_CLOSE(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	IOB = address of iob
optional-parameter	{ OPTIONS = option-keyword } { USER = user-defined value } { SUCCESS = address of action routine } { FAILURE = address of action routine }
option-keyword	REMEMBER

NOTE: The keyword OPTIONS may be shortened to OPTION.

A.19.2 Parameter Semantics

IOB = address of IOB

specifies the address of the IOB that describes the file to be closed. This parameter must be specified.

OPTIONS = option-keyword

indicates a processing option to be applied to the file being closed.

<u>Option</u>	<u>Meaning</u>
REMEMBER	Remember relevant file attributes (e.g., resultant file specification) so that the file can be reprocessed (e.g., reopened, renamed, deleted, backed up). This option is assumed by default when a temporary file is closed (see Section 3.5).

Macro Descriptions
\$XPO_CLOSE - Close a File

File processing options (including this option) are not remembered, whether or not this option is specified. If this option is not specified by file close time, the IOB is reset to an initialized state after a successful file close.

If this parameter is not specified, the IOB option field is not changed.

USER = user-defined value
specifies an application-dependent fullword value to be placed in the IOB field IOB\$Z_USER. If this parameter is not specified, the IOB user field is not changed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the CLOSE operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the CLOSE operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.19.3 Usage Guidelines

If a file is closed with the REMEMBER option, a subsequent OPEN, DELETE, or RENAME operation will not perform file-specification resolution for the file, but will instead use the 'remembered' resultant-file-specification string described in the IOB.

A.19.4 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G_COMP_CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G_2ND_CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Success Code:
XPO\$_NORMAL The file was successfully closed.

Macro Descriptions
\$XPO_CLOSE - Close a File

Error Codes:

XPO\$_CLOSED	The file has already been closed.
XPO\$_FILE_LOCK	JFN is locked; file cannot be closed.
XPO\$_FREE_MEM +	Error freeing IOB-related memory.
XPO\$_IN_USE	The file is currently in use.
XPO\$_IO_ERROR +	A hardware-level I/O error occurred.
XPO\$_NETWORK +	A network error has occurred.
XPO\$_NO_ACCESS +	The file cannot be accessed.
XPO\$_NO_CLOSE	The file cannot be closed by this process.
XPO\$_NO_MEMORY	Insufficient dynamic memory space.
XPO\$_NO_SPACE	Quota exceeded or disk full.
XPO\$_NO_SUPPORT +	The requested function is not supported.
XPO\$_NO_WRITE	The file is write-protected.
XPO\$_NOT_EXPIRE	The file-expiration date is not past.
XPO\$_NOT_ONLINE	The device was not ready.
XPO\$_NOT_OPEN	The file was not open.

Fatal Error Codes:

XPO\$_BAD_IOB +	The IOB is invalid.
XPO\$_BAD_LOGIC	An XPORT logic error was detected.

Macro Descriptions

\$XPO_DELETE - Delete a File

A.20 \$XPO_DELETE - Delete a File

The \$XPO_DELETE macro calls the XPORT I/O facility to delete an existing file. This operation, like OPEN and RENAME, performs file-specification resolution as necessary.

A.20.1 Syntax

delete a file	\$XPO_DELETE(parameter ,...)
parameter	{ required-parameter } { primary-parameter } { optional-parameter }
required-parameter	IOB = address of iob
primary-parameter	FILE_SPEC = char-string-info
optional-parameter	{ DEFAULT = char-string-info } { RELATED = char-string-info } { USER = user-defined value } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character-string descriptor } { 'literal ascii string' } { (count , pointer) }

A.20.2 Parameter Semantics

IOB = address of IOB

specifies the address of an IOB for the file to be deleted. This IOB must be initialized, but it must not be open when the DELETE call is made. This parameter must be specified.

FILE_SPEC = character-string-info

describes the file specification given by the end user. Unless the IOB was previously closed with the REMEMBER option, this file specification is combined with the default and related file specifications, if any, to form the resultant file specification (see Section 3.6.1). If this parameter is not specified, the corresponding IOB fields are not changed.

Macro Descriptions
\$XPO_DELETE - Delete a File

DEFAULT = character-string-info
describes a default file specification. During file-specification resolution, this file specification is combined with the user and related file specifications, if any, to form the resultant file specification (see Section 3.6.1). If this parameter is not specified, the corresponding IOB fields are not changed.

RELATED = character-string-info
describes a file specification that is related to the file being deleted. During file-specification resolution, this file specification is combined with the user and default file specifications, if any, to form the resultant file specification (see Section 3.6.1). If this parameter is not specified, the corresponding IOB fields are not changed.

USER = user-defined value
specifies an application-dependent fullword value to be placed in the IOB field IOB\$Z_USER. If this parameter is not specified, the IOB user field is not changed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the delete operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the delete operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.20.3 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G_COMP_CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G_2ND_CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Success Code:
XPO\$_NORMAL The file was successfully deleted.

Macro Descriptions

\$XPO_DELETE - Delete a File

Error Codes:

XPO\$ _BAD _ACCT	Invalid account field.
XPO\$ _BAD _ATTR	Invalid attribute field in file spec.
XPO\$ _BAD _DELIM	Invalid punctuation used in a quoted string.
XPO\$ _BAD _DEVICE	A nonexistent device was specified.
XPO\$ _BAD _DFLT +	The default file specification is invalid.
XPO\$ _BAD _DIRECT	Directory-access privilege required, or invalid directory format.
XPO\$ _BAD _NAME	No filename was specified.
XPO\$ _BAD _PROT	Invalid protection field.
XPO\$ _BAD _REQ +	The XPORT request was invalid.
XPO\$ _BAD _RLTD +	The related file specification is invalid.
XPO\$ _BAD _RSLT +	The resultant file specification is invalid.
XPO\$ _BAD _SPEC +	The user file specification is invalid.
XPO\$ _BAD _TEMP	Multiple ";"s specified.
XPO\$ _BAD _VER	Generation number not numeric.
XPO\$ _CHANNEL +	A channel-assignment error occurred.
XPO\$ _CORRUPTED	Invalid FDB format, or FDB not found.
XPO\$ _FREE MEM +	Error freeing IOB-related memory.
XPO\$ _GET MEM +	A memory-allocation error occurred.
XPO\$ _IN _USE	The file is currently in use.
XPO\$ _IO _ERROR +	A hardware-level I/O error occurred.
XPO\$ _NETWORK +	A network error has occurred.
XPO\$ _NO _ACCESS +	The file cannot be accessed.
XPO\$ _NO _CHANNEL	No I/O channel was available.
XPO\$ _NO _CONCAT	Concatenated file-spec not allowed.
XPO\$ _NO _DELETE	The file cannot be deleted.
XPO\$ _NO _DIRECT	The indicated directory was not found.
XPO\$ _NO _FILE	The file does not exist.
XPO\$ _NO _SPACE	Disk working space is exhausted.
XPO\$ _NO _SUBDIR	The sub-directory does not exist.
XPO\$ _NO _SUPPORT +	The requested function is not supported.
XPO\$ _NO _WRITE	The file is write-protected.
XPO\$ _NOT _EXPIRE	The file-expiration date is not past.
XPO\$ _NOT _ONLINE	The device was not ready.
XPO\$ _OPEN	The file is currently open.
XPO\$ _PROTECTED	Access to the file is denied.

Fatal Error Codes:

XPO\$ _BAD _IOB +	The IOB is invalid.
XPO\$ _BAD _LOGIC	An XPORT logic error was detected

Macro Descriptions

\$XPO_DESCRIPTOR - Declare a Data Descriptor

A.21 \$XPO_DESCRIPTOR - Declare a Data Descriptor

The \$XPO_DESCRIPTOR macro generates an attribute list for a binary data descriptor within an OWN, GLOBAL, LOCAL, MAP or BIND declaration. These attributes (1) indicate that the descriptor is a BLOCK structure of a given length, (2) specify the set of field-names that can be used to reference portions of the descriptor, and (3) specify initial descriptor-field values.

A detailed description of the four types of descriptors (fixed, dynamic, bounded, and dynamic-bounded) appears in Section 7.1.

A descriptor must be initialized before it can be used. If the descriptor is declared within an OWN or GLOBAL declaration, the descriptor can be statically initialized. Otherwise, it must be initialized by use of the \$XPO_DESC_INIT macro.

A.21.1 Syntax

declare-a-data-descriptor	\$XPO_DESCRIPTOR({optional-parameter ,...})
optional-parameter	{ CLASS = class-keyword } { BINARY_DATA = binary-data-info }
class-keyword	{ FIXED DYNAMIC } { BOUNDED DYNAMIC_BOUNDED }
binary-data-info	(size , address { , <u>FULLWORDS</u> } { , <u>UNITS</u> }) { nothing })

A.21.2 Restrictions

The BINARY_DATA parameter may only be specified within an OWN or GLOBAL declaration.

A.21.3 Parameter Semantics

CLASS = class-keyword
indicates the class of descriptor being declared. See Section 7.1 for a complete description of the descriptor classes. If this parameter is not specified, CLASS=FIXED is assumed.

Macro Descriptions
\$XPO_DESCRIPTOR - Declare a Data Descriptor

BINARY_DATA = binary-data-info
describes the binary data item to be described by the descriptor.
If this parameter is not specified, the size and length fields of
the descriptor are not statically initialized.

Macro Descriptions

\$XPO_DESC_INIT - Initialize a Data Descriptor

A.22 \$XPO_DESC_INIT - Initialize a Data Descriptor

The \$XPO_DESC_INIT macro dynamically initializes a binary data descriptor; that is, this macro generates the executable code necessary to initialize all of the fields of a given descriptor.

A.22.1 Syntax

setup-a-data-descriptor	\$XPO_DESC_INIT(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	DESCRIPTOR = address of descriptor
optional parameter	{ CLASS = class-keyword } { BINARY_DATA = binary-data-info }
class-keyword	{ FIXED DYNAMIC } { BOUNDED DYNAMIC_BOUNDED }
binary-data-info	{ address of binary data descriptor } { (size , address { , FULLWORDS }) } { (size , address { , UNITS }) } { nothing }

A.22.2 Parameter Semantics

DESCRIPTOR = address of descriptor
specifies the address of the descriptor to be initialized. This parameter must be specified.

CLASS = class-keyword
indicates the class of descriptor being initialized. See Section 7.1 for a complete description of the descriptor classes. If this parameter is not specified, CLASS=FIXED is assumed.

BINARY_DATA = binary-data-info
describes the binary data item to be described by the descriptor. If this parameter is not specified, the corresponding descriptor fields are not initialized.

Macro Descriptions
\$XPO_DESC_INIT - Initialize a Data Descriptor

A.22.3 Completion Code

Success Code:

XPO\$_NORMAL	The descriptor was successfully initialized.
--------------	--

Macro Descriptions

\$XPO_FREE_MEM - Release a Memory Element

A.23 \$XPO_FREE_MEM - Release a Memory Element

The \$XPO_FREE_MEM macro calls the XPORT MEMORY facility to release a previously allocated element of memory. The memory element may optionally be cleared before it is released.

A.23.1 Syntax

release memory	\$XPO_FREE_MEM(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ STRING = char-string-info } { BINARY_DATA = binary-data-info }
optional-parameter	{ FILL = memory clearing value } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character string descriptor } { (count , pointer) }
binary-data-info	{ address of binary data descriptor } { (size , address { , <u>FULLWORDS</u> }) } { (size , address { , <u>UNITS</u> }) } { (size , address { , nothing }) }

A.23.2 Restrictions

The STRING and BINARY_DATA parameters are mutually exclusive. As indicated in the syntax diagram, a literal-string is not valid as a character-string-info parameter in this macro.

A.23.3 Parameter Semantics

STRING = char-string-info

describes a character-string memory element to be released. The element described must begin on a BLISS fullword boundary; its length is rounded up, if necessary, to the next fullword boundary. If a string descriptor is specified, the count and pointer fields are zeroed to reflect the memory released. Either this parameter or the BINARY_DATA parameter must be specified.

Macro Descriptions
\$XPO_FREE_MEM - Release a Memory Element

BINARY_DATA = binary-data-info
describes a binary-data memory element. The data area described must begin on a BLISS fullword boundary; its length is rounded up, if necessary, to the next fullword boundary. If a binary-data descriptor is specified, the size and pointer fields are zeroed to reflect the memory released. Either this parameter or the **STRING** parameter must be specified.

FILL = memory clearing value
specifies a fullword binary value which is to be used to overwrite the memory element before it is released. If this parameter is omitted, no memory clearing is performed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the release memory operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the release memory operation is unsuccessful. If **FAILURE=0** is specified, no failure action routine is called. If this parameter is not specified, **FAILURE=XPO\$FAILURE** is assumed (see Section 3.8).

A.23.4 Completion Codes

Success Code:

XPO\$_NORMAL	The element was released.
---------------------	---------------------------

Error Codes:

XPO\$_BAD_ADDR	The element is not in allocated dynamic memory.
XPO\$_BAD_ALIGN	Invalid string/data alignment.
XPO\$_BAD_DESC	Invalid string/data descriptor.

Fatal Error Code:

XPO\$_BAD_LOGIC	An XPORT logic error was detected.
------------------------	------------------------------------

Macro Descriptions

\$XPO_GET - Read From a File

A.24 \$XPO_GET - Read From a File

The \$XPO_GET macro calls the XPORT I/O facility (1) to read the next logical record in an input file, or (2) to read a specified amount of character or binary data.

At the completion of a successful \$XPO_GET operation, the IOB is updated to reflect the data that has been read into an internal XPORT buffer. Note that \$XPO_GET is a "locate mode" operation; that is, the data is not read into a caller-provided buffer.

A.24.1 Syntax

read from a file	\$XPO_GET(parameter,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	IOB = address of iob
optional-parameter	{ PROMPT = char-string-info } { CHARACTERS = number of characters } { FULLWORDS = number of binary fullwords } { UNITS = number of binary units } { USER = user-defined value } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character-string descriptor } { 'literal ascii string' } { (count , pointer) } { string conversion pseudo-function }

A.24.2 Parameter Semantics

IOB = address of IOB
specifies the address of the IOB that describes the file to be read. This parameter must be specified.

PROMPT = char-string-info
describes a terminal input-prompt string. This parameter is ignored if the input device is not a terminal. If this parameter is not specified, the IOB prompt descriptor is not changed.

Macro Descriptions
\$XPO_GET - Read From a File

CHARACTERS = number of characters
FULLWORDS = number of binary fullwords
UNITS = number of binary units
specify the amount of data to be read. Only one of these parameters may be specified. The CHARACTERS parameter must be given for character-stream GET operations (see ATTRIBUTES=STREAM in A.28). This parameter specifies the number of characters to be read. The FULLWORDS or UNITS parameter must be given for binary GET operations (see ATTRIBUTES=BINARY in A.28). These parameters specify the amount of data to be read in terms of BLISS fullwords (FULLWORDS) or addressable units (UNITS). Specifying UNITS limits program transportability; that is, a program usually will not behave correctly in both the 36-bit and 16/32-bit environments if UNITS is specified. These parameters are ignored for record-mode GET operations (see ATTRIBUTES=RECORD in A.28). If no I/O length parameter is specified, the IOB I/O length field is not changed.

USER = user-defined value
specifies an application-dependent fullword value to be placed in the IOB field IOB\$Z_USER. If this parameter is not specified, the IOB user field is not changed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the \$XPO_GET operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the \$XPO_GET operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.24.3 Usage Guidelines

The following IOB fields are updated at the completion of an \$XPO_GET character operation:

IOB\$T_STRING	Character string descriptor:
IOB\$A_STRING	pointer to the character string
IOB\$H_STRING	number of characters read
IOB\$H_PAGE_NUMB	Page number (sequenced file only)
IOB\$Z_SEQ_NUMB	Record sequence number (sequenced file only) or record number (non-sequenced file)

The following IOB fields are updated at the completion of an \$XPO_GET binary operation:

IOB\$T_DATA	Binary data descriptor:
-------------	-------------------------

Macro Descriptions

\$XPO_GET - Read From a File

IOB\$A_DATA	address of the binary data
IOB\$H_UNITS	number of addressable units read
IOB\$H_FULLWORDS	number of fullwords read

A.24.4 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G COMP_CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G 2ND_CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Success Code:

XPO\$_NORMAL	The \$XPO_GET operation was successful - the IOB data pointer and length fields describe the data which has just been read.
XPO\$_INCOMPLETE	An incomplete amount of data has been read (STREAM or BINARY mode only).
XPO\$_NEW_FILE	The first read on a concatenated input file was successful - implies new page.
XPO\$_NEW_PAGE	The first read on a new page was successful.

Warning Code:

XPO\$_END_FILE	No more data exists in the input file, i.e., attempt to read past end-of-file.
----------------	--

Error Codes:

XPO\$ BAD_PROMPT +	The input prompt is invalid.
XPO\$ BAD_RECORD +	An invalid record was encountered.
XPO\$ BAD_REQ +	The I/O request is invalid.
XPO\$ FREE_MEM +	Error freeing IOB-related memory.
XPO\$ GET_MEM +	A memory allocation error occurred.
XPO\$ IO_BUFFER +	An I/O buffering error occurred.
XPO\$ IO_ERROR +	An I/O error occurred reading the file.
XPO\$ NETWORK +	A network error has occurred.
XPO\$ NO_MEMORY	Insufficient dynamic memory space.
XPO\$ NO_SPACE	Quota exceeded or disk full.
XPO\$ NO_SUPPORT +	The requested function is not supported.
XPO\$ NO_WRITE	The file is write-protected.
XPO\$ NOT_INPUT	The file is not open for input.
XPO\$ NOT_ONLINE	The device was not ready.
XPO\$ NOT_OPEN	The file is not open.
XPO\$ REC_LOCK	A record is locked by another task.

Fatal Error Codes:

XPO\$ BAD_IOB +	The IOB is invalid.
XPO\$ BAD_LOGIC	An XPORT logic error was detected.

Macro Descriptions
\$XPO_GET - Read From a File

If input-file concatenation is in effect (see Section 3.2.2.1), OPEN or CLOSE failure codes are returned if either of these automatic operations fails.

Macro Descriptions

\$XPO_GET_MEM - Allocate Dynamic Memory Element

A.25 \$XPO_GET_MEM - Allocate Dynamic Memory Element

The \$XPO_GET_MEM macro calls the XPORT MEMORY facility to allocate an element of dynamic memory. An allocated memory element may optionally be initialized.

A.25.1 Syntax

allocate memory	\$XPO_GET_MEM(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	size-parameter result-parameter
size-parameter	{ CHARACTERS = number of characters } { FULLWORDS = number of fullwords } { UNITS = number of addressable units }
result-parameter	{ RESULT = address for result } { DESCRIPTOR = address of descriptor }
optional-parameter	{ FILL = memory initialization value } { SUCCESS = address of action routine } { FAILURE = address of action routine }

A.25.2 Restrictions

The CHARACTERS, FULLWORDS, and UNITS parameters are mutually exclusive.

A.25.3 Parameter Semantics

CHARACTERS = number of characters

FULLWORDS = number of fullwords

UNITS = number of addressable units

specify the size of the memory element to be allocated. One, and only one, of these parameters must be specified.

Macro Descriptions
\$XPO_GET_MEM - Allocate Dynamic Memory Element

RESULT = address of result
specifies the address of an area in which the allocation operation (if successful) will store a pointer to the allocated string element, or will store the address of the allocated binary-data element, respectively. Note that the RESULT parameter or the DESCRIPTOR parameter must be specified.

DESCRIPTOR = address of descriptor
specifies the address of a DYNAMIC or DYNAMIC_BOUNDED descriptor. If the allocation operation is successful, the descriptor is updated to describe the allocated string or binary-data element.

The following indicates the descriptor fields that are changed:

STR\$H_LENGTH or XPO\$H_LENGTH:
size of element if DYNAMIC and zero if DYNAMIC_BOUNDED
STR\$H_MAXLEN or XPO\$H_MAXLEN:
size of element only if DYNAMIC_BOUNDED
STR\$A_POINTER or XPO\$A_ADDRESS:
element POINTER or ADDRESS

Note that the DESCRIPTOR parameter or the RESULT parameter must be specified.

FILL = memory initialization value
specifies an appropriate value to be used to initialize each character, fullword, or unit in an allocated memory element. If this parameter is not specified, the content of an allocated memory element is unpredictable.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the memory allocation operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the memory allocation operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.25.4 Completion Codes

Success Code:
XPO\$_NORMAL The memory element was successfully allocated.

Macro Descriptions
\$XPO_GET_MEM - Allocate Dynamic Memory Element

Error Codes:

XPO\$_NO_MEMORY	Insufficient memory to satisfy request.
XPO\$_BAD_DESC	Invalid string/data descriptor.

