IEEE Standard for Information Technology—POSIX FORTRAN 77 Language Interfaces—Part 1: Binding for System Application Program Interface (API)

Sponsor
Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society

Approved June 18, 1992

IEEE Standards Board

Abstract: This standard provides a standardized interface for accessing the system services of ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990, also known as POSIX.1), and support routines to access constructs not directly accessible with FORTRAN 77. This standard supports application portability at the source level through the binding between ANSI X3.9-1978 and POSIX.1, and a standardized definition of language-specific services. The goal is to provide standardized interfaces to the POSIX.1 system services via a FORTRAN 77 language interface. Terminology and general requirements, process primitives, the process environment, files and directories, input and output primitives, device- and class-specific functions, the FORTRAN 77 language library, and system databases are covered.
Keywords: application portability, FORTRAN 77, interfaces, interoperability, POSIX, system interfaces

The Institute of Electrical and Electronics Engineers, Inc.

345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-230-3

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of the IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, the IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Introduction

(This introduction is not a normative part of IEEE Std 1003.9-1992, IEEE Standard for Information Technology—POSIX FORTRAN 77 Language Interfaces—Part 1: Binding for System Application Program Interface (API), but is included for information only.)

The purpose of this standard is to support application portability at the source level through the definition of:

1) An interface between the ANSI X3.9-1978 FORTRAN Standard (archival) and ISO/IEC 9945-1: 1990, Information technology — Portable Operating System Interface (POSIX) — Part 1: System application interface (API) [C language]
2) A standardized interface for language-specific services.

The focus of this standard is to provide standardized interfaces to the ISO/IEC 9945-1: 1990 system services via a FORTRAN 77 language interface. Future work will consist of interfaces to other parts of ISO/IEC 9945 and the possible use of new functionality provided in ISO/IEC 1539: 1991 (Fortran 90).

Organization of This Standard

1) Statement of scope and list of normative references (Section 1)
2) Definitions and global concepts (Section 2)
3) The various interface facilities (Section 3 through 9)

The FORTRAN 77 language interface for each service interface is given in the subclause labeled Synopsis. The correspondence of the ISO/IEC 9945-1: 1990 system service interface to the FORTRAN 77 language interface is described in the Description subclause. Additional information on the creation of specific actual arguments is provided for some interfaces. The Description subclause provides a specification of the operation performed for the language-specific services. In most cases, there is also an Errors subclause that describes error handling. References are used to direct the reader to related sections in ISO/IEC 9945-1: 1990 and in POSIX.9. Additional material to complement sections in this standard may be found in the Rationale and Notes (Annex A). This annex provides historical perspectives into the technical choices made by developers of this standard. It also elaborates on the information provided in the corresponding section of this standard.

Informative annexes are not part of the draft standard and are provided for information only. A normative annex is part of the standard and imposes requirements, but there are currently no such normative annexes in this standard.

In publishing this standard, its developers simply intend to provide a basis upon which various FORTRAN 77 interfaces to ISO/IEC 9945-1: 1990 can be measured for conformance. It is not the intent of the developers to measure or rate any products, to reward or sanction any vendors of products for conformance or lack of conformance to this standard, or to enforce this standard by these or any other means. The responsibility for determining the degree of conformance or lack thereof with this standard rests solely with the individual who is evaluating the product claiming to be in conformance with this standard.

Background

The developers of this standard represent a cross section of hardware manufacturers, user organizations, software designers, applications programmers, and others. In the course of the development of this standard, the developers received guidance from members of the ANS Committee on Fortran, X3J3.

ISO/IEC 9945-1: 1990 describes a set of fundamental system services. Access to these services has been provided by defining an interface using the FORTRAN 77 programming language in this standard.

Audience

The intended audience for this standard is all persons concerned with an industrywide standard FORTRAN 77 interface to the system services described in ISO/IEC 9945-1: 1990.

Purpose

Several principles guided the development of this standard:

Application Oriented

The basic goal was to promote portability of FORTRAN 77 application programs on systems conforming to ISO/IEC 9945-1: 1990.

Interface, Not Implementation

This standard defines an interface, not an implementation. No details of the implementation of any function are given, although historical practice may be indicated in Annex A.

Source, Not Object, Portability

This standard has been written so that a program written and translated for execution on one conforming implementation may also be translated for execution on another conforming implementation. This standard does not guarantee that executable (object or binary) code will execute under a different conforming implementation than that for which it was translated, even if the underlying hardware is identical.

The FORTRAN 77 Language

This standard is written in terms of the standard FORTRAN 77 language as specified in the FORTRAN 77 standard {3}. See 1.3.3. It contains the single extension of 31-character names.

Minimal Interface, Minimally Defined

In keeping with the rules of FORTRAN 77, this standard uses subroutine and function calls to interface to the ISO/IEC 9945-1: 1990. The 31-character name extension was added to allow the use of interface names from ISO/IEC 9945-1: 1990.

Related Standards Activities

Activities to extend this standard to address additional requirements are being considered.

The following activities are under active consideration at this time, or are expected to become active in the near future:

1) ISO 1539: 1991 (Fortran 90) Programming Language binding to a language-independent version of ISO/IEC 9945-1: 1990.
2) Fortran binding to Shell and Utilities facilities
3) Fortran binding to Realtime facilities
4) Fortran binding to Security

If you have interest in participating in the TCOS working groups addressing these issues, please send your name, address, and telephone number to the Secretary, IEEE Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, and ask to have this forwarded to the chairperson of the appropriate TCOS working group. If you have interest in participating in this work at the international level, contact your ISO/IEC national body.

This standard was prepared by the 1003.9 Working Group, sponsored by the Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society. At the time this standard was approved, the membership of the 1003.9 Working Group was as follows:

Technical Committee on Operating Systems and Application Environments (TCOS)

Chair: Jehan-François Pâris

TCOS Standards Subcommittee

Chair: Jim Isaak
Vice-Chairs: Ralph Barker
Hal Jespersen
Lorraine Kevra
Pete Meier
Andrew Twigger
Treasurer: Quin Hahn
Secretary: Shane McCarron

1003.9 Working Group Officials

Chair: John J. McGrory II
Vice-Chair: Michael Hannah
Editor: Daniel J. Magenheimer
Secretary: Larry Diegel

Working Group

Joanne Brixius
Loren Buhle
Cynthia M. Cox
Mike Hunter
Joseph King

The following persons were members of the balloting group that approved this standard for submission to the IEEE Standards Board:

Roger E. Anderson
Bengt Asker
David Athersych
Edward Benson
Jerry Berkman
Keith Bierman
Andy Bihain
James M. Bishop
Andy Cheese
Kilnam Chon
Cynthia M. Cox
Larry Diegel
Ron Elliott
Roger Golliver
E. Howard Green
Robert M. Gross
Mark Guzzi
Charles Hammons
Michael Hannah
Kurt W. Hirchert
Don Huebschman
Michael T. Hunter
Jim Isaak
Hal Jespersen
Jens Kolind
Ip-Beau Phillip Law
Donald Lewine
F. C. Lim
Daniel J. Magenheimer
Roger Martin
Patrick McGehearty
John J. McGrory II
Robert McWhirter
Lyle Meier
Martha Nalebuff
Daniel Nissen
Fred Noz
Paul E. Renaud
Steve M. Rowan
Lorne H. Schachter
Gerhard Schmitt
Leonard Seagren
Richard Seibel
Dan Shia
Andrew D. Tait
Ravi Tavakley
Donn S. Terry
Mark-Rene Uchida

Michael W. Vannier	Janusz Zalewski	Joseph King *(DECUS Institutional Representative)*
Stephen Walli	John Zolnowsky	
Richard W. Weaver	Joanne Brixius *(ANSI X3J3 Liaison)*	
Frederick N. Webb		

When the IEEE Standards Board approved this standard on June 18, 1992, it had the following membership:

Marco W. Migliaro, *Chair*
Donald C. Loughry, *Vice Chair*
Andrew G. Salem, *Secretary*

Dennis Bodson	Donald N. Heirman	T. Don Michael*
Paul L. Borrill	Ben C. Johnson	John L. Rankine
Clyde Camp	Walter J. Karplus	Wallace S. Read
Donald C. Fleckenstein	Ivor N. Knight	Ronald H. Reimer
Jay Forster*	Joseph Koepfinger*	Gary S. Robinson
David F. Franklin	Irving Kolodny	Martin V. Schneider
Ramiro Garcia	D. N. “Jim” Logothetis	Terrance R. Whittemore
Thomas L. Hannan	Lawrence V. McCall	Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal	Richard B. Engelman	Stanley Warshaw
James Beall	David E. Soffrin	

Mary Lynne Nielsen, *IEEE Standards Project Editor*

CLAUSE PAGE

CLAUSE PAGE

IEEE Standard for Information Technology—POSIX FORTRAN 77 Language Interfaces—Part 1: Binding for System Application Program Interface (API)

1. General

1.1 Scope

This standard provides the binding between the ANSI X3.9-1978 (FORTRAN 77) programming language and the system services defined in ISO/IEC 9945-1: 1990 (hereinafter referred to as POSIX.1 {2}).

As with the definition of the service interfaces in POSIX.1 {2}, this FORTRAN 77 language binding is defined exclusively at the source code level. The objective is that a Strictly Conforming Application may be developed in FORTRAN 77 and compiled to execute on a POSIX.1 {2} conforming implementation.

It is intended that this FORTRAN 77 language binding may coexist on a system with any other language binding.

The following areas are outside the scope of this standard:

1) Extensions to the FORTRAN 77 language, other than the required longer external names.
2) Bindings to any system interfaces using new or changed features in the revision to FORTRAN 77 (ISO/IEC 1539: 1991).
3) Bindings for system interfaces other than those defined in POSIX.1 {2}.

1.2 Normative References

[1] ISO/IEC 9899:1990, *Information technology—Programming languages—C.*[1]

[1]ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse. IEEE documents can be obtained from the Institute of Electrical and Electronic Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ, 08855-1331, USA.

[2] ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990), *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*.

[3] ANSI X3.9-1978, *American National Standard Programming Language FORTRAN*.[2]

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.1 Requirements

A *conforming implementation* shall meet all of the following criteria:

1) The system shall support all required interfaces defined within this standard. These interfaces shall support the functional behavior described herein.
2) The system may provide additional routines or facilities not required by this standard. Nonstandard extensions should be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of routines or facilities defined by this standard. The conformance document shall define an environment in which an application can be made to run with the behavior specified by this standard. In no case shall such an environment require modification of a Strictly Conforming POSIX.9 Application.

See POSIX.1 {2} 1.3 for a description of a POSIX.1 {2} conforming implementation.

1.3.1.2 Documentation

A conformance document with the information described in POSIX.1 {2} 1.3 and in this standard shall be available for an implementation claiming conformance to POSIX.1 {2} and to this standard. The conformance document shall have the same structure as that described in POSIX.1 {2} 1.3 and this standard, with the information presented in appropriately-numbered sections. The conformance document shall not contain information about extended facilities or capabilities outside the scope of POSIX.1 {2} 1.3, this standard, and the applicable standard described in 1.3.3.1.

The conformance document shall contain a statement that indicates the full name, number, and date of the standards that apply. The conformance document may also list international software standards that are available for use by a Conforming POSIX.1 {2} and POSIX.9 Application. Applicable characteristics where documentation is required by one of these standards, or by standards of government bodies, may also be included.

The conformance document shall describe the behavior of the implementation for all implementation-defined features defined in this standard. This requirement shall be met by listing these features and providing either a specific reference to the system documentation or providing full syntax and semantics of these features. The conformance may specify the behavior of the implementation for those features where this standard states that implementations may vary or where features are identified as undefined or unspecified.

The phrases “shall document” or “shall be documented” in this standard mean that documentation of the feature shall appear in the conformance document, as described previously, unless the system documentation is explicitly mentioned.

The system documentation should also contain the information found in the conformance document.

See POSIX.1 {2} 1.3 for a description of a POSIX.1 {2} conformance document.

[2]This is an archival standard which is identical to the obsolete ISO 1539:1980. ANSI documents can be obtained from the American National Standards Institute, 1430 Broadway, New York, NY 10018, USA.

1.3.2 Application Conformance

All applications claiming conformance to this standard shall use only ANSI X3.9-1978 (FORTRAN 77) as described in 1.3.3.1 and shall fall within one of the following categories:

1.3.2.1 Strictly Conforming POSIX.9 Application

A Strictly Conforming POSIX.9 Application is an application that requires only the facilities in POSIX.9 and the facilities described in POSIX.1 {2} 1.3 and the language standard described in 1.3.3.1 for a *Strictly Conforming POSIX.1 Application.* Such an application shall accept any behavior described in this standard as *unspecified* or *implementation-defined.*

1.3.2.2 Conforming POSIX.9 Application

An IEEE Conforming POSIX.9 Application is an application that uses only the facilities described in this standard and the facilities described in POSIX.1 {2} 1.3 for a *Conforming POSIX.1 Application.*

1.3.3 Language-Dependent Services for the FORTRAN 77 Programming Language

ANSI X3.9-1978 {3} (FORTRAN 77), will provide the definition of any FORTRAN 77 language-dependent features used by POSIX.9. Section 8 provides new facilities and amplifications to facilities provided by the FORTRAN 77 standard. Any implementation claiming conformance to POSIX.9 shall provide the facilities of the FORTRAN 77 standard {3} that are referenced in Section 8 of POSIX.9 and any additions and amplifications required by Section 8 and 1.3.3.1.

Although POSIX.9 references FORTRAN 77 features to describe its own requirements, conformance to the FORTRAN 77 standard {3} is unnecessary for conformance to this standard. Any Fortran implementation that does not conflict with FORTRAN 77 and provides the facilities stipulated in Section 8 and 1.3.3.1 may claim conformance. However, it shall clearly state that its Fortran language does not conform to the FORTRAN 77 standard.

NOTE — FORTRAN 77 is considered to be contained within the ISO/IEC 1539: 1991 (Fortran 90), i.e., all features in FORTRAN 77 are considered to be part of Fortran 90. While Fortran 90 features need not be acceptable to FORTRAN 77, FORTRAN 77 features are acceptable under Fortran 90. To be able to use the Fortran 90 features, a separate bindings standard for Fortran 90 will be developed at a later time.

1.3.3.1 FORTRAN 77 Language Binding

ANSI X3.9-1978 {3} (FORTRAN 77) is used as the basis for this FORTRAN 77 language bindings to ISO/IEC 9945-1: 1990 {2}. Implementations claiming conformance to this standard must supply the FORTRAN 77 features required by this document, such as the intrinsic function facility.

One extension to FORTRAN 77 is required by POSIX.9. FORTRAN 77 specifies that “a symbolic name takes the form of one to six letters or digits, the first of which must be a letter.” This document assumes that the language implementation can accept specified symbolic names that are longer than six characters. Subroutine and function names, in particular, are assumed to be longer than six characters by this standard. Furthermore, to permit a FORTRAN 77 implementation to claim conformance to POSIX.1 {2}, names that differ in or before the 31st character position are required to be recognized as distinct names by the language implementation (see POSIX.1 {2} 1.3.5).

1.3.4 Other Language Related Specifications

The FORTRAN 77 standard {3} specifies that at least the 49 defined FORTRAN 77 characters shall exist with some FORTRAN 77-specified ordering requirements in an implementation-defined collating sequence. The FORTRAN 77 functions *CHAR*() and *ICHAR*() shall perform conversions based on that collating sequence.

A FORTRAN 77 character in a POSIX.9-conforming implementation shall be capable of representing all values of a byte define by POSIX.1 {2}. The POSIX.9-conforming implementation shall use a collating sequence which conforms to the FORTRAN 77 standard {3}.

2. Terminology and General Requirements

2.1 Conventions

2.1.1 Typographical Conventions

The following typographical conventions are used in this standard:

1) The *italic* font is used for:
   — Cross references to defined terms within 1.2, 2.2.1, and 2.2.2, or within these sections in POSIX.1 {2}.
   — Symbolic parameters in Synopsis subclauses and in the text that are generally substituted with real values by the application.
   — FORTRAN language data types, variable names, and subroutine/function names (except in Synopsis subclauses)
2) The **bold** font is used with a word in all capital letters, such as
   PATH
   to represent an environment variable, as described in 2.6. It is also used for the term "**NULL** pointer."
3) The `constant-width` (Courier) font is used:
   — For FORTRAN 77 language data types and function names within function Synopsis subclauses
   — To illustrate examples of system input or output where exact usage is depicted
   — For references to C-language syntax and headers
4) HELVETICA font is used in 8.3 to represent a "generic" data type for which the appropriate actual FORTRAN 77 data type is substituted.
5) Symbolic constants returned by many functions and subroutines as error numbers are represented as:
   [ERRNO]
   See 2.4.
6) Symbolic constants or limits defined in certain POSIX.1 {2} headers are represented as:
   {LIMIT}
   See 2.8 and 2.9.

In some cases tabular information is presented "inline"; in others it is presented in a separately labeled table. This arrangement was employed purely for ease of typesetting, and there is no normative difference between these two cases.

The conventions listed previously are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

Notes provided as parts of labeled tables and figures are integral parts of this standard (normative). Footnotes and notes within the body of the text are for information only (nonnormative).

2.1.2 Namespace Conventions

The following naming conventions are used in this standard:

2.1.2.1 subroutine naming:

This standard defines a FORTRAN 77 subroutine interface to POSIX.1 {2} system calls and language-specific service routines. This standard prefixes the names of the POSIX.1 {2} system calls and service routines with the characters

PXF to create a unique name for the corresponding FORTRAN 77 procedures. For consistency, the names for the other service subroutines defined in this standard are also prefixed with the same characters.

2.1.2.2 function naming:

This standard defines a FORTRAN 77 function interface to the functionality provided by certain POSIX.1 {2} macros and service functions. All service functions in this standard that return an integer value are prefixed with the characters *IPXF.*

2.1.2.3 argument naming:

The names of all integer items in the actual argument list of each defined procedure statement in the Synopsis sections begin with one of the letters I, L, M, or N. The names of all items that are *structure handles* or *subroutine handles* (see 2.2.2) in the actual argument list are prefixed with the letter J.

2.2 Definitions

2.2.1 Terminology

For the purposes of this standard, the following definitions from POSIX.1 {2} apply:

2.2.1.1 conformance document: A document provided by an implementor that contains implementation details as described in POSIX.1 {2} 1.3.1.2.

2.2.1.2 implementation defined: An indication that the implementation shall define and document the requirements for correct program constructs and correct data of a value or behavior.

2.2.1.3 may: An indication of an optional feature.

With respect to implementations, the word *may* is to be interpreted as an optional feature that is not required in this standard but can be provided. With respect to Strictly Conforming POSIX.9 Applications, the word *may* means that the optional feature shall not be used.

2.2.1.4 obsolescent: An indication that a certain feature may be considered for withdrawal in future revisions of this standard.

Obsolescent features are retained in this version because of their widespread use. Their use in new applications is discouraged.

2.2.1.5 shall: An indication of a requirement on the implementation or on Strictly Conforming POSIX.9 Applications, where appropriate.

2.2.1.6 should:

1) With respect to implementations, an indication of an implementation recommendation, but not a requirement.
2) With respect to applications, an indication of a recommended programming practice for applications and a requirement for Strictly Conforming POSIX.9 Applications.

2.2.1.7 supported: A condition regarding optional functionality.

Certain functionality in this standard is optional, but the interfaces to that functionality are always required. If the functionality is *supported*, the interfaces work as specified by this standard (except that they do not return the error condition indicated for the unsupported case). If the functionality is not *supported*, the interface shall always return the indication specified for this situation.

2.2.1.8 system documentation: All documentation provided with an implementation, except the conformance document.

Electronically distributed documents for an implementation are considered part of the system documentation.

2.2.1.9 undefined: An indication that this standard imposes no portability requirements on an application's use of an indeterminate value or its behavior with erroneous program constructs or erroneous data.

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application using such behaviors is using extensions, as defined in POSIX.1 {2} 1.3.

2.2.1.10 unspecified: An indication that this standard imposes no portability requirements on applications for a correct program construct or correct data.

Implementations (or other standards) may specify the result of using that value or causing that behavior. An application requiring a specific behavior, rather than tolerating any behavior when using that functionality, is using extensions, as defined in POSIX.1 {2} 1.3.

2.2.2 General Terms

In addition to those terms defined in ISO/IEC 9945-1: 1990 (see POSIX.1 {2} 2.2), the terms defined in this section are used in this standard.

2.2.2.1 component: A member, element, or field of a structure.

2.2.2.2 handle: An integer value that refers to a structure handle or subroutine handle.

Unless otherwise specified in this standard, *handle* refers to a structure handle.

2.2.2.3 structure handle: An integer value that refers to a unique instance of a structure.

An existing (structure) handle is a handle that references an existing instance of a structure.

2.2.2.4 subroutine handle: An integer value that refers to a unique instance of a subroutine.

2.2.2.5 intent: A description of whether the actual argument is used by a defined subprogram as an *input* argument (intent=IN), as an *output* argument (intent=OUT), or as both an input and output argument (intent=INOUT).

2.2.2.6 newline delimited: An indication that the newline character is used as a delimiter.

2.2.2.7 POSIX-based FORTRAN I/O file: A FORTRAN 77 file associated with a POSIX.1 {2} file descriptor that is connected to a FORTRAN 77 unit.

2.2.2.8 significant trailing blanks: One or more blanks at the end of a character string that are intended to be a meaningful part of the contents of the string.

Unlike strings in the C language, character variables in FORTRAN 77 are of a fixed length and are padded with blanks. That is, if a character variable is assigned a value that contains fewer characters than declared, the remainder of the variable is filled with blanks. Because of this characteristic, it is difficult to determine the difference between a string (e.g., a value assigned to an environment variable) that contains trailing blanks that are part of the string (significant trailing blanks) and a string for which the trailing blanks are only FORTRAN 77-required padding. For example, the strings "myprompt" and "mypromptΔ" (where Δ represents a significant blank that is part of the string) can be indistinguishable, but they are different legally valid prompts.

2.2.2.9 structure: An aggregate data type that allows the grouping of multiple data elements of possibly differing type into a single unit.

Two common implementations of structures are the *struct* in C and the *record* in Pascal. FORTRAN 77 provides no such aggregate data type.

2.2.2.10 text file: A file that contains characters organized into one or more lines.

The lines shall not contain NUL characters and none shall exceed {LINE_MAX} bytes in length, including the newline.

2.2.3 Abbreviations

For the purposes of this standard, the following abbreviations apply:

2.2.3.1 POSIX.1: This standard assumes and uses POSIX.1 {2} to mean ISO/IEC 9945-1: 1990 {2}. References to sections and terms in that standard will be indicated using this term, e.g., "POSIX.1 {2} 2.3" will imply a reference to Section 2.3 of ISO/IEC 9945-1: 1990 and "POSIX.1 {2} *fork*()" will imply the function *fork*() described in that standard.

2.2.3.2 POSIX.9: This standard.

2.2.3.3 FORTRAN 77: This standard assumes and uses FORTRAN 77 to mean ANSI X3.9-1978 {3} plus the extension for long identifier names described in 1.3.3.1.

2.3 FORTRAN 77 Language Bindings Concepts

The following subsections present many of the design issues addressed in the development of this FORTRAN 77 Bindings standard. The discussion here is intended to be largely functional, i.e., describing only specific problems and their solutions. The accompanying rationale (Annex A) discusses in more detail the design objectives and alternatives that were considered in the development of this standard. With the exception of 2.3, the sections of the rationale correspond directly to the sections of this standard and to the sections of the POSIX.1 {2} rationale.

Because the POSIX.1 {2} system services are defined in the C language and use many language features that are not available in FORTRAN 77, many of the issues presented below are the result of differences and incompatibilities between these two languages. The main goals were to achieve access to all required POSIX.1 {2} functionality while following FORTRAN 77 as closely as possible and to allow consistent treatment of the exceptional cases.

2.3.1 System Headers

System headers containing definitions of *symbolic constants* and *macros* are used extensively throughout POSIX.1 {2}. These header files are intended for inclusion in application programs through the use of the C-language #include mechanism; however, FORTRAN 77 provides no similar inclusion capability, so methods had to be devised to allow the required header definitions to be accessed from FORTRAN 77 programs.

2.3.1.1 Symbolic Constants

The POSIX.1 {2} system headers contain the definitions of symbolic constants intended for use throughout the POSIX.1 {2} programming environment. These symbolic constants can be accessed from FORTRAN 77 with defined procedures. An overview of these procedures is given below:

- An integer function that returns the value of the named constant. This function can be used as an in-line call and provides no error checking.
- A logical function that indicates if the named constant is defined. This functionality is similar to the *feature test macro* in a C-based POSIX.1 {2} system.
- A subroutine that returns an argument containing the named constant. This subroutine provides full error checking.

Further information, including definitions of the interfaces that shall be provided, is given in 8.2.

2.3.1.2 Macros

Where functionality in POSIX.1 {2} is provided by macros, this standard specifies FORTRAN 77 functions. Definitions and descriptions for the functions that provide the functionality of those POSIX.1 {2} macros are included in the appropriate sections of this standard.

2.3.2 Data Types

Incompatibilities between the data types of the C and FORTRAN 77 languages caused many issues in the development of this standard. These incompatibilities can be divided into two categories: language-defined data types and data type definition capabilities.

2.3.2.1 Primitive Data Types

FORTRAN 77 does not provide a facility for type definition such as the *typedef* in the C language. Each of the primitive data types defined in POSIX.1 {2}, as well as any additional implementation-defined primitive types, is equated with a corresponding FORTRAN 77 intrinsic type in order to be used in FORTRAN 77 applications. Specifically, the FORTRAN 77 INTEGER data type shall be used as a substitute for defined arithmetic types (see POSIX.1 {2} 2.5).

2.3.2.2 Numeric Range of Integer Data

Many functions defined in POSIX.1 {2} make use of the *unsigned integer* data type provided by C to double the range of an argument or returned value. FORTRAN 77 does not provide such a data type. An implementation of this standard may choose to utilize an available sign bit of the FORTRAN 77 INTEGER data type to extend the range of such values to a range equivalent to that provided by the C bindings. Instances where this is allowable are indicated throughout the text of this standard. A support routine is defined (see 8.11) to allow these extended-range integer values to be compared.

2.3.2.3 Aggregate Data Types

Many of the service interfaces defined in POSIX.1 {2} require the use of aggregate data types that do not map to FORTRAN 77. FORTRAN 77 does not provide any mechanism for the construction or use of aggregate data types, creating a serious conflict.

The solution to this problem in the FORTRAN 77 bindings involves the use of *data abstraction*: Through the use of additional subroutines to access and manipulate the aggregate data, the underlying data structures are largely hidden from the FORTRAN 77 source code. It is the responsibility of the FORTRAN 77 programmer to maintain variables corresponding to the individual components of the aggregate data, but the programmer need not worry about the details of the actual implementation of the aggregate. The basic model of this data abstraction is used as follows:

— The programmer calls a subroutine to “create” an instance of the desired aggregate data type; this subroutine returns a handle that the programmer subsequently uses in order to reference and/or manipulate the data. The handle is guaranteed to fit in an integer variable, and a valid handle is guaranteed to be nonzero.
— The programmer uses additional subroutines to load values into or extract values from the aggregate data. These subroutines are passed the handle of the desired aggregate and the name of the specific component that is to be accessed. Notice that the programmer has direct control over only one component at a time.
— When an instance of an aggregate is no longer required, a subroutine can be called to release it.
— A subroutine is defined to duplicate contents of an instance of an aggregate.

Further information, including definitions of the subroutines that shall be provided, is given in 8.3.

2.3.2.3.1 List of Aggregate Data Types

The structure types shown in Table 2.1 shall be accessible through the techniques described previously in this section. The components of these structures are enumerated and described in the section indicated.

Table 2.1—Structure Types

Structure Name	Reference
sigset	3.3.1 and 3.3.3
sigaction	3.3.4
utsname	4.4.1
tms	4.5.2
dirent	5.1.1
stat	5.6.1
utimbuf	5.6.6
flock	6.5.2
termios	7.1.2
group	9.2.1
passwd	9.2.2

An implementation may provide additional structures to be used in an implementation-defined manner. Any such structures may be defined by an implementation to be able to be manipulated using these same structure manipulation subroutines.

POSIX.1 {2} allows the implementation to support additional components of many of these structures. In an implementation of POSIX.9 that corresponds to an implementation of POSIX.1 {2} that allows such extensions, access to these additional components shall be provided using these same structure manipulation subroutines.

2.3.2.4 Character Variables and String Manipulation

Data contained in a FORTRAN 77 *string* [a dummy argument declared as CHARACTER*(*)] is padded with blanks if necessary to match the declared string length. Because of this, it is difficult to differentiate between string data that is intended to contain trailing blanks and data that has simply been padded with blanks in order to match the declared string variable length. To allow this distinction, an extra argument is passed to or from procedures that have a string argument.

For procedures in which the string is *returned from* the system, this extra argument shall contain the *actual* length of the data assigned to the string. This length value can be zero, which indicates the equivalent of a **NULL** string indicating that the value of the string is undefined. If the length of the character argument is insufficient to contain the data to be returned from the system, *IERROR* shall be set to *[ETRUNC],* the data shall be truncated to fit the string, and the length argument shall contain the original length of the data before truncation.

For procedures in which the string is being *passed to* the system, this extra argument contains the *intended* length of the string contents, which is not necessarily the fixed, maximum length of the string variable. A value of zero passed as the length of the string data shall indicate that trailing blanks are to be stripped and ignored, or, if the string contains only blanks, shall indicate the equivalent of a **NULL** string.

2.3.2.5 Pointers

C-language pointers are used throughout POSIX.1 {2}; however, FORTRAN 77 does not have a pointer data type. In cases where POSIX.1 {2} specifies functionality dependent on the use or detection of a **NULL** pointer, the behavior has been modified slightly in this binding.

All uses of structures in POSIX.1 {2} are through pointers, i.e., structures are passed by reference to system functions. In conjunction with the methods defined in this binding for accessing and manipulating structured data, an object called a *handle* is used in the FORTRAN 77 interfaces where POSIX.1 {2} uses pointers to structures. A handle is an abstract reference to the aggregate data and does not require any direct manipulation by the FORTRAN 77 programmer. See 2.3.2.3 and 8.3 for further discussion of aggregate data and the use of handles.

Although FORTRAN 77 permits subroutines and functions declared as EXTERNAL to be passed as arguments to another procedure, there is no way in FORTRAN 77 to store a pointer to a subroutine for later use. This functionality shall be provided by the subroutines described in 8.4.

2.4 Error Numbers

Most functions in POSIX.1 {2} provide an error number in the external system variable *errno*, which is defined in the C language as:

```
extern int errno;
```

In this standard, the interface specification for subroutines that can result in error conditions contains an extra out argument, *IERROR*. Unless otherwise specified, a value of zero returned in this argument indicates that no error has occurred and a nonzero value indicates that an error *has* occurred. In this case, the value of other out arguments are undefined unless otherwise specified.

The following symbolic names identify additional errors that can occur in the use of this standard. The values represented by these names shall be unique and shall not conflict with error numbers specified in POSIX.1 {2}.

[ENONAME]	Invalid constant, structure, or component name.
[ENOHANDLE]	Handle not created.
[ETRUNC]	The declared length of the out character argument is insufficient to contain the string to be returned. (See 2.3.2.4.)
[EARRAYLEN]	For *get* routines, the number of array elements to be returned exceeds *IALEN*, and only the first *IALEN* elements of the array argument have been set. For *set* routines, *IALEN* exceeds the number of array elements in the target array. Only the available elements of the array in the target array have been set.
[EEND]	End of file, record, or directory stream has been encountered.

2.5 Primitive System Data Types

Because all of the primitive system data types shall be *arithmetic* types (see POSIX.1 {2} 2.5), the FORTRAN 77 INTEGER data type shall be used as a substitute for each of the listed types. However, when a primitive data type is defined in the C bindings to POSIX.1 {2} as an unsigned integer, other issues may arise. See 2.3.2.1 and 2.3.2.2 for further discussion.

2.6 Environment Description

The individual members of the environment (see POSIX.1 {2} 2.6) are examined using the *PXFGETENV*() subroutine, modified using the *PXFSETENV*() subroutine, and cleared by the *PXFCLEARENV*() subroutine (see 4.6.1).

2.7 FORTRAN 77 Language Definitions

FORTRAN language terms and symbols used in this standard are defined by FORTRAN 77.

2.8 Numerical Limits

2.8.1 FORTRAN 77 Language Limits

Certain limits used in this standard are defined in the FORTRAN 77 programming language.

2.8.2 Minimum Values

The symbolic constants specifying minimum values described in POSIX.1 {2} 2.8.2 shall be accessible through calls to any of the *PXFCONST* procedures (see 8.2).

2.8.3 Run-Time Increasable Values

The magnitude limitations specified in POSIX.1 {2} 2.8.3 shall be accessible through a call to *PXFSYSCONF*() (see 4.8).

2.8.4 Run-Time Invariant Values (Possible Indeterminate)

The run time invariant values specified in POSIX.1 {2} 2.8.4 shall be accessible through a call to *PXFSYSCONF*() (see 4.8).

2.8.5 Pathname Variable Values

The pathname variable values specified in POSIX.1 {2} 2.8.5 shall be accessible through a call to *PXFPATHCONF*() (see 5.7).

2.8.6 Invariant Values

The invariant values specified in POSIX.1 {2} 2.8.6 shall be accessible through a call to *PXFSYSCONF*() (see 4.8).

2.9 Symbolic Constants

The symbolic constants defined in POSIX.1 {2} (see POSIX.1 {2} 2.9) and POSIX.9 shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

2.9.1 Constants for FORTRAN 77 I/O to STDIO Translation

The following symbolic constants shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

STDIN_UNIT	The value of the FORTRAN 77 unit identifier associated with a preconnected input file.
STDOUT_UNIT	The value of the FORTRAN 77 unit identifier associated with a preconnected output file.
STDERR_UNIT	The value of the FORTRAN 77 unit identifier associated with a preconnected error file.

The values of these constants shall be integers in the range 0–9. Portable applications using units for other files should use values outside these ranges.

3. Process Primitives

3.1 Process Creation and Execution

3.1.1 Process Creation

Subroutine: *PXFFORK*()

3.1.1.1 Synopsis

```
SUBROUTINE PXFFORK (IPID, IERROR)
INTEGER IPID, IERROR
```

3.1.1.2 Description

The *PXFFORK*() subroutine shall provide the same functionality as the POSIX.1 {2} function *fork*() (see POSIX.1 {2} 3.1) except that files opened with the FORTRAN 77 OPEN statement are not required to have file descriptors (see 8.5).

Arguments for *PXFFORK*() correspond to the arguments for *fork*(), as shown in Table 3.1.

Table 3.1—Arguments for *PXFFORK()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IPID	ret_value	OUT	
IERROR	ret_value/errno	OUT	

3.1.1.3 Errors

Possible error conditions for *PXFFORK*() are identical to those for the POSIX.1 {2} function *fork*(). Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.1.2 Execute a File

Subroutines: *PXFEXECV*(), *PXFEXECVE*(), *PXFEXECVP*()

3.1.2.1 Synopsis

```
      SUBROUTINE PXFEXECV (PATH, LENPATH, ARGV, LENARGV, IARGC, IERROR)
      INTEGER LENPATH, LENARGV(0:IARGC-1), IARGC, IERROR
      CHARACTER*(*) PATH, ARGV(0:IARGC-1)

      SUBROUTINE PXFEXECVE (PATH, LENPATH, ARGV, LENARGV, IARGC,
     +   ENV, LENENV, IENVC, IERROR)
      INTEGER LENPATH, LENARGV(0:IARGC-1), IARGC, LENENV(IENVC), IENVC, IERROR
      CHARACTER*(*) PATH, ARGV(0:IARGC-1), ENV(IENVC)
```



```
SUBROUTINE PXFEXECVP (FILE, LENFILE, ARGV, LENARGV, IARGC, IERROR)
INTEGER LENFILE, LENARGV(0:IARGC-1), IARGC, IERROR
CHARACTER*(*) FILE, ARGV(0:IARGC-1)
```

3.1.2.2 Description

The *PXFEXECV* subroutines shall provide the same functionality as the corresponding POSIX.1 {2} *exec* functions in POSIX.1 {2} (see POSIX.1 {2} 3.1).

The lengths of the *ARGV* and *ENV* arrays are explicitly passed in the arguments *IARGC* and *IENVC* respectively. The arrays *ARGV* and *LENARGV* shall be dimensioned at least as large as *IARGC*, and the arrays *ENV* and *LENENV* shall be dimensioned at least as large as *IENVC*. While these arrays may be dimensioned greater than required, the arguments *IARGC* and *IENVC* specify the number of the elements at the beginning of the respective arrays that are to be used by the subroutine. The string length of each element of the *ARGV* and *ENV* arrays is passed in the corresponding element of the *LENARGV* and *LENENV* arrays.

Arguments for these subroutines correspond to the arguments for the corresponding POSIX.1 {2} process creation and execution functions, as shown in Table 3.2.

Table 3.2—Arguments for the *PXFEXEC...()* Subroutines

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
LENPATH	--	IN	Length of PATH; see 2.3.2.4
FILE	file	IN	
LENFILE	--	IN	Length of FILE; see 2.3.2.4
ARGV	argv	IN	
IARGC	--	IN	Number of elements in ARGV
LENARGV	--	IN	Length of elements in ARGV; see 2.3.2.4
ENV	envp	IN	
IENVC	--	IN	Number of elements in ENV
LENENV	--	IN	Length of elements in ENV; see 2.3.2.4
IERROR	ret_value/errno	OUT	

3.1.2.3 Errors

Possible error conditions for the *PXFEXECV*() family of subroutines are identical to those for the POSIX.1 {2} *exec*() family. Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.2 Process Termination

Process termination shall provide the functionality defined by the POSIX.1 {2} function *_exit*() (see POSIX.1 {2} 3.2.2). There are two kinds of process termination, normal and abnormal. Normal termination occurs by the execution of the FORTRAN 77 END statement in the FORTRAN 77 main program, the FORTRAN 77 STOP statement, or the

PXFFASTEXIT() or *PXFEXIT*() subroutine (see 3.2.2 and 8.12). Abnormal termination occurs when certain signals are received, as defined in POSIX.1 {2} 3.3.