Fatal Error Code:

XPO\$_BAD_LOGIC	An XPORT logic error was detected.
-----------------	------------------------------------

Macro Descriptions
\$XPO_IOB - Declare an IOB

A.26 \$XPO_IOB - Declare an IOB

The \$XPO_IOB macro is used, in a data- or bind-data-declaration, to create and possibly initialize an XPORT I/O control block (IOB). The \$XPO_IOB macro generates a structure-attribute and field-attribute for an IOB data-segment name, and causes the data segment to be initialized if it is allocated in permanent storage.

The macro itself is specified as an attribute in an OWN, GLOBAL, LOCAL, MAP or BIND declaration. The generated attributes (1) indicate that the IOB is a BLOCK structure of a given length, and (2) define the field-names that can be used to reference portions of the IOB.

An IOB that is created in temporary storage (LOCAL declaration) or in dynamic storage must be explicitly initialized with the \$XPO_IOB_INIT macro before it can be used. (See Section A.25.)

A.26.1 Syntax

declare an iob	\$XPO_IOB()
----------------	--------------

A.26.2 Parameter Semantics

The \$XPO_IOB macro takes no parameters at the present time.

A.26.3 Examples

```
OWN
    input_iob : $XPO_IOB(),
    output_iob : $XPO_IOB();
LOCAL
    temporary_iob : $XPO_IOB();
MAP
    dynamic_iob : REF $XPO_IOB();
BIND
    selected_iob = iobset[.index, 0,0,0,0] : $XPO_IOB() ;
```

Macro Descriptions

\$XPO_IOB_INIT - Initialize an IOB

A.27 \$XPO_IOB_INIT - Initialize an IOB

The \$XPO_IOB_INIT macro dynamically initializes an XPORT IOB; that is, it generates the executable code necessary to initialize all of the fields of an IOB. Dynamic initialization is necessary for IOBs created in temporary storage (i.e., declared as LOCAL) or in dynamically-acquired storage.

The user-related IOB fields are initialized to a user-specified value (for subsequent operations), or to zeroes if no values are specified.

A.27.1 Syntax

setup an iob	\$XPO_IOB_INIT(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	IOB=address of iob
optional-parameter	{ USER = user-defined value } { \$XPO_CLOSE parameters } { \$XPO_DELETE parameters } { \$XPO_GET parameters } { \$XPO_OPEN parameters } { \$XPO_PUT parameters } { \$XPO_RENAME parameters }

A.27.2 Restrictions

The SUCCESS and FAILURE parameters may not be specified (as optional parameters of the other macros listed above). In addition, the parameters that are unique to the \$XPO_RENAME macro -- NEW_SPEC, NEW_DEFAULT, and NEW_RELATED -- may not be specified.

A.27.3 Parameter Semantics

IOB = address of IOB
specifies the address of the IOB to be initialized. This parameter must be specified.

Macro Descriptions
\$XPO_IOB_INIT - Initialize an IOB

USER = user-defined value
specifies an application-dependent fullword value to be placed in
the IOB field IOB\$Z_USER. If this parameter is not specified, no
user value is assumed.

\$XPO_CLOSE parameter
\$XPO_DELETE parameter
\$XPO_GET parameter
\$XPO_OPEN parameter
\$XPO_PUT parameter
\$XPO_RENAME parameter
initialize IOB fields for subsequent I/O operations.

A.27.4 Completion Code

Success Code:
XPO\$_NORMAL The IOB was successfully initialized.

Macro Descriptions
\$XPO_OPEN - Open a File

A.28 \$XPO_OPEN - Open a File

The \$XPO_OPEN macro calls the XPORT I/O facility to prepare a file for reading or writing. Before a file is opened, the IOB defaults are established and an IOB validity check is made. When a file is opened for input, the file attributes are checked against those specified in the IOB to ensure against a conflict (e.g., RECORD vs. BINARY, see below). This operation performs file-specification resolution as necessary.

A.28.1 Syntax

open a file	\$XPO_OPEN(parameter ,...)
parameter	{ required-parameter } { primary-parameter } { optional-parameter }
required-parameter	IOB = address of iob
primary-parameter	{ FILE_SPEC = char-string-info } { OPTIONS = (option-keyword ,...) } { ATTRIBUTES = (attribute-keyword ,...) }
optional-parameter	{ DEFAULT = char-string-info } { RELATED = char-string-info } { RECORD_SIZE = fixed length of record } { RECORD_SIZE = VARIABLE } { BLOCK_SIZE = fixed length of block } { USER = user-defined value } { SUCCESS = address of action routine } { FAILURE = address of action routine } { any \$XPO_CLOSE parameter } { any \$XPO_GET parameter } { any \$XPO_PUT parameter }
char-string-info	{ address of character string descriptor } { 'literal ascii string' } { (count , pointer) }
option-keyword	{ INPUT } { OUTPUT } { OVERWRITE APPEND }
continued on the next page	

Macro Descriptions
\$XPO_OPEN - Open a File

attribute-keyword	{ RECORD STREAM BINARY }
	{ SEQUENCED }

NOTE: The keywords OPTIONS and ATTRIBUTES may be shortened to OPTION and ATTRIBUTE respectively.

A.28.2 Parameter Semantics

IOB = address of IOB
specifies the address of the IOB that describes the file to be opened. This parameter must be specified.

FILE_SPEC = character-string-info
describes a file specification provided by an end user. Unless the IOB was previously closed with the REMEMBER option, this user file specification is combined with the default and related file specifications, if any, to form the resultant file specification (see Section 3.6.1). Unless otherwise specified by file open time, a null user file specification is assumed.

OPTIONS = (option-keyword ,...)
indicates the processing options that apply to the file being opened. One or more of the options described in the following table can be specified. Note that some of these options are mutually exclusive; any errors will be detected at file open time.

<u>Option</u>	<u>Description</u>
INPUT	The file is opened as an input file (default option).
OUTPUT	The file is opened as an output file. If neither OVERWRITE nor APPEND is specified, a new output file is created. If this is not possible, the file opening will fail.
OVERWRITE	If a file being opened for <u>output</u> already exists, it is overwritten from the beginning. If this option is specified, OUTPUT is assumed.
APPEND	If a file being opened for <u>output</u> already exists, it is appended to rather than overwritten. If this option is specified, OUTPUT is assumed.

Macro Descriptions

\$XPO_OPEN - Open a File

A device that is not file-structured and that can be both read and written (e.g., terminal, DECnet link), may be opened for both INPUT and OUTPUT, and will be so opened by default; for all other devices, the INPUT and OUTPUT options are mutually exclusive. Unless otherwise specified by file open time, OPTIONS=INPUT is assumed.

ATTRIBUTES = attribute-keyword ,...
indicates the attributes that apply to the file described by the IOB. One or more of the attributes described in the following table can be specified. Note that some of these attributes are mutually exclusive; any errors will be detected at file open time.

<u>Attribute</u>	<u>Description</u>
RECORD	<u>Character</u> data is read and written in terms of logical records.
STREAM	<u>Character</u> data is read and written as <u>streams</u> of characters. (Control characters, e.g., CR, LF, VT, may be read and written in this mode). GET operations in STREAM mode require specification of the CHARACTERS= parameter (see A.24).
SEQUENCED	All output records are to have an associated sequence number. If this attribute is specified, RECORD is assumed.
BINARY	The file contains binary data.

Unless otherwise specified by file open time, ATTRIBUTES=RECORD is assumed.

DEFAULT = character-string-info
describes a default file specification. During file-specification resolution, this file specification is combined with the user and related file specifications, if any, to form the resultant file specification (see Section 3.6.1). Unless otherwise specified by file open time, no default file specification is assumed.

RELATED = character-string-info
describe a file specification that is related to the file being opened. For example, an application that creates an output file as a modification or "update" of an input file, e.g., a text editor, will typically treat the input file as a related file.

Macro Descriptions
\$XPO_OPEN - Open a File

During file-specification resolution, a related file specification is combined with the user and default file specifications, if any, to form the resultant file specification (see Section 3.6.1). Unless otherwise specified by file open time, no related file specification is assumed.

RECORD_SIZE = fixed length of record
specifies the size of a record in an output file. This parameter should only be specified for fixed-length, record-oriented output files (see ATTRIBUTES=RECORD above). The record length is expressed in terms of characters. Unless otherwise specified by file open time, "RECORD_SIZE = VARIABLE" is assumed.

RECORD_SIZE = VARIABLE
specifies that an output-file record is of variable length.. This parameter is the RECORD_SIZE default for record-oriented output files (see ATTRIBUTES=RECORD above).

BLOCK_SIZE = fixed length of block
specifies the physical block size of an output file. It has meaning only for those devices which support a user-specified block size (e.g., magnetic tape). On a system which uses Files-11 disk-file structure, this block-size parameter corresponds to the "RMS bucket size". This parameter should only be specified for output files. A block size is expressed in terms of "bytes" (i.e., 9 bits on a PDP-10, 8 bits on a PDP-11 or VAX-11). Unless otherwise specified by file open time, a device-dependent block size is assumed.

/REVIEWERS: SHOULD THE BLOCK-SIZE VALUE BE FULLWORDS OR BYTES?/

USER = user-defined value
specifies an application-dependent fullword value to be placed in the IOB field IOB\$Z_USER. If this parameter is not specified, the IOB user field is not changed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the OPEN operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the OPEN operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

any \$XPO_CLOSE parameter
any \$XPO_GET parameter
any \$XPO_PUT parameter
initialize IOB fields for subsequent I/O operations.

Macro Descriptions
\$XPO_OPEN - Open a File

A.28.3 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G COMP CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G 2ND CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Success Code:

XPO\$ NORMAL	The file was successfully opened.
XPO\$ CREATED	The file was created and successfully opened.

Error Codes:

XPO\$ BAD ACCT	Invalid account field.
XPO\$ BAD ATTR	Invalid attribute field in file spec.
XPO\$ BAD DELIM	Invalid punctuation used in a quoted string.
XPO\$ BAD DEVICE	An invalid device was specified.
XPO\$ BAD DFLT +	The default file specification is invalid.
XPO\$ BAD DIRECT	Directory-access privilege required, or invalid directory format.
XPO\$ BAD FORMAT	The record format is invalid.
XPO\$ BAD IO OPT	An invalid I/O option was specified (e.g., binary terminal I/O).
XPO\$ BAD NAME	No filename was specified.
XPO\$ BAD ORG	The file organization is invalid.
XPO\$ BAD PROT	Invalid protection field.
XPO\$ BAD RECORD	An invalid record was encountered.
XPO\$ BAD REQ +	The XPORT request was invalid.
XPO\$ BAD RLTD +	The related file specification is invalid.
XPO\$ BAD RSLT +	The resultant file specification is invalid.
XPO\$ BAD SPEC +	The user file specification is invalid.
XPO\$ BAD TEMP	Multiple ";T"s specified.
XPO\$ BAD VER	Generation number not numeric.
XPO\$ CHANNEL +	A channel-assignment error occurred.
XPO\$ CORRUPTED	The file header contains invalid information.
XPO\$ EXISTS	The file already exists.
XPO\$ FILE LOCK	The file is locked.
XPO\$ FREE MEM +	Error freeing IOB-related memory.
XPO\$ GET MEM +	A memory-allocation error occurred.
XPO\$ IN USE	The file is currently in use.
XPO\$ IO ERROR +	A hardware-level I/O error occurred.
XPO\$ NETWORK +	A network error has occurred.
XPO\$ NO ACCESS +	The file cannot be accessed.
XPO\$ NO CHANNEL	All I/O channels are currently in use.
XPO\$ NO CREATE +	The file could not be created.
XPO\$ NO DIRECT	The indicated directory was no found.
XPO\$ NO FILE	The file does not exist.

Macro Descriptions
\$XPO_OPEN - Open a File

XPO\$_OPEN	The file has already been opened.
XPO\$_NO_SPACE	Insufficient space on the requested or implied device.
XPO\$_NO_SUBDIR	The sub-directory does not exist.
XPO\$_NO_SUPPORT +	The requested function is not supported.
XPO\$_NO_WRITE	The file is write-protected.
XPO\$_NOT_EXPIRE	The file-expiration date is not past.
XPO\$_NOT_ONLINE	The device was not ready.
XPO\$_PROTECTED	Access to the file is denied.
Fatal Error Codes:	
XPO\$_BAD_IOB +	The IOB is invalid.
XPO\$_BAD_LOGIC	An XPORT logic error was detected.

Macro Descriptions

\$XPO_PARSE_SPEC - Parse a File Specification

A.29 \$XPO_PARSE_SPEC - Parse a File Specification

The \$XPO_PARSE_SPEC macro calls the XPORT I/O facility to parse a file-specification string into its component parts (e.g., device name, file name, file version). The parsing operation includes a target-system-dependent syntax check of the file specification.

A failure completion code indicates that an invalid component was encountered in the parsing operation. The results of the parse, including any diagnostic information, is stored in a user-supplied File Specification Block (see the \$XPO_SPEC_BLOCK macro, Section A.31).

A.29.1 Syntax

parse a file-spec	\$XPO_PARSE_SPEC(parameter ,...)
parameter	{ required-parameter } { optional-parameter }
required-parameter	{ FILE_SPEC = char-string-info } { SPEC_BLOCK = address of file-spec block }
optional-parameter	{ SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character string descriptor } { 'literal ascii string' } { (count , pointer) }

A.29.2 Parameter Semantics

FILE_SPEC = character-string-info
describes the file specification to be parsed. This parameter must be specified.

SPEC_BLOCK = address of file-spec block
specifies the address of an XPORT File Specification Block (see A.31) that is to receive the results of the parsing. This parameter must be specified.

Macro Descriptions
\$XPO_PARSE_SPEC - Parse a File Specification

SUCCESS = address of action routine
specifies the address of an action routine to be called upon the successful completion of the PARSE_SPEC operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the PARSE_SPEC operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.29.3 Completion Codes

Success Code:
XPO\$_NORMAL The file specification was successfully parsed.

Error Codes:

XPO\$_BAD_DELIM	Invalid delimiter.
XPO\$_BAD_DEVICE +	Invalid device name.
XPO\$_BAD_DIRECT +	Invalid directory specification.
XPO\$_BAD_NAME +	Invalid file name.
XPO\$_BAD_NODE +	Invalid node name.
XPO\$_BAD_TYPE +	Invalid file type (or extension).
XPO\$_BAD_VER +	Invalid file version.

Macro Descriptions

\$XPO_PUT - Write to a File

A.30 \$XPO_PUT - Write to a File

The \$XPO_PUT macro calls the XPORT I/O facility to write data to an output file at its current position. See the OPTIONS parameter of the \$XPO_OPEN macro (A.28) for information about initial positioning of an output file.

A.30.1 Syntax

write into a file	\$XPO_PUT(parameter ,...)
parameter	{ required-parameter } { primary-parameter } { optional-parameter }
required-parameter	IOB = address of iob
primary-parameter	{ STRING = char-string-info } { BINARY_DATA = binary-data-info }
optional-parameter	{ USER = user-defined value } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character string descriptor } { 'literal ascii string' } { (count , pointer) } { string conversion pseudo-function }
binary-data-info	{ address of binary data descriptor } { { , <u>FULLWORDS</u> } } { (size , address { , <u>UNITS</u> }) } { { nothing } }

A.30.2 Restrictions

The character-string parameters (STRING, PAGE_NUMBER, SEQUENCE_NUMBER) and the BINARY_DATA parameter are mutually exclusive.

Macro Descriptions
\$XPO_PUT - Write to a File

A.30.3 Parameter Semantics

IOB = address of IOB
specifies the address of the IOB that describes the file being written. This parameter must be specified.

STRING = character-string-info
describes an output record or stream composed of characters. If this parameter is not specified, the IOB character-output-string descriptor is not changed.

BINARY_DATA = binary-data-info
describes an output binary data stream. If this parameter is not specified, the IOB binary-output-data descriptor is not changed.

USER = user-defined value
specifies an application-dependent fullword value to be placed in the IOB field IOB\$Z_USER. If this parameter is not specified, the IOB user field is not changed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the \$XPO_PUT operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the \$XPO_PUT operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.30.4 Usage Guidelines

For a character-stream PUT operation (see ATTRIBUTES=STREAM in \$XPO_OPEN, A.28) no carriage return, line feed, or any other format-control characters are supplied by XPORT in the output record. In this mode, it is the user's exclusive responsibility to supply in the character string any and all control character that might be desired in the output record.

A.30.5 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G_COMP_CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G_2ND_CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

Macro Descriptions
\$XPO_PUT - Write to a File

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Success Code:

XPO\$_NORMAL	The record was successfully written.
--------------	--------------------------------------

Error Codes:

XPO\$_BAD_REQ +	The I/O request is invalid.
XPO\$_CORRUPTED	The file header contains invalid information.
XPO\$_FREE_MEM +	Error freeing IOB-related memory.
XPO\$_GET_MEM +	A memory-allocation error occurred.
XPO\$_IO_ERROR +	An I/O error occurred writing the file.
XPO\$_NETWORK +	A network error has occurred.
XPO\$_NO_ACCESS +	The file cannot be accessed.
XPO\$_NO_SPACE	The file cannot be extended, device is full.
XPO\$_NO_SUPPORT +	The requested function is not supported.
XPO\$_NO_WRITE	The file is write-protected.
XPO\$_NOT_ONLINE	The device was not ready.
XPO\$_NOT_OPEN	The file is not open.
XPO\$_NOT_OUTPUT	The file is not open for output.
XPO\$_REC_LOCK	A record is locked by another task.
XPO\$_TRUNCATED	A truncated record was successfully written.

Fatal Error Codes:

XPO\$_BAD_IOB +	The IOB is invalid.
XPO\$_BAD_LOGIC	An XPORT logic error was detected.

Macro Descriptions

\$XPO_PUT_MSG - Send a Message

A.31 \$XPO_PUT_MSG - Send a Message

The \$XPO_PUT_MSG macro calls the XPORT MESSAGE facility to send a single- or multiple-line message to the user of the program.

The routing of a message depends on its associated severity. All messages, regardless of severity, are sent to the standard XPORT output device (\$XPO_OUTPUT), normally the user's terminal in the case of an interactive program. Messages with a severity other than SUCCESS are also sent to the standard XPORT error device (\$XPO_ERROR) if that device is different from the standard output device.

Sending a FATAL message will result in automatic program termination at the completion of message processing.

A.31.1 Syntax

send a message	\$XPO_PUT_MSG(parameter ,...)
parameter	{ required-parameter } { primary-parameter } { optional-parameter }
required-parameter	{ CODE = completion code } { STRING = char-string-info }
primary-parameter	SEVERITY = { SUCCESS WARNING ERROR FATAL }
optional-parameter	{ SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character string descriptor } { 'literal ascii string' } { (count , pointer) }

A.31.2 Parameter Semantics

CODE = completion code
specifies an XPORT completion code. The message text corresponding to this completion code is retrieved and sent to the appropriate devices (see SEVERITY below). This parameter may be specified more than once in a single macro call; each occurrence results in a single message. Either this parameter or the STRING parameter must be specified.

Macro Descriptions
\$XPO_PUT_MSG - Send a Message

STRING = character-string-info
describes a message string. This parameter may be specified more than once in a single macro call; each occurrence results in a single message. Either this parameter or the CODE parameter must be specified.

SEVERITY = severity-level keyword
specifies the severity to be associated with the message. A FATAL level will result in automatic program termination. If this parameter is not specified and CODE is specified, the severity code is determined from the completion code. Otherwise, if this parameter is not specified, SEVERITY=ERROR is assumed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the message output operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the message output operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

A.31.3 Completion Codes

Success Code:

XPO\$NORMAL	The message was successfully processed.
-------------	---

Error Codes:

XPO\$BAD_ARGS	The message argument list is invalid.
STR\$BAD_SOURCE	The string descriptor is invalid.
...	(Failure completion codes from \$XPO_GET_MEM.)
...	(Failure completion codes from \$XPO_PUT, see below.)

Fatal Error Codes:

XPO\$BAD_LOGIC	An XPORT logic error was detected.
----------------	------------------------------------

Note that \$XPO_PUT may be used to output messages to the standard XPORT output and/or error devices. If the \$XPO_PUT operation fails, the failure completion code from that operation is passed back as the \$XPO_PUT_MSG completion code.

Macro Descriptions
\$XPO_RENAME - Rename a File

A.32 \$XPO_RENAME - Rename a File

The \$XPO_RENAME macro calls the XPORT I/O facility to change one or more of the following attributes of an existing file: directory specification, file name, file extension (or type), and file version (if any). After a successful file renaming, the IOB user and resultant file specifications are updated to reflect the changes that were made.