A parent process may suspend its execution to wait for the termination of a child process with the *PXFWAIT*() or *PXFWAITPID*() subroutines.

3.2.1 Wait for Process Termination

Subroutines: *PXFWAIT*(), *PXFWAITPID*()

3.2.1.1 Synopsis

```
SUBROUTINE PXFWAIT (ISTAT, IRETPID, IERROR)
INTEGER ISTAT, IRETPID, IERROR

SUBROUTINE PXFWAITPID (IPID, ISTAT, IOPTIONS, IRETPID, IERROR
INTEGER IPID, ISTAT, IOPTIONS, IRETPID, IERROR)
```

3.2.1.2 Description

The PXFWAIT() and PXFWAITPID() subroutines shall provide the same functionality as the POSIX.1 {2} functions wait() and waitpid() (see POSIX.1 {2} 3.2).

The value for the IOPTIONS arguments to the PXFWAITPID() subroutine is based on the symbolic constants defined for waitpid(). These constants shall be accessible through any of the PXFCONST() procedures (see 8.2). These values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7). Arguments for PXFWAIT() and PXFWAITPID() correspond to the arguments for wait() and waitpid(), as shown in Table 3.3.

Table 3.3—Arguments for *PXFWAIT()* and *PXFWAITPID()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISTAT	stat_loc	OUT	
IPID	pid	IN	
IRETPID	ret_value	OUT	
IOPTIONS	options	IN	
IERROR	ret_value/errno	OUT	

The following functions may be used to interpret the ISTAT argument, as defined in POSIX.1 {2} 3.2.

```
LOGICAL FUNCTION PXFWIFEXITED (ISTAT)
INTEGER ISTAT

INTEGER FUNCTION IPXFWEXITSTATUS (ISTAT)
INTEGER ISTAT

LOGICAL FUNCTION PXFWIFSIGNALED (ISTAT)
INTEGER ISTAT
```



```
INTEGER FUNCTION IPXFWTERMSIG (ISTAT)
INTEGER ISTAT

LOGICAL FUNCTION PXFWIFSTOPPED (ISTAT)
INTEGER ISTAT

INTEGER FUNCTION IPXFWSTOPSIG (ISTAT)
INTEGER ISTAT
```

3.2.1.3 Errors

Possible error conditions for *PXFWAIT*() and *PXFWAITPID*() are identical to those for the POSIX.1 {2} functions *wait*() and *waitpid*(). *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.2.2 Terminate a Process

Subroutine: *PXFFASTEXIT*()

3.2.2.1 Synopsis

```
SUBROUTINE PXFFASTEXIT (ISTATUS)
INTEGER ISTATUS
```

3.2.2.2 Description

The *PXFFASTEXIT*() subroutine shall provide the same functionality as the POSIX.1 {2} function *_exit*() (see POSIX.1 {2} 3.2). There is no possible return from *PXFFASTEXIT*(), and no *IERROR* argument is defined for *PXFFASTEXIT*(). Arguments for *PXFFASTEXIT*() correspond to the arguments for *_exit*(), as shown in Table 3.4.

Table 3.4—Arguments for *PXFFASTEXIT()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISTATUS	status	IN	

3.3 Signals

3.3.1 Signal Concepts

3.3.1.1 Signal Names

The values for use with the signal procedures are based on the symbolic constants defined for POSIX.1 {2} signals. These constants shall be accessible through any of the *PXFCONST*() procedures (see 8.2).

The symbolic constants *SIG_DFL* and *SIG_IGN* represent values that shall not be identical to any value returned by *PXFGETSUBHANDLE*() (see 8.4 and 3.3.1.3). When used as the handle for the signal-catching subroutine, they shall cause the signal-specific default action or ignore signal action respectively.

The subroutine *PXFSTRUCTCREATE*() with the string ‘sigset’ given as the *STRUCTNAME* argument may be used to obtain an instance of the *sigset_t* type as defined in POSIX.1 {2} 3.3.1. There are no defined components of this

structure, and the contents of the structure may not be altered with the structure-component manipulation subroutines. Instead, the subroutines defined in 3.3.3 shall be used.

3.3.1.2 Signal Generation and Delivery

3.3.1.3 Signal Actions

On delivery of a signal (see POSIX.1 {2} 3.3), the system may call a user-defined signal-catching subroutine (see 3.3.4). This signal-catching subroutine shall be defined with a single integer argument. The argument contains the number of the signal being delivered.

3.3.1.4 Signal Effects on Other Subroutines

Signals may affect the behavior of certain procedures defined by this standard if delivered to a process while it is executing such a procedure. Specifically, nonzero values of *IERROR* for each of the system services are not guaranteed to be reliable in the presence of signals.

3.3.2 Send a Signal to a Process

Subroutine: *PXFKILL*()

3.3.2.1 Synopsis

```
SUBROUTINE PXFKILL (IPID, ISIG, IERROR)
INTEGER IPID, ISIG, IERROR
```

3.3.2.2 Description

The *PXFKILL*() subroutine shall provide the same functionality as the POSIX.1 {2} function *kill*() (see POSIX.1 {2} 3.3).

The value of the desired signal (specified by *ISIG*) shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2). Arguments for *PXFKILL*() correspond to the arguments for *kill*(), as shown in Table 3.5.

Table 3.5—Arguments for *PXFKILL()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IPID	pid	IN	
ISIG	sig	IN	
IERROR	ret_value/errno	OUT	

3.3.2.3 Errors

Possible error conditions for *PXFKILL* are identical to those for the POSIX.1 {2} function *kill*().()*IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.3.3 Manipulate Signal Sets

Subroutines: *PXFSIGEMPTYSET*(), *PXFSIGFILLSET*() *PXFSIGADDSET*(), *PXFSIGDELSET*(), *PXFSIGISMEMBER*()

3.3.3.1 Synopsis

```
SUBROUTINE PXFSIGEMPTYSET (JSIGSET, IERROR)
INTEGER JSIGSET, IERROR

SUBROUTINE PXFSIGFILLSET (JSIGSET, IERROR)
INTEGER JSIGSET, IERROR

SUBROUTINE PXFSIGADDSET (JSIGSET, ISIGNO, IERROR)
INTEGER JSIGSET, ISIGNO, IERROR

SUBROUTINE PXFSIGDELSET (JSIGSET, ISIGNO, IERROR)
INTEGER JSIGSET, ISIGNO, IERROR

SUBROUTINE PXFSIGISMEMBER (JSIGSET, ISIGNO, ISMEMBER, IERROR)
INTEGER JSIGSET, ISIGNO, IERROR
LOGICAL ISMEMBER
```

3.3.3.2 Description

These subroutines shall provide the same functionality as the equivalent POSIX.1 {2} signal set manipulation functions (see POSIX.1 {2} 3.3).

The *PXFSIGISMEMBER()* procedure shall return a logical value .TRUE. in the argument *ISMEMBER* if the specified signal is a member of the specified set or a value of .FALSE. if it is not.

Applications shall call either *PXFSIGEMPTYSET()* or *PXFSIGFILLSET()* at least once for each *sigset* structure prior to any other use of that structure. If the structure is not initialized in this way, the results are undefined.

This standard defines *sigset* as a structure. An instance of a sigset shall be created using *PXFSTRUCTCREATE()* before manipulation using these subroutines.

Arguments for these subroutines correspond to the arguments for the corresponding POSIX.1 {2} signal set manipulation functions, as shown in Table 3.6.

Table 3.6—Arguments for the *PXFSIG...()* Subroutines

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
JSIGSET	set	IN	1.
ISMEMBER	ret_value	OUT	
ISIGNO	signo	IN	
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘sigset’,...); see 8.3.1.

3.3.3.3 Errors

Possible error conditions for these subroutines are identical to those for the corresponding signal set manipulation functions defined in POSIX.1 {2}. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.3.4 Examine and Change Signal Action

Subroutine: *PXFSIGACTION()*

3.3.4.1 Synopsis

```
SUBROUTINE PXFSIGACTION (ISIG, JSIGACT, JOSIGACT, IERROR)
INTEGER ISIG, JSIGACT, JOSIGACT, IERROR
```

3.3.4.2 Description

The *PXFSIGACTION()* subroutine shall provide the same functionality as the POSIX.1 {2} function *sigaction()* (see POSIX.1 {2} 3.3). Arguments for *PXFSIGACTION()* correspond to the arguments for *sigaction()*, as shown in Table 3.7.

Table 3.7—Arguments for *PXFSIGACTION()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISIG	sig	IN	
JSIGACT	act	IN	1.
JOSIGACT	oact	OUT	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘sigaction’,...); see 8.3.1.

The functionality obtained in the POSIX.1 {2} function *sigaction()* by passing a **NULL** can be obtained in *PXFSIGACTION* by passing a handle argument with a value of zero.

The values of the symbolic constants *SIG_DFL* and *SIG_IGN* shall be accessible through calls to any of the *PXFCONST()* procedures (see 8.2) and can be used as values for the signal handler component. They shall cause the signal-specific default action or ignore signal action respectively, as defined by POSIX.1 {2} 3.3.1.3.

The *PXFSTRUCTCREATE()* subroutine (see 8.3.1) with the string ‘sigaction’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *sigaction* structure as defined in POSIX.1 {2} 3.3. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JSIGACTION, COMPNAM, IVALUE, IERROR)
INTEGER JSIGACTION, IVALUE, IERROR

SUBROUTINE PXFINTSET(JSIGACTION, COMPNAM, IVALUE, IERROR)
INTEGER JSIGACTION, IVALUE, IERROR
```

where *JSIGACTION* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 3.8.

Table 3.8—Components for *sigaction* Structure

POSIX.1 Component	COMPNAM	Structure Procedure Used to Access
sa_handler	'sa_handler'	PXFINTGET,PXFINTSET
sa_mask	'sa_mask'	PXFINTGET,PXFINTSET
sa_flags	'sa_flags'	PXFINTGET,PXFINTSET

The *sa_handler* component shall be a subroutine handle obtained from a call to *PXFGETSUBHANDLE()* (see 8.4), obtained from a previous call to *PXFSIGACTION()*, or that shall contain the value of one of the symbolic constants *SIG_DFL* or *SIG_IGN*. The *sa_mask* component shall be a sigset structure handle (see 3.3) obtained from a call to *PXFSTRUCTCREATE()*.

Values of the *sa_flags* component can be used to modify the behavior of the signal specified in a call to *PXFSIGACTION()*. Values of *sa_flags* are composed of the flag bits used by the *sigaction()* function as defined in POSIX.1 {2} 3.3. The values of these flags shall be bitwise distinct and can be combined with the use of the inclusive OR function (see 8.7). The flag names are constants for which the values shall be accessible through calls to any of the *PXFCONST()* procedures (see 8.2).

3.3.4.3 Errors

Possible error conditions for *PXFSIGACTION()* are identical to those for the POSIX.1 {2} function *sigaction()*. Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.3.5 Examine and Change Blocked Signals

Subroutine: *PXFSIGPROCMASK()*

3.3.5.1 Synopsis

```
SUBROUTINE PXFSIGPROCMASK (IHOW, JSIGSET, JOSIGSET, IERROR)
INTEGER IHOW, JSIGSET, JOSIGSET, IERROR
```

3.3.5.2 Description

The *PXFSIGPROCMASK()* subroutine shall provide the same functionality as the POSIX.1 {2} function *sigprocmask()* (see POSIX.1 {2} 3.3). Arguments for *PXFSIGPROCMASK()* correspond to the arguments for *sigprocmask()*, as shown in Table 3.9.

Table 3.9—Arguments for *PXFSIGPROCMASK()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IHOW	how	IN	
JSIGSET	set	IN	1.
JOSIGSET	oset	OUT	1.
IERROR	ret_value/errno	OUT	

1.Handle obtained from PXFSTRUCTCREATE ('sigset',...); see 8.3.1.

The functionality obtained in the POSIX.1 {2} function *sigprocmask()* by passing a **NULL** may be obtained in *PXFSIGPROCMASK* by passing a handle argument with a value of zero.

3.3.5.3 Errors

Possible error conditions for *PXFSIGPROCMASK()* are identical to those for the POSIX.1 {2} function *sigprocmask()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.3.6 Examine Pending Signals

Subroutine: *PXFSIGPENDING()*

3.3.6.1 Synopsis

```
SUBROUTINE PXFSIGPENDING (JSIGSET, IERROR
INTEGER JSIGSET, IERROR
```

3.3.6.2 Description

The *PXFSIGPENDING()* subroutine shall provide the same functionality as the POSIX.1 {2} function *sigpending()* (see POSIX.1 {2} 3.3). Arguments for *PXFSIGPENDING()* correspond to the arguments for *sigpending()*, as shown in Table 3.10.

Table 3.10—Arguments for *PXFSIGPENDING()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
JSIGSET	set	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE ('sigset',...); see **8.3.1**.

3.3.6.3 Errors

Possible error conditions for *PXFSIGPENDING()* are identical to those for the POSIX.1 {2} function *sigpending()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.3.7 Wait for a Signal

Subroutine: *PXFSIGSUSPEND()*

3.3.7.1 Synopsis

```
SUBROUTINE PXFSIGSUSPEND (JSIGSET, IERROR)
INTEGER JSIGSET, IERROR
```

3.3.7.2 Description

The *PXFSIGSUSPEND()* subroutine shall provide the same functionality as the POSIX.1 {2} function *sigsuspend()* (see POSIX.1 {2} 3.3). Arguments for *PXFSIGSUSPEND()* correspond to the arguments for *sigsuspend()*, as shown in Table 3.11.

Table 3.11—Arguments for *PXFSIGSUSPEND()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
JSIGSET	sigmask	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE ('sigset',...); see 8.3.1.

3.3.7.3 Errors

Since the *PXFSIGSUSPEND()* subroutine suspends process execution indefinitely, there is no successful completion return value.

Possible error conditions for *PXFSIGSUSPEND()* are identical to those for the POSIX.1 {2} function *sigsuspend()*. If any of these conditions occur, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.4 Timer Operations

3.4.1 Schedule Alarm

Subroutine: *PXFALARM()*

3.4.1.1 Synopsis

```
SUBROUTINE PXFALARM (ISECONDS, ISECLEFT, IERROR)
INTEGER ISECONDS, ISECLEFT, IERROR
```

3.4.1.2 Description

The *PXFALARM()* subroutine shall provide the same functionality as the POSIX.1 {2} function *alarm()* (see POSIX.1 {2} 3.4).

If there is a previous *PXFALARM()* request with time remaining, the number of seconds until the previous request would have generated a SIGALARM signal is returned in *ISECLEFT.* Otherwise, *ISECLEFT* shall contain a value of zero upon return.

Arguments for *PXFALARM()* correspond to the arguments for *alarm(),* as shown in Table 3.12.

Table 3.12—Arguments for *PXFALARM()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISECONDS	seconds	IN	
ISECLEFT	ret_value	OUT	1.
IERROR	ret_value/errno	OUT	

1. Value may exceed the range of a signed integer; see 2.3.2.2.

3.4.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *alarm()* function. Upon successful completion of *PXFALARM(),* the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.4.2 Suspend Process Execution

Subroutine: *PXFPAUSE()*

3.4.2.1 Synopsis

```
SUBROUTINE PXFPAUSE (IERROR)
INTEGER IERROR
```

3.4.2.2 Description

The *PXFPAUSE()* subroutine shall provide the same functionality as the POSIX.1 {2} function *pause()* (see POSIX.1 {2} 3.4). Arguments for *PXFPAUSE()* correspond to the arguments for *pause(),* as shown in Table 3.13.

Table 3.13—Arguments for *PXFPAUSE()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IERROR	ret_value/errno	OUT	

3.4.2.3 Errors

Since the *PXFPAUSE()* subroutine suspends process execution indefinitely, there is no successful completion return value.

Possible error conditions for *PXFPAUSE()* are identical to those for the POSIX.1 {2} function *pause().* If any of these conditions occur, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

3.4.3 Delay Process Execution

Subroutine: *PXFSLEEP()*

3.4.3.1 Synopsis

```
SUBROUTINE PXFSLEEP (ISECONDS, ISECLEFT, IERROR)
INTEGER ISECONDS, ISECLEFT, IERROR
```

3.4.3.2 Description

The *PXFSLEEP()* subroutine shall provide the same functionality as the POSIX.1 {2} function *sleep()* (see POSIX.1 {2} 3.4).

If *PXFSLEEP()* returns because the requested time has elapsed, the value of *ISECLEFT* is set to zero. If *PXFSLEEP()* returns due to delivery of a signal, *ISECLEFT* shall contain upon return the unslept amount (the requested time minus the time actually slept) in seconds.

Arguments for *PXFSLEEP()* correspond to the arguments for *sleep()*, as shown in Table 3.14.

Table 3.14—Arguments for *PXFSLEEP()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISECONDS	seconds	IN	
ISECLEFT	ret_value	OUT	1.
IERROR	ret_value/errno	OUT	

1. Value may exceed the range of a signed integer; see 2.3.2.2.

3.4.3.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *sleep()* function. Upon successful completion of *PXFSLEEP()*, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4. Process Environment

4.1 Process Identification

4.1.1 Get Process and Parent Process IDs

Subroutines: *PXFGETPID()*, *PXFGETPPID()*

4.1.1.1 Synopsis

```
SUBROUTINE PXFGETPID (IPID, IERROR)
INTEGER IPID, IERROR

SUBROUTINE PXFGETPPID (IPID, IERROR)
INTEGER IPID, IERROR
```

4.1.1.2 Description

The *PXFGETPID()* and *PXFGETPPID()* subroutines shall provide the same functionality as the POSIX.1 {2} functions *getpid()* and *getppid()* (see POSIX.1 {2} 4.1). Arguments for *PXFGETPID()* and *PXFGETPPID()* correspond to the arguments for *getpid()* and *getppid()*, as shown in Table 4.1.

Table 4.1—Arguments for *PXFGETPID()* and *PXFGETPPID()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IPID	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.1.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *getpit()* function or the *getppid()* function. Upon successful completion of *PXFGETPID()* or *PXFGETPPID()*, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.2 User Identification

4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs

Subroutines: *PXFGETUID(), PXFGETEUID(), PXFGETGID(), PXGETEGID()*

4.2.1.1 Synopsis

```
SUBROUTINE PXFGETUID (IUID, IERROR)
INTEGER IUID, IERROR

SUBROUTINE PXFGETUEID (IEUID, IERROR)
INTEGER IEUID, IERROR

SUBROUTINE PXFGETGID (IGID, IERROR)
INTEGER IGID, IERROR

SUBROUTINE PXFGETEGID (IEGID, IERROR)
INTEGER IEGID, IERROR
```

4.2.1.2 Description

The *PXFGETUID(), PXFGETEUID(), PXFGETGID(),* and *PXFGETEGID()* subroutines shall provide the same functionality as the POSIX.1 {2} functions *getuid(), geteuid(), getgid(),* and *getegid()* (see POSIX.1 {2} 4.2). Arguments for these subroutines correspond to the arguments for the corresponding POSIX.1 {2} user identification functions, as shown in Table 4.2.

Table 4.2—Arguments for the *PXFGET...ID()* Subroutines

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IUID	ret_value	OUT	
IEUID	ret_value	OUT	
IGID	ret_value	OUT	
IEGID	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.2.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *get...id()* family of functions. Upon successful completion of any of the *PXFGET...ID()* family of subroutines, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.2.2 Set User and Group IDs

Subroutines: *PXFSETUID()*, *PXFSETGID()*

4.2.2.1 Synopsis

```
SUBROUTINE PXFSETUID (IUID, IERROR)
INTEGER IUID, IERROR

SUBROUTINE PXFSETGID (IGID, IERROR)
INTEGER IGID, IERROR
```

4.2.2.2 Description

The *PXFSETUID()* and *PXFSETGID()* subroutines shall provide the same functionality as the POSIX.1 {2} functions *setuid()* and *setgid()* (see POSIX.1 {2} 4.2). Arguments for *PXFSETUID()* and *PXFSETGID()* correspond to the arguments for *setuid()* and *setgid()*, as shown in Table 4.3.

Table 4.3—Arguments for *PXFSETUID()* and *PXFSETGID()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IUID	uid	IN	
IGID	gid	IN	
IERROR	ret_value/errno	OUT	

4.2.2.3 Errors

Possible error conditions for *PXFSETUID0* and *PXFGETUID()* are identical to those for the POSIX.1 {2} functions *setuid()* and *getuid()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.2.3 Get Supplementary Group IDs

Subroutine: *PXFGETGROUPS()*

4.2.3.1 Synopsis

```
SUBROUTINE PXFGETGROUPS (IGIDSETSIZE, IGROUPLIST, NGROUPS, IERROR)
INTEGER IGIDSETSIZE, IGROUPLIST(IGIDSETSIZE), NGROUPS, IERROR
```

4.2.3.2 Description

The *PXFGETGROUPS()* subroutine shall provide the same functionality as the POSIX.1 {2} function *getgroups()* (see POSIX.1 {2} 4.2), including the special case behavior when the *IGIDSETSIZE* argument is zero.

Arguments for *PXFGETGROUPS()* correspond to the arguments for *getgroups()*, as shown in Table 4.4.

Table 4.4—Arguments for *PXFGETGROUPS()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IGROUPLIST	grouplist	OUT	
IGIDSETSIZE	gidsetsize	IN	
NGROUPS	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.2.3.3 Errors

Possible error conditions for *PXFGETGROUPS()* are identical to those for the POSIX.1 {2} function *getgroups()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.2.4 Get User Name

Subroutine: *PXFGETLOGIN()*

4.2.4.1 Synopsis

```
SUBROUTINE PXFGETLOGIN (S, ILEN, IERROR)
CHARACTER*(*) S
INTEGER ILEN, IERROR
```

4.2.4.2 Description

PXFGETLOGIN() shall provide the same functionality as the POSIX.1 {2} function *getlogin()* (see POSIX.1 {2} 4.2). Arguments for *PXFGETLOGIN()* correspond to the arguments for *getlogin()*, as shown in Table 4.5.

Table 4.5—Arguments for *PXFGETLOGIN()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
S	ret_value	OUT	
ILEN	—	OUT	Length of S; see 2.3.2.4
IERROR	ret_value/errno	OUT	

4.2.4.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *getlogin*() function. Upon successful completion of *PXFGETLOGIN*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETLOGIN*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ETRUNC] The declared length of the argument *S* is insufficient to contain the string to be returned. (See 2.3.2.4.)

4.3 Process Groups

4.3.1 Get Process Group ID

Subroutine: *PXFGETPGRP*()

4.3.1.1 Synopsis

```
SUBROUTINE PXFGETPGRP (IPGRP, IERROR)
INTEGER, IPGRP, IERROR
```

4.3.1.2 Description

The *PXFGETPGRP*() subroutine shall provide the same functionality as the POSIX.1 {2} function *getpgrp*() (see POSIX.1 {2} 4.3). Arguments to *PXFGETPGRP*() correspond to the arguments for *getpgrp()*, as shown in Table 4.6.

Table 4.6—Arguments for *PXFGETPGRP()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IPGRP	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.3.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *getpgrp*() function. Upon successful completion of *PXFGETPGRP()*, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.3.2 Create Session and Set Process Group ID

Subroutine: *PXFSETSID*()

4.3.2.1 Synopsis

```
SUBROUTINE PXFSETSID (ISID, IERROR)
INTEGER ISID, IERROR
```

4.3.2.2 Description

The *PXFSETSID*() subroutine shall provide the same functionality as the POSIX.1 {2} function *setsid*() (see POSIX.1 {2} 4.3). Arguments for *PXFSETSID*() correspond to the arguments for *setsid()*, as shown in Table 4.7.

Table 4.7—Arguments for *PXFSETSID()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISID	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.3.2.3 Errors

Possible error conditions for *PXFSETSID*() are identical to those for the POSIX.1 {2} function *setsid(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.3.3 Set Process Group ID for Job Control

Subroutine: *PXFSETPGID*()

4.3.3.1 Synopsis

```
SUBROUTINE PXFSETPGID (IPID, IPGID, IERROR)
INTEGER IPID, IPGID, IERROR
```

4.3.3.2 Description

The *PXFSETPGID*() subroutine shall provide the same functionality as the POSIX.1 {2} function *setpgid*() (see POSIX.1 {2} 4.3). Arguments for *PXFSETPGID*() correspond to the arguments for *setpgid()*, as shown in Table 4.8.

Table 4.8—Arguments for *PXFSETPGID()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IPID	pid	IN	
IPGID	pgid	IN	
IERROR	ret_value/errno	OUT	

4.3.3.3 Errors

Possible error conditions for *PXFSETPGID*() are identical to those for the POSIX.1 {2} function *setpgid(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.4 System Identification

4.4.1 Get System Name

Subroutine: *PXFUNAME*()

4.4.1.1 Synopsis

```
SUBROUTINE PXFUNAME (JUTSNAME, IERROR)
INTEGER JUSTSNAME, IERROR
```

4.4.1.2 Description

The *PXFUNAME*() subroutine shall provide the same functionality as the POSIX.1 {2} function *uname*() (see POSIX.1 {2} 4.4). Arguments for *PXFUNAME*() correspond to the arguments for *uname()*, as shown in Table 4.9.

Table 4.9—Arguments for *PXFUNAME()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
JUTSNAME	name	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘utsname’,...); see 8.3.1.

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘utsname’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *utsname* structure as defined in POSIX.1 {2} 4.4. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFSTRGET (JUTSNAME, COMPNAM, SVALUE, ILEN, IERROR)
INTEGER JUTSNAME, ILEN, IERROR
CHARACTER*(*) SVALUE
```

where *JUTSNAME* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 4.10.

Table 4.10—Components for *utsname* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
sysname	‘sysname’	PXFSTRGET
nodename	‘nodename’	PXFSTRGET
release	‘release’	PXFSTRGET
version	‘version’	PXFSTRGET
machine	‘machine’	PXFSTRGET

4.4.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *uname*() function. Upon successful completion of *PXFUNAME(),* the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero vaue to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.5 Time

4.5.1 Get System Time

Subroutine: *PXFTIME*()

4.5.1.1 Synopsis

```
SUBROUTINE PXFTIME (ITIME, IERROR)
INTEGER ITIME, IERROR
```

4.5.1.2 Description

The value of time is always returned in the argument *ITIME*. No indirection or separate return value is used or necessary. Otherwise, the *PXFTIME*() subroutine shall provide the same functionality as the POSIX.1 {2} function *time*() (see POSIX.1 {2} 4.5). Arguments for *PXFTIME*() correspond to the arguments for *time*(), as shown in Table 4.11.

Table 4.11—Arguments for *PXFTIME()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ITIME	*tloc	OUT	1.
IERROR	ret_value/errno	OUT	

1. Value may exceed the range of a signed integer; see 2.3.2.2.

4.5.1.3 Errors

Possible error conditions for *PXFTIME*() are identical to those for the POSIX.1 {2} function *time*(). *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.5.2 Get Process Times

Subroutine: *PXFTIMES*()

4.5.2.1 Synopsis

```
SUBROUTINE PXFTIMES (JTMS, ITIME, IERROR)
INTEGER JTMS, ITIME, IERROR
```

4.5.2.2 Description

The *PXFTIMES*() subroutine shall provide the same functionality as the POSIX.1 {2} function *times*() (see POSIX.1 {2} 4.5). Arguments for *PXFTIMES* correspond to the arguments for *times()*, as shown in Table 4.12.

Table 4.12—Arguments for *PXFTIMES()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
JTMS	buffer	IN	1.
ITIME	ret_value	OUT	2.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘tms’,...); see 8.3.1.

2. Value may exceed the range of a signed integer; see 2.3.2.2.

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘tms’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *tms* structure as defined in POSIX.1 {2} 4.5. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JTMS, COMPNAM, IVALUE, IERROR)
INTEGER JTMS, IVALUE, IERROR
```

where *JTMS* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 4.13.

Table 4.13—Components for *tms* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
tms_utime	‘tms_utime’	PXFINTGET
tms_stime	‘tms_stime’	PXFINTGET
tms_cutime	‘tms_cutime’	PXFINTGET
tms_cstime	‘tms_cstime’	PXFINTGET

The value of the *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* components may exceed the range of a signed integer. See 2.3.2.2.

4.5.2.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *times*() function. Upon successful completion of *PXFTIMES()*, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.6 Environment Variables

4.6.1 Environment Access

Subroutines: *PXFGETENV*(), *PXFSETENV*(), *PXFCLEARENV*()

4.6.1.1 Synopsis

```
SUBROUTINE PXFGETENV (NAME, LENNAME, VALUE, LENVAL, IERROR)
CHARACTER*(*) NAME, VALUE
INTEGER LENNAME, LENVAL, IERROR

SUBROUTINE PXFSETENV (NAME, LENNAME, NEW, LENNEW, IOVERWRITE, IERROR)
CHARACTER*(*) NAME, NEW
INTEGER LENNAME, LENNEW, IOVERWRITE, IERROR

SUBROUTINE PXFCLEARENV (IERROR)
INTEGER IERROR
```

4.6.1.2 Description

The argument *VALUE* shall be a valid user-space character variable; the static return area provided by POSIX.1 {2} is not supported. Upon completion of *PXFGETENV*(), *VALUE* shall contain the value for the specified name *NAME*, and *LENVAL* shall contain the length of the value. If the specified variable is found but has no value, the value of *LENVAL* shall be set to zero and *VALUE* shall be filled with blanks. If the specified variable cannot be found, the condition *EINVAL* shall be returned in *IERROR*. Otherwise, the *PXFGETENV*() subroutine shall provide the same functionality as the POSIX.1 {2} function *getenv*() (see POSIX.1 {2} 4.6).

The *PXFSETENV*() subroutine shall search the environment list (see POSIX.1 {2} 2.6) for a string of the form *name=value*, where *name* is the contents of the character argument *NAME*. If no such string is present, *PXFSETENV*() shall add a string of the form *name=new* to the environment list, where *new* is the contents of the character argument *NEW*. Otherwise, if the *IOVERWRITE* argument is nonzero, *PXFSETENV*() either shall change the existing value to the contents of *NEW* or shall delete the string *name=value* and add the string *name=new*. The values assigned to the environment variables are restricted as specified in POSIX.1 {2} 2.6.

The *PXFCLEARENV*() subroutine shall clear the process environment. No environment variables are defined immediately after a call to *PXFCLEARENV*().

Arguments for *PXFGETENV*() correspond to the arguments for *getenv*(), as shown in Table 4.14.

Table 4.14—Arguments for *PXFGETENV()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
NAME	name	IN	
LENNAME	--	IN	Length of NAME; see 2.3.2.4
VALUE	ret_value	OUT	
LENVAL	--	OUT	Returned length of VALUE
IERROR	ret_value/errno	OUT	

4.6.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *getenv*() function. Upon successful completion of *PXFGETENV*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETENV*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[EINVAL] The variable *NAME* is not in the environment list.

[ETRUNC] The declared length of the argument *VALUE* is insufficient to contain the string to be returned. (See 2.3.2.4.)

Upon successful completion of *PXFSETENV*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFSETENV*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ENOMEM] Not enough memory is available to create the necessary structures.

Upon successful completion of *PXFLCEARENV*(), the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.7 Terminal Identification

4.7.1 Generate Terminal Pathname

Subroutine: *PXFCTERMID*()

4.7.1.1 Synopsis

```
SUBROUTINE PXFCTERMID (S, ILEN, IERROR)
CHARACTER*(*) S
INTEGER ILEN, IERROR
```

4.7.1.2 Description

The argument *S* shall be a valid user-space character variable; the static return area provided by POSIX.1 {2} is not supported, and the maximum length of the returned string indicated by the symbolic constant L_ctermid provided by

POSIX.1 {2} is not supported. The argument *ILEN* shall contain zero if the pathname that would refer to the controlling terminal cannot be determined or if *PXFCTERMID* is unsuccessful. If the length of the returned value is longer than the length of the passed character variable *S*, the return value shall be truncated.

Otherwise, the *PXFCTERMID*() subroutine shall provide the same functionality as the POSIX.1 {2} function *ctermid*() (see POSIX.1 {2} 4.7).

Upon completion, *S* shall contain a string that represents the controlling terminal for the current process, and *ILEN* shall contain the length of the string. Arguments for *PXFCTERMID*() correspond to the arguments for *ctermid*(), as shown in Table 4.15.

Table 4.15—Arguments for *PXFCTERMID()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
S	s	OUT	
ILEN	--	OUT	Length of S; see 2.3.2.4
IERROR	ret_value/errno	OUT	

4.7.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *ctermid*() function. Upon successful completion of *PXFCTERMID*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFCTERMID*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ETRUNC] The declared length of the argument *S* is insufficient to contain the string to be returned. (See 2.3.2.4.)

4.7.2 Determine Terminal Device Name

Subroutines: *PXFTTYNAME*(), *PXFISATTY*()

4.7.2.1 Synopsis

```
SUBROUTINE PXFTTYNAME (IFILDES, S, ILEN, IERROR)
INTEGER IFILDES, ILEN, IERROR
CHARACTER*(*) S

SUBROUTINE PXFISATTY (IFILDES, ISATTY, IERROR)
INTEGER IFILDES, IERROR
LOGICAL ISATTY
```

4.7.2.2 Description

PXFTTYNAME() and *PXFISATTY*() shall provide the same functionality as the corresponding POSIX.1 {2} functions *ttyname*() and *isatty*() (see POSIX.1 {2} 4.7). Upon return, the value of *ISATTY* shall be .TRUE. if *IFILDES* contains a valid file descriptor associated with a terminal. Otherwise, it shall be .FALSE..

Upon completion of *PXFTTYNAME*(), *S* shall contain the terminal pathname, and *ILEN* shall contain the length of the string. If the length of the returned value is longer than the length of the passed character variable *S*, the return value shall be truncated.

Arguments for *PXFTTYNAME*() and *PXFISATTY*() correspond to the arguments for *ttyname*() and *isatty*(), as shown in Table 4.16.

Table 4.16—Arguments for *PXFTTYNAME()* and *PXFISATTY()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
S	ret_value	OUT	
ILEN	--	OUT	Length of S; see 2.3.2.4
ISATTY	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.7.2.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *ttyname*() function. Upon successful completion of *PXFTTYNAME*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFTTYNAME*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ETRUNC]	The declared length of the argument *S* is insufficient to contain the string to be returned. (See 2.3.2.4.)
[EBADF]	*IFILDES* is not a valid file descriptor.

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *isatty*() function. Upon successful completion of *PXFISATTY*(), the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

4.8 Configurable System Variables

4.8.1 Get Configurable System Variables

Subroutine: *PXFSYSCONF*()

4.8.1.1 Synopsis

```
SUBROUTINE PXFSYSCONF (NAME, IVAL, IERROR)
INTEGER NAME
INTEGER IVAL, IERROR
```

4.8.1.2 Description

The *PXFSYSCONF*() subroutine shall provide the same functionality as the POSIX.1 {2} function *sysconf*() (see POSIX.1 {2} 4.8). *NAME* is an integer value representing a symbolic system variable. Values for *NAME* shall be obtained through calls to any of the *PXFCONST*() procedures (see 8.2).

Access to the special symbol {CLK_TCK} is not included since POSIX.1 {2} declares such access to be obsolescent.

Arguments for *PXFSYSCONF*() correspond to the arguments for *sysconf*(), as shown in Table 4.17.

Table 4.17—Arguments for *PXFSYSCONF()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
NAME	name	IN	
IVAL	ret_value	OUT	
IERROR	ret_value/errno	OUT	

4.8.1.3 Errors

Possible error conditions for *PXFSYSCONF*() are identical to those for the POSIX.1 {2} function *sysconf(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5. Files and Directories

5.1 Directories

5.1.1 Format of Directory Entries

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘dirent’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *dirent* structure as defined in POSIX.1 {2} 5.1. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFSTRGET(JDIRENT, COMPNAM, SVALUE, ILEN, IERROR)
INTEGER, JDIRENT, ILEN, IERROR
CHARACTER*(*) SVALUE
```

where *JDIRENT* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 5.1.

Table 5.1—Components for *dirent* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
d_name	‘d_name’	PXFSTRGET

5.1.2 Directory Operations

Subroutines: *PXFOPENDIR()*, *PXFREADDIR()*, *PXFREWINDDIR()*, *PXFCLOSEDIR*()

5.1.2.1 Synopsis

```
SUBROUTINE PXFOPENDIR (DIRNAME, LENDIRNAME, IOPENDIRID, IERROR)
CHARACTER*(*) DIRNAME
INTEGER LENDIRNAME, IOPENDIRID, IERROR

SUBROUTINE PXFREADDIR (IDIRID, JDIRENT, IERROR)
INTEGER IDIRID, JDIRENT, IERROR

SUBROUTINE PXFREWINDDIR (IDIRID, IERROR)
INTEGER IDIRID, IERROR

SUBROUTINE PXFCLOSEDIR (IDIRID, IERROR)
INTEGER IDIRID, IERROR
```

5.1.2.2 Description

The type DIR (see POSIX.1 {2} 5.1) is represented by a directory identifier contained in the integer arguments *IDIRID* and *IOPENDIRID*. This integer shall contain an identifier for a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. A unique value of *IOPENDIRID* shall be returned by a call to *PXFOPENDIR*(), and *IDIRID* shall become undefined upon the matching call to *PXFCLOSEDIR*(). Otherwise *PXFOPENDIR()*, *PXFCLOSEDIR()*, *PXFREWINDDIR*(), and *PXFREADDIR*() shall provide the same functionality as the corresponding POSIX.1 {2} functions (see POSIX.1 {2} 5.1).

When the end of the directory stream is reached by *PXFREADDIR*(), the components in the *dirent* structure shall be undefined, and *IERROR* shall be set to the value indicated in 5.1.2.3.

Arguments for these subroutines correspond to the arguments for the corresponding POSIX.1 {2} directory operations (see POSIX.1 {2} 5.1), as shown in Table 5.2.

Table 5.2—Arguments for the *PXF...DIR()* Subroutines

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
DIRNAME	dirname	IN	
LENDIRNAME	dirname	IN	Length of DIRNAME; see 2.3.2.4
IOPENDIRID	ret_value	OUT	
IDIRID	dirp	IN	
JDIRENT	--	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘dirent’,...); see 8.3.1.