This operation, like OPEN and DELETE, performs file-specification resolution as necessary.

A.32.1 Syntax

rename a file	\$XPO_RENAME(parameter ,...)
parameter	{ required-parameter } { primary-parameter } { optional-parameter }
required-parameter	IOB=address of iob
primary-parameter	{ FILE SPEC = char-string-info } { NEW_SPEC = char-string-info }
optional-parameter	{ DEFAULT = char-string-info } { NEW DEFAULT = char-string-info } { RELATED = char-string-info } { NEW RELATED = char-string-info } { OPTIONS = option-keyword } { USER = user-defined value } { SUCCESS = address of action routine } { FAILURE = address of action routine }
char-string-info	{ address of character string descriptor } { 'literal ascii string' } { (count , pointer) }
option-keyword	REMEMBER

NOTE: The keyword OPTIONS may be shortened to OPTION.

Macro Descriptions
\$XPO_RENAME - Rename a File

A.32.2 Parameter Semantics

- IOB = address of IOB
specifies the address of an IOB for the file to be renamed. This IOB must be initialized, but it must not be open when the RENAME call is made. This parameter must be specified.
- FILE_SPEC = character-string-info
describes a file specification provided by the end user. Unless the IOB was previously closed with the REMEMBER option, this user file specification is combined with the default and related file specifications, if any, to form the resultant file specification (see Section 3.6.1). If this parameter is not specified, the corresponding IOB string-descriptor field is not changed.
- NEW_SPEC = character-string-info
describes the new file specification, typically also provided by the end user. This new file specification is combined with the new-default and new-related file specifications, if any, to form the resultant new file specification (see Section 3.6.1). If this parameter is not specified, the corresponding IOB string-descriptor field is not changed.
- DEFAULT = character-string-info
describes a default file specification. During file-specification resolution, this file specification is combined with the user and related file specifications, if any, to form the resultant file specification. If this parameter is not specified, the corresponding IOB string-descriptor field is not changed.
- NEW_DEFAULT = character-string-info
describes a default for the new file specification. This file specification is combined with the new and new-related file specifications, if any, to form the resultant new file specification. If this parameter is not specified, the corresponding IOB string-descriptor field is not changed.
- RELATED = character-string-info
describes a file specification that is related to the file being renamed. During file-specification resolution, this file specification is combined with the user and default file specifications, if any, to form the resultant file specification. If this parameter is not specified, the corresponding IOB string-descriptor field is not changed.

Macro Descriptions
\$XPO_RENAME - Rename a File

NEW_RELATED = character-string-info
describes a file specification that is related to the new file specification for the file being renamed. This file specification is combined with the new and new-default file specifications, if any, to form the resultant new file specification. If this parameter is not specified, the resultant old file specification is assumed as the new-related file specification.

OPTIONS = option-keyword
indicates a processing option to be applied to the file being renamed.

<u>Option</u>	<u>Meaning</u>
REMEMBER	Remember relevant file <u>attributes</u> (e.g., resultant file specification) so that the file can be reprocessed (e.g., opened, read, written, backed up). File processing <u>options</u> (including this option) are <u>not</u> remembered, whether or not this option is specified. If this option is not specified by file rename time, the IOB is reset to an initialized state after successful renaming.

If this parameter is not specified, the IOB option field is not changed.

USER = user-defined value
specifies an application-dependent fullword value to be placed in the IOB field IOB\$Z_USER. If this parameter is not specified, the IOB user field is not changed.

SUCCESS = address of action routine
specifies the address of an action routine to be called upon successful completion of the rename operation. If this parameter is not specified, no success action routine is assumed.

FAILURE = address of action routine
specifies the address of an action routine to be called if the rename operation is unsuccessful. If FAILURE=0 is specified, no failure action routine is called. If this parameter is not specified, FAILURE=XPO\$FAILURE is assumed (see Section 3.8).

Macro Descriptions
\$XPO_RENAME - Rename a File

A.32.3 Completion Codes

A primary completion code is returned as the routine-call value, and is also available in the IOB\$G_COMP_CODE field of the IOB. A secondary completion code (if any) is available in the IOB\$G_2ND_CODE field of the IOB. Secondary completion codes, where applicable, are indicated by a plus sign (+) following the associated primary code in the listing below. The secondary completion codes are listed and described in Appendix C.

NOTE: Some of the completion codes listed below may not apply to all operating systems.

Success Code:

XPO\$_NORMAL	The file was successfully renamed.
--------------	------------------------------------

Error Codes:

XPO\$_BAD_ACCT	Invalid account field.
XPO\$_BAD_ATTR	Invalid attribute field in file spec.
XPO\$_BAD_DELIM	Invalid punctuation used in a quoted string.
XPO\$_BAD_DEVICE	An invalid device was specified.
XPO\$_BAD_DIRECT +	Error entering filename in directory.
XPO\$_BAD_DFLT +	The default file specification is invalid.
XPO\$_BAD_NAME	No filename was specified.
XPO\$_BAD_NEW +	The new file specification is invalid.
XPO\$_BAD_PROT	Invalid protection field.
XPO\$_BAD_REQ +	The XPORT request was invalid.
XPO\$_BAD_RLTD +	The related file specification is invalid.
XPO\$_BAD_TEMP	Multiple ";T"s specified.
XPO\$_BAD_VER	Generation number not numeric.
XPO\$_BAD_RSLT +	The resultant file specification is invalid.
XPO\$_BAD_SPEC +	The user file specification is invalid.
XPO\$_CHANNEL +	A channel-assignment error occurred.
XPO\$_FREE_MEM +	Error freeing IOB-related memory.
XPO\$_GET_MEM +	A memory-allocation error occurred.
XPO\$_IO_ERROR +	A hardware-level I/O error occurred.
XPO\$_NO_ACCESS +	The file cannot be accessed.
XPO\$_NO_CHANNEL	No I/O channel was available.
XPO\$_NO_DIRECT	The indicated directory was not found.
XPO\$_NO_FILE	The file does not exist.
XPO\$_NO_RENAME +	The file cannot be renamed.
XPO\$_NO_SPACE	File space exhausted.
XPO\$_NO_SUBDIR	The sub-directory does not exist.
XPO\$_NO_SUPPORT +	The requested function is not supported.
XPO\$_NO_WRITE	The file is write-protected.
XPO\$_NOT_ONLINE	The device was not ready.
XPO\$_OPEN	The file is currently open.
XPO\$_PROTECTED	Access to the file is denied.

Fatal Error Codes:

XPO\$_BAD_IOB +	The IOB is invalid.
XPO\$_BAD_LOGIC	An XPORT logic error was detected.

Macro Descriptions
\$XPO_SPEC_BLOCK - Declare a File Specification Block

A.33 \$XPO_SPEC_BLOCK - Declare a File Specification Block

The \$XPO_SPEC_BLOCK macro generates an attribute list for an XPORT file specification block allocated within an OWN, GLOBAL, LOCAL, MAP or BIND declaration. These attributes (1) indicate that the file specification block is a BLOCK structure of a given length, and (2) define the field-names that can be used to reference portions of the block.

A file specification block is used in conjunction with the \$XPO_PARSE_SPEC macro (see A.27). The format of a file specification block is given in Appendix B.

A.33.1 Syntax

declare spec-block	\$XPO_SPEC_BLOCK
--------------------	------------------

A.33.2 Examples

```
OWN
    input_spec_blk : $XPO_SPEC_BLOCK,
    output_spec_blk : $XPO_SPEC_BLOCK;

LOCAL
    temp_spec_blk : $XPO_SPEC_BLOCK;

MAP
    passed_spec_blk : REF $XPO_SPEC_BLOCK;
```

Macro Descriptions
\$XPO_TERMINATE - Terminate Program Execution

A.34 \$XPO_TERMINATE - Terminate Program Execution

The \$XPO_TERMINATE macro terminates program execution after sending the user a termination message. The message that is issued is determined by the completion code associated with the macro call.

A.34.1 Syntax

+-----+-----+	
terminate program	\$XPO_TERMINATE({optional-parameter})
+-----+-----+	
optional-parameter	CODE = completion code
+-----+-----+	

A.34.2 Parameter Semantics

CODE = completion code
specifies an XPORT completion code. This code is used to select the program termination message that is sent to the user. If this parameter is not specified, CODE = XPO\$_TERMINATE is assumed.

A.34.3 Routine Value

This XPORT function does not return a value or a completion code since it results in program termination. However, the specified or assumed completion code becomes the program termination code.

APPENDIX B CONTROL BLOCKS

B.1	INPUT/OUTPUT BLOCK (IOB)	B-2
B.2	STRING DESCRIPTORS	B-4
B.3	BINARY DATA DESCRIPTORS	B-5
B.4	FILE SPECIFICATION PARSE BLOCK	B-6

APPENDIX B

CONTROL BLOCKS

This appendix presents a detailed description of the control blocks that are involved in the use of XPORT facilities. This appendix is intended for reference use. A tutorial discussion of these control blocks and their usage is given earlier in this manual, beginning with Chapter 3.

Control Blocks INPUT/OUTPUT BLOCK (IOB)

B.1 INPUT/OUTPUT BLOCK (IOB)

The following table describes the IOB fields and literals that may be of interest to an XPORT I/O user. The entries under the columns "Used By" and "Set By" indicate which XPORT I/O functions use the corresponding field value (typically set by macro keyword parameters), and which functions set the corresponding field value.

Symbol	Type	Used By*	Set By*	Description
IOB\$H_LENGTH	integer	all	init	Length of IOB (number of elements)
IOB\$T_RESULTANT	desc	close, backup	open, delete,	Resultant file specification descriptor
IOB\$A_ASSOC_IOB	address	backup, rename		Address of associated IOB
IOB\$B_FUNCTION	byte	all		I/O function code:
IOB\$K_OPEN	1			open file
IOB\$K_CLOSE	2			close file
IOB\$K_DELETE	3			delete file
IOB\$K_RENAME	4			rename file
IOB\$K_BACKUP	5			create backup copy of input file
IOB\$K_GET	6			get record (locate mode)
IOB\$K_PUT	7			put record (move mode)
IOB\$V_OPTIONS	16 bits			I/O option flags:
IOB\$V_INPUT	bit	open, get		open for input
IOB\$V_OUTPUT	bit	open, put		open for output
IOB\$V_OVERWRITE	bit	open-out		overwrite existing output file
IOB\$V_APPEND	bit	open-out		append to existing output file
IOB\$V_REMEMBER	bit	close		file will be reprocessed after close
IOB\$V_MAX_VERSI	bit	open, rename		maximize file version number (internal)
IOB\$V_ATTRIBUTE	16 bits			File attributes:
IOB\$V_BINARY	bit	open, get, put		binary data
IOB\$V_STREAM	bit	open, get, put		stream-oriented character data
IOB\$V_RECORD	bit	open, get, put		record-oriented character data
IOB\$V_SEQUENCED	bit	open-out, put	open-in	sequence-numbered records
IOB\$V_STATUS	16 bits			Current file status:
IOB\$V_OPEN	bit	all	open	file is open
IOB\$V_EOF	bit	get, put	get, put	end-of-file detected
IOB\$V_CLOSED	bit	open	close	file is closed
IOB\$V_AUTO_CONC	bit	open	get-conc	input file switching in progress
IOB\$V_TERMINAL	bit	get, put	open	I/O device is a terminal
IOB\$V_TEMPORARY	bit	open, close	open	XPORT temporary file
IOB\$V_CONC_SPEC	bit	close	open	primary file-spec is a concatenated file-spec
IOB\$V_CH_ASSIGN	bit	open, delete, rename	open, delete, rename	channel has been assigned
IOB\$T_STRING	desc			Character input string descriptor:
IOB\$H_STRING	2 bytes	get-stream	get-char	length of the character string
IOB\$A_STRING	pointer		get-char	pointer to the character string
IOB\$T_DATA	desc			Binary input data descriptor (overlays IOB\$T_STRING):
IOB\$H_UNITS	2 bytes	get-bin	get-bin	length of the data in addressable units
IOB\$A_DATA	address		get-bin	address of the data
IOB\$H_FULLWORDS	2 bytes		get-full	length of the data in BLISS fullwords

continued on the next page

Control Blocks INPUT/OUTPUT BLOCK (IOB)

Symbol	Type	Used By*	Set By*	Description
IOB\$H_PAGE_NUMB	integer	put-seq	get-seq	Current page number
IOB\$G_SEQ_NUMB	integer		get	Sequence number of current record
IOB\$G_PREV_REC	integer			Number of last direct record read or written (future)
IOB\$G_REC_NUMB	integer	open-out		Direct-access record number (future)
IOB\$G_REC_SIZE	integer		open	Fixed record size (0 = variable length records)
IOB\$G_BLK_SIZE	integer	open-out	open	Block size
IOB\$G_COMP_CODE	integer		all	Completion code of current operation
IOB\$G_2ND_CODE	integer		all	Secondary completion code
IOB\$Z_USER	integer			User-defined value
IOB\$G_USER_CODE	integer			User-defined completion code
IOB\$A_BUFFER_CB	address	get, put	open	Address of TOPS-10 buffer control block
IOB\$A_RMS_FAB	address	close	open	Address of RMS FAB (system-specific)
IOB\$A_RMS_RAB	address	get, put	open	Address of RMS RAB (system-specific)
IOB\$A_FCS_FDB	address	get put	open	Address of FCS FDB (system-specific)
		close		
IOB\$A_RSTS_CB	address	get put	open	Address of RSTS control block
		close		
IOB\$H_CHANNEL	integer	get put	open	I/O channel number (system-specific)
		close		

* Code-names for XPORT I/O functions:

all = All I/O functions
 init = IOB initialization
 open = Open either input file or output file
 open-in = Open input file
 open-out = Open output file
 open-conc = Automatic open of concatenated input file
 close = Close file
 get = All binary and character get operations
 get-char = Get character string
 get-stream = Get character stream data
 get-conc = Get with automatic input concatenation
 get-bin = Get binary data
 get-full = Get binary fullwords
 put = All binary and character put operations
 put-seq = Put sequenced output record
 delete = Delete file
 rename = Rename file
 backup = Backup file

Declaration macro: \$XPO_IOB

Initialization macro: \$XPO_IOB_INIT

Control Blocks
STRING DESCRIPTORS

B.2 STRING DESCRIPTORS

The following table describes the fields of a string descriptor (see Section 6.1), and the literals associated with these descriptors.

Symbol	Type	Description
STR\$H_LENGTH	2 bytes	Number of characters in the string
STR\$B_DTYPE	byte	Atomic data type code:
STR\$K_DTYPE_XXX	0	Erroneous XPORT temporary string
STR\$K_DTYPE_T	14	ASCII text string
STR\$B_CLASS	byte	Descriptor class code:
STR\$K_CLASS_Z	0	unspecified
STR\$K_CLASS_F	1	fixed string
STR\$K_CLASS_D	2	dynamic string
STR\$K_CLASS_B	3	bounded string
STR\$K_CLASS_DB	190	dynamic bounded string
STR\$K_CLASS_XT	189	XPORT temporary string (dynamic)
STR\$A_POINTER	pointer	Pointer to the character string
STR\$H_MAXLEN	2 bytes	Length of the container string
STR\$H_PFXLEN	2 bytes	Length of the prefix string

Declaration macro: \$STR_DESCRIPTOR

Initialization macro: \$STR_DESC_INIT

Control Blocks
BINARY DATA DESCRIPTORS

B.3 BINARY DATA DESCRIPTORS

The following table describes the fields of a binary data descriptor (see Section 7.1), and the literals associated with these descriptors.

Symbol	Type	Description
XPO\$H_LENGTH	2 bytes	Length of the binary data units
XPO\$B_DTYPE	byte	Atomic data type code:
XPO\$K_DTYPE_BU	2	XPORT binary data (binary units)
XPO\$B_CLASS	byte	Descriptor class code:
XPO\$K_CLASS_Z	0	unspecified
XPO\$K_CLASS_F	1	fixed binary data
XPO\$K_CLASS_D	2	dynamic binary data
XPO\$K_CLASS_B	3	bounded binary data
XPO\$K_CLASS_DB	190	dynamic bounded binary data
XPO\$A_ADDRESS	pointer	Address of the binary data
XPO\$H_MAXLEN	2 bytes	Maximum length of the binary data
XPO\$H_PFXLEN	2 bytes	Length of the binary data prefix

Declaration macro: \$XPO_DESCRIPTOR

Initialization macro: \$XPO_DESC_INIT

Control Blocks
FILE SPECIFICATION PARSE BLOCK

B.4 FILE SPECIFICATION PARSE BLOCK

The following table describes all XPORT File Specification Parse Block fields and literals (see Section 3.6.2).

Symbol	Type	Description
XPO\$V_SPEC_STAT	16 bits	File specification indicators:
XPO\$V_DIR_NAME	bit	<directory-name> specified
XPO\$V_PPN	bit	[project,programmer] specified
XPO\$V_SFD	bit	[,,SFD] specified (TOPS-10 only)
XPO\$V_WILD_CARD	bit	wild-card somewhere in file-spec
XPO\$V_WILD_NODE	bit	wild-card node name
XPO\$V_WILD_DEV	bit	wild-card device name
XPO\$V_WILD_DIR	bit	wild-card in directory name
XPO\$V_WILD_NAME	bit	wild-card file name
XPO\$V_WILD_TYPE	bit	wild-card file type (extension)
XPO\$V_WILD_VER	bit	wild-card file version number
XPO\$V_WILD_ATTR	bit	wild-card file attributes
XPO\$T_NODE	desc	Network node name descriptor:
XPO\$H_NODE	2 bytes	length of the node name
XPO\$A_NODE	pointer	pointer to the node name
XPO\$T_DEVICE	desc	Device name descriptor:
XPO\$H_DEVICE	2 bytes	length of the device name
XPO\$A_DEVICE	pointer	pointer to the device name
XPO\$T_DIRECT	desc	Directory specification descriptor:
XPO\$H_DIRECT	2 bytes	length of the directory spec
XPO\$A_DIRECT	pointer	pointer to the directory spec
XPO\$H_PROJ_NUMB	2 bytes	Project number (binary)
XPO\$H_PGMN_NUMB	2 bytes	Programmer number (binary)
XPO\$T_FILE_NAME	desc	File name descriptor:
XPO\$H_FILE_NAME	2 bytes	length of the file name
XPO\$A_FILE_NAME	pointer	pointer to the file name
XPO\$T_FILE_TYPE	desc	File type (extension) descriptor:
XPO\$H_FILE_TYPE	2 bytes	length of the file type
XPO\$A_FILE_TYPE	pointer	pointer to the file type
XPO\$T_FILE_VER	desc	File version number descriptor:
XPO\$H_FILE_VER	2 bytes	length of the file version
XPO\$A_FILE_VER	pointer	pointer to the file version
continued on the next page		

Control Blocks
FILE SPECIFICATION PARSE BLOCK

Symbol	Type	Description
XPOST_FILE_PROT	desc	File protection descriptor (RSTS only):
XPO\$H_FILE_PROT	2 bytes	length of the protection
XPO\$A_FILE_PROT	pointer	pointer to the protection
XPOST_EXTRA	desc	File 'EXTRA' information descriptor:
XPO\$H_EXTRA	2 bytes	length
XPO\$A_EXTRA	pointer	pointer

Declaration macro: \$XPO_SPEC_BLOCK

Initialization macro: Not needed

APPENDIX C
COMPLETION CODES

The following table lists all XPORT completion codes together with their numeric values and corresponding message texts. Note that the numeric values are given for debugging purposes only; they should not be 'hard-coded' into a program.

COMPLETION CODES

Completion Code Name	Severity	BLISS-16/36	Code Value	BLISS-32	Message Text
STR\$ FAILURE	warning	0 \$0'000000'	2129920	\$X'208000'	unsuccessful completion
STR\$ NORMAL	success	1 \$0'000001'	2129921	\$X'208001'	normal completion
XPO\$ NORMAL	success	1 \$0'000001'	2129921	\$X'208001'	normal completion
XPO\$ CREATED	success	9 \$0'000011'	2129929	\$X'208009'	file was successfully created and opened
XPO\$ INCOMPLETE	success	17 \$0'000021'	2129937	\$X'208011'	incomplete amount of data read
XPO\$ NEW FILE	success	25 \$0'000031'	2129945	\$X'208019'	first read on concatenated file was successful
XPO\$ NEW PAGE	success	33 \$0'000041'	2129953	\$X'208021'	first read on a new page was successful
STR\$ END STRING	success	2049 \$0'004001'	2394113	\$X'248801'	end of string reached
STR\$ TRUNCATED	success	2057 \$0'004011'	2394121	\$X'248809'	string was truncated
STR\$ NOT TEMP	success	2065 \$0'004021'	2394129	\$X'248811'	not a temporary string
XPO\$ END FILE	warning	4096 \$0'010000'	2134016	\$X'209000'	end-of-file has been reached
XPO\$ BAD ADDR	error	8194 \$0'020002'	2138114	\$X'20A002'	invalid memory address
XPO\$ BAD ALIGN	error	8202 \$0'020012'	2138122	\$X'20A00A'	memory element not on a fullword boundary
XPO\$ BAD ARGS	error	8210 \$0'020022'	2138130	\$X'20A012'	invalid argument list
XPO\$ BAD CONCAT	error	8218 \$0'020032'	2138138	\$X'20A01A'	invalid concatenated file specification
XPO\$ BAD DELIM	error	8226 \$0'020042'	2138146	\$X'20A022'	invalid punctuation
XPO\$ BAD DESC	error	8234 \$0'020052'	2138154	\$X'20A02A'	invalid descriptor
XPO\$ BAD DEVICE	error	8242 \$0'020062'	2138162	\$X'20A032'	invalid device
XPO\$ BAD DFLT	error	8250 \$0'020072'	2138170	\$X'20A03A'	invalid default file specification
XPO\$ BAD DIRECT	error	8258 \$0'020102'	2138178	\$X'20A042'	invalid directory
XPO\$ BAD DTYPE	error	8266 \$0'020112'	2138186	\$X'20A04A'	invalid data type
XPO\$ BAD FORMAT	error	8274 \$0'020122'	2138194	\$X'20A052'	invalid record format
XPO\$ BAD TO OPT	error	8282 \$0'020132'	2138202	\$X'20A05A'	invalid I/O option
XPO\$ BAD LENGTH	error	8290 \$0'020142'	2138210	\$X'20A062'	invalid length
XPO\$ BAD NAME	error	8298 \$0'020152'	2138218	\$X'20A06A'	invalid file name
XPO\$ BAD NEW	error	8306 \$0'020162'	2138226	\$X'20A072'	invalid new file
XPO\$ BAD NODE	error	8314 \$0'020172'	2138234	\$X'20A07A'	invalid node
XPO\$ BAD ORG	error	8322 \$0'020202'	2138242	\$X'20A082'	invalid file organization
XPO\$ BAD PROMPT	error	8330 \$0'020212'	2138250	\$X'20A08A'	invalid prompt
XPO\$ BAD RECORD	error	8338 \$0'020222'	2138258	\$X'20A092'	invalid record
XPO\$ BAD REQ	error	8346 \$0'020232'	2138266	\$X'20A09A'	invalid request
XPO\$ BAD RLTD	error	8354 \$0'020242'	2138274	\$X'20A0A2'	invalid related file specification
XPO\$ BAD RSLT	error	8362 \$0'020252'	2138282	\$X'20A0AA'	invalid resultant file specification
XPO\$ BAD SEC	error	8370 \$0'020262'	2138290	\$X'20A0B2'	invalid file specification
XPO\$ BAD TYPE	error	8378 \$0'020272'	2138298	\$X'20A0BA'	invalid file type
XPO\$ BAD VER	error	8386 \$0'020302'	2138306	\$X'20A0C2'	invalid file version
XPO\$ CHANNEL	error	8394 \$0'020312'	2138314	\$X'20A0CA'	I/O channel assignment error
XPO\$ CLOSED	error	8402 \$0'020322'	2138322	\$X'20A0D2'	file is already closed
XPO\$ CONFLICT	error	8410 \$0'020332'	2138330	\$X'20A0DA'	conflicting options or attributes
XPO\$ CORRUPTED	error	8418 \$0'020342'	2138338	\$X'20A0E2'	file is corrupted
XPO\$ EXISTS	error	8426 \$0'020352'	2138346	\$X'20A0EA'	file already exists
XPO\$ FILE LOCK	error	8434 \$0'020362'	2138354	\$X'20A0F2'	file is locked
XPO\$ FREE MEM	error	8442 \$0'020372'	2138362	\$X'20A0FA'	dynamic memory deallocation error
XPO\$ GET MEM	error	8450 \$0'020402'	2138370	\$X'20A102'	dynamic memory allocation error
XPO\$ IN USE	error	8458 \$0'020412'	2138378	\$X'20A10A'	file is currently in use

continued on the next page

COMPLETION CODES

Completion Code Name	Severity	BLISS-16/36	Code Value	BLISS-32	Message Text
XPO\$ IO BUFFER	error	8466 \$0'020422'	2138386	\$X'20A112'	I/O buffering error
XPO\$ IO ERROR	error	8474 \$0'020432'	2138394	\$X'20A11A'	I/O error
XPO\$ MISSING	error	8482 \$0'020442'	2138402	\$X'20A122'	required parameter, option or attribute missing
XPO\$ NETWORK	error	8490 \$0'020452'	2138410	\$X'20A12A'	network error
XPO\$ NO ACCESS	error	8498 \$0'020462'	2138418	\$X'20A132'	file cannot be accessed
XPO\$ NO BACKUP	error	8506 \$0'020472'	2138426	\$X'20A13A'	file cannot be backed up
XPO\$ NO CHANNEL	error	8514 \$0'020502'	2138434	\$X'20A142'	all I/O channels are in use
XPO\$ NO CLOSE	error	8522 \$0'020512'	2138442	\$X'20A14A'	file cannot be closed
XPO\$ NO CONCAT	error	8530 \$0'020522'	2138450	\$X'20A152'	concatenated file specification not allowed
XPO\$ NO CREATE	error	8538 \$0'020532'	2138458	\$X'20A15A'	file cannot be created
XPO\$ NO DELETE	error	8546 \$0'020542'	2138466	\$X'20A162'	file cannot be deleted
XPO\$ NO DIRECT	error	8554 \$0'020552'	2138474	\$X'20A16A'	directory does not exist
XPO\$ NO FILE	error	8562 \$0'020562'	2138482	\$X'20A172'	file does not exist
XPO\$ NO MEMORY	error	8570 \$0'020572'	2138490	\$X'20A17A'	insufficient dynamic memory
XPO\$ NO OPEN	error	8578 \$0'020602'	2138498	\$X'20A182'	file cannot be opened
XPO\$ NO READ	error	8586 \$0'020612'	2138506	\$X'20A18A'	file cannot be read
XPO\$ NO RENAME	error	8594 \$0'020622'	2138514	\$X'20A192'	file cannot be renamed
XPO\$ NO SPACE	error	8602 \$0'020632'	2138522	\$X'20A19A'	insufficient space
XPO\$ NO SUBDIR	error	8610 \$0'020642'	2138530	\$X'20A1A2'	sub-directory does not exist
XPO\$ NO SUPPORT	error	8618 \$0'020652'	2138538	\$X'20A1AA'	requested function not supported
XPO\$ NO WRITE	error	8626 \$0'020662'	2138546	\$X'20A1B2'	file cannot be written
XPO\$ NOT CLOSED	error	8634 \$0'020672'	2138554	\$X'20A1BA'	file has not been closed
XPO\$ NOT EXPIRE	error	8642 \$0'020702'	2138562	\$X'20A1C2'	expiration date has not been reached
XPO\$ NOT INPUT	error	8650 \$0'020712'	2138570	\$X'20A1CA'	file is not open for input
XPO\$ NOT ONLINE	error	8658 \$0'020722'	2138578	\$X'20A1D2'	device is not online
XPO\$ NOT OPEN	error	8666 \$0'020732'	2138586	\$X'20A1DA'	file has not been opened
XPO\$ NOT OUTPUT	error	8674 \$0'020742'	2138594	\$X'20A1E2'	file is not open for output
XPO\$ OPEN	error	8682 \$0'020752'	2138602	\$X'20A1EA'	file is currently open
XPO\$ PREV ERROR	error	8690 \$0'020762'	2138610	\$X'20A1F2'	Program terminated due to previous error
XPO\$ PRIVILEGED	error	8698 \$0'020772'	2138618	\$X'20A1FA'	Privileged operation
XPO\$ PROTECTED	error	8706 \$0'021002'	2138626	\$X'20A202'	file protection denies access
XPO\$ PUT MSG	error	8714 \$0'021012'	2138634	\$X'20A20A'	message output error
XPO\$ REC LOCK	error	8722 \$0'021022'	2138642	\$X'20A212'	record is locked
XPO\$ RENAME NEW	error	8730 \$0'021032'	2138650	\$X'20A21A'	new file cannot be renamed
XPO\$ RENAME OLD	error	8738 \$0'021042'	2138658	\$X'20A222'	old file cannot be renamed
XPO\$ TRUNCATED	error	8746 \$0'021052'	2138666	\$X'20A22A'	record was truncated
XPO\$ WILDCARD	error	8754 \$0'021062'	2138674	\$X'20A232'	wildcard error
XPO\$ BAD ACCT	error	8762 \$0'021072'	2138682	\$X'20A23A'	invalid account attribute
XPO\$ BAD ATTR	error	8770 \$0'021102'	2138690	\$X'20A242'	invalid attribute
XPO\$ BAD DATA	error	8778 \$0'021112'	2138698	\$X'20A24A'	invalid data
XPO\$ BAD MEDIA	error	8786 \$0'021122'	2138706	\$X'20A252'	disk/tape cannot be read/written
XPO\$ BAD MEMORY	error	8794 \$0'021132'	2138714	\$X'20A25A'	free storage chain is invalid
XPO\$ BAD PROT	error	8802 \$0'021142'	2138722	\$X'20A262'	invalid protection attribute

continued on the next page

COMPLETION CODES

Completion Code Name	Severity	BLISS-16/36	Code Value	BLISS-32	Message Text
XPO\$ BAD_PTR	error	8810	\$0'021152'	2138730	\$X'20A26A' invalid character pointer
XPO\$ BAD_RECNUM	error	8818	\$0'021162'	2138738	\$X'20A272' invalid record number
XPO\$ BAD_SIZE	error	8826	\$0'021172'	2138746	\$X'20A27A' invalid size
XPO\$ BAD_TEMP	error	8834	\$0'021202'	2138754	\$X'20A282' invalid temporary file attribute
XPO\$ CHAN_USED	error	8842	\$0'021212'	2138762	\$X'20A28A' I/O channel is currently in use
XPO\$ HOST_ERROR	error	8850	\$0'021222'	2138770	\$X'20A292' host operating system error
XPO\$ NO_NODE	error	8858	\$0'021232'	2138778	\$X'20A29A' network node does not exist
XPO\$ NO_STACK	error	8866	\$0'021242'	2138786	\$X'20A2A2' insufficient stack space
XPO\$ SYS_ERROR	error	8874	\$0'021252'	2138794	\$X'20A2AA' unexpected operating system error
XPO\$ BAD_CLASS	error	8882	\$0'021262'	2138802	\$X'20A2B2' invalid descriptor class
XPO\$ NO_TEMP	error	8890	\$0'021272'	2138810	\$X'20A2BA' temporary file not permitted
XPO\$ FOREGROUND	error	8898	\$0'021302'	2138818	\$X'20A2C2' foreground jobs not permitted
XPO\$ NO_APPEND	error	8906	\$0'021312'	2138826	\$X'20A2CA' append function not permitted
XPO\$ NO_SEQ	error	8914	\$0'021322'	2138834	\$X'20A2D2' sequenced files not permitted
XPO\$ BAD_ORDER	error	8922	\$0'021332'	2138842	\$X'20A2DA' field is misplaced or duplicated
XPO\$ BAD_SYNTAX	error	8930	\$0'021342'	2138850	\$X'20A2E2' invalid syntax
STR\$ BAD_CHAR	error	10242	\$0'024002'	2402306	\$X'24A802' invalid character
STR\$ BAD_CLASS	error	10250	\$0'024012'	2402314	\$X'24A80A' invalid descriptor class
STR\$ BAD_DESC	error	10258	\$0'024022'	2402322	\$X'24A812' invalid string descriptor
STR\$ BAD_DTYPE	error	10266	\$0'024032'	2402330	\$X'24A81A' invalid descriptor data type
STR\$ BAD_LENGTH	error	10274	\$0'024042'	2402338	\$X'24A822' invalid string length
STR\$ BAD_MAXLEN	error	10282	\$0'024052'	2402346	\$X'24A82A' invalid maximum string length
STR\$ BAD_PATRN	error	10290	\$0'024062'	2402354	\$X'24A832' invalid pattern string
STR\$ BAD_PTR	error	10298	\$0'024072'	2402362	\$X'24A83A' invalid string pointer
STR\$ BAD_REQ	error	10306	\$0'024102'	2402370	\$X'24A842' invalid string request
STR\$ BAD_SOURCE	error	10314	\$0'024112'	2402378	\$X'24A84A' invalid source string
STR\$ BAD_STRNG1	error	10322	\$0'024122'	2402386	\$X'24A852' invalid primary string
STR\$ BAD_STRNG2	error	10330	\$0'024132'	2402394	\$X'24A85A' invalid secondary string
STR\$ BAD_TARGET	error	10338	\$0'024142'	2402402	\$X'24A862' invalid target string
STR\$ CONFLICT	error	10346	\$0'024152'	2402410	\$X'24A86A' conflicting string function arguments
STR\$ NO_SPACE	error	10354	\$0'024162'	2402418	\$X'24A872' insufficient space
STR\$ NO_STRING	error	10362	\$0'024172'	2402426	\$X'24A87A' no string specified
STR\$ NO_SUPPORT	error	10370	\$0'024202'	2402434	\$X'24A882' requested function not supported
STR\$ NO_TEMP	error	10378	\$0'024212'	2402442	\$X'24A88A' temporary string not permitted
STR\$ NULL_STRNG	error	10386	\$0'024222'	2402450	\$X'24A892' null string not permitted
STR\$ OUT_RANGE	error	10394	\$0'024232'	2402458	\$X'24A89A' integer value out of range
XPO\$ BAD_IOB	fatal	16388	\$0'040004'	2146308	\$X'20C004' invalid IOB
XPO\$ BAD_LOGIC	fatal	16396	\$0'040014'	2146316	\$X'20C00C' XPORT logic error detected
XPO\$ TERMINATE	fatal	16404	\$0'040024'	2146324	\$X'20C014' program terminated due to program request
STR\$ BAD_LOGIC	fatal	18436	\$0'044004'	2410500	\$X'24C804' XPORT string logic error detected

APPENDIX D
SAMPLE PROGRAM

This appendix presents a programming example that uses most of the XPORT facilities in a realistic context. The program reads a BLISS source file and merges into it the contents of any REQUIRE files referred to in the original.

Sample Program

```

MODULE MERGER ( IDENT = 'V1.0-1',           %TITLE 'BLISS REQUIRE File Merger'
               MAIN = MERGER
               %BLISS32(,ADDRESSING_MODE( EXTERNAL=LONG_RELATIVE ) )
               ) =
BEGIN

!++
!
! FACILITY:  BLISS User Program Library
!
! ABSTRACT:
!
!       This program "merges" the contents of any REQUIRE files declared
!       in a primary source file into that primary file. It also provides
!       a visually-distinctive header for each segment of REQUIRE-file
!       code.
!
! ENVIRONMENT:  User Mode with XPORT Facility; system independent.
!
! AUTHORS:  Ed Williams and Ward Clark,  CREATION DATE:  21 June 1979
!
!--

!
! TABLE OF CONTENTS:
!
FORWARD ROUTINE
    MERGER,                               ! Primary file merging control routine
    REQUIRE_DECL,                         ! REQUIRE declaration parser
    OPEN_REQ_FAIL;                       ! Open failure action routine

!
! INCLUDE FILES:
!

LIBRARY 'BLI:XPORT';                     ! XPORT definitions

!
! MACROS:
!

MACRO
    skip( string ) =
        (%STRING( %CHAR(cr), %CHAR(lf), string )) %;

!
! EQUATED SYMBOLS:
!

LITERAL
    true = 1,
    false = 0,
    ht = %O'11',
    cr = %O'15',
    lf = %O'12',
    ff = %O'14';

!
! OWN STORAGE:
!

OWN
    terminal : $XPO_IOB(),               ! IOB for terminal I/O
    primary_file : $XPO_IOB(),           ! IOB for primary input file
    require_file : $XPO_IOB(),           ! IOB for current REQ file
    output_file : $XPO_IOB();            ! IOB for output file

!
! EXTERNAL REFERENCES:
!

```

Sample Program

```

ROUTINE MERGER =                                ! Program entry point

!++
! FUNCTIONAL DESCRIPTION:
!
!   This routine performs all of file merging except for the parsing
!   of BLISS source statements (see the REQUIRE_DECL routine) and trying
!   alternate REQUIRE file default file-types (see the OPEN_REQ_FAIL
!   action routine).
!
! FORMAL PARAMETERS:
!
!   None
!
! IMPLICIT INPUTS:
!
!   None
!
! IMPLICIT OUTPUTS:
!
!   None
!
! COMPLETION CODES:
!
!   XPO$NORMAL - Successful completion
!
! SIDE EFFECTS:
!
!   None
!
!--

BEGIN
!+
!   Open the user's terminal for input/output and greet the user.
!-
!- $XPO_OPEN( IOB = terminal, FILE_SPEC = $XPO_INPUT );
!
! $XPO_PUT( IOB = terminal, STRING = skip('BLISS REQUIRE File Merger') );
!+
!   Ask the user for a BLISS source file-spec and open the file.
!-
!- WHILE 1 DO                                     ! Loop until the user gives a valid file-spec.
!-   BEGIN
!-     IF NOT $XPO_GET( IOB = terminal,
!-       PROMPT = skip('Enter name of BLISS source file (.BLI assumed): ') )
!-     THEN
!-       RETURN XPO$NORMAL;                         ! Exit if the user types ^Z.
!-     IF $XPO_OPEN( IOB = primary_file,
!-       FILE_SPEC = terminal[IOB$T_STRING],
!-       DEFAULT = '.BLI',
!-       FAILURE = XPO$IO_FAILURE )
!-     THEN
!-       EXITLOOP;
!-     END;
!+
!   Ask the user for an output file-spec and open the output file.
!-
!- IF NOT $XPO_GET( IOB = terminal,
!-   PROMPT = skip('Enter name of output file (*.BLI assumed): ') )
!- THEN
!-   RETURN XPO$NORMAL;                             ! Exit if the user types ^Z.
!-   output_file[IOB$V_SEQUENCED] =                 ! Make the output file sequenced
!-     .primary_file[IOB$V_SEQUENCED];             ! if the input file is sequenced.
!-   IF .terminal[IOB$H_STRING] GTR 0                ! If the user gave a file-spec,
!-   THEN                                             !
!-     $XPO_OPEN( IOB = output_file,                ! open the specified file for output.
!-       FILE_SPEC = terminal[IOB$T_STRING],
!-       DEFAULT = '*.BLI',
!-       RELATED = primary_file[IOB$T_RESULTANT],
!-       OPTION = OUTPUT )
!-   ELSE
!-     $XPO_OPEN( IOB = output_file,                ! Otherwise, open a temporary output file.
!-       FILE_SPEC = $XPO_TEMPORARY,
!-       OPTION = OUTPUT );
!+
!   Primary input file processing loop.
!-
!- WHILE $XPO_GET( IOB = primary_file ) DO          ! Read until end-of-file.
!-   BEGIN
!-     IF NOT REQUIRE_DECL()                        ! If this line is not a REQUIRE statement,

```

Sample Program

```

THEN
    $XPO_PUT( IOB = output_file,
              STRING = primary_file[IOB$T_STRING],
              SEQUENCE_NUMBER = .primary_file[IOB$G_SEQ_NUMB] )
!+
! REQUIRE file processing loop.
!-
ELSE
    BEGIN
    WHILE 1 DO
        IF $XPO_OPEN( IOB = require_file,
                     FAILURE = OPEN_REQ_FAIL )
        THEN
            ! REQUIRE_DECL sets up FILE_SPEC= and DEFAULT=.
            ! Action routine sets up alternate file types.
            EXITLOOP;

        $XPO_PUT( IOB = output_file,
                  STRING = %STRING( %CHAR(ff) ),
                  SEQUENCE_NUMBER = 0 );
        ! Put an SOS page mark before the REQUIRE file.

        WHILE $XPO_GET( IOB = require_file ) DO
            $XPO_PUT( IOB = output_file,
                      STRING = require_file[IOB$T_STRING],
                      SEQUENCE_NUMBER = .require_file[IOB$G_SEQ_NUMB] );
            ! Copy the entire REQUIRE file
            ! into the output file.

        $XPO_PUT( IOB = output_file,
                  STRING = %STRING( %CHAR(ff) ),
                  SEQUENCE_NUMBER = 0 );
        ! Put an SOS page mark after the REQUIRE file.

        $XPO_CLOSE( IOB = require_file );
        ! Then close the REQUIRE file
        END;
        ! and continue processing the primary source file.
    END;

!+
! Cleanup after reaching the end of the primary source file.
!-
$XPO_CLOSE( IOB = primary_file,
            OPTION = REMEMBER );
! Close the source file and remember its name.

$XPO_CLOSE( IOB = output_file );
! Close the output file (Note: its name will be
! remembered if it is a temporary file).

!+
! Rename the output file and backup the input file if the user
! did not provide an output file specification.
!-
IF .output_file[IOB$V_TEMPORARY]
THEN
    $XPO_BACKUP( OLD_IOB = primary_file,
                 NEW_IOB = output_file );

!+
! Tell the user that the file merging was successfully completed.
!-
$XPO_PUT( IOB = terminal,
          STRING = skip('REQUIRE file merging successfully completed') );

$XPO_CLOSE( IOB = terminal );

RETURN XPOS_NORMAL
END;