5.1.2.3 Errors

Possible error conditions for these subroutines include those for the directory operations defined in POSIX.1 {2}, as well as the conditions listed below. If any of these conditions occur, the argument *IERROR* shall be set to the

corresponding nonzero value specified by the POSIX.1 {2} function. In addition to the POSIX.1 {2} specified conditions, if any of the following conditions occur, these subroutines shall set the argument to the corresponding value. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[EEND] Following a call to *PXFREADDIR*(), indicates that all directory entries have been read.

5.2 Get Working Directory

5.2.1 Change Current Working Directory

Subroutine: *PXFCHDIR*()

5.2.1.1 Synopsis

```
SUBROUTINE PXFCHDIR (PATH, ILEN, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IERROR
```

5.2.1.2 Description

The *PXFCHDIR*() subroutine shall provide the same functionality as the POSIX.1 {2} function *chdir*() (see POSIX.1 {2} 5.2). Arguments for *PXFCHDIR*() correspond to the arguments for *chdir*(), as shown in Table 5.3.

Table 5.3—Arguments for *PXFCHDIR()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IERROR	ret_value/errno	OUT	

5.2.1.3 Errors

Possible error conditions for *PXFCHDIR*() are identical to those for the POSIX.1 {2} function *chdir*(). Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.2.2 Get Working Directory Pathname

Subroutine: *PXFGETCWD*()

5.2.2.1 Synopsis

```
SUBROUTINE PXFGETCWD (BUF, ILEN, IERROR)
CHARACTER*(*) BUF
INTEGER ILEN, IERROR
```


5.2.2.2 Description

The *size* argument in the POSIX.1 {2} function *getcwd*() is superfluous in *PXFGETCWD*() since the size of *BUF* is defined by the declaration of the character variable. *ILEN* is the returned length of the string written into *BUF*. If *PXFGETCWD*() is unsuccessful, *ILEN* is set to zero. Otherwise, *PXFGETCWD*() shall provide the same functionality as the POSIX.1 {2} function *getcwd*() (see POSIX.1 {2} 5.2). Arguments for *PXFGETCWD*() correspond to the arguments for *getcwd*(), as shown in Table 5.4.

Table 5.4—Arguments for *PXFGETCWD()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
BUF	buf	OUT	
--	size	--	
ILEN	--	OUT	Length of returned string in BUF
IERROR	ret_value/errno	OUT	

5.2.2.3 Errors

Except for replacing the ERANGE error with the ETRUNC error below, possible error conditions for *PXFGETCWD*() are identical to those for the POSIX.1 {2} function *getcwd*(). Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ETRUNC] The declared length of the argument *BUF* is insufficient to contain the string that is to be returned. (See 2.3.2.4.)

5.3 General File Creation

5.3.1 Open a File

Subroutine: *PXFOPEN*()

5.3.1.1 Synopsis

```
SUBROUTINE PXFOPEN (PATH, ILEN, IOPENFLAG, IMODE, IFILDES, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IOPENFLAG, IMODE, IFILDES, IERROR
```

5.3.1.2 Description

The *PXFOPEN*() subroutine shall provide the same functionality as the POSIX.1 {2} function *open*() (see POSIX.1 {2} 5.3).

The values of the symbolic constants defined in POSIX.1 {2} for *open*() and necessary for construction of the *IOPENFLAG* and *IMODE* arguments shall be accessible through any of the *PXFCONST*() procedures (see 8.2). These values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7).

Arguments for *PXFOPEN*() correspond to the arguments for *open*(), as shown in Table 5.5.

Table 5.5—Arguments for *PXFOPEN()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IOPENFLAG	oflag	IN	
IMODE	mode	IN	1.
IFILDES	ret_value	OUT	
IERROR	ret_value/errno	OUT	

1. Utilized only if IOPENFLAG contains O_CREAT and if the file did not previously exist.

5.3.1.3 Errors

Possible error conditions for *PXFOPEN*() are identical to those for the POSIX.1 {2} function *open(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.3.2 Create a New File or Rewrite an Existing One

Subroutine: *PXFCREAT*()

5.3.2.1 Synopsis

```
SUBROUTINE PXFCREAT (PATH, ILEN, IMODE, IFILDES, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IMODE, IFILDES, IERROR
```

5.3.2.2 Description

The *PXFCREAT*() subroutine shall provide the same functionality as the POSIX.1 {2} function *creat*() (see POSIX.1 {2} 5.3).

The values of the symbolic constants defined in POSIX.1 {2} for *creat*() and necessary for construction of the *IMODE* argument shall be accessible through any of the *PXFCONST*() procedures (see 8.2). The values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7). Arguments for *PXFCREAT*() correspond to the arguments for *creat*(), as shown in Table 5.6.

Table 5.6—Arguments for *PXFCREAT()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IMODE	mode	IN	
IFILDES	ret_value	OUT	
IERROR	ret_value/errno	OUT	

5.3.2.3 Errors

Possible error conditions for *PXFCREAT*() are identical to those for the POSIX.1 {2} function *creat*(). *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.3.3 Set File Creation Mask

Subroutine: *PXFUMASK*()

5.3.3.1 Synopsis

```
SUBROUTINE PXFUMASK (ICMASK, IPREVCMASK, IERROR)
INTEGER ICMASK, IPREVCMASK, IERROR
```

5.3.3.2 Description

The *PXFUMASK*() subroutine shall provide the same functionality as the POSIX.1 {2} function *umask*() (see POSIX.1 {2} 5.3).

The values of the symbolic constants necessary to compose the *ICMASK*() argument shall be accessible through any of the *PXFCONST*() procedures (see 8.2). The values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7).

The file creation mask of the process shall also be used when determining the permission bits for the creation of POSIX-based FORTRAN I/O files (see 8.5.1). Arguments for *PXFUMASK*() correspond to the arguments for *umask*(), as shown in Table 5.7.

Table 5.7—Arguments for *PXFUMASK()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ICMASK	cmask	IN	
IPREVCMASK	ret_value	OUT	
IERROR	ret_value/errno	OUT	

5.3.3.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *umask*() function. Upon successful completion of *PXFUMASK*(), the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.3.4 Link to a File

Subroutine: *PXFLINK*()

5.3.4.1 Synopsis

```
SUBROUTINE PXFLINK (EXISTING, LENEXIST, NEW, LENNEW, IERROR)
CHARACTER*(*) EXISTING, NEW
INTEGER LENEXIST, LENNEW, IERROR
```

5.3.4.2 Description

The *PXFLINK*() subroutine shall provide the same functionality as the POSIX.1 {2} function *link*() (see POSIX.1 {2} 5.3). Arguments for *PXFLINK*() correspond to the arguments for *link*(), as shown in Table 5.8.

Table 5.8—Arguments for *PXFLINK()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
EXISTING	existing	IN	
LENEXIST	--	IN	Length of EXISTING; see 2.3.2.4
NEW	new	IN	
LENNEW	--	IN	Length of NEW; see 2.3.2.4
IERROR	ret_value/errno	OUT	

5.3.4.3 Errors

Possible error conditions for *PXFLINK*() are identical to those for the POSIX.1 {2} function *link(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.4 Special File Creation

5.4.1 Make a Directory

Subroutine: *PXFMKDIR*()

5.4.1.1 Synopsis

```
SUBROUTINE PXFMKDIR (PATH, ILEN, IMODE, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IMODE, IERROR
```

5.4.1.2 Description

The *PXFMKDIR*() subroutine shall provide the same functionality as the POSIX.1 {2} function *mkdir*() (see POSIX.1 {2} 5.4).

The values of the symbolic constants defined in POSIX.1 {2} for *mkdir*() and necessary for construction of the *IMODE* argument shall be accessible through any of the *PXFCONST*() procedures (see 8.2). These values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7). Arguments for *PXFMKDIR*() correspond to the arguments for *mkdir*(), as shown in Table 5.9.

Table 5.9—Arguments for *PXFMKDIR()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IMODE	mode	IN	
IERROR	ret_value/errno	OUT	

5.4.1.3 Errors

Possible error conditions for *PXFMKDIR*() are identical to those for the POSIX.1 {2} function *mkdir(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.4.2 Make a FIFO Special File

Subroutine: *PXFMKFIFO*()

5.4.2.1 Synopsis

```
SUBROUTINE PXFMKFIFO (PATH, ILEN, IMODE, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IMODE, IERROR
```

5.4.2.2 Description

The *PXFMKFIFO*() subroutine shall provide the same functionality as the POSIX.1 {2} function *mkfifo*() (see POSIX.1 {2} 5.4).

The values of the symbolic constants defined in POSIX.1 {2} for *mkfifo*() and necessary for construction of the *IMODE* argument shall be accessible through any of the *PXFCONST*() procedures (see 8.2). These values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7). Arguments for *PXFMKFIFO*() correspond to the arguments for *mkfifo*(), as shown in Table 5.10.

Table 5.10—Arguments for *PXFMKFIFO()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IMODE	mode	IN	
IERROR	ret_value/errno	OUT	

5.4.2.3 Errors

Possible error conditions for *PXFMKFIFO*() are identical to those for the POSIX.1 {2} function *mkfifo(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.5 File Removal

5.5.1 Remove Directory Entries

Subroutine: *PXFUNLIN*()

5.5.1.1 Synopsis

```
SUBROUTINE PXFUNLINK (PATH, ILEN, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IERROR
```

5.5.1.2 Description

The *PXFUNLINK*() subroutine shall provide the same functionality as the POSIX.1 {2} function *unlink*() (see POSIX.1 {2} 5.5). Arguments for *PXFUNLINK*() correspond to the arguments for *unlink*(), as shown in Table 5.11.

Table 5.11—Arguments for *PXFFUNLINK()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IERROR	ret_value/errno	OUT	

5.5.1.3 Errors

Possible error conditions for *PXFUNLINK*() are identical to those for the POSIX.1 {2} function *unlink(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.5.2 Remove a Directory

Subroutine: *PXFRMDIR*()

5.5.2.1 Synopsis

```
SUBROUTINE PXFRMDIR (PATH, ILEN, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IERROR
```

5.5.2.2 Description

The *PXFRMDIR*() subroutine shall provide the same functionality as the POSIX.1 {2} function *rmdir*() (see POSIX.1 {2} 5.5). Arguments for *PXFRMDIR*() correspond to the arguments for *rmdir*(), as shown in Table 5.12.

Table 5.12—Arguments for *PXFRMDIR()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IERROR	ret_value/errno	OUT	

5.5.2.3 Errors

Possible error conditions for *PXFRMDIR*() are identical to those for the POSIX.1 {2} function *rmdir(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.5.3 Rename a File

Subroutine: *PXFRENAME*()

5.5.3.1 Synopsis

```
SUBROUTINE PXFRENAME (OLD, LENOLD, NEW, LENNEW, IERROR)
CHARACTER*(*) OLD, NEW
INTEGER LENOLD, LENNEW, IERROR
```

5.5.3.2 Description

The *PXFRENAME*() subroutine shall provide the same functionality as the POSIX.1 {2} function *rename*() (see POSIX.1 {2} 5.5). Arguments for *PXFRENAME*() correspond to the arguments for *rename*(), as shown in Table 5.13.

Table 5.13—Arguments for *PXFRENAME()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
OLD	old	IN	
LENOLD	--	IN	Length of OLD; see 2.3.2.4
NEW	new	IN	
LENNEW	--	IN	Length of NEW; see 2.3.2.4
IERROR	ret_value/errno	OUT	

5.5.3.3 Errors

Possible error conditions for *PXFRENAME*() are identical to those for the POSIX.1 {2} function *rename*(). *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.6 File Characteristics

5.6.1 File Characteristics: Header and Data Structure

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘stat’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *stat* structure as defined in POSIX.1 {2} 5.6. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JSTAT, COMPNAM, IVALUE, IERROR)
INTEGER JSTAT, IVALUE, IERROR
```

where *JSTAT* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 5.14.

Table 5.14—Components for *stat* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
st_mode	‘st_mode’	PXFINTGET
st_ino	‘st_ino’	PXFINTGET
st_dev	‘st_dev’	PXFINTGET
st_nlink	‘st_nlink’	PXFINTGET
st_uid	‘st_uid’	PXFINTGET
st_gid	‘st_gid’	PXFINTGET
st_size	‘st_size’	PXFINTGET
st_atime	‘st_atime’	PXFINTGET
st_mtime	‘st_mtime’	PXFINTGET
st_ctime	‘st_ctime’	PXFINTGET

The value of the *st_atime*, *st_mtime*, and *st_ctime* components may exceed the range of a signed integer. See 2.3.2.2.

5.6.1.1 File Types

The following functions shall test whether a file is of the specified type, performing the same functions and returning the same logical result as the macros defined in POSIX.1 {2} 5.6. The value *M* supplied to the functions is the value of *st_mode* obtained from PXFINTGET(JSTAT, 'st_mode',...).

```
LOGICAL FUNCTION PXFISDIR (M)
INTEGER M

LOGICAL FUNCTION PXFISCHR (M)
INTEGER M

LOGICAL FUNCTION PXFISBLK (M)
INTEGER M

LOGICAL FUNCTION PXFISREG (M)
INTEGER M

LOGICAL FUNCTION PXFISFIFO (M)
INTEGER M
```

5.6.1.2 File Modes

All constants and masks defined in POSIX.1 {2} 5.6 for encoding the *st_mode* value shall be recognized by any of the *PXFCONST*() procedures (see 8.2).

5.6.1.3 Time Entries

The time-related structure components shall be interpreted as described in POSIX.1 {2} 5.6.1.3.

5.6.2 Get File Status

Subroutines: *PXFSTAT*(), *PXFFSTAT*()

5.6.2.1 Synopsis

```
SUBROUTINE PXFSTAT (PATH, ILEN, JSTAT, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, JSTAT, IERROR

SUBROUTINE PXFFSTAT (IFILDES, JSTAT, IERROR)
INTEGER IFILDES, JSTAT, IERROR
```

5.6.2.2 Description

The *PXFSTAT*() and *PXFFSTAT*() subroutines shall provide the same functionality as the POSIX.1 {2} functions *stat*() and *fstat*() (see POSIX.1 {2} 5.6). Arguments for *PXFSTAT*() and *PXFFSTAT*() correspond to the arguments for *stat*() and *fstat*(), as shown in Table 5.15.

Table 5.15—Arguments for *PXFSTAT()* and *PXFFSTAT()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IFILDES	fildes	IN	
JSTAT	buf	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE ('stat',...); see 8.3.1.

5.6.2.3 Errors

Possible error conditions for *PXFSTAT*() and *PXFFSTAT*() are identical to those for the POSIX.1 {2} functions *stat*() and *fstat*(). *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.6.3 Check File Accessibility

Subroutine: *PXFACCESS*()

5.6.3.1 Synopsis

```
SUBROUTINE PXFACESS (PATH, ILEN, IAMODE, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IAMODE, IERROR
```

5.6.3.2 Description

The *PXFACCESS*() subroutine shall provide the same functionality as the POSIX.1 {2} function *access*() (see POSIX.1 {2} 5.6).

The values of the symbolic constants defined in POSIX.1 {2} for *access*() and necessary for construction of the *IAMODE* argument shall be accessible through any of the *PXFCONST*() procedures (see 8.2). These values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7).

Arguments for *PXFACCESS*() correspond to the arguments for *access*(), as shown in Table 5.16.

Table 5.16—Arguments for *PXFACCESS()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IAMODE	amode	IN	
IERROR	ret_value/errno	OUT	

5.6.3.3 Errors

Possible error conditions for *PXFACCESS*() are identical to those for the POSIX.1 {2} function *access*(). Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.6.4 Change File Modes

Subroutine: *PXFCHMOD*()

5.6.4.1 Synopsis

```
SUBROUTINE PXFCHMOD (PATH, ILEN, IMODE, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, IMODE, IERROR
```

5.6.4.2 Description

The *PXFCHMOD*() subroutine shall provide the same functionality as the POSIX.1 {2} function *chmod*() (see POSIX.1 {2} 5.6).

The values of the symbolic constants defined in POSIX.1 {2} for *chmod*() and necessary for construction of the *IMODE* argument shall be accessible through any of the *PXFCONST*() procedures (see 8.2). These values of the symbolic constants shall be distinct and can be combined with the use of the inclusive OR function (see 8.7). Arguments for *PXFCHMOD(* correspond to the arguments for *chmod*(), as shown in Table 5.17.

Table 5.17—Arguments for *PXFCHMOD()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IMODE	mode	IN	
IERROR	ret_value/errno	OUT	

5.6.4.3 Errors

Possible error conditions for *PXFCHMOD*() are identical to those for the POSIX.1 {2} function *chmod*(). *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.6.5 Change Owner and Group of a File

Subroutine: *PXFCHOWN*()

5.6.5.1 Synopsis

```
SUBROUTINE PXFCHOWN (PATH, ILEN, IOWNER, IGROUP, IERROR)
```

```
CHARACTER*(*) PATH
INTEGER ILEN, IOWNER, IGROUP, IERROR
```

5.6.5.2 Description

The *PXFCHOWN*() subroutine shall provide the same functionality as the POSIX.1 {2} function *chown*() (see POSIX.1 {2} 5.6). Arguments for *PXFCHOWN*() correspond to the arguments for *chown*(), as shown in Table 5.18.

Table 5.18—Arguments for *PXFCHOWN()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IOWNER	owner	IN	
IGROUP	group	IN	
IERROR	ret_value/errno	OUT	

5.6.5.3 Errors

Possible error conditions for *PXFCHOWN*() are identical to those for the POSIX.1 {2} function *chown*(). Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.6.6 Set File Access and Modification Times

Subroutine: *PXFUTIME*()

5.6.6.1 Synopsis

```
SUBROUTINE PXFUTIME (PATH, ILEN, JUTIMBUF, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, JUTIMBUF, IERROR
```

5.6.6.2 Description

The *PXFUTIME*() subroutine shall provide the same functionality as the POSIX.1 {2} function *utime*() (see POSIX.1 {2} 5.6).

The functionality obtained in the POSIX.1 {2} function *utime*() by passing a **NULL** can be obtained in *PXFUTIME* by passing a handle argument with a value of zero. Arguments or *PXFUTIME*() correspond to the arguments for *utime*(), as shown in Table 5.19.

Table 5.19—Arguments for *PXFUTIME()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
JUTIMBUF	times	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘utimbuf,...); see 8.3.1.

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘utimbuf’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *utimbuf* structure as defined in POSIX.1 {2} 5.6. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2.):

```
SUBROUTINE PXFINTSET(JUTIMBUF, COMPNAM, IVALUE, IERROR)
INTEGER JUTIMBUF, IVALUE, IERROR
```

where *JUTIMBUF* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 5.20.

Table 5.20—Components for *utimbuf* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
actime	‘actime’	PXFINTSET
modtime	‘modtime’	PXFINTSET

The value of the *actime* and *modtime* components may exceed the range of a signed integer. See 2.3.2.2.

5.6.6.3 Errors

Possible error conditions for *PXFUTIME*() are identical to those for the POSIX.1 {2} function *utime()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

5.7 Configurable Pathname Variables

5.7.1 Get Configurable Pathname Variables

Subroutines: *PXFPATHCONF(), PXFFPATHCONF*()

5.7.1.1 Synopsis