```

Sample Program

```

ROUTINE REQUIRE_DECL =
!++
! FUNCTIONAL DESCRIPTION:
!
!     This routine determines whether the current source line is a REQUIRE file declaration.
! FORMAL PARAMETERS:
!
!     None
! IMPLICIT INPUTS:
!
!     primary_file[IOBST_STRING] = current source line descriptor
! IMPLICIT OUTPUTS:
!
!     require_file[IOBSA_FILE_SPEC] = address of REQUIRE file name descriptor
! COMPLETION CODES:
!
!     true - line is a REQUIRE declaration
!     false - line is not a REQUIRE declaration
! SIDE EFFECTS:
!
!     None
!--

BEGIN
OWN
    statement : $STR_DESCRIPTOR( CLASS = DYNAMIC, STRING = (0,0) ),
    line_scan : $STR_DESCRIPTOR( CLASS = BOUNDED ),
    spaces : $STR_DESCRIPTOR( STRING = %STRING(' ',%CHAR(ht)) );
!+
! Create a local, upper-case version of the BLISS statement.
!-
$STR_COPY( STRING = primary_file[IOBST_STRING], TARGET = statement, OPTION = UP_CASE );
!+
! Initialize the BLISS source line descriptor.
!-
$STR_DESC_INIT( DESCRIPTOR = line_scan, CLASS = BOUNDED, STRING = statement );
!+
! Bypass any leading spaces and/or tabs.
!-
$STR_SCAN( REMAINDER = line_scan, SPAN = spaces, SUBSTRING = line_scan );
!+
! Test for a REQUIRE statement.
!-
$STR_SCAN( REMAINDER = line_scan, STOP = spaces, SUBSTRING = line_scan );
IF NOT $STR_EQ( STRING1 = line_scan, STRING2 = 'REQUIRE' )
THEN
    RETURN false;
!+
! Locate the name of the REQUIRE file in the source line.
!-
IF $STR_SCAN( REMAINDER = line_scan, STOP = '', SUBSTRING = line_scan ) NEQ STR$END_STRING
THEN
    BEGIN
        line_scan[STR$H_LENGTH] = .line_scan[STR$H_LENGTH] + 1;
        IF $STR_SCAN( REMAINDER = line_scan, STOP = '', SUBSTRING = line_scan ) NEQ STR$END_STRING
        THEN
            BEGIN
!+
! Put the REQUIRE file name into the REQUIRE-file IOB.
!-
                $XPO_IOB_INIT( IOB = require_file, FILE_SPEC = line_scan, DEFAULT = '.REQ' );
!+
! Return a success code to the caller.
!-
                RETURN true
            END;
        END;
!+
! Tell the user about an invalid REQUIRE declaration and terminate program execution.
!-
        $XPO_PUT_MSG( STRING = 'Cannot find a file-spec in the following BLISS REQUIRE statement:',
                     STRING = primary_file[IOBST_STRING],
                     SEVERITY = FATAL )
! NOTE: Fatal message terminates program.
    END;
END;

```

Sample Program

```

ROUTINE OPEN_REQ_FAIL( function_code, primary_code, secondary_code, iob ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This failure action routine is used to supply a series of default
!     file types to open a REQUIRE file.
!
! FORMAL PARAMETERS:
!
!     function_code - action routine function code - ignored
!     primary_code - primary $XPO_OPEN failure completion code
!     secondary_code - secondary $XPO_OPEN failure completion code
!     iob - address of the REQUIRE file IOB
!
! IMPLICIT INPUTS:
!
!     iob[IOB$Z_USER] - number of times this routine has been called
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     XPO$NORMAL - alternate REQUIRE file successfully opened
!
! SIDE EFFECTS:
!
!     This routine terminates program execution if a REQUIRE file
!     cannot be successfully opened.
!
!--

BEGIN

MAP
    iob : REF $XPO_IOB();

BIND
    recursion_level = iob[IOB$Z_USER];

EXTERNAL ROUTINE
    XPO$FAILURE;                                ! Default I/O failure action routine
!+
! Perform default file type sequencing only for "file not found" errors.
!-
IF .primary_code NEQ XPO$_NO_FILE
THEN
    BEGIN
        XPO$FAILURE( .function_code, .primary_code, .secondary_code, .iob );
        RETURN .primary_code
    END;
!+
! Increment the recursion counter.
!-
recursion_level = .recursion_level + 1;
!+
! Try a new default file type for the REQUIRE file.
!-
CASE .recursion_level FROM 1 TO 8 OF
    SET
        [ 1 ] : iob[IOB$A_DEFAULT] = %ASCID'.R16';
        [ 2 ] : iob[IOB$A_DEFAULT] = %ASCID'.R32';
        [ 3 ] : iob[IOB$A_DEFAULT] = %ASCID'.R36';
        [ 4 ] : iob[IOB$A_DEFAULT] = %ASCID'.BLI';
        [ 5 ] : iob[IOB$A_DEFAULT] = %ASCID'.B16';
        [ 6 ] : iob[IOB$A_DEFAULT] = %ASCID'.B32';
        [ 7 ] : iob[IOB$A_DEFAULT] = %ASCID'.B36';
        [ 8 ] : $XPO_PUT_MSG( STRING = 'cannot open the file named in the following REQUIRE declaration:',
                                STRING = primary_file[IOB$T_STRING],
                                SEVERITY = FATAL );      ! NOTE: Fatal message terminates program.
    TES;
!+
! Return the primary failure completion code to the caller.
!-
RETURN .primary_code
END;

END
ELUDOM

```


APPENDIX E

ACTION ROUTINES

E.1	ACTION-ROUTINE CALLS AND RETURNS	E-1
E.1.1	Action Routine Calls	E-1
E.1.2	Action Routine Return Values	E-3
E.2	XFAIL.BLI FAILURE-ACTION ROUTINE LISTING	E-4
E.3	SFAIL.BLI FAILURE-ACTION ROUTINE LISTING	E-15

APPENDIX E

ACTION ROUTINES

This appendix contains information related to the coding of user-supplied action routines, and presents the XPORT default failure-action routines, XPOSFAILURE and STR\$FAILURE, as examples of and models for such routines.

This appendix is intended for reference use. A tutorial discussion of action routines and their use is given earlier in this manual, beginning with Chapter 3.

E.1 ACTION-ROUTINE CALLS AND RETURNS

Any action routine that applies to a given operation is called by the XPORT function in question just before it returns control to the original caller. The routine is passed several values related to the operation. (A success-action routine is called in the case of a success completion code; a failure-action routine is called for any other code including a warning code.)

On completing its processing, the action routine returns a value to its caller that becomes the completion code returned to the original call site.

E.1.1 Action Routine Calls

All action routines called by XPORT functions are passed parameters which describe the function, its arguments, success or failure completion codes, etc. Although the number of parameters and the meaning of each parameter varies somewhat depending on the calling function, the general format of an action-routine call is:

```
routine-address( function-code, primary-code,  
                 secondary-code, action-argument, ... )
```

Table E.1 describes the action-routine arguments that are actually passed by each XPORT function.

Action Routines
ACTION-ROUTINE CALLS AND RETURNS

Table E.1
Action Routine Arguments

Function	Argument Description
\$XPO_OPEN \$XPO_CLOSE \$XPO_GET \$XPO_PUT \$XPO_DELETE \$XPO_RENAME \$XPO_BACKUP	1: XPO\$K_IO 2: Primary completion code 3: Secondary completion code 4: Address of IOB
\$XPO_PARSE_SPEC	1: XPO\$K_PARSE 2: Primary completion code 3: Secondary completion code 4: Address of file-spec string descriptor
\$XPO_GET_MEM	1: XPO\$K_GET_MEM 2: Primary completion code 3: Secondary completion code 4: Address of memory-request descriptor
\$XPO_FREE_MEM	1: XPO\$K_FREE_MEM 2: Primary completion code 3: Secondary completion code 4: Address of allocated-memory descriptor
\$XPO_PUT_MSG	1: XPO\$K_PUT_MSG 2: Primary completion code 3: Secondary completion code 4: Serverity of first message
\$STR_EQL \$STR_NEQ \$STR_LSS \$STR_LEQ \$STR_GEQ \$STR_GTR \$STR_COMPARE	1: STR\$K_COMPARE 2: Primary completion code 3: Secondary completion code 4: Address of relationship string descriptor 5: Address of primary-string descriptor 6: Address of secondary-string descriptor

Action Routines
ACTION-ROUTINE CALLS AND RETURNS

Table E.1 (Continued)
Action Routine Arguments

Function	Argument Description
\$STR_COPY	1: STR\$K_COPY 2: Primary completion code 3: Secondary completion code 4: 0 5: Address of source-string descriptor 6: Address of target-string descriptor
\$STR_APPEND	1: STR\$K_APPEND 2: Primary completion code 3: Secondary completion code 4: 0 5: Address of source-string descriptor 6: Address of target-string descriptor
\$STR_SCAN	1: STR\$K_SCAN 2: Primary completion code 3: Secondary completion code 4: Internal \$STR_SCAN function code 5: Address of source-string descriptor 6: Address of pattern-string descriptor
\$STR_BINARY	1: STR\$K_BINARY 2: Primary completion code 3: Secondary completion code 4: Internal \$STR_BINARY function code 5: Address of source-string descriptor 6: Address of internal result area

E.1.2 Action Routine Return Values

An action routine must have a return value. This value is returned to the original call site as the final completion code of the XPORT function.

Action Routines ACTION-ROUTINE CALLS AND RETURNS

Action routines for I/O functions are passed the primary and secondary I/O completion codes in the associated IOB as well as in the action-routine argument list. The corresponding IOB fields are IOB\$G_COMP_CODE and IOB\$G_2ND_CODE respectively. XPORT updates the contents of IOB\$G_COMP_CODE (only) to agree with the completion code returned by the action routine -- assuming it differs from the original. The action routine may modify the content of IOB\$G_2ND_CODE as necessary before returning.

E.2 XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

The source module XFAIL, reproduced below, contains the default XPORT failure-action routine XPO\$FAILURE, together with the four function-specific routines called by XPO\$FAILURE. These supporting routines are XPO\$IO_FAILURE for I/O functions, XPO\$GM_FAILURE for the get-memory function, XPO\$FM_FAILURE for the free-memory function, and XPO\$PM_FAILURE for the put-message function.

Note that XPO\$FAILURE terminates program execution for any error condition, after calling one of the function-specific routines. The function-specific routines simply issue appropriate error messages. Since these routines have the same parameter list as XPO\$FAILURE, any of them can be used directly in place of the default failure-action routine (e.g., FAILURE = XPO\$IO_FAILURE for I/O calls).

Action Routines

XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

MODULE XFAIL ( IDENT = 'V1.0-14'                %TITLE 'XPOSFAILURE - Default Failure Action Routine'
               %BLISS32( ,ADDRESSING MODE( EXTERNAL=LONG RELATIVE ) )
               %BLISS36( ,ENTRY( XPOSFAILURE, XPOSIO FAILURE, XPOSFS FAILURE, XPOSGM_FAILURE,
                                XPOSFM_FAILURE, XPOSPM_FAILURE ),OTS='' )
               ) =

BEGIN

!
!               COPYRIGHT (c) 1981 BY
!               DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
! ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
! INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
! COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
! OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
! TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!
!++
!
! FACILITY:  BLISS Library
!
! ABSTRACT:
!
!       This module is the default XPORT failure action routine.
!
! ENVIRONMENT:  User Mode - system-independent
!
! AUTHOR:  Ward Clark,  CREATION DATE:  11 July 1978
!
!--

!
! TABLE OF CONTENTS:
!
!
! FORWARD ROUTINE
!   XPOSFAILURE;                                ! Failure action routine dispatcher
!
! IF %BLISS(BLISS16) %THEN
!   EXTERNAL ROUTINE
! %ELSE
!   FORWARD ROUTINE
! %FI
!   XPOSIO FAILURE,                            ! XPORT I/O failure action routine
!   XPOSFS FAILURE,                            ! $XPO_PARSE_SPEC failure action routine
!   XPOSGM FAILURE,                           ! $XPO_GET_MEM failure action routine
!   XPOSFM FAILURE,                           ! $XPO_FREE_MEM failure action routine
!   XPOSPM_FAILURE;                           ! $XPO_PUT_MSG failure action routine
!
!
! INCLUDE FILES:
!
!
! LIBRARY 'XPORT' ;                            ! Public XPORT control block and macro definitions
! LIBRARY 'XPOSYS' ;                          ! Internal XPORT macro definitions
!
!   $XPO_SYS_TEST( $ALL )
!
!
! MACROS:
!
!
! EQUATED SYMBOLS:
!
!
! PSECT DECLARATIONS:
!
!   $XPO_PSECTS                                ! Declare XPORT PSECT names and attributes
!
!

```

Action Routines
XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```
! OWN STORAGE:
!
!   See each function-specific failure action routine.
!
! EXTERNAL REFERENCES:
!
!   See each function-specific failure action routine.
```

Action Routines XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE XPO$FAILURE( function_code, primary_code, secondary_code, action_argument ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine dispatches a failure action routine call to the
!     appropriate processing routine for the function which failed.
!
! FORMAL PARAMETERS:
!
!     function_code - XPORT failure action routine function code
!     primary_code - primary failure completion code
!     secondary_code - secondary failure completion code
!     action_argument - function-specific action routine argument
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! ROUTINE VALUE:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     This routine returns to the caller if the completion code
!     severity is SUCCESS or WARNING. If the severity is ERROR or
!     FATAL, this routine terminates program execution.
!
!--
!
! BEGIN
!
! LOCAL
!     action_routine;
!
! Select the appropriate failure processing routine.
!
!     action_routine = ( CASE .function_code FROM 1 to XPO$K_PUT_MSG OF
!                         SET
!                         [ XPO$K_IO ] :          XPO$IO_FAILURE;
!                         [ XPO$K_PARSE ] :       XPO$PS_FAILURE;
!                         [ XPO$K_GET_MEM ] :     XPO$GM_FAILURE;
!                         [ XPO$K_FREE_MEM ] :    XPO$FM_FAILURE;
!                         [ XPO$K_PUT_MSG ] :     XPO$PM_FAILURE;
!                         TES );
!
! Call the action routine.
!
!     (.action_routine)( .function_code, .primary_code, .secondary_code, .action_argument );
!
! Terminate program execution or return to the caller.
!
!     IF .primary_code OR
!     THEN .primary_code<0,3,0> EQL XPO$_WARNING
!     RETURN .primary_code
!     ELSE
!         $XPO_TERMINATE( CODE = XPO$PREV_ERROR )
!     END;
!
! If the completion code is a success code
! or has a WARNING severity,
! return the input completion code to the caller.
! Otherwise, terminate program execution.
!
! $XPO_MODULE( XFAIL1 )

```


Action Routines XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE XPO$IO_FAILURE( function_code, primary_code, secondary_code, iob ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!         ? error opening 'file-spec' as input
!         -     primary completion code message
!         -     secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - failure action routine function code (XPO$K_IO)
!     primary_code  - primary I/O failure completion code
!     secondary_code - secondary failure completion code
!     iob           - address of the associated IOB
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--

BEGIN

MAP
    iob : REF $XPO_IOB();

BIND
    file_spec = .iob[IOB$A_FILE_SPEC] : $STR_DESCRIPTOR(),
    resultant = iob[IOB$T_RESULTANT] : $STR_DESCRIPTOR();

OWN
    initial_text : $STR_DESCRIPTOR( STRING = 'error ' ),
    open_text : $STR_DESCRIPTOR( STRING = 'opening ' ),
    close_text : $STR_DESCRIPTOR( STRING = 'closing ' ),
    delete_text : $STR_DESCRIPTOR( STRING = 'deleting ' ),
    rename_text : $STR_DESCRIPTOR( STRING = 'renaming ' ),
    backup_text : $STR_DESCRIPTOR( STRING = 'backing up ' ),
    put_text : $STR_DESCRIPTOR( STRING = 'writing to ' ),
    get_text : $STR_DESCRIPTOR( STRING = 'reading from ' ),
    auto_open_text : $STR_DESCRIPTOR( STRING = 'auto-opening ' ),
    auto_close_text : $STR_DESCRIPTOR( STRING = 'auto closing ' ),
    bad_func_text : $STR_DESCRIPTOR( STRING = 'invalid operation on ' ),
    input_text : $STR_DESCRIPTOR( STRING = ' for input ' ),
    and_output_text : $STR_DESCRIPTOR( STRING = ' and output' ),
    output_text : $STR_DESCRIPTOR( STRING = ' for output' ),
    to_text : $STR_DESCRIPTOR( STRING = ' to ' );

EXTERNAL ROUTINE
    XST$INIT_MSG : NOVALUE,                ! Failure message initialization routine
    XST$STRING : NOVALUE,                  ! Append string to failure message routine
    XST$QUOTED : NOVALUE,                  ! Append quoted string to failure message routine

EXTERNAL
    XST$MESSAGE;                          ! Failure message string descriptor

!
! Don't issue a message for SUCCESS or WARNING conditions.
!

IF .primary_code OR
    .primary_code < 0, 3, 0 > EQL XPO$_WARNING
THEN
    RETURN .primary_code;

!
! Create the initial function-specific message.
!

```

XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

IF .lob[IOB$B_FUNCTION] LEQ IOB$K_PUT      ! All messages except "invalid function" start with
"error".                                     !
THEN                                         !
    XST$INIT_MSG( initial_text );           !
CASE .lob[IOB$B_FUNCTION]                   !
FROM IOB$K_OPEN TO IOB$K_PUT OF            ! Use the XPORT function code to select
SET                                          ! the next part of the message.
[ OUTRANGE ] : XST$INIT MSG( bad_func_text );
[ IOB$K_OPEN ] : XST$STRING( open_text );
[ IOB$K_CLOSE ] : XST$STRING( close_text );
[ IOB$K_DELETE ] : XST$STRING( delete_text );
[ IOB$K_RENAME ] : XST$STRING( rename_text );
[ IOB$K_BACKUP ] : XST$STRING( backup_text );
[ IOB$K_PUT ] : XST$STRING( put_text );
[ IOB$K_GET ] :
    IF .lob[IOB$V_AUTO_CONC]               ! If input switching is in progress,
    THEN                                    ! special open and close text will be needed.
        IF .lob[IOB$V_OPEN]
        THEN
            XST$STRING( auto_close_text )
        ELSE
            XST$STRING( auto_open_text )
        ELSE
            XST$STRING( get_text );         ! Otherwise, use the normal input failure text.
    TES;
IF .resultant[STR$H_LENGTH] NEQ 0          ! Put the best file name into the message:
THEN                                       !
    XST$QUOTED( resultant )              !
ELSE                                      !
    XST$QUOTED( file_spec );             !
                                           !
SELECTONE .lob[IOB$B_FUNCTION] OF        !
SET                                       !
[ IOB$K_OPEN, IOB$K_CLOSE ] :             ! Special OPEN/CLOSE message suffix:
    IF .lob[IOB$V_INPUT]                 ! Indicate whether this is an
    THEN                                  ! input or an output file.
        BEGIN
            XST$STRING( input_text );
            IF .lob[IOB$V_OUTPUT]
            THEN
                XST$STRING( and_output_text );
            END
        ELSE
            XST$STRING( output_text );
    [ IOB$K_RENAME ] :                   ! Special RENAME message suffix:
        BEGIN
            BIND
            new_lob = .lob[IOB$A_ASSOC_IOB] :
                                $XPO_IOB();
            new_result = new_lob[IOB$T_RESULTANT] :
                                $STR_DESCRIPTOR();
            XST$STRING( to_text );
            IF .new_result[STR$H_LENGTH] NEQ 0
            THEN
                XST$QUOTED( new_result )   ! Add the new resultant file-spec
            ELSE                             !
                XST$QUOTED(.new_lob[IOB$A_FILE_SPEC]); ! or the new primary file-spec.
        END
    [ SEND_FAILURE ] :
        BEGIN
            IF .lob[IOB$G_COMP_CODE] EQL XPO$ BAD_NEW
            AND .lob[IOB$G_2ND_CODE] EQL 0 ! Test for special linked RENAME messages.
            THEN
                BEGIN
                    $XPO_PUT_MSG( STRING = XST$MESSAGE,
                                CODE = XPO$ BAD_NEW,
                                CODE = .new_lob[IOB$G_COMP_CODE],
                                CODE = .new_lob[IOB$G_2ND_CODE],
                                FAILURE = 0 );
                    RETURN .primary_code     ! Return after a special RENAME message.
                END;
            END;
        TES;
    $XPO_PUT_MSG( STRING = XST$MESSAGE,
                  CODE = .lob[IOB$G_COMP_CODE],
                  CODE = .lob[IOB$G_2ND_CODE],

```

Action Routines
XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```
        FAILURE = 0 );  
  
!  
! Return to the caller.  
!  
        RETURN .primary_code                ! Return the original completion code to the caller.  
        END;  
SXPO_MODULE( XFAIL2 )
```

Action Routines

XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE XPO$PS_FAILURE( function_code, primary_code, secondary_code, file_spec ) =
!++
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!         ? error parsing 'file-spec'
!         - primary completion code message
!         - secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - failure action routine function code (XPO$K_PARSE)
!     primary_code  - primary $XPO_PARSE_SPEC failure completion code
!     secondary_code - secondary failure completion code
!     file_spec     - address of file-spec string descriptor
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! OWN
!     initial_text : $STR_DESCRIPTOR( STRING = 'error parsing ' );
!
! EXTERNAL ROUTINE
!     XST$INIT MSG : NOVALUE,
!     XST$QUOTED : NOVALUE;
!
! EXTERNAL
!     XST$MESSAGE;
!
! Create the initial function-specific message.
!
!     XST$INIT MSG( initial_text );
!     XST$QUOTED( .file_spec );
!
! Send a multi-line failure message to the user.
!
!     $XPO_PUT_MSG( STRING = XST$MESSAGE,
!                   CODE = .primary_code,
!                   CODE = .secondary_code,
!                   FAILURE = 0 );
!
! Tell the user that $XPO_PARSE_SPEC failed
! and what the failure was.
!
! Return to the caller.
!
!     RETURN .primary_code
!
! Return the original completion code to the caller.
!
! END;
!
! $XPO_MODULE( XFAIL3 )

```

Action Routines XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE XPO$GM_FAILURE( function_code, primary_code, secondary_code, descriptor ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!       This routine sends the user a message sequence similar to the following:
!
!               ? dynamic memory allocation error
!               -   primary completion code message
!               -   secondary completion code message
!
! FORMAL PARAMETERS:
!
!       function_code - XPORT failure action routine function code (ignored)
!       primary_code - primary $XPO GET MEM failure completion code
!       secondary_code - secondary failure completion code
!       descriptor - address of $XPO_GET_MEM request descriptor
!
! IMPLICIT INPUTS:
!
!       None
!
! IMPLICIT OUTPUTS:
!
!       None
!
! COMPLETION CODES:
!
!       .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!       None
!
!--
!
! BEGIN
!
! MAP
!       descriptor : REF $STR_DESCRIPTOR();           ! Redefine the descriptor argument.
!
! ! Send a three-line error message to the user.
!
!       $XPO_PUT_MSG( CODE = XPOS_GET_MEM,           ! Tell the user that $XPO_GET_MEM failed
!                     CODE = .primary_code,           ! and what the failure was.
!                     CODE = .secondary_code,
!                     FAILURE = 0 );
!
!       RETURN .primary_code                          ! Return the original completion code to the caller.
!
! END;
!
! $XPO_MODULE( XFAIL4 )

```

Action Routines XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE XPO$FM_FAILURE( function_code, primary_code, secondary_code, descriptor ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!         ? dynamic memory deallocation error
!         -   primary completion code message
!         -   secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - XPORT failure action routine function code (ignored)
!     primary_code - primary $XPO_FREE_MEM failure completion code
!     secondary_code - secondary failure completion code
!     descriptor - address of $XPO_FREE_MEM request descriptor
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! MAP
!     descriptor : REF $STR_DESCRIPTOR();           ! Redefine the descriptor argument.
!
! ! Send a three-line error message to the user.
!
!     $XPO_PUT_MSG( CODE = XPO$FREE_MEM,
!                   CODE = .primary_code,
!                   CODE = .secondary_code,
!                   FAILURE = 0 );
!
!     ! Tell the user that $XPO_FREE_MEM failed
!     ! and what the failure was.
!
!     RETURN .primary_code
!
!     ! Return the original completion code to the caller.
!
! END;
!
! $XPO_MODULE( XFAIL5 )

```

Action Routines XFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE XPO$PM_FAILURE( function_code, primary_code, secondary_code, actual_severity ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!           ? message output error
!           -   primary completion code message
!           -   secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - XPORT failure action routine function code (ignored)
!     primary_code  - primary $XPO_PUT_MSG failure completion code
!     secondary_code - secondary failure completion code
!     actual_severity - actual severity of 1st message
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! Send a three-line error message to the user.
!
!     $XPO_PUT_MSG( CODE = XPO$PUT_MSG,           ! Tell the user that PUT_MESSAGE failed
!                  CODE = .primary_code,         ! and what the failure was
!                  CODE = .secondary_code,       !
!                  FAILURE = 0 );                ! (blocking failure recursion).
!
!     RETURN .primary_code                      ! Return the original completion code to the caller.
!
! END;
! ELUDOM

```

Action Routines
SF_{AIL}.BLI FAILURE-ACTION ROUTINE LISTING

E.3 SF_{AIL}.BLI FAILURE-ACTION ROUTINE LISTING

The source module SF_{AIL}, reproduced below, contains the default XPORT failure-action routine STR\$FAILURE, together with the five function-specific routines called by STR\$FAILURE. These supporting routines are STR\$X_FAILURE for comparison functions, STR\$C_FAILURE for the copy function, STR\$A_FAILURE for the append function, STR\$S_FAILURE for the scan function, and STR\$B_FAILURE for the conversion-to-binary function.

Note that STR\$FAILURE terminates program execution for any error condition, after calling one of the function-specific routines. The function-specific routines simply issue appropriate error messages. Since these routines have the same parameter list as STR\$FAILURE, any of them can be used directly in place of the default failure-action routine (e.g., FAILURE = STR\$X_FAILURE for string-comparison calls).

Action Routines

SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

MODULE SFAIL ( IDENT = 'V1.0-8'          %TITLE 'STR$FAILURE - String Failure Action Routine'
               %BLISS32( ,ADDRESSING MODE( EXTERNAL=LONG RELATIVE ) )
               %BLISS36( ,ENTRY( STR$FAILURE, STR$X FAILURE,
                                STR$C FAILURE, STR$A FAILURE,
                                STR$S_FAILURE, STR$B_FAILURE ),OTS='' )
               ) =

BEGIN

!
!               COPYRIGHT (c) 1981 BY
!               DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASS.
!
! THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED
! ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE
! INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER
! COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
! OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY
! TRANSFERRED.
!
! THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
! AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT
! CORPORATION.
!
! DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
! SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.
!

!++
!
! FACILITY: BLISS Library
!
! ABSTRACT:
!
!       This module includes all standard String Handling failure
!       action routine processing.
!
! ENVIRONMENT: User mode - multiple host operating/file systems
!
! AUTHOR: Ward Clark, CREATION DATE: 28 February 1980
!
!--

!
! TABLE OF CONTENTS:
!
!
! FORWARD ROUTINE
!   STR$FAILURE;                                ! Failure action routine dispatcher
! %IF %BLISS(BLISS16) %THEN
!   EXTERNAL ROUTINE
! %ELSE
!   FORWARD ROUTINE
! %FI
!   STR$X FAILURE,                                ! String comparison failure action routine
!   STR$C FAILURE,                                ! $STR COPY failure action routine
!   STR$A FAILURE,                                ! $STR_APPEND failure action routine
!   STR$S_FAILURE,                                ! $STR_SCAN failure action routine
!   STR$B_FAILURE;                                ! $STR_BINARY failure action routine
!
!
! INCLUDE FILES:
!
!
! LIBRARY 'BLI:XPORT' ;                          ! Public XPORT control block and macro definitions
! LIBRARY 'XPOSYS' ;                             ! Internal XPORT macro definitions
!
!   $XPO_SYS_TEST( $ALL )
!
!
! MACROS:
!
!
! EQUATED SYMBOLS:
!
!
! PSECT DECLARATIONS:
!
!
!   $XPO_PSECTS                                  ! Declare XPORT PSECT names and attributes

```

Action Routines
SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```
!  
! OWN STORAGE:  
!  
!   See each function-specific failure action routine.  
!  
! EXTERNAL REFERENCES:  
!  
!   See each function-specific failure action routine.
```

Action Routines

SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE STR$FAILURE( function_code, primary_code, secondary_code, action_arg1, action_arg2,
action_arg3 ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine dispatches a failure action routine call to the
!     appropriate processing routine for the function which failed.
!
! FORMAL PARAMETERS:
!
!     function_code - String Handling failure action routine function code
!     primary_code - primary failure completion code
!     secondary_code - secondary failure completion code
!     action_arg1,2,3 - function-specific action routine arguments
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! ROUTINE VALUE:
!
!     primary completion code (value passed as a formal parameter)
!
! SIDE EFFECTS:
!
!     This routine returns to the caller if the completion code
!     severity is SUCCESS or WARNING. If the severity is ERROR or
!     FATAL, this routine terminates program execution.
!
!--
!
! BEGIN
!
! LOCAL
!     action_routine;                                ! Address of action routine to be called
!
! Select the appropriate failure processing routine.
!
!     action_routine = ( CASE .function_code FROM 1 TO STR$K_BINARY OF
!                         SET
!                         [ STR$K_COMPARE ] :    STR$X_FAILURE;
!                         [ STR$K_COPY ] :       STR$C_FAILURE;
!                         [ STR$K_APPEND ] :     STR$A_FAILURE;
!                         [ STR$K_SCAN ] :       STR$S_FAILURE;
!                         [ STR$K_BINARY ] :     STR$B_FAILURE;
!                         TES );
!
! Call the action routine.
!
!     (.action_routine)( .function_code, .primary_code, .secondary_code, .action_arg1, .action_arg2,
!     .action_arg3 );
!
! Terminate program execution or return to the caller.
!
! IF .primary_code OR
!     .primary_code<0,3,0> EQL XPOS_WARNING          ! If the completion code is a success code
! THEN                                                ! or has a WARNING severity,
!     RETURN .primary_code                          ! return the input completion code to the caller.
! ELSE
!     $XPO_TERMINATE( CODE = XPOS_PREV_ERROR )        ! Otherwise, terminate program execution.
!
! END;
!
! $XPO_MODULE( SFAIL )

```

Action Routines

SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE STR$X_FAILURE( function_code, primary_code, secondary_code, relation, string1, string2 ) =
|++
|
| FUNCTIONAL DESCRIPTION:
|
|     This routine sends the user a message sequence similar to the following:
|
|           ? comparison error: 'string1' equal to 'string2'
|           -     primary completion code message
|           -     secondary completion code message
|
| FORMAL PARAMETERS:
|
|     function_code - action routine function code (STR$K_COMPARE)
|     primary_code  - primary completion code
|     secondary_code - secondary completion code
|     relation      - comparison relationship string (e.g., 'compared to ')
|     string1       - address of primary string descriptor
|     string2       - address of secondary string descriptor
|
| IMPLICIT INPUTS:
|
|     None
|
| IMPLICIT OUTPUTS:
|
|     None
|
| COMPLETION CODES:
|
|     .primary_code - primary completion code passed by caller
|
| SIDE EFFECTS:
|
|     None
|--
|
| BEGIN
|
| OWN
|     initial_text : $STR_DESCRIPTOR( STRING = 'comparison error:  ' );
|
| EXTERNAL ROUTINE
|     XST$INIT MSG : NOVALUE,
|     XST$STRING  : NOVALUE,
|     XST$QUOTED  : NOVALUE;
|
| EXTERNAL
|     XST$MESSAGE;
|
| ! Failure message initialization routine
| ! Append string to failure message routine
| ! Append quoted string to failure message routine
|
| ! Failure message string descriptor
|
| Create the initial function-specific message.
|
|     XST$INIT MSG( initial_text );
|     XST$QUOTED( .string1 );
|     XST$STRING( .relation );
|     XST$QUOTED( .string2 );
|
| Send a multi-line failure message to the user.
|
|     $XPO_PUT_MSG( STRING = XST$MESSAGE,
|                   CODE = .primary_code,
|                   CODE = .secondary_code,
|                   FAILURE = 0 );
|
| ! Function-specific message
| ! Primary failure completion code
| ! Secondary failure completion code
|
| Return to the caller.
|
|     RETURN .primary_code
|
| END;
|
| $XPO_MODULE( SFAIL2 )

```

Action Routines SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE STR$C_FAILURE( function_code, primary_code, secondary_code, dummy, string, target ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!           ? error copying 'string'
!           -   primary completion code message
!           -   secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - action routine function code (STR$K_COMPARE)
!     primary_code  - primary completion code
!     secondary_code - secondary completion code
!     dummy         - dummy argument (not used)
!     string        - address of source string descriptor
!     target        - address of target string descriptor
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! OWN
!     initial_text : $STR_DESCRIPTOR( STRING = 'error copying ' );
!
! EXTERNAL ROUTINE
!     XST$INIT_MSG : NOVALUE,                ! Failure message initialization routine
!     XST$QUOTED   : NOVALUE;                ! Append quoted string to failure message routine
!
! EXTERNAL
!     XST$MESSAGE;                ! Failure message string descriptor
!
! Create the initial function-specific message.
!
!     XST$INIT_MSG( initial_text );
!     XST$QUOTED( .string );
!
! Send a multi-line failure message to the user.
!
!     $XPO_PUT_MSG( STRING = XST$MESSAGE,                ! Function-specific message
!                   CODE = .primary_code,                ! Primary failure completion code
!                   CODE = .secondary_code,              ! Secondary failure completion code
!                   FAILURE = 0 );
!
! Return to the caller.
!
!     RETURN .primary_code
!
! END;
!
! $XPO_MODULE( SFAIL3 )

```

Action Routines SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE STR$A_FAILURE( function_code, primary_code, secondary_code, dummy, string, target ) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!           ? error appending 'string' to 'target'
!           -   primary completion code message
!           -   secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - action routine function code (STR$K_COMPARE)
!     primary_code - primary completion code
!     secondary_code - secondary completion code
!     dummy - dummy argument (not used)
!     string - address of source string descriptor
!     target - address of target string descriptor
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! OWN
!     initial_text : $STR_DESCRIPTOR( STRING = 'error appending ' ),
!     to_text : $STR_DESCRIPTOR( STRING = ' to ' );
!
! EXTERNAL ROUTINE
!     XST$INIT_MSG : NOVALUE,                                ! Failure message initialization routine
!     XST$STRING : NOVALUE,                                  ! Append string to failure message routine
!     XST$QUOTED : NOVALUE;                                  ! Append quoted string to failure message routine
!
! EXTERNAL
!     XST$MESSAGE;                                           ! Failure message string descriptor
!
! Create the initial function-specific message.
!
!     XST$INIT_MSG( initial_text );
!     XST$QUOTED( .string );
!     XST$STRING( to_text );
!     XST$QUOTED( .target );
!
! Send a multi-line failure message to the user.
!
!     $XPO_PUT_MSG( STRING = XST$MESSAGE,                    ! Function-specific message
!                   CODE = .primary_code,                    ! Primary failure completion code
!                   CODE = .secondary_code,                  ! Secondary failure completion code
!                   FAILURE = 0 );
!
! Return to the caller.
!
!     RETURN .primary_code
!
! END;
!
! $XPO_MODULE( SFAIL4 )

```

Action Routines

SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE STR$S_FAILURE( function_code, primary_code, secondary_code, scan_function, string, pattern )
=
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!         ? error scanning 'string' to find 'pattern'
!         - primary completion code message
!         - secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - action routine function code (STR$K_COMPARE)
!     primary_code - primary completion code
!     secondary_code - secondary completion code
!     scan_function - $STR_SCAN function code
!     string - address of source string descriptor
!     pattern - address of pattern string descriptor
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! OWN
!
!     initial_text : $STR_DESCRIPTOR( STRING = 'error scanning ' ),
!     find_text : $STR_DESCRIPTOR( STRING = ' to find ' ),
!     span_text : $STR_DESCRIPTOR( STRING = ', spanning ' ),
!     stop_text : $STR_DESCRIPTOR( STRING = ', stopping at ' );
!
! EXTERNAL ROUTINE
!
!     XST$INIT_MSG : NOVALUE,           ! Failure message initialization routine
!     XST$STRING : NOVALUE,             ! Append string to failure message routine
!     XST$QUOTED : NOVALUE,            ! Append quoted string to failure message routine
!
! EXTERNAL
!
!     XST$MESSAGE,                     ! Failure message string descriptor
!
! !
! ! Create the initial function-specific message.
! !
!
!     XST$INIT_MSG( initial_text );
!     XST$QUOTED( .string );
!
! CASE .scan_function FROM STR$K_FIND TO STR$K_STOP OF
!     SET
!     [ STR$K_FIND ] : XST$STRING( find_text );
!     [ STR$K_SPAN ] : XST$STRING( span_text );
!     [ STR$K_STOP ] : XST$STRING( stop_text );
!     TES;
!
!     XST$QUOTED( .pattern );
!
! !
! ! Send a multi-line failure message to the user.
! !
!
!     $XPO_PUT_MSG( STRING = XST$MESSAGE,           ! Function-specific message
!                   CODE = .primary_code,           ! Primary failure completion code
!                   CODE = .secondary_code,         ! Secondary failure completion code
!                   FAILURE = 0 );
!
! !
! ! Return to the caller.
! !

```

Action Routines
SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```
        RETURN .primary_code  
    END;  
$XPO_MODULE( SFAIL5 )
```


Action Routines SFAIL.BLI FAILURE-ACTION ROUTINE LISTING

```

GLOBAL ROUTINE STR$B_FAILURE( function_code, primary_code, secondary_code, convert_function, string, result
) =
!++
!
! FUNCTIONAL DESCRIPTION:
!
!     This routine sends the user a message sequence similar to the following:
!
!         ? error converting 'string' to binary
!         - primary completion code message
!         - secondary completion code message
!
! FORMAL PARAMETERS:
!
!     function_code - action routine function code (STR$K_COMPARE)
!     primary_code - primary completion code
!     secondary_code - secondary completion code
!     convert_function - $STR_BINARY function code
!     string - address of source string descriptor
!     result - address of result area
!
! IMPLICIT INPUTS:
!
!     None
!
! IMPLICIT OUTPUTS:
!
!     None
!
! COMPLETION CODES:
!
!     .primary_code - primary completion code passed by caller
!
! SIDE EFFECTS:
!
!     None
!
!--
!
! BEGIN
!
! OWN
!     initial_text : $STR_DESCRIPTOR( STRING = 'error converting ' ),
!     binary_text : $STR_DESCRIPTOR( STRING = ' to binary' );
!
! EXTERNAL ROUTINE
!     XST$INIT MSG : NOVALUE,                                ! Failure message initialization routine
!     XST$STRING : NOVALUE,                                ! Append string to failure message routine
!     XST$QUOTED : NOVALUE;                                ! Append quoted string to failure message routine
!
! EXTERNAL
!     XST$MESSAGE;                                           ! Failure message string descriptor
!
! ! Create the initial function-specific message.
! !
!     XST$INIT_MSG( initial_text );
!     XST$QUOTED( .string );
!     XST$STRING( binary_text );
!
! ! Send a multi-line failure message to the user.
! !
!     $XPO_PUT_MSG( STRING = XST$MESSAGE,                    ! Function-specific message
!                   CODE = .primary_code,                    ! Primary failure completion code
!                   CODE = .secondary_code,                  ! Secondary failure completion code
!                   FAILURE = 0 );
!
! ! Return to the caller.
! !
!     RETURN .primary_code
!
! END;
END
ELUDOM

```

APPENDIX F COMPILING AND LINKING

F.1	DEFINING A TRANSPORTABLE LOGICAL DEVICE	F-1
F.2	COMPILING	F-2
F.3	LINKING	F-3

COMPILING AND LINKING

This appendix contains information related to the compilation and linking of BLISS programs that contain references to the programming tools described in this manual. It contains the relevant source and object file specifications for all target systems on which these tools are currently implemented. It also suggests an operational technique that facilitates both transportable compilation and more convenient program linking.