```
SUBROUTINE PXFPATHCONF (PATH, ILEN, NAME, IVAL, IERROR)
CHARACTER*(*) PATH
INTEGER ILEN, NAME, IVAL, IERROR

SUBROUTINE PXFPATHCONF (IFILDES, NAME, IVAL, IERROR)
```

```
INTEGER IFILDES, NAME, IVAL, IERROR
```

5.7.1.2 Description

The *PXFPATHCONF*() and *PXFFPATHCONF*() subroutines shall provide the same functionality as the POSIX.1 {2} functions *pathconf*() and *fpathconf*() (see POSIX.1 {2} 5.7). *NAME* is an integer value representing a symbolic pathname variable. Values for *NAME* shall be obtained through calls to any of the *PXFCONST*() procedures (see 8.2). Arguments for *PXFPATHCONF*() and *PXFFPATHCONF*() correspond to the arguments for *pathconf*() and *fpathconf*(), as shown in Table 5.21.

Table 5.21—Arguments for *PXFPATHCONF()* and *PXFFPATHCONF()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
PATH	path	IN	
ILEN	--	IN	Length of PATH; see 2.3.2.4
IFILDES	fildes	IN	
NAME	name	IN	
IVAL	ret_value	OUT	
IERROR	ret_value/errno	OUT	

5.7.1.3 Errors

Possible error conditions for *PXFPATHCONF*() and *PXFFPATHCONF*() are identical to those for the POSIX.1 {2} functions *pathconf*() and *fpathconf*(). Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6. Input and Output Primitives

6.1 Pipes

6.1.1 Create an Inter-Process Channel

Subroutine: *PXFPIPE*()

6.1.1.1 Synopsis

```
SUBROUTINE PXFPIPE (IREADFD, IWRTFD, IERROR)
INTEGER IREADFD, IWRTFD, IERROR
```

6.1.1.2 Description

The *PXFPIPE*() subroutine shall provide the same functionality as the POSIX.1 {2} function *pipe*() (see POSIX.1 {2} 6.1). Arguments for *PXFPIPE*() correspond to the arguments for *pipe*(), as shown in Table 6.1.

Table 6.1—Arguments for *PXFPIPE()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IREADFD	fildes[0]	IN	
IWRTFD	fildes[1]	IN	
IERROR	ret_value/errno	OUT	

6.1.1.3 Errors

Possible error conditions for *PXFPIPE*() are identical to those for the POSIX.1 {2} function *pipe(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6.2 File Descriptor Manipulation

6.2.1 Duplicate an Open File Descriptor

Subroutines: *PXFDUP(), PXFDUP2*()

6.2.1.1 Synopsis

```
SUBROUTINE PXFDUP (IFILDES, IFID, IERROR)
INTEGER IFILDES, IFID, IERROR

SUBROUTINE PXFDUP2 (IFILDES, IFILDES2, IERROR)
INTEGER IFILDES, IFILDES2, IFID, IERROR
```

6.2.1.2 Description

The *PXFDUP*() and *PXFDUP2*() subroutines shall provide the same functionality as the POSIX.1 {2} functions *dup*() and *dup2*() (see POSIX.1 {2} 6.2).

If *PXFDUP2*() succeeds, then the context of the file open on *IFILDES* has been duplicated into *IFILDES2*. If *PXFDUP2*() fails, then *IFILDES2* should be considered closed or invalid, depending on the value in *IERROR*. Arguments for *PXFDUP*() and *PXFDUP2*() correspond to the arguments for *dup*() and *dup2*(), as shown in Table 6.2.

Table 6.2—Arguments for *PXFDUP()* and *PXFDUP2()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IFILDES2	fildes2	IN	
IFID	ret_value	OUT	
IERROR	ret_value/errno	OUT	

6.2.1.3 Errors

Possible error conditions for *PXFDUP*() and *PXFDUP2*() are identical to those for the POSIX.1 {2} functions *dup*() and *dup2(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6.3 File Descriptor Deassignment

6.3.1 Close a File

Subroutine: *PXFCLOSE*()

6.3.1.1 Synopsis

```
SUBROUTINE PXFCLOSE (IFILDES, IERROR)
INTEGER IFILDES, IERROR
```

6.3.1.2 Description

The *PXFCLOSE()* subroutine shall provide the same functionality as the POSIX.1 {2} function *close*() (see POSIX.1 {2} 6.3). Arguments for *PXFCLOSE*() correspond to the arguments for *close(),* as shown in Table 6.3.

Table 6.3—Arguments for *PXFCLOSE()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IERROR	ret_value/errno	OUT	

6.3.1.3 Errors

Possible error conditions for *PXFCLOSE*() are identical to those for the POSIX.1 {2} function *close(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6.4 Input and Output

6.4.1 Read From a File

Subroutine: *PXFREAD*()

6.4.1.1 Synopsis

```
SUBROUTINE PXFREAD (IFILDES, BUF, NBYTE, NREAD, IERROR)
INTEGER IFILDES
CHARACTER BUF*
INTEGER NBYTE, NREAD, IERROR
```

6.4.1.2 Description

The *PXFREAD*() subroutine shall provide the same functionality as the POSIX.1 {2} function *read*() (see POSIX.1 {2} 6.4). Arguments for *PXFREAD*() correspond to the arguments for *read()*, as shown in Table 6.4.

Table 6.4—Arguments for *PXFREAD()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
BUF	buf	OUT	
NBYTE	nbyte	IN	
NREAD	ret_value	OUT	Undefined if error occurs
IERROR	ret_value/errno	OUT	

6.4.1.3 Errors

Possible error conditions for *PXFREAD*() are identical to those for the POSIX.1 {2} function *read(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6.4.2 Write to a File

Subroutine: *PXFWRITE*()

6.4.2.1 Synopsis

```
SUBROUTINE PXFWRITE (IFILDES, BUF, NBYTE, NWRITTEN, IERROR)
INTEGER IFILDES
CHARACTER BUF(*))
INTEGER NBYTE, NWRITTEN, IERROR
```

6.4.2.2 Description

The *PXFWRITE*() subroutine shall provide the same functionality as the POSIX.1 {2} function *write*() (see POSIX.1 {2} 6.4). Arguments for *PXFWRITE*() correspond to the arguments for *write()*, as shown in Table 6.5.

Table 6.5—Arguments for *PXFWRITE()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
BUF	buf	IN	
NBYTE	nbyte	IN	
NWRITTEN	ret_value	OUT	Undefined if error occurs
IERROR	ret_value/errno	OUT	

6.4.2.3 Errors

Possible error conditions for *PXFWRITE*() are identical to those for the POSIX.1 {2} function *write(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6.5 Control Operations on Files

6.5.1 Data Definitions for File Control Operations

Values for all of the command and control constants defined in POSIX.1 {2} for the *fcntl*() and *open*() functions shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

6.5.2 File Control

Subroutine: *PXFFCNTL*()

6.5.2.1 Synopsis

```
SUBROUTINE PXFFCNTL (IFILDES, ICMD, IARGIN, IARGOUT, IERROR)
INTEGER IFILDES, ICMD, IARGIN, IARGOUT, IERROR
```

6.5.2.2 Description

The *PXFFCNTL*() subroutine shall provide the same functionality as the POSIX.1 {2} function *fcntl*() (see POSIX.1 {2} 6.5), with the exception that the third argument is always of integer type: It can be a (integer) handle for an instance of the *flock* structure or an integer (representing a numeric value), depending on the argument *ICMD* under the conditions defined in POSIX.1 {2} 6.5. The value returned in *IARGOUT* shall also depend on the *ICMD* argument.

The constant values for use in specifying *ICMD* shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2). Arguments for *PXFFCNTL*() correspond to the arguments for *fcntl()*, as shown in Table 6.6.

Table 6.6—Arguments for *PXFFCNTL()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
ICMD	cmd	IN	
IARGIN	arg	IN	1.(or integer value)
IARGOUT	ret_value	OUT	
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘flock’,...); see 8.3.1.

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘flock’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *flock* structure as defined in POSIX.1 {2} 6.5. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JFLOCK, COMPNAM, IVALUE, IERROR)
INTEGER JFLOCK, IVALUE, IERROR
```



```
SUBROUTINE PXFINTSET(JFLOCK, COMPNAM, IVALUE, IERROR)
INTEGER JFLOCK, IVALUE, IERROR
```

where *JFLOCK* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 6.7.

Table 6.7—Components for *flock* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
l_type	'l_type'	PXFINTGET,PXFINTSET
l_whence	'l_whence'	PXFINTGET,PXFINTSET
l_start	'l_start'	PXFINTGET,PXFINTSET
l_len	'l_len'	PXFINTGET,PXFINTSET
l_pid	'l_pid'	PXFINTGET,PXFINTSET

6.5.2.3 Errors

Possible error conditions for *PXFFCNTL*() are identical to those for the POSIX.1 {2} function *fcntl()*. Under the circumstances specified by POSIX.1 {2}, the argument *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

6.5.3 Reposition Read/Write File Offset

Subroutine: *PXFLSEEK*()

6.5.3.1 Synopsis

```
SUBROUTINE PXFLSEEK (IFILDES, IOFFSET, IWHENCE, IPOSITION, IERROR)
INTEGER IFILDES, IOFFSET, IWHENCE, IPOSITION, IERROR
```

6.5.3.2 Description

The *PXFLSEEK*() subroutine shall provide the same functionality as the POSIX.1 {2} function *lseek*() (see POSIX.1 {2} 6.5).

The file-positioning constants defined in POSIX.1 {2} and used for the argument *IWHENCE* shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2). Arguments for *PXFLSEEK*() correspond to the arguments for *lseek()*, as shown in Table 6.8.

Table 6.8—Arguments for *PXFLSEEK()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IOFFSET	offset	IN	1.
IWHENCE	whence	IN	
IPOSITION	ret_value	OUT	1.
IERROR	ret_value/errno	OUT	

1. Value may exceed the range of a signed integer, see 2.3.2.2

6.5.3.3 Errors

Possible error conditions for *PXFLSEEK*() are identical to those for the POSIX.1 {2} function *lseek(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

7. Device- and Class-Specific Procedures

7.1 General Terminal Interface

The terminal interface model shall be the same as defined in POSIX.1 {2}.

7.1.1 Interface Characteristics

The interface characteristics shall be the same as defined in POSIX.1 {2}.

7.1.2 Parameters That Can Be Set

7.1.2.1 *termios* Structure

Any application that needs to control certain terminal I/O characteristics shall do so by using the *termios* structure (see POSIX.1 {2} 7.1).

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘termios’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *termios* structure as defined in POSIX.1 {2} 7.1. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JTERMIOS, COMPNAM, IVALUE, IERROR)
INTEGER JTERMIOS, IVALUE, IERROR

SUBROUTINE PXFINTGET(JTERMIOS, COMPNAM, IVALUE, IERROR)
INTEGER JTERMIOS, IVALUE, IERROR

SUBROUTINE PXFAINTGET(JTERMIOS, COMPNAM, IAVALUE, IALEN, IERROR)
INTEGER JTERMIOS, IAVALUE(IALEN), IALEN, IERROR

SUBROUTINE PXFAINTGET(JTERMIOS, COMPNAM, IAVALUE, IALEN, IERROR)
```

```
INTEGER JTERMIOS, IAVALUE(IALEN), IALEN, IERROR

SUBROUTINE PXFEINTGET(JTERMIOS, COMPNAM, IVALUE, INDEX, IERROR)
INTEGER JTERMIOS, IVALUE, INDEX, IERROR

SUBROUTINE PXFEINTGET(JTERMIOS, COMPNAM, IVALUE, INDEX, IERROR)
INTEGER JTERMIOS, IVALUE, INDEX, IERROR
```

where *JTERMIOS* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 7.1.

Table 7.1—Components for *termios* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
c_iflag	‘c_iflag’	PXFINTSET,PXFINTGET
c_oflag	‘c_oflag’	PXFINTSET,PXFINTGET
c_cflag	‘c_cflag’	PXFINTSET,PXFINTGET
c_lflag	‘c_lflag’	PXFINTSET,PXFINTGET
c_cc	‘c_cc	PXFAINTSET,PXFAINTGET, PXFEINTSET,PXFEINTGET

The component *c_cc* is an array of integers that can be accessed as an entire array or an element at a time. The number of elements in *c_cc* is the value of the constant NCCS, which shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

7.1.2.2 Input Modes

Values of the *c_iflag* component describe the basic terminal input control and are composed of the bit masks described in POSIX.1 {2} 7.1.2.2). The values of these masks shall be bitwise distinct and can be combined with the use of the inclusive OR function (see 8.7). The mask names are constants for which the values shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

7.1.2.3 Output Modes

Values of the *c_oflag* component describe the basic terminal output control and are composed of the bit masks described in POSIX.1 {2} 7.1.2.3). The values of these masks shall be bitwise distinct and can be combined with the use of the inclusive OR function (see 8.7). The mask names are constants for which the values shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

7.1.2.4 Control Modes

Values of the *c_cflag* component describe the basic terminal hardware control and are composed of the bit masks described in POSIX.1 {2} 7.1.2.4). The values of these masks shall be bitwise distinct and can be combined with the use of the inclusive OR function (see 8.7). The mask names are constants for which the values shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

7.1.2.5 Local Modes

Values of the *c_lflag* component describe the control of various functions and are composed of the bit masks described in POSIX.1 {2} 7.1.2.5). The values of these masks shall be bitwise distinct and can be combined with the use of the

inclusive OR function (see 8.7). The mask names are constants for which the values shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

7.1.2.6 Special Control Characters

The values of special control characters are defined by the array component *c_cc*, as described by POSIX.1 {2} 7.1.2.6). The subscript names are symbolic constants for which the values shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2). The elements of the *c_cc* array contain integer representations of the control characters. (See 1.3.4).

7.1.2.7 Baud Rate Values

Baud rate values described in 7.1.3 can be set into the *termios* structure by the baud rate subroutines in 7.1.3. The baud rates are specified by symbolic constants for which the values shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

7.1.3 Baud Rate Subroutines

Subroutines: *PXFCFGETOSPEED()*, *PXFCFSETOSPEED()*, *PXFCFGETISPEED()*, *PXFCFSETISPEED*()

7.1.3.1 Synopsis

```
SUBROUTINE PXFCFGETOSPEED (JTERMIOS, IOSPEED, IERROR)
INTEGER JTERMIOS, IOSPEED, IERROR

SUBROUTINE PXFCFSETOSPEED (JTERMIOS, ISPEED, IERROR)
INTEGER JTERMIOS, ISPEED, IERROR

SUBROUTINE PXFCFGETISPEED (JTERMIOS, IOSPEED, IERROR)
INTEGER JTERMIOS, IOSPEED, IERROR

SUBROUTINE PXFCFSETISPEED (JTERMIOS, ISPEED, IERROR)
INTEGER JTERMIOS, ISPEED, IERROR
```

7.1.3.2 Description

These subroutines shall provide the same functionality as the POSIX.1 {2} baud rate functions (see POSIX.1 {2} 7.1.3). Arguments for these subroutines correspond to the arguments for the corresponding POSIX.1 {2} baud rate functions, as shown in Table 7.2.

Table 7.2—Arguments for the *PXFCF..SPEED()* Subroutines

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
JTERMIOS	termios_p	IN	1.
ISPEED	speed	IN	
IOSPEED	ret_value	OUT	
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘termios’,...); see 8.3.1

7.1.3.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *cf...speed*() family of functions. Upon successful completion of any of the *PXFCF...SPEED*() family of subroutines, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

7.2 General Terminal Interface Control Subroutines

7.2.1 Get and Set State

Subroutines: *PXFTCGETATTR(), PXFTCSETATTR*()

7.2.1.1 Synopsis

```
SUBROUTINE PXFTCGETATTR (IFILDES, JTERMIOS, IERROR)
INTEGER IFILDES, JTERMIOS, IERROR

SUBROUTINE PXFTCSETATTR (IFILDES, IOPTACTS, JTERMIOS, IERROR)
INTEGER IFILDES, IOPTACTS, JTERMIOS, IERROR
```

7.2.1.2 Description

The *PXFTCGETATTR*() and *PXFTCSETATTR*() subroutines shall provide the same functionality as the POSIX.1 {2} functions *tcgetattr*() and *tcsetattr*() (see POSIX.1 {2} 7.2). Arguments for *PXFTCSETATTR*() and *PXFTCGETATTR*() correspond to the arguments for *tcsetattr*() and *tcgetattr()*, as shown in Table 7.3.

Table 7.3—Arguments for *PXFTCSETATTR()* and *PXFTCGETATTR()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IOPTACTS	optional_actions	IN	
JTERMIOS	termios_p	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘termios’,...); see 8.3.1.

7.2.1.3 Errors

Possible error conditions for *PXFTCGETATTR*() and *PXFTCSETATTR*() are identical to those for the POSIX.1 {2} functions *tcgetattr*() and *tcsetattr(). IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

7.2.2 Line Control Subroutines

Subroutines: *PXFTCSENDBREAK(), PXFTCDRAIN(), PXFTCFLUSH(), PXFTCFLOW*()

7.2.2.1 Synopsis

```
SUBROUTINE PXFTCSENDBREAK (IFILDES, IDURATION, IERROR)
INTEGER IFILDES, IDURATION, IERROR

SUBROUTINE PXFTCDRAIN (IFILDES, IERROR)
INTEGER IFILDES, IERROR

SUBROUTINE PXFTCFLUSH (IFILDES, IQUEUE, IERROR)
INTEGER IFILDES, IQUEUE, IERROR

SUBROUTINE PXFTCFLOW (IFILDES, IACTION, IERROR)
INTEGER IFILDES, IACTION, IERROR
```

7.2.2.2 Description

PXFTCSENDBREAK(), *PXFTCDRAIN()*, *PXFTCFLUSH()*, and *PXFTCFLOW*() shall provide the same functionality as their respective POSIX.1 {2} functions *tcsendbreak()*, *tcdrain()*, *tcflush()*, and *tcflow*() (see POSIX.1 {2} 7.2).

The constant values for use in specifying *IQUEUE* and *IACTION* shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

Arguments for these subroutines correspond to the arguments for the corresponding POSIX.1 {2} line control functions, as shown in Table 7.4.

Table 7.4—Arguments for the *PXFTC..()* Subroutines

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IDURATION	duration	IN	
IQUEUE	queue_selector	IN	
IACTION	action	IN	
IERROR	ret_value/errno	OUT	

7.2.2.3 Errors

Possible error conditions for these subroutines are identical to those for the corresponding line control functions defined in POSIX.1 {2}. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

7.2.3 Get Foreground Process Group ID

Subroutine: *PXFTCGETPGRP()*

7.2.3.1 Synopsis

```
SUBROUTINE PXFTCGETPGRP (IFILDES, IPGID, IERROR)
INTEGER IFILDES, IPGID, IERROR
```


7.2.3.2 Description

The *PXFTCGETPGRP()* subroutine shall provide the same functionality as the POSIX.1 {2} function *tcgetpgrp()* (see POSIX.1 {2} 7.2), except that the process group is returned in *IPGID*. Arguments for *PXFTCGETPGRP()* correspond to the arguments for *tcgetpgrp()*, as shown in Table 7.5.

Table 7.5—Arguments for *PXFTCGETPGRP()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IPGID	ret_value	OUT	
IERROR	ret_value/errno	OUT	

7.2.3.3 Errors

Possible error conditions for *PXFTCGETPGRP()* are identical to those for the POSIX.1 {2} function *tcgetpgrp()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon succesful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

7.2.4 Set Foreground Process Group ID

Subroutine: *PXFTCSETPGRP()*

7.2.4.1 Synopsis