F.1 DEFINING A TRANSPORTABLE LOGICAL DEVICE

For purposes of both transportability and operational convenience, we suggest that you define, on each development system, a 'logical device name' -- say BLI -- that stands for the device, directory, etc., in which the XPORT files reside on that particular system. The uses of such a name are illustrated in subsequent sections.

The system-level command that would be used to define a logical name on several BLISS-development systems are as follows:

- ```
o TOPS-10 : (No user command available; see below)
o TOPS-20 : @DEFINE BLI: partial-file-spec
o VAX/VMS : $ DEFINE BLI partial-file-spec
 or
 $ ASSIGN partial-file-spec BLI
```

This type of command would normally be executed from your login-command file; see below.

## Compiling and Linking DEFINING A TRANSPORTABLE LOGICAL DEVICE

A typical example of such a command on TOPS-20 might be:

```
@DEFINE BLI: PS:<SUBSYS>
```

and for VAX/VMS:

```
$ DEFINE BLI SYS$LIBRARY:
```

In order to fill in the "partial-file-spec" in the formats given above, you must find out where in your system the XPORT files have been installed. You can do this either by inspection, i.e., by 'looking through' the file system, or by asking your system manager.

The reason that we suggest the name BLI is that, in many cases, the system manager will already have appropriately defined that name on a system-wide basis. If this is the case, you do not have to perform the definition yourself, of course. (You cannot under TOPS-10 in any case.)

Since the commands described above are only effective for the duration of the session in which they are executed, it is strongly recommended that the appropriate command be placed in your command file that is automatically executed at login time, e.g., LOGIN.COM (VMS), LOGIN.CMD (TOPS-20).

### F.2 COMPILING

Assuming that a definition exists for the logical name BLI as discussed in Section F.1, include the following LIBRARY declaration in your BLISS source modules:

```
LIBRARY 'BLI:XPORT' ;
```

Given appropriate definitions of BLI, this single declaration will work for all BLISS compilers; the relevant XPORT.Lnn Library file will be selected by default in each case: XPORT.L32, or XPORT.L36.

The source-time portion of the XPORT package, containing macro, literal, and control-block definitions, is provided in two forms. One is the BLISS Library file just discussed, XPORT.Lnn, which is intended for use in normal compilations. The other is a BLISS Require file, XPORT.REQ, which is provided only for non-standard uses such as the construction of a user Library file that contains XPORT source as well as other code, in precompiled form. (Use of Require files rather than Library files in standard compilations incurs a significant increase in compilation cost.)

## Compiling and Linking LINKING

### F.3 LINKING

A standard (default) XPORT object file is provided on each BLISS development system. Use of this file presumes that the development system and the intended target system are one and the same. The name of this file is either XPORT.OLB (for VAX) or XPORT.REL (for TOPS-10 and TOPS-20). A LINK command of the general form

```
LINK program, BLI:XPORT [...]
```

(or its equivalent) will incorporate the standard XPORT object file for the system in use.

Additional object files are provided where necessary for debugging (36-bit systems only) and for cross-linking, that is, linking for a target system other than the development system. Also, an object file named specifically for the host system (and identical to the default object file) is provided on each system; for example, XPOT10.REL is the same as XPORT.REL on a TOPS-10 system.

The special debug objects provided on the 36-bit systems, which have the file-type .DBG, are for use with the SIX12 Debugger. Note that they can only be used if the module containing the program entry point has been compiled with the /DEBUG switch.

A listing of the currently available object files follows. Arrows (->) preceding the file names indicate which pairs of files are identical in content.

- o On a TOPS-10 system:
  - > XPORT.REL and XPORT.DBG
  - > XPOT10.REL and XPOT10.DBG
- o On a TOPS-20 system:
  - > XPORT.REL and XPORT.DBG
  - XPOT10.REL and XPOT10.DBG
  - > XPOT20.REL and XPOT20.DBG
- o On a VAX/VMS system:
  - > XPORT.OLB
  - > XPOVMS.OLB

Note that special debug object files are not required for 32-bit target systems.

## Compiling and Linking LINKING

Typical program-linking command sequences for the several development systems are shown below. These command sequences assume that the logical name BLI has been defined as discussed in Section F.1.

- o On a TOPS-10 system:

```
.R LINK
*program, BLI:XPORT/SEARCH/GO
.SAVE
```

- o On a TOPS-20 system:

```
@LINK
*program, BLI:XPORT/SEARCH/GO
@SAVE
```

- o On a VAX/VMS system:

```
$LINK program, BLI:XPORT/LIBRARY/NOTRACEBACK [/DEBUG]
```

APPENDIX G            XDUMP UTILITY PROGRAM

|       |                                                                       |     |
|-------|-----------------------------------------------------------------------|-----|
| G.1   | XDUMP - XPORT DATA STRUCTURE DISPLAY UTILITY . . .                    | G-1 |
| G.1.1 | Running the XDUMP Utility . . . . .                                   | G-1 |
| G.1.2 | Compiling a Structure Display Module . . . . .                        | G-2 |
| G.1.3 | Linking a Structure Display Module . . . . .                          | G-3 |
| G.1.4 | Displaying a User Declared Structure While<br>Debugging . . . . .     | G-3 |
| G.2   | XDESC, XIOB, and XSPEC - XPORT STRUCTURE DISPLAY<br>MODULES . . . . . | G-3 |
| G.2.1 | Linking an XPORT Structure Display Module . . .                       | G-3 |
| G.2.2 | Displaying an XPORT Structure While Debugging .                       | G-4 |

## APPENDIX G

### XDUMP UTILITY PROGRAM

This appendix describes the use of the XDUMP Utility program during debugging of user programs that employ the XPORT data-structuring facilities (see Chapter 2).

#### G.1 XDUMP - XPORT DATA STRUCTURE DISPLAY UTILITY

The XPORT XDUMP utility generates an executable BLISS module that can be called during program debugging to symbolically display the contents of a specific program data structure.

##### G.1.1 Running the XDUMP Utility

The XDUMP utility is provided in executable form as part of an XPORT release. Although its location on your system is determined by your system manager, it will typically be located in the standard system executable-image library (e.g., SYS\$SYSTEM on VAX/VMS).

To invoke the XDUMP utility, enter a command similar to the following VAX/VMS DCL command:

```
$ RUN XDUMP
```

XDUMP will then ask for the name of a BLISS REQUIRE file which includes the definition of the desired program data structure, as follows:

```
BLISS REQUIRE file name (default = .REQ)?
```

XDUMP will then ask for the name of the desired data structure:

```
Name of structure?
```

XDUMP then searches the REQUIRE file for a BLISS comment statement of



XDUMP UTILITY PROGRAM  
XDUMP - XPORT DATA STRUCTURE DISPLAY UTILITY

the form

```
! structure-name . . .
```

where "structure-name" is an arbitrary name which you assign to your structure. This comment statement may include text following the structure name.

NOTE

It is recommended that a structure-name be no more than six characters long since this name will become the name of the generated display module and the name of the display routine within this module.

The data structure must be defined using the XPORT Transportable Data Structure definition facilities (see Chapter 2). If the desired structure definition is not the last structure definition in the REQUIRE file, a comment statement of the following form must follow the structure definition:

```
! End of structure-name
```

Upon completion of XDUMP processing, the name of the generated display module will be reported, as follows:

```
structure-name.BLI display module generated
```

### G.1.2 Compiling a Structure Display Module

To compile the structure display module generated by the XDUMP utility, enter a command similar to the following VAX/VMS DCL command:

```
$ BLISS structure-name
```

This compilation command assumes that the logical device BLI: has been defined to point to the device and directory in which the XPORT LIBRARY file exists (see Appendix F).

XDUMP UTILITY PROGRAM  
XDUMP - XPORT DATA STRUCTURE DISPLAY UTILITY

### G.1.3 Linking a Structure Display Module

The structure-display object module can be included in a linked program image in the same manner as any other program subroutine. For example, a command similar to the following VAX/VMS DCL command should be entered:

```
$ LINK main-program, structure-name, ..., BLI:XPORT/LIBRARY/DEBUG
```

### G.1.4 Displaying a User Declared Structure While Debugging

While debugging, the current contents of a data structure can be displayed by entering a command similar to the following VAX/VMS DEBUG command:

```
DBG>CALL structure-name (address-of-structure)
```

NOTE

Some current debuggers (e.g., DDT, ODT) do not include a CALL-type command and are therefore not compatible with the XDUMP utility.

The contents of specified structure will be displayed, one field per line, in a form similar to the following:

```
field-name = symbolic-field-value
```

where "field-name" is the name of the field in the structure definition and "symbolic-field-value" is the current contents of the field displayed in symbolic form based on the "type" of the field.

## G.2 XDESC, XIOB, and XSPEC - XPORT STRUCTURE DISPLAY MODULES

### G.2.1 Linking an XPORT Structure Display Module

The XPORT distribution includes structure display modules which can be used during program debugging to display an XPORT descriptor, an XPORT IOB structure, or an XPORT File Specification Parse Block.

If any of these standard XPORT display modules (XDESC, XIOB, XSPEC) is desired during debugging, each must be explicitly requested during program linking. For example, on a VAX/VMS system:

```
$ LINK program, BLI:XPORT/LIBRARY/INCLUDE=XIOB/DEBUG
```

XDUMP UTILITY PROGRAM  
XDESC, XIOB, and XSPEC - XPORT STRUCTURE DISPLAY MODULES

Alternatively, you can include any of the statements shown below in your program to force automatic inclusion of these standard XPORT display modules in your program image:

```
EXTERNAL ROUTINE XDESC;
EXTERNAL ROUTINE XIOB;
EXTERNAL ROUTINE XSPEC;
```

### G.2.2 Displaying an XPORT Structure While Debugging

While debugging, the XPORT data structures (i.e., XDESC, XIOB, XSPEC) can be displayed by entering a command similar to the VAX/VMS DEBUG command shown in Section G.1.4.

```
DBG>CALL XIOB (name of IOB)
```

## APPENDIX Z EASY-TO-USE I/O PACKAGE (EZIO)

|       |                                               |     |
|-------|-----------------------------------------------|-----|
| Z.1   | OVERVIEW . . . . .                            | Z-1 |
| Z.2   | LIMITATIONS . . . . .                         | Z-1 |
| Z.3   | FUNCTIONAL DESCRIPTION . . . . .              | Z-2 |
| Z.3.1 | The FILOPN Routine . . . . .                  | Z-2 |
| Z.3.2 | The FILIN Routine . . . . .                   | Z-4 |
| Z.3.3 | The FILOUT Routine . . . . .                  | Z-4 |
| Z.3.4 | The FILCLS Routine . . . . .                  | Z-5 |
| Z.3.5 | Restrictions . . . . .                        | Z-5 |
| Z.3.6 | Example . . . . .                             | Z-6 |
| Z.4   | LOADING EZIO WITH USER PROGRAM . . . . .      | Z-7 |
| Z.4.1 | EZIOFC - File Services 11 (RSX-11M) . . . . . | Z-7 |
| Z.4.2 | EZIORT - RT-11 . . . . .                      | Z-8 |
| Z.4.3 | EZIO10 - TOPS-10 . . . . .                    | Z-8 |
| Z.4.4 | EZIO20 - TOPS-20 . . . . .                    | Z-8 |
| Z.4.5 | EZIOVX - VAX/VMS . . . . .                    | Z-8 |
| Z.5   | PACKAGING . . . . .                           | Z-8 |

## APPENDIX Z

### EASY-TO-USE I/O PACKAGE (EZIO)

#### Z.1 OVERVIEW

EZIO is a very basic I/O package for BLISS programs that predates the development of XPORT. (It is documented here for historical reasons only; it is not a supported product.)

EZIO provides only sequential character-string I/O, such as line input and line output. It is functionally the same on all major Digital operating systems.

EZIO was intended for the BLISS programmer who wanted to write a 'throwaway' program, e.g., a prototype or a temporary tool, both transportably and with a minimum of effort. (Prior to the development of XPORT I/O, the only alternative for I/O was direct monitor calls.) That is to say, EZIO was intended to be a disposable tool. It is very easy to use, but is not a particularly capable tool.

#### Z.2 LIMITATIONS

The EZIO package does not provide a complete set of I/O facilities for any system. More specifically, the following functionality is not provided:

- o Binary I/O
- o Sequenced files (record and page numbers, as with SOS)
- o Random I/O
- o More than 3 file I/O channels open concurrently.
- o Concurrent use of monitor calls for I/O processing (which may be precluded due to the way that EZIO does file I/O).

## Easy-to-Use I/O Package (EZIO) LIMITATIONS

The maximum number of I/O channels that may be opened can be increased in most implementations by modifying and recompiling the EZIO sources; see Section Z.3.5.

### Z.3 FUNCTIONAL DESCRIPTION

The EZIO package provided in the BLISS Library is a set of routines that are called to perform I/O operations. The routines are:

1. FILOPN - Opens a specified file on a logical channel.
2. FILIN - Reads a line of text from a file opened on the specified channel. It returns the number of characters actually transferred.
3. FILOUT - Transfers data from a string to a file opened on the specified channel.
4. FILCLS - Closes the file on a specified channel.

With the exception of FILIN, all the functions return 1 to indicate a successful completion of the operation, or 0 to indicate a failure. FILIN returns a negative value to indicate a failure to complete its operation, or a positive value (possibly zero) to indicate the number of characters read.

All of the routines take a logical channel number as a parameter. (This is not guaranteed to correspond to any logical unit number of the underlying file system). The channel number should be within the range of -1 to the maximum number of channels supported by EZIO, normally the range [-1, 0, 1, 2] (see Section Z.3.5).

Channel -1 is reserved for terminal operations. All calls using this channel do a minimum of buffering, and use operating-system primitives for communicating with the terminal (if such exist).

#### Z.3.1 The FILOPN Routine

This routine call requires four parameters. The first specifies a logical channel that the file is to be opened on. The second parameter specifies the length of the file-spec string. The third is a character-string pointer to the file-spec (constructed by CH\$PTR or equivalent). The last parameter specifies either 0 or 1 to indicate that the file is to be opened for input or output respectively.

Easy-to-Use I/O Package (EZIO)  
FUNCTIONAL DESCRIPTION

For example, to open a file for output whose file-spec was dynamically constructed:

```
FILOPN(1, .NAMLEN, CH$PTR(FILNAM), 1)
```

The value returned by the routine is 1 if the file was opened and subsequent I/O can be done to its channel; 0 otherwise. A return value of 0 usually indicates that either the file is not present, there was an error in the syntax of the file-spec, or the file could not be accessed in the specified mode.

The following conventions apply to programs containing "hard coded" file-specs that may be transported to other systems:

1. File names should consist of a maximum of six characters, and contain only the characters A-Z (upper or lower case is permitted, but lower will be converted to upper ) and 0-9.
2. File types (extensions) should consist of a maximum of three characters, and contain only the characters A-Z and 0-9. Also, they should follow the file name and be separated from the file name by a period. (The same rule applies to case as for file names.)
3. Hard coded device names should be avoided since they differ across operating systems. The default device is disk for I/O to channels 0 through 2. All I/O to channel -1 will go to the terminal.
4. Account information (e.g., PPNs or directory names) should not be specified if a program is to be fully transportable. In all implementations, the default is to use the directory established as the default when EZIO is first invoked (e.g., the logged in directory).

An example of a transportable file-spec is:

```
UPLIT('FILE.EXT')
```

Note that all of the above can be avoided if the program asks the user to enter his file rather than storing them as static data in the module. The user can then type in a file-spec in the host system's own syntax because EZIO uses the system file-spec parser in all implementations except TOPS-10 (where the parser is PIP like).

For all implementations, the character string specifying the file-spec should not exceed a length of 127 characters. However, there may be other restrictions for specific operating systems.

Easy-to-Use I/O Package (EZIO)  
FUNCTIONAL DESCRIPTION

Z.3.2 The FILIN Routine

This routine call requires three parameters. The first specifies the logical channel associated with the file to be read from. The second parameter specifies the maximum number of characters that will fit into the destination string. The last parameter is a character-string pointer to the string that will receive the data read in.

The value of the routine is -1 if a read beyond the end-of-file was attempted. -2 is returned if any I/O error occurred. Otherwise, an integer value greater than -1 is returned to indicate the number of characters transferred to the destination string.

For example, to read a line of text from the file opened on channel 1 and to put it in string BUFR:

```
LEN = FILIN(1, 80, CH$PTR(BUFR));
```

In this example, the variable LEN is set with the number of characters actually put in the string at BUFR.

A "line of text" is a string of ASCII characters delimited according to the host file system's conventions. Any delimiting factors (i.e., length field, vertical motion characters, etc) will act as a stream "break" and will be stripped on input. On output, the format expected by the system's principal editors will be used.

NOTE

On some systems, there is more than one way to delimit a line of text, and an effort has been made to use the methods of the principal editors in each case.

Z.3.3 The FILOUT Routine

This routine call has three parameters. The first specifies the logical channel associated with the file to be written on. The second parameter specifies the number of characters that will be transferred to the file. The last parameter is a character string pointer to the string that is to be written.

For example, to write a line from string TITLE to the file open on channel 2:

```
FILOUT(2, .TITLIN, CH$PTR(TITLE));
```



Easy-to-Use I/O Package (EZIO)  
FUNCTIONAL DESCRIPTION

Of course, if there were any doubt that the operation might not complete successfully, an IF statement could be used to check the return value of the call; the value 1 indicating success and 0 indicating failure.

The string is output as a line of text. That is to say, the host system's conventions are used to output the string so that it will appear as one line when printed. (Some systems delimit lines with CRLF, others store it as a counted string.) In all cases, EZIO should be able to read files that it creates.

#### Z.3.4 The FILCLS Routine

This routine call has one parameter, which is the logical channel associated with the file to be closed.

The FILCLS routine ensures that all EZIO buffers containing data are written out before informing the file system that all data transfers to/from the file are complete.

For example:

```
FILCLS(-1);
```

Of course, the call could be included in a conditional expression to determine whether the operation succeeded or failed (e.g., the case of not enough disk space).

#### Z.3.5 Restrictions

There are no logical restrictions in the EZIO package. However, because EZIO is dependent on them, the restrictions of the underlying file systems and of BLISS apply to all programs using the EZIO package.

There are physical constraints on the number of EZIO channels that can be open at any one time. The maximum as distributed is three plus channel -1 for terminal I/O. The reason is to reduce the number of buffers and tables within EZIO.

#### NOTE

The maximum number of channels that can be concurrently opened is controlled by the compile-time literal MAXCHANS in most of the implementations. The user may change this parameter to change the number of channels to his/her liking.

Easy-to-Use I/O Package (EZIO)  
FUNCTIONAL DESCRIPTION

### Z.3.6 Example

This example program prints a file on the terminal:

```

MODULE LISTER (MAIN = LSTR) =
!+
! This program asks for a filename, opens the named file,
! and copies the file to the terminal.
!--
BEGIN

EXTERNAL ROUTINE
 FILOPN, ! EZIO open
 FILCLS, ! EZIO Close
 FILOUT, ! EZIO Output
 FILIN; ! EZIO Input

OWN ! Holds one line of text
 BUF : VECTOR[CH$ALLOCATION(120)];

MACRO
 MSGS(S) =
 FILOUT(-1, %CHARCOUNT(S), CH$PTR(UPLIT(S))) %;

ROUTINE LSTR =
 BEGIN

 LOCAL
 LEN, ! Length of string
 PTR; ! Pointer to buf
!
! Open the TTY. Note: no filespec
!
 FILOPN(-1, 0, 0, 0);
 PTR = CH$PTR(BUF); ! Get pointer
 MSG('Enter file name'); ! Prompt
 LEN = FILIN(-1, 60, .PTR); ! Get file name
!
! Open the file on channel 0.
!
 IF NOT FILOPN(0, .LEN, .PTR, 0)
 THEN
 BEGIN ! Open failed.
 MSG('Open failed.');
```

Easy-to-Use I/O Package (EZIO)  
FUNCTIONAL DESCRIPTION

```
! Process each line
!
 WHILE 1 DO
 BEGIN
 LEN = FILIN(0, 120, .PTR);
 IF .LEN EQL -1
 THEN
 EXITLOOP; ! End of file
 FILOUT(-1, .LEN, .PTR); ! Output the string
 END;

 FILCLS(0);
 MSG('DONE');
 END;
 END
ELUDOM
```

#### Z.4 LOADING EZIO WITH USER PROGRAM

The procedures shown below describe the steps necessary to load EZIO with a user program on each system. These examples assume that the user's object program is in a file named TEST which has the proper file type or extension for the system in question.