```
SUBROUTINE PXFTCSETPGRP (IFILDES, IPGID, IERROR)
INTEGER IFILDES, IPGID, IERROR
```

7.2.4.2 Description

The *PXFTCSETPGRP()* subroutine shall provide the same functionality as the POSIX.1 {2} function *tcsetpgrp()* (see POSIX.1 {2} 7.2). Arguments for *PXFTCSETPGRP()* correspond to the arguments for *tcsetpgrp()*, as shown in Table 7.6.

Table 7.6—Arguments for *PXFTCSETPGRP()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IFILDES	fildes	IN	
IPGID	pgrp_id	IN	
IERROR	ret_value/errno	OUT	

7.2.4.3 Errors

Possible error conditions for *PXFTCSETPGRP()* are identical to those for the POSIX.1 {2} function *tcsetpgrp()*. *IERROR* shall be set to the corresponding nonzero value specified by the POSIX.1 {2} function. Upon successful completion, the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

8. FORTRAN 77 Language Library

8.1 FORTRAN 77 Intrinsics

For general information regarding these functions, see FORTRAN 77 {3}.

8.2 System Symbolic Constant Access

For general information regarding these subroutines, see 2.3.1.

8.2.1 Access and Verify Symbolic Constants

Subroutine: *PXFCONST()*

Functions: *IPXFCONST(), PXFISCONST()*

8.2.1.1 Synopsis

```
INTEGER FUNCTION IPXFCONST (CONSTNAME)
CHARACTER*(*) CONSTNAME

LOGICAL FUNCTION PXFISCONST (CONSTNAME)
CHARACTER*(*) CONSTNAME

SUBROUTINE PXFCONST (CONSTNAME, IVAL, IERROR)
CHARACTER*(*) CONSTNAME
INTEGER IVAL, IERROR
```

8.2.1.2 Description

The argument *CONSTNAME* is the character representation of the name of any constant defined in a POSIX.1 {2} header or in POSIX.9. *CONSTNAME* is case-sensitive, and trailing blanks in the argument shall be ignored.

The function *IPXFCONST()* shall provide an integer return value but no error checking. If the argument passed corresponds to a defined constant in POSIX.1 {2} or POSIX.9, the return value is the integer value associated with the constant; if the argument is not a defined constant, the behavior is implementation defined. The *PXFISCONST()* function shall confirm whether the argument is a valid constant defined by POSIX.1 {2} or POSIX.9. *PXFISCONST()* shall return .TRUE. if and only if *IPXFCONST()* would return a valid value for the same *CONSTNAME*.

The subroutine *PXFCONST()* shall provide error checking and a return value in the same call. Upon successful completion, the argument *IVAL* shall be set to the integer value associated with the symbolic constant.

The alteration of a constant value by an implementation should require recompilation of an application utilizing any of these *PXFCONST()* procedures to access the altered constant.

Table 8.1—Arguments for Symbolic Constant Procedures

FORTRAN Argument	Intent	Notes
CONSTNAME	IN	
IVAL	OUT	
IERROR	OUT	

8.2.1.3 Errors

Upon successful completion of *PXFCONST()*, the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFCONST()* shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[ENONAME]	Invalid constant name.

8.3 Structure Creation and Manipulation

For general information regarding these subroutines, see 2.3.2.

There are two common usage patterns associated with accessing the aggregate data: passing information to the system service procedures and obtaining information from the system procedures. The following steps should be followed when using system procedures that require the use of aggregate data:

- *PXFSTRUCTCREATE()* can be called to create an instance of the desired structure and to obtain a *handle* with which to reference it.
- If an application passes information to the system, the *PXF<TYPE>SET()* subroutines shall be called, once for each member, *before* calling the system procedure (i.e., the structure is loaded before the system call).
- The desired system procedure is called.
- If an application needs to get information from the system, the *PXF<TYPE>GET()* subroutines should be called, once for each member, *after* calling the system procedure (i.e., the information is only available in the structure after the system call).
- *PXFSTRUCTFREE()* can be called to remove the instance of the structure.

When calling the actual system procedure, the calling sequence is equivalent to the C binding as shown in POSIX.1 {2}, except that a handle is used in place of the POSIX.1 {2} structure (pointer) argument.

8.3.1 Structure Creation

Subroutine: *PXFSTRUCTCREATE()*

8.3.1.1 Synopsis

```
SUBROUTINE PXFSTRUCTCREATE (STRUCTNAME, JHANDLE, IERROR)
CHARACTER*(*) STRUCTNAME
INTEGER JHANDLE, IERROR
```

8.3.1.2 Description

The subroutine *PXFSTRUCTCREATE()* creates an instance of the desired structure and returns a nonzero handle in the argument *JHANDLE*. All further references to this instance of this structure are through this handle. A list of POSIX.9-defined values for *STRUCTNAME* is provided in 2.3.2.3. The initial values of components within the new instance of the structure are undefined.

Table 8.2—Arguments for *PXFSTRUCTCREATE()*

FORTRAN Argument	Intent	Notes
STRUCTNAME	IN	
JHANDLE	OUT	
IERROR	OUT	

8.3.1.3 Errors

Upon successful completion of *PXFSTRUCTCREATE()*, the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFSTRUCTCREATE()* shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[ENONAME]	Invalid structure name.
[ENOHANDLE]	Instance of the structure could not be created.

8.3.2 Structure-Component Manipulation

Subroutines: *PXF<TYPE>SET(), PXF<TYPE>GET(), PXFA<TYPE>SET(), PXFA<TYPE>GET(), PXFE<TYPE>SET(), PXFE<TYPE>GET(),*

8.3.2.1 Synopsis

```
SUBROUTINE PXF<TYPE>SET (JHANDLE, COMPNAM, VALUE [,ILEN], IERROR)
INTEGER JHANDLE, [ILEN], IERROR)
CHARACTER*(*) COMPNAM
TYPE VALUE

SUBROUTINE PXF<TYPE>GET (JHANDLE,COMPNAM, VALUE [,ILEN], IERROR)
INTEGER JHANDLE, [ILEN,] IERROR)
CHARACTER*(*) COMPNAM
TYPE VALUE

SUBROUTINE PXFA<TYPE>SET (JHANDLE, COMPNAM, VALUE, IALEN [,ILEN], IERROR)
INTEGER JHANDLE, IALEN, [ILEN(IALEN),] IERROR)
CHARACTER*(*) COMPNAM
TYPE VALUE(IALEN)

SUBROUTINE PXFA<TYPE>GET (JHANDLE, COMPNAM, VALUE, IALEN [,ILEN], IERROR)
INTEGER JHANDLE, IALEN, [ILEN(IALEN),] IERROR)
CHARACTER*(*) COMPNAM
TYPE VALUE(IALEN)

SUBROUTINE PXFE<TYPE>SET (JHANDLE, COMPNAM, INDEX, VALUE [,ILEN], IERROR)
INTEGER JHANDLE, INDEX, [ILEN,] IERROR)
CHARACTER*(*) COMPNAM
TYPE VALUE
```



```
SUBROUTINE PXFE<TYPE>GET (JHANDLE, COMPNAM, INDEX, VALUE [,ILEN], IERROR)
 INTEGER JHANDLE, INDEX, [ILEN,] IERROR)
 CHARACTER*(*) COMPNAM
 TYPE VALUE
```

NOTE — The argument *ILEN* only appears in the interface definition when the type of TYPE *VALUE* is CHARACTER*(*).

8.3.2.2 Description

The *PXF<TYPE>SET()* subroutines allow components of a structure to be set or modified, while the *PXF<TYPE>GET()* subroutines allow values stored in individual components to be extracted and used. There is a separate subroutine for handling each unique base FORTRAN 77 data type that may occur within a structure. Substituting one of the following character sequences for <TYPE> in the generic names shown shall result in access to a structure component of the indicated data type. A conforming implementation shall provide all access routines required to access the structures described in 2.3.2.3.1.

Table 8.3—*<TYPE>*s of Structure Element Subroutines

<TYPE>	**TYPE**
INT	INTEGER
REAL	REAL
LGCL	LOGICAL
STR	CHARACTER*(*)
CHAR	CHARACTER*1
DBL	DOUBLE PRECISION
CPLX	COMPLEX

The subroutines *PXFAT<YPE>SET*() and *PXFA<TYPE>GET*() are analogous subroutines that are used when the structure component is an array. The entire array is accessed (read/written) as a unit when these subroutines are used. *PXFE<TYPE>SET*() and *PXFe<TYPE>GET*() can be used to access a *single* element of a structure component that is an array. The array element is selected with the argument *INDEX*. Note that, unlike in the C binding of POSIX.1 {2}, these FORTRAN 77 arrays are one-based for indexing.

For all subroutines, the arguments named *JHANDLE, COMPNAM*, and *INDEX* are “in” arguments, and *IERROR* is an “out” argument. The intent of the *VALUE, ILEN*, and *IALEN* arguments are “in” for the *PXF<TYPE>SET*() subroutines or any of the analagous array or array element subroutines, and “out” for the *PXF<TYPE>GET*() subroutines or any of the analagous array or array element subroutines.

8.3.2.3 Errors

Upon successful completion of any of the *PXF<TYPE>SET*() or *PXF<TYPE>GET*() subroutines or any of the analagous array or array element subroutines, the argument *IERROR* shall be set to zero. If any of the following conditions occur, the subroutine shall set the argument to the corresponding value.

[EINVAL]	Invalid value for *INDEX*.
[ENONAME]	Component name is not defined for this structure.
[ETRUNC]	The declared length of the character argument is insufficient to contain the string to be returned. (See 2.3.2.4.)
[EARRAYLEN]	For *PXF<TYPE>GET* subroutines, the number of array elements to be returned exceeds *IALEN*, and only the first *IALEN* elements of the array argument have been set. For *PXF<TYPE>SET* subroutines, *IALEN* exceeds the number of array elements in the structure component. Only the available elements of the array in the structure component have been set.

Access to a structure component that does not belong to the structure referenced by *JHANDLE* or use of a subroutine of the wrong class (e.g., the use of an array subroutine to access a scalar structure component) or the wrong type (e.g., the use of a STR routine when the component is an integer) is undefined.

8.3.3 Structure Deletion

Subroutine: *PXFSTRUCTFREE*()

8.3.3.1 Synopsis

```
SUBROUTINE PXFSTRUCTFREE (JHANDLE, IERROR)
INTEGER JHANDLE, IERROR
```

8.3.3.2 Description

The subroutine *PXFSTRUCTFREE*() deletes the instance of the structure referenced by *JHANDLE*.

Table 8.4—Arguments for *PXFSTRUCTFREE()*

FORTRAN Argument	Intent	Notes
JHANDLE	IN	structure handle
IERROR	OUT	

8.3.3.3 Errors

Upon successful completion of *PXFSTRUCTFREE*(), the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

8.3.4 Structure Copy

Subroutine: *PXFSTRUCTCOPY*()

8.3.4.1 Synopsis

```
SUBROUTINE PXFSTRUCTCOPY (STRUCTNAME, JHANDLE1, JHANDLE2, IERROR)
INTEGER JHANDLE1, JHANDLE2, IERROR
CHARACTER*(*) STRUCTNAME
```

8.3.4.2 Description

The subroutine *PXFSTRUCTCOPY*() copies the contents of the structure referenced by *JHANDLE1* to the structure referenced by *JHANDLE2*. Both handles shall have been created by *PXFSTRUCTCREATE* using the same *STRUCTNAME*. A list of POSIX.9 defined values for *STRUCTNAME* is provided in 2.3.2.3.

Table 8.5—Arguments for *PXFSTRUCTCOPY()*

FORTRAN Argument	Intent	Notes
STRUCTNAME	IN	
JHANDLE1	IN	structure handle
JHANDLE2	IN	structure handle
IERROR	OUT	

8.3.4.3 Errors

Upon successful completion of *PXFSTRUCTCOPY*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFSTRUCTCOPY*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[ENONAME] Invalid structure name.

8.4 Subroutine-Handle Manipulation

These subroutines shall provide the subroutine pointer facility described in 2.3.2.5.

8.4.1 Save and Reference Subroutine Handle

Subroutine: *PXFGETSUBHANDLE()*, *PXFCALLSUBHANDLE*()

8.4.1.1 Synopsis

```
SUBROUTINE PXFGETSUBHANDLE (SUB, JHANDLE1, IERROR)
INTEGER JHANDLE, IERROR
EXTERNAL SUB

SUBROUTINE PXFCALLSUBHANDLE(JHANDLE2, IVAL, IERROR)
INTEGER JHANDLE, IVAL, IERROR
```

8.4.1.2 Description

Given a subroutine external argument, *PXFGETSUBHANDLE*() returns a subroutine handle for that subroutine in the argument *JHANDLE1*. The argument *SUB* shall not be a function, an intrinsic, or an entry point and shall be defined with exactly one integer argument.

Given a subroutine handle obtained from a previous call to *PXFGETSUBHANDLE*() or *PXFSIGACTION()*, *PXFCALLSUBHANDLE*() calls the subroutine associated with that handle, with *IVAL* as the one integer argument.

Table 8.6—Arguments for Subroutine-Handle Manipulation Subroutines

FORTRAN Argument	Intent	Notes
SUB	IN	
JHANDLE1	OUT	subroutine handle
JHANDLE2	IN	subroutine handle
IVAL	IN	
IERROR	OUT	

The values of the symbolic constants *SIG_DFL* and *SIG_IGN* are reserved and never returned as a value for the handle by *PXFGETSUBHANDLE*() nor may they be passed as the *SUB* argument in a call to *PXFCALLSUBHANDLE*().

8.4.1.3 Errors

Upon successful completion of *PXFGETSUBHANDLE*() or *PXFCALLSUBHANDLE*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETSUBHANDLE*() and *PXFCALLSUBHANDLE*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[ENOHANDLE] Instance of the subroutine handle could not be created by *PXFGETSUBHANDLE*().

8.5 External Unit and File Description Interaction

This section describes the interaction of FORTRAN 77 external units with file descriptors. A unit identifier is local to a single process. After an *PXFFORK*() call, an open file description shall be shared by parent and child. The *PXFFDOPEN*() subroutine shall connect a unit to a file descriptor (see 8.5.3). When a file is opened using the FORTRAN 77 OPEN statement, an external unit shall be connected to a file descriptor if the value of the POSIX I/O flag (see 8.5.1) is one (1) upon execution of the OPEN statement. External units not described in this section may be connected to file descriptors.

The preconnected units identified by STDIN_UNIT, STDOUT_UNIT, and STDERR_UNIT shall each be connected to file descriptors. In addition, records read from or written to these units shall be accessed as if they are newline delimited (see 8.5.1).

8.5.1 POSIX-Based FORTRAN I/O

Subroutine: *PXFPOSIXIO*()

8.5.1.1 Synopsis

```
SUBROUTINE PXFPOSIXIO (NEW, OLD, IERROR)
INTEGER NEW, OLD, IERROR
```

8.5.1.2 Description

The *PXFPOSIXIO*() subroutine sets and returns the current value of the POSIX I/O flag. The POSIX I/O flag is set to the value of *NEW*. The previous value of the POSIX I/O flag is returned in *OLD*. The initial state of the POSIX I/O flag is unspecified.

Table 8.7—Arguments for *PXFPOSIXIO()*

FORTRAN Argument	Intent	Notes
NEW	IN	
OLD	OUT	
IERROR	OUT	

If a file is opened with a FORTRAN 77 OPEN statement when the value of the POSIX I/O flag is one (1), the unit shall be connected to a file descriptor. In addition, records within formatted sequential access files shall be accessed as if the records are newline delimited, even if the file does not contain records that are delimited by a newline character. When the value of the POSIX I/O flag is zero (0) upon execution of the FORTRAN 77 OPEN statement, a connection to a file descriptor is not assumed, and the records in the file are not required to be accessed as if they are newline delimited. If the value of the POSIX I/O flag is other than zero or one, the interpretation is unspecified.

If the file is already open and another FORTRAN 77 OPEN statement is only used to change the BLANK= specifier on the same file, the selection of POSIX-based FORTRAN I/O is not changed on that file.

8.5.1.3 Errors

Upon successful completion of *PXFPOSIXIO*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFPOSIXIO*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL] Value of *NEW* is neither zero nor one and is not supported.

8.5.2 Map a Unit to a File Descriptor

Subroutine: *PXFFILENO*()

8.5.2.1 Synopsis

```
SUBROUTINE PXFFILENO (IUNIT, IFILDES, IERROR)
INTEGER IUNIT, IFILDES, IERROR
```

8.5.2.2 Description

The *PXFFILENO*() subroutine shall return in *IFILDES* the file descriptor to which the unit identified by *IUNIT* is connected.

Table 8.8—Arguments for *PXFFILENO()*

FORTRAN Argument	Intent	Notes
IUNIT	IN	
IFILDES	OUT	
IERROR	OUT	

The units associated with the preconnected files identified by STDIN_UNIT, STDOUT_UNIT, and STDERR_UNIT (see 2.9.1) are connected to the file descriptors defined by the symbolic constants STDIN_FILENO. STDOUT_FILENO, and STDERR_FILENO respectively (see Table 8.9). When performing FORTRAN 77 read operations on a file connected to the processor-determined external unit specified by the asterisk (*), this unit is connected to the file descriptor defined by the symbolic constant STDIN_FILENO, When performing FORTRAN 77 write operations on a file connected to the processor-determined external unit specified by the asterisk (*), this unit is connected to the file descriptor defined by the symbolic constant STDOUT_FILENO. The symbolic constants shall be accessible through calls to any of the *PXFCONST*() procedures (see 8.2).

Table 8.9—File Descriptor Constants

Name	Description	File Descriptor Value
STDIN_FILENO	Standard input file descriptor	0
STDOUT_FILENO	Standard output file descriptor	1
STDERR_FILENO	Standard error file descriptor	2

8.5.2.3 Errors

Upon successful completion of *PXFFILENO*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFFILENO*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL] *IUNIT* is not an open unit.

[EBADF] *IUNIT* is not connected with a file descriptor.

8.5.3 Open a Unit

Subroutine: *PXFFDOPEN*()

8.5.3.1 Synopsis