##### Z.4.1 EZIOFC - File Services 11 (RSX-11M)

1. Run the Task Builder:

```
MCR>TKB TEST=TEST,EZIOFC,EISLIB
```

The TEST to the left of the equal sign is the name of the task-image file. The TEST to the right of the equal sign is the object module of the main program. EZIOFC is the object module of EZIO. And EISLIB is the EIS BLISS-16 library of runtime routines.

2. When TKB finishes, give the run command to invoke the program:

```
MCR>RUN TEST
```

Easy-to-Use I/O Package (EZIO)  
LOADING EZIO WITH USER PROGRAM

Z.4.2 EZIORT - RT-11

For RT-11, simply use the DCL command EXECUTE:

```
.EXECUTE TEST,EZIORT,EISLIB
```

Z.4.3 EZIO10 - TOPS-10

After your program has been compiled with BLISS-36, use the TOPS-10 EXECUTE command:

```
.EXECUTE TEST.REL,BLI:EZIO10
```

Note that a library file does not have to be specified (BLISS-36 generates a load request automatically).

Z.4.4 EZIO20 - TOPS-20

After your program has been compiled by BLISS-36 (with the /TOPS-20 compilation switch, if necessary), simply use the EXECUTE command:

```
@EXECUTE TEST,BLI:EZIO20
```

Note that a library file does not have to be specified (BLISS-36 generates a load request automatically).

Z.4.5 EZIOVX - VAX/VMS

After your program has been compiled with BLISS-32, use the VAX/VMS LINK command, as follows:

```
$ LINK TEST,SYS$LIBRARY:EZIOVX
```

Z.5 PACKAGING

EZIO is distributed as several files: EZIOFC for a Files-11 based system, EZIORM for a RMS-11 based file system, EZIORT for programs that run under RT-11, EZIOVX for the VAX/VMS operating system, EZIO10 for TOPS-10 and EZIO20 for TOPS-20.

Easy-to-Use I/O Package (EZIO)  
PACKAGING

EZIOFC, EZIORM and EZIORT are to be compiled using BLISS-16. EZIOVX must be compiled using BLISS-32. And EZIO10 and EZIO20 must be compiled with BLISS-36 using the /TOPS10 and /TOPS20 switches respectively. Some of the EZIO sources have REQUIRE statements. These require the various system-interface modules also distributed in the BLISS Library, which must be present in order to successfully modify the sources.



## INDEX

\$ADDRESS field-type, 2-7  
  usage guidelines, 2-8  
\$ALIGN, 2-2  
  usage, 2-13  
\$BIT field-type, 2-6  
\$BITS(n) field-type, 2-6  
\$BYTE  
  field-type, 2-7  
  See also BYTE  
\$BYTES vs. \$STRING usage, 2-21  
\$BYTES(n) field-type, 2-7  
\$CONTINUE, 2-2  
  usage, 2-14  
\$DESCRIPTOR(class) field-type, 2-7  
  usage guidelines, 2-10  
\$DISTINCT, 2-2  
  usage, 2-15  
\$FIELD declaration  
  example of, 2-3  
  general form of, 2-5  
  usage rules, 2-6  
\$FIELD keyword, 2-2  
\$field-types, 2-2  
\$FIELD SET SIZE, 2-2  
  usage, 2-12  
\$INTEGER field-type, 2-7  
\$LITERAL, 2-2  
  usage, 2-15  
\$LONG INTEGER field-type, 2-7  
\$OVERLAY, 2-2  
  usage, 2-14  
\$POINTER field-type, 2-7  
  usage guidelines, 2-8  
\$REF\_DESCRIPTOR field-type, 2-7  
\$SHORT\_INTEGER field-type, 2-7  
\$SHOW, 2-2  
\$SIXBIT(n) field-type, 2-8  
\$STR\_APPEND macro  
  completion codes, A-6  
  definition of, A-4  
\$STR\_ASCII macro  
  definition of, A-7  
\$STR\_BINARY macro  
  completion codes, A-11  
  definition of, A-9  
\$STR\_COMPARE macro  
  completion codes, A-13  
  definition of, A-12  
\$STR\_CONCAT macro  
  definition of, A-14  
\$STR\_COPY macro  
  completion codes, A-18  
  definition of, A-16  
\$STR\_DESC\_INIT macro  
  completion codes, A-22  
  definition of, A-21  
  examples of, 6-3  
\$STR\_DESCRIPTOR macro  
  definition of, A-19  
  examples of, 6-2  
\$STR\_EQL macro  
  completion codes, A-25  
  definition of, A-23  
\$STR\_FORMAT macro  
  definition of, A-26  
\$STR\_GEQ macro  
  completion codes, A-31  
  definition of, A-29  
\$STR\_GTR macro  
  completion codes, A-34  
  definition of, A-32  
\$STR\_LEQ macro  
  completion codes, A-37  
  definition of, A-35  
\$STR\_LSS macro  
  completion codes, A-40  
  definition of, A-38  
\$STR\_NEQ macro  
  completion codes, A-43  
  definition of, A-41  
\$STR\_SCAN macro  
  completion codes, A-47  
  definition of, A-44  
\$STRING vs. \$BYTES usage, 2-21  
\$STRING(n) field-type, 2-7  
  usage guidelines, 2-11  
\$SUB\_BLOCK(len) field-type, 2-7  
  usage guidelines, 2-9  
\$SUB\_FIELD, 2-2  
\$TINY\_INTEGER field-type, 2-7  
\$XPO\_BACKUP macro

- completion codes, A-49
- definition of, A-48
- example of, 3-14
- \$XPO\_CLOSE macro
  - completion codes, A-52
  - definition of, A-51
  - examples of, 3-12
- \$XPO\_DELETE macro
  - completion codes, A-55
  - definition of, A-54
  - examples of, 3-13
- \$XPO\_DESC\_INIT macro
  - completion codes, A-60
  - definition of, A-59
  - examples of, 7-3
- \$XPO\_DESCRIPTOR macro
  - definition of, A-57
  - examples of, 7-2
- \$XPO\_ERROR
  - standard message device, 3-22
- \$XPO\_FREE\_MEM macro, 4-2
  - completion codes, A-62
  - definition of, A-61
- \$XPO\_GET macro
  - completion codes, A-65
  - definition of, A-63
  - examples of, 3-15
- \$XPO\_GET\_MEM macro, 4-2
  - definition of, A-67
- \$XPO\_INPUT, 3-11
  - standard input device, 3-22
- \$XPO\_IOB macro
  - definition of, A-70
  - examples of, A-70
- \$XPO\_IOB\_INIT macro
  - completion codes, A-72
  - definition of, A-71
- \$XPO\_OPEN macro
  - completion codes, A-77
  - definition of, A-73
  - examples of, 3-11
- \$XPO\_OUTPUT
  - standard output device, 3-22
- \$XPO\_PARSE\_SPEC macro
  - completion codes, A-80
  - definition of, A-79
  - use of, 3-26
- \$XPO\_PUT macro
  - completion codes, A-82
  - definition of, A-81
  - examples of, 3-16
- \$XPO\_PUT\_MSG macro
  - completion codes, A-85
  - definition of, A-84
  - example of, 5-1
- \$XPO\_RENAME macro
  - completion codes, A-89
  - definition of, A-86
  - examples of, 3-13
- \$XPO\_SPEC\_BLOCK macro, 3-26
  - definition of, A-90
  - examples of, A-90
- \$XPO\_TEMPORARY
  - temporary work file, 3-22
- \$XPO\_TERMINATE macro
  - definition of, A-91
  - example of, 5-3
- \$XPO xxxx macro calls
  - format of, 3-10
- Action routines
  - I/O, 3-28
  - memory management, 4-4
  - put-message, 5-3
- Addresses vs. pointers, 2-8, 3-17
- ALIGN - See \$ALIGN
- APPEND option, 3-7, 3-12
- Asterisk character (\*)
  - See Wild-card character, 3-26
- Automatic file concatenation, 3-6
- Automatic program termination, 5-2
- BACKUP function, 3-4
- BACKUP macro call
  - see \$XPO\_BACKUP
- Backup operation
  - explanation of, 3-4
- Binary descriptor, 7-1
- BINARY mode, 3-6
  - GET example, 3-16
  - PUT example, 3-17
- Block size, fixed, 3-8
- BLOCK structures, transportable
  - difficulties with, 2-1
  - efficiency considerations, 2-22
  - example of, 2-3



- problem areas, 2-20
- Boundary alignment
  - by default, 2-12
  - efficiency considerations, 2-22
- BOUNDED descriptor, 6-7, 7-7
- BYTE alignment keyword, 2-13
- Caveats
  - on file-spec parsing, 3-26
  - on use of STREAM mode, 3-6
  - on use of UNITS (I/O), 3-6
- Character string
  - See String
- Character-string fields
  - coding of, 2-21
- Classes of descriptors, 6-4, 7-4
  - See also Descriptors
- CLOSE function, 3-4
- CLOSE macro call
  - see \$XPO\_CLOSE
- Completion codes
  - I/O, 3-22, 3-27
  - program termination, 5-4
  - put-message, 5-1, 5-3
- Concatenated input files, 3-6
- Concatenation of strings, logical, A-14
- CONTINUE - See \$CONTINUE
- Conversion
  - ASCII to ASCII, A-26
  - binary to ASCII, A-7
  - logical string concatenation, A-14
- Conversion pseudo-functions
  - definition of
    - \$STR\_ASCII, A-7
    - \$STR\_CONCAT, A-14
    - \$STR\_FORMAT, A-26
- Data descriptors, 6-1, 7-1
  - See also Descriptors
- Data parameter formats, A-2
- Data structures, transportable
  - difficulties with, 2-1
  - efficiency considerations, 2-22
  - example of, 2-3
  - field positioning, 2-12
  - literal definition, 2-14
- problem areas, 2-20
- DECnet links, 3-8
- Default action routines
  - for I/O, 3-28
  - memory management, 4-5
  - put-message, 5-3
- Default file specification, 3-9
- Defaulting technique
  - for file-specs, 3-23
- Defaults
  - file-type for BACKUP, 3-14
  - for descriptor class, 6-2, 7-2
  - for file specifications, 3-8
  - for NEW\_RELATED, 3-13
- DELETE macro call
  - see \$XPO\_DELETE
- Descriptor usage, 3-11, 3-15 to 3-17, 3-20
- Descriptors
  - BOUNDED, 6-6, 7-6
  - classes of, 6-2, 7-1
  - details, 6-4, 7-4
  - typical uses, 6-8
  - usage rules for, 6-6, 7-6
  - compile-time initialization of, 6-2, 7-2
  - creation of, 6-2, 7-2
  - DYNAMIC, 6-6, 7-6
  - DYNAMIC BOUNDED, 6-6, 7-6
  - FIXED, 6-6, 7-6
  - initialization of, 7-3
  - introduction to, 6-1, 7-1
  - run-time initialization of, 6-3
  - typical uses of, 6-8
  - UNDEFINED, 6-6, 7-6
- DISTINCT - See \$DISTINCT
- DYNAMIC descriptor, 6-7, 7-7
- Dynamic memory, 4-1
- DYNAMIC BOUNDED descriptor, 6-7, 7-7
- Editing of strings, A-26
- End of file (EOF)
  - on concatenated input, 3-6
- Error correction, I/O, 3-28
- Error message generation, 5-1
- FAILURE parameter, 3-10
- Failure-action routines

- I/O, 3-28
  - memory management, 4-4
- FATAL severity level, 5-2
- FIELD - See \$FIELD
- Field alignment
  - by default, 2-12
  - efficiency considerations, 2-22
- FIELD declaration, standard, 2-2
- Field positioning, 2-12
- Field positioning features
  - \$ALIGN, 2-13
  - \$CONTINUE, 2-14
  - \$OVERLAY, 2-14
- Field-names
  - IOB, 3-19
- Field-type usage guidelines
  - \$ADDRESS, 2-8
  - \$DESCRIPTOR, 2-10
  - \$POINTER, 2-8
  - \$STRING, 2-11
  - \$SUB BLOCK, 2-9
- Field-type, nontransportable
  - \$SIXBIT(n), 2-8
- Field-types, transportable, 2-6
- FIELD SET SIZE - See \$FIELD\_SET\_SIZE
- File backup operation, 3-4
- File concatenation, input, 3-6
- File specification
  - default, 3-9
  - primary, 3-9
  - related, 3-9
  - resultant, 3-9
- File specifications
  - concatenated input files, 3-6
  - defaults for, 3-8, 3-23
  - parsing of, 3-26
  - processing of, 3-23
  - remembered, 3-4
  - resolution of, 3-8, 3-23
  - rules for resolution, 3-24
- File-copy coding example, 3-21
- File-level capabilities, 3-3
- File-level functions
  - BACKUP, 3-4
  - CLOSE, 3-4
  - OPEN, 3-4
  - summary of, 3-3
- File-level macros
  - examples of, 3-11
- File-spec resolution
  - detailed discussion of, 3-23
  - general description, 3-8
  - introduction, 3-4
  - rules for, 3-24
- FILL parameter, memory, 4-2 to 4-3
- FIXED descriptor, 6-7, 7-7
- Freeing memory
  - examples of, 4-3
  - rule for, 4-4
- FULLWORD alignment keyword, 2-13
- Fullword, definition of, 2-5
- GET macro call
  - see \$XPO GET
- Get operations
  - concatenated input files, 3-7
  - in BINARY mode, 3-5
  - in RECORD mode, 3-5
  - in STREAM mode, 3-5
- Getting memory, examples, 4-2
- I/O action routines, 3-28
- I/O capabilities
  - concatenated input files, 3-6
  - get and put
    - BINARY mode, 3-6
    - in general, 3-5
    - RECORD mode, 3-6
    - STREAM mode, 3-6
- I/O completion codes, 3-22, 3-27
- I/O control block - See "IOB"
- I/O control blocks
  - description of, 3-18
  - introduction to, 3-2
- I/O devices, standard, 3-22
- I/O functions, summary of, 3-3
- I/O macros, 3-9
  - common parameters, 3-10
  - examples of, 3-15
  - general format, 3-10
- I/O routine example, 3-21
- I/O services
  - characteristics, 3-2
  - file-level capabilities, 3-3
  - implementation, 3-2
  - input/output capabilities, 3-5
  - introduction, 3-1
  - keyword macros, 3-2

- linking with program, 3-2
- I/O transportability, 3-1
- Input & output devices, 3-8
  - standard error, 3-22
  - standard input, 3-22
  - standard output, 3-22
- Input file concatenation, 3-6
- INPUT option, default, 3-8
- Input/Output - See "I/O"
- Interactive terminals, 3-8
- IOB (control block)
  - creation of, 3-18
  - description of, 3-18
  - field names, 3-2
  - fields
    - function of, 3-19
    - usage example, 3-20 to 3-21
  - initialization of, 3-19
  - introduction to, 3-2
- IOB keyword parameter, 3-10
- IOB\$G\_2ND\_CODE field, 3-28
- IOB\$G\_COMP\_CODE field, 3-27
- Keyword parameters, I/O, 3-10
  - examples of use, 3-20
- LIBRARY declaration, 3-2
- LINK command example, 3-2
- LITERAL - See \$LITERAL
- Literal-defining features
  - \$DISTINCT, 2-14
  - \$LITERAL, 2-14
- Macro calls
  - definitions of, A-4, A-19, A-57
  - general format for I/O, 3-10
- Macros
  - See \$STR\_xxxx
  - See \$XPO\_xxxx
- Memory management
  - action routines, 4-4
  - introduction, 4-1
  - routine values, 4-4
- Miscellaneous services, 5-1
- Mode keywords, I/O, 3-5
- NEW\_RELATED default, 3-13
- Notation for macro definitions, A-1
- OPEN function, 3-4
- OPEN macro call
  - see \$XPO\_OPEN
- Opening
  - concatenated input files, 3-6
  - defaults, 3-8
  - for both input and output, 3-8
  - interactive terminal, 3-11
  - output files, 3-7
- Output files, opening of, 3-7
- OUTPUT option, 3-7
- OVERLAY - See \$OVERLAY
- OVERWRITE option, 3-7
- Parameter keywords, I/O, 3-10
- Physical block size, fixed, 3-8
- Placeholder \$SUB\_BLOCK usage, 2-9
- Pointers vs. addresses, 2-8, 3-17
- Primary file specification, 3-9
- Program
  - brief example, 1-5
- Program termination
  - automatic, 5-2
  - requested, 5-3
- Prompted read operation, 3-15
- PUT macro call
  - see \$XPO PUT
- Put operations
  - in BINARY mode, 3-5
  - in RECORD mode, 3-5
  - in STREAM mode, 3-5
- Put-Message function, 5-1
- RECORD mode, 3-6
  - GET example, 3-15
  - PUT example, 3-16
- RECORD\_SIZE parameter, 3-8
- Related file specification, 3-9
- Releasing memory
  - rule for, 4-4
- REMEMBER file-close option, 3-4 to 3-5
- REMEMBER option, 3-12 to 3-14
- Remembered file specifications, 3-4
- RENAME macro call
  - see \$XPO RENAME
- Resultant file specification,

- 3-9
- Routine values
  - I/O, 3-27
  - memory management, 4-4
- Sample program
  - brief, 1-5
- SEQUENCE\_NUMBER parameter, 3-17
- SEQUENCED attribute, 3-5, 3-12, 3-22
- Sequenced file
  - PUT example, 3-17
- SEVERITY parameter
  - put-message, 5-2
- Spaceholder \$SUB\_BLOCK usage, 2-9
- STREAM mode, 3-6
  - caveat regarding, 3-6
  - GET example, 3-15
  - PUT example, 3-16
- String concatenation, logical, A-14
- String conversion
  - see also Conversion
- String conversion pseudo-functions
  - definition of
    - \$STR\_ASCII, A-7
    - \$STR\_CONCAT, A-14
    - \$STR\_FORMAT, A-26
- String descriptors, 6-1
  - See also Descriptors
- String editing, A-26
- String parameter formats, A-2
- Structures - See Data structures
- Sub-fields - See \$SUB\_FIELD
- Sub-structures - See \$SUB\_BLOCK
- SUCCESS parameter, 3-11
- Success-action routines
  - I/O, 3-28
  - memory management, 4-4
- Syntax notation, A-1
- Temporary files, 3-23
- Terminal I/O, 3-22
- Termination function, 5-3
- Transportability problem areas, 2-20
- Transportable BLOCK structures
  - see BLOCK structures
- Transportable data structures
  - see Data structures
- Transportable field-types, 2-6
  - see also Field-type
- Transportable I/O services, 3-1
- UNDEFINED descriptor, 6-6, 7-6
- UNIT alignment keyword, 2-13
- Usage guidelines
  - \$XPO\_BACKUP macro, A-49
  - \$XPO\_CLOSE macro, A-52
  - \$XPO\_PUT macro, A-82
- USER parameter, 3-11
- Wild-card character (\*), 3-26
- WORD alignment keyword, 2-13
- XPO\$NORMAL
  - completion code, 3-27
- XPO\$TERMINATE code, 5-3
- XPO\$FAILURE routine, 3-28, 4-5, 5-3
- XPO\$FM\_FAILURE routine, 4-5
- XPO\$GM\_FAILURE routine, 4-5
- XPO\$IO\_FAILURE routine, 3-28
- XPO\$PM\_FAILURE routine, 5-3
- XPORT I/O devices
  - standard, 3-22
- XPORT I/O facilities - See "I/O"

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

---

---

---

---

---

---

---

---

---

---

Did you find errors in this manual? If so, specify the error and the page number.

---

---

---

---

---

---

---

---

---

---

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)\_\_\_\_\_

Name\_\_\_\_\_ Date\_\_\_\_\_

Organization\_\_\_\_\_

Street\_\_\_\_\_

City\_\_\_\_\_ State\_\_\_\_\_ Zip Code\_\_\_\_\_

or  
Country

Please cut along this line.

— — — Do Not Tear - Fold Here and Tape — — —

**digital**



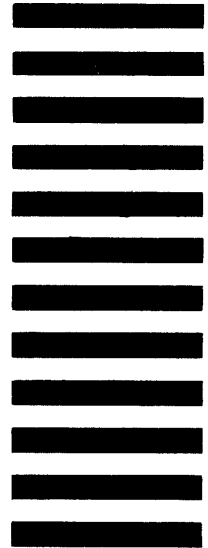
No Postage  
Necessary  
if Mailed in the  
United States

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J3-5  
DIGITAL EQUIPMENT CORPORATION  
110 SPIT BROOK ROAD  
NASHUA, NEW HAMPSHIRE 03061



— — — Do Not Tear - Fold Here — — —

Cut Along Dotted Line