```
SUBROUTINE PXFFDOPEN (IFILDES, IUNIT, ACCESS, IERROR)
INTEGER IUNIT, IFILDES, IERROR
CHARACTER*(*) ACCESS
```

8.5.3.2 Description

The *PXFFDOPEN*() subroutine connects an external unit identified by *IUNIT*, to a file descriptor, *IFILDES*. If the unit is connected to a file, the file shall be closed before the unit becomes connected to the file descriptor. See the OPEN statement in FORTRAN 77 {3}.

Table 8.10—Arguments for *PXFFDOPEN()*

FORTRAN Argument	Intent	Notes
IFILDES	IN	
IUNIT	IN	
ACCESS	IN	
IERROR	OUT	

The *ACCESS* argument is a character string that specifies the attributes of the connection. This string consists of one or more keyword/value pairs, described in Table 8.11. Keywords shall be separated from their values by the equals (=) character. Keyword/value pairs shall be separated by the comma (,) character. Blanks shall be ignored.

Table 8.11—Values for *ACCESS* Argument

Keyword	Values	Function	Default
'NEWLINE'	'YES'	I/O type	'YES' 'NO'
'BLANK'	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
'STATUS'	'OLD' 'SCRATCH' 'UNKNOWN'	File status at open	'UNKNOWN'
'FORM'	'FORMATTED' 'UNFORMATTED'	Format type	'FORMATTED'

Records within a formatted file shall be accessed as if they are newline delimited when the NEWLINE keyword is set to the value YES. When the FORM keyword is set to the value 'UNFORMATTED', the NEWLINE keyword shall be ignored.

The meaning and behavior of the BLANK and FORM keywords and its values shall be as defined for the FORTRAN 77 OPEN statement.

The meaning and behavior of the STATUS keyword and its values shall be as defined for the FORTRAN 77 OPEN statement with the following exceptions. When the STATUS keyword is set to the value 'OLD', the file offset associated with the file description shall not be changed as a result of calling *PXFFDOPEN*().

Additional ACCESS argument keywords and values may be present.Their interpretation is implementation defined.

8.5.3.3 Errors

Upon successful completion of *PXFFDOPEN*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFFDOPEN*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL]	The ACCESS keyword specifies invalid options.
[EACCES]	Access is not permitted by the file permissions of the file.
[EBADF]	The *IFILDES* argument is not a valid file descriptor or the IUNIT argument does not specify a valid external unit.

8.5.4 Flush Output

Subroutine: *PXFFFLUSH*()

8.5.4.1 Synopsis

```
SUBROUTINE PXFFFLUSH (IUNIT, IERROR)
INTEGER IUNIT, IERROR
```

8.5.4.2 Description

The *PXFFFLUSH*() subroutine shall write any buffered output to the file connected to the unit *IUNIT*. End-of-record is not implied by a call to *PXFFFLUSH*().

Table 8.12—Arguments for *PXFFFLUSH()*

FORTRAN Argument	Intent	Notes
IUNIT	IN	
IERROR	OUT	

If the *IUNIT* argument is not connected for POSIX-based FORTRAN I/O (see 8.5), the results of *PXFFFLUSH*() are undefined. *PXFFFLUSH*() shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file if the file is writable, the call results in a transfer of data to the file, and if data has not yet been written to the file.

8.5.4.3 Errors

Upon successful completion of *PXFFFLUSH*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFFFLUSH*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL]	The *IUNIT* argument is not a valid external unit identifier.
[EFBIG]	An attempt was made to write a file that exceeds an implementation-defined maximum file size.
[ENOSPC]	There is no free space remaining on the device containing the file.
[ESPIPE]	An attempt is made to write a pipe (or FIFO) that is not open for reading by a process. A SIGPIPE signal shall also be sent to the process.

8.5.5 FORTRAN Language I/O Statements

This section describes the behavior of FORTRAN 77 I/O that is special because the underlying operating system is POSIX based. It defines special procedures that provide I/O capabilities specific to this environment. In particular, this section describes interactions of FORTRAN 77 I/O statements with POSIX.9. All interactions specified in this section apply only to POSIX-based FORTRAN I/O files. These interactions define behavior that is undefined or unspecified by FORTRAN 77 and does not modify or replace any behavior that is defined in FORTRAN 77.

The set of allowable names for a file (see Section 12.2.2 of FORTRAN 77 {3}) shall include pathnames as defined by POSIX.9. A connected unit is a unit that has been opened by the FORTRAN 77 statement OPEN or by *PXFFDOPEN*.

FORTRAN 77 — formatted sequential I/O shall read and write files that are accessed as if they are newline delimited, but is not limited to reading and writing these files.

8.5.5.1 General Interactions of FORTRAN I/O Statements

A single open file description can be accessed through units and file descriptors. This section defines the interaction of units and file descriptors with an open file description.

A unit is explicitly closed in FORTRAN 77 by a CLOSE statement. It is implicitly closed through OPEN, STOP, or END statements as specified by FORTRAN 77. A unit is implicitly closed through *PXFFDOPEN*() and *PXFEXIT*(). A file descriptor is explicitly closed by *PXFCLOSE*() and implicitly closed by *PXFFASTEXIT*() or by one of the *PXFEXEC*() calls under the conditions specified in 3.1. When a unit is closed, the underlying file descriptor is also closed.

A unit is connected to a file descriptor when the unit and the file descriptor access the same open file description. A file descriptor is connected to a unit when the unit and the file descriptor access the same open file description. POSIX.9 subroutines that could affect the file offset are *PXFLSEEK*(), *PXFREAD*(), and *PXFWRITE*().

For direct access files, operations that could directly affect the file offset are undefined.

For unformatted sequential access files, when a POSIX.9 procedure that operates directly on a file descriptor affects the file offset, and that file descriptor is connected to a unit, the results of subsequent FORTRAN 77 I/O statements using the connected unit are undefined.

For formatted sequential access files, operations that directly affect the file offset may be used in conjunction with FORTRAN 77 I/O operations. When a POSIX.9 procedure that operates directly on a file descriptor affects the file offset, and that file descriptor is connected to a unit, the results of subsequent FORTRAN 77 I/O statements using the connected unit are undefined unless *PXFFFLUSH*() was called to flush the connected unit prior to such operations. After a call to *PXFFFLUSH*(), the subsequent I/O operation on the connected unit shall reestablish the file position from the file offset, as the first action of the operation.

A file connected to a unit shall become connected to two units in two separate processes after a *PXFFORK*(). In addition, a file could become connected to two different units as a result of calling *PXFFDOPEN*(). I/O operations on these units shall be coordinated by the application. For direct access files, I/O operations are not defined on a file connected to more than one unit at a time. For sequential files, I/O operations on subsequent units connected to the same file at the same time are defined under one or more of the following conditions:

1) No operation was performed on the initial unit that could affect the file offset.
2) The initial unit has been closed, unless a subsequent unit was connected by *PXFFORK*().
3) The subroutine *PXFFLUSH*() was executed on the initial unit, and no subsequent I/O operation was performed on that unit that could affect the file offset.
4) Following 47 *PXFFORK*(), the process that connected the initial unit has not performed any I/O operation on that unit that could affect the file offset and has successfully executed any one of the *PXFEXEC*...() or *PXFFASTEXIT*() subroutines.
5) Prior to *PXFFORK*(), conditions (1) or (3) are met.

I/O operations on the initial unit are defined only if the same conditions are met for subsequent units. If more than two units are connected to the same open file description, these conditions should be met for all other units before performing I/O operations on any one unit. If these conditions are met, no data shall be duplicated or lost. If these conditions are not met, the results of performing I/O operations on these units are undefined.

For formatted sequential access files, the file position (see Section 12.2.3 of FORTRAN 77 {3}) could be manipulated with *PXFFSEEK*(), *PXFGETC*(), *PXFFGETC*(), *PXFPUTC*(), and *PXFFPUTC*() (see 8.6). These routines shall access the bytes of a file. FORTRAN 77 I/O operations access the records of the file. The file position is updated after

each byte or record access. A byte access operation that follows a record access operation shall behave as if the byte position of the file is the byte following the newline that delimited the record accessed. After a byte access operation, the current, next, and preceding records are defined according to the file position and the newline record delimiter.

If a byte access positions the file on a byte other than a newline delimiter, the next record shall begin with the byte at the file position. If a byte access positions the file on a newline delimiter, the next record shall begin with the byte following the file position. If the number of records is zero, or if the file is positioned at its terminal point, there is no next record.

The preceding record shall begin with the byte following the preceding newline. If there is no preceding newline, the record shall begin at the file initial point. If the number of records is zero, or if the file is positioned at its initial point, there is no preceding record.

After a byte access operation, the current record is undefined.

8.5.5.2 Interactions With FORTRAN 77 OPEN Statement

The FORTRAN 77 OPEN statement shall allocate a file descriptor with at least the consequences of calling *PXFOPEN*(). When creating a new file, OPEN shall have at least the consequences of calling *PXFOPEN*() with a value of

```
      IOR(IPXFCONST('S_IRUSR'),IOR(IPXFCONST('S_IWUSR'),
     + IOR(IPXFCONST('S_IRGRP'),IOR(IPXFCONST('S_IWGRP'),
     + IOR(IPXFCONST('S_IROTH'),IPXFCONST('S_IWOTH'))))))
```

for the mode argument.

In the FORTRAN 77 OPEN statement, the interaction of POSIX.9 with open list specifiers shall be as follows:

— IOSTAT
  If the OPEN statement fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT specifier shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.

8.5.5.3 Interactions With FORTRAN 77 INQUIRE Statement

In the FORTRAN 77 INQUIRE statement, the interaction of POSIX.9 with inquire list specifiers shall be as follows:

— IOSTAT
  If the INQUIRE statement fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT specifier shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.
— NAMED
  Files opened with *PXFFDOPEN*() do not have names. If a second unit is connected by execution of *PXFFORK*() and the first unit has a name, the second unit shall have a name.
— NAME
  If the file has a name, the value returned by the *NAME* argument shall be the complete pathname for the file. If the file does not have a name, the value returned by the *NAME* argument shall be a string of all blanks. if an absolute pathname cannot be determined.

8.5.5.4 Interactions With FORTRAN 77 CLOSE Statement

The results of the FORTRAN 77 CLOSE statement shall have at least the consequences of *PXFCLOSE*() called with the file descriptor connected to the unit. It shall also mark for update the *st_ctime* and *st_mtime* fields of the file, if the unit is writable and if buffered data has not been written to the file.

In the FORTRAN 77 CLOSE statement, the interaction of POSIX.9 with closed list specifiers shall be as follows:

— IOSTAT
If the CLOSE statement fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT keyword shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.

8.5.5.5 Interactions With FORTRAN 77 READ Statement

FORTRAN 77 sequential READ, *PXFFGETC*(), and *PXFGETC*() (see 8.6) shall have at least the consequences of *PXFREAD*() when the open file description is accessed, except the condition [EINTR] shall not cause failure. The *st_atime* field shall be marked for update by the first successful execution of READ (sequential or direct), *PXFFGETC*(), or *PXFGETC*() that results in data transferred from the file.

Before a READ, *PXFFGETC*(), or *PXFGETC*() operation on the controlling terminal, data buffered as a result of a WRITE, *PXFFPUTC*(), or *PXFPUTC*() operation shall be written.

In the FORTRAN 77 READ statement, the interaction of POSIX.9 with READ control information list specifiers shall be as follows:

— IOSTAT
If the READ statement fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT specifier shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.

8.5.5.6 Interactions With FORTRAN 77 WRITE Statement

FORTRAN 77 sequential WRITE shall have at least the consequences of *PXFWRITE*() when the open file description is accessed, except the condition [EINTR] shall not cause failure. The *st_ctime* and *st_mtime* shall be marked for update by the first successful execution of WRITE (sequential or direct), *PXFFPUTC*(), or *PXFPUTC*() (see 8.6) that results in data being transferred to the file.

In the FORTRAN 77 WRITE statement, the interaction of POSIX.9 with WRITE control information list specifiers shall be as follows:

— IOSTAT
If the WRITE statement fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT specifier shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.

8.5.5.7 Interactions With FORTRAN 77 BACKSPACE and REWIND Statements

FORTRAN 77 BACKSPACE, FORTRAN 77 REWIND, and *PXFFSEEK*() shall have at least the consequences of calling *PXFLSEEK*() for the equivalent file positioning. Provided the unit is connected to a file that exists, is writable, and unbuffered data has not yet been written to the file, BACKSPACE, REWIND, and *PXFFSEEK*() shall have at least

the consequences of *PXFWRITE*(), except the condition [EINTR] shall not cause failure. In addition, *st_ctime* and *st_mtime* shall be marked for update. The results of REWIND shall have at least the consequences of calling *PXFLSEEK*() with the *IOFFSET* argument set to zero and the *IWHENCE* argument set to IPXFCONST('SEEK_SET'). FORTRAN 77 I/O shall consider the file to be at its initial point.

In the FORTRAN 77 BACKSPACE and REWIND statements, the interaction of POSIX.9 with the corresponding auxiliary list specifiers shall be as follows:

— IOSTAT
If the BACKSPACE or REWIND fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT specifier shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.

8.5.5.8 Interactions With FORTRAN 77 ENDFILE Statement

FORTRAN 77 ENDFILE shall have at least the consequences of calling *PXFWRITE*(), except the condition [EINTR] shall not cause failure. ENDFILE shall mark the *st_ctime* and *st_mtime* of the file for update.

In the FORTRAN 77 ENDFILE statement, the interaction of POSIX.9 with the auxiliary list specifiers shall be as follows:

— IOSTAT
If ENDFILE fails due to a POSIX.9 error condition, the value returned in the argument of the IOSTAT specifier shall be the POSIX.9 error value. The implementation-defined values for the POSIX.9-defined error conditions shall be different from any processor-defined values for additional FORTRAN 77 processor-defined error conditions.

8.6 Stream I/O

Stream I/O shall provide byte access to a POSIX-based FORTRAN I/O file (see 8.5). These files, including record-control information contained in these files, shall be accessible through the stream I/O subroutines. The results of the procedures in this section are undefined for files that are not POSIX-based FORTRAN I/O files and files opened for unformatted FORTRAN I/O.

8.6.1 Modify a File Position

Subroutine: *PXFFSEEK*()

8.6.1.1 Synopsis

```
SUBROUTINE PXFFSEEK (IUNIT, IOFFSET, IWHENCE, IERROR)
INTEGER IUNIT, IOFFSET, IWHENCE, IERROR
```

8.6.1.2 Description

The subroutine *PXFFSEEK*() shall modify the file position of the file connected to the unit *IUNIT*. The *IUNIT* argument shall refer to an open unit. The *IOFFSET* argument is an offset in bytes relative to the position specified by *IWHENCE*.

Table 8.13—Arguments for *PXFFSEEK()*

FORTRAN Argument	Intent	Notes
IUNIT	IN	
IOFFSET	IN	1.
IWHENCE	IN	
IERROR	OUT	

1. Value may exceed the range of a signed integer; see 2.3.2.2.

The file-positioning constants used for the argument *IWHENCE* are the same as those used for the argument *IWHENCE* for the procedure *PXFLSEEK*() (see 6.5.3).

8.6.1.3 Errors

Upon successful completion of *PXFFSEEK*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFFSEEK*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL]	No file is connected to *IUNIT*, or *IWHENCE* is not a proper value, or the resulting file offset would be invalid.
[ESPIPE]	The *IUNIT* argument is connected to a pipe or FIFO.
[EEND]	The end of file was encountered.

8.6.2 Read a File Position

Subroutine: *PXFFTELL*()

8.6.2.1 Synopsis

```
SUBROUTINE PXFFTELL (IUNIT, IOFFSET, IERROR)
INTEGER IUNIT, IOFFSET, IERROR
```

8.6.2.2 Description

The subroutine *PXFFTELL*() shall return the file position for the file connected to the unit *IUNIT*. The file position returned in the argument *IOFFSET* shall be the number of bytes from the beginning of the file.

Table 8.14—Arguments for *PXFFTELL()*

FORTRAN Argument	Intent	Notes
IUNIT	IN	
IOFFSET	OUT	1.
IERROR	OUT	

1. Value may exceed the range of a signed integer; see 2.3.2.2.

8.6.2.3 Errors

Upon successful completion of *PXFFTELL*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFFTELL*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL]	No file is connected to *IUNIT*.
[ESPIPE]	The *IUNIT* argument is connected to a pipe or FIFO.

8.6.3 Get a Character

Subroutine: *PXFGETC*(), *PXFFGETC*()

8.6.3.1 Synopsis

```
SUBROUTINE PXFGETC (CHAR, IERROR)
CHARACTER*1 CHAR
INTEGER IERROR

SUBROUTINE PXFFGETC (IUNIT, CHAR, IERROR)
CHARACTER*1 CHAR
INTEGER IUNIT, IERROR
```

8.6.3.2 Description

These subroutines shall read a byte from a file connected to an external unit. When a byte is read, the current file position shall be incremented by one byte. FORTRAN 77 record processing shall not apply to bytes read using these subroutines.

The *PXFGETC*() subroutine shall read from the unit connected to standard input STDIN_UNIT and is equivalent to the call

```
PXFFGETC(IPXFCONST('STDIN_UNIT'), CHAR, IERROR)
```

Table 8.15—Arguments for *PXFGETC()* and *PXFFGETC()*

FORTRAN Argument	Intent	Notes
IUNIT	IN	
CHAR	OUT	
IERROR	OUT	

8.6.3.3 Errors

Upon successful completion of *PXFGETC*() or *PXFFGETC*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETC*() and *PXFFGETC*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EEND]	The end of file has been encountered.

8.6.4 Write a Character

Subroutine: *PXFPUTC*(), *PXFFPUTC*()

8.6.4.1 Synopsis

```
SUBROUTINE PXFPUTC (CHAR, IERROR)
CHARACTER*1 CHAR
INTEGER IERROR

SUBROUTINE PXFFPUTC (IUNIT, CHAR, IERROR)
CHARACTER*1 CHAR
INTEGER IUNIT, IERROR
```

8.6.4.2 Description

These subroutines shall write a byte to a file connected to an external unit. When a byte is written, the current file position shall be incremented by one byte. FORTRAN 77 record processing shall not apply to bytes written using these subroutines.

Table 8.16—Arguments for *PXFPUTC()* and *PXFFPUTC()*

FORTRAN Argument	Intent	Notes
IUNIT	IN	
CHAR	IN	
IERROR	OUT	

The *PXFPUTC*() subroutine writes to the unit connected to standard output {STDOUT_UNIT} and is equivalent to the call

```
PXFFPUTC(IPXFCONST('STDOUT_UNIT'), CHAR, IERROR)
```

8.6.4.3 Errors

Upon successful completion of *PXFPUTC*() or *PXFFPUTC*(), the argument *IERROR* shall be set to zero. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

8.7 Bit Field Manipulation

The following subroutines and functions shall be provided to construct and manipulate bit patterns within an integer variable. This functionality is required in order to achieve the range of capability provided by the system-defined integer constants (i.e., the ability to combine such values into a single value to be sent to a system call).

8.7.1 Inclusive OR

Function: *IOR*()

8.7.1.1 Synopsis

```
INTEGER FUNCTION IOR (M, N)
INTEGER M, N
```

8.7.1.2 Description

The *IOR*() function returns the inclusive-or result of the bit patterns contained in the input arguments *M* and *N*, as shown in Table 8.17.

Table 8.17—Definition of Inclusive-Or

Bit in Argument *M*	Bit in Argument *N*	Bit in Result
0	0	0
0	1	1
1	0	1
1	1	1

8.7.1.3 Errors

The *IOR*() function is always successful, and no return argument is specified to indicate an error.

8.7.2 Logical AND

Function: *IAND*()

8.7.2.1 Synopsis

```
INTEGER FUNCTION IAND (M, N)
INTEGER M, N
```

8.7.2.2 Description

The *IAND*() function returns the logical-and result of the bit patterns contained in the input arguments *M* and *N*, as shown in Table 8.18.

Table 8.18—Definition of Logical-And

Bit in Argument *M*	Bit in Argument *N*	Bit in Result
0	0	0
0	1	0
1	0	0
1	1	1

8.7.2.3 Errors

The *IAND*() function is always successful, and no return argument is specified to indicate an error.

8.7.3 Bitwise NOT

Function: *NOT*()

8.7.3.1 Synopsis

```
INTEGER FUNCTION NOT (M)
INTEGER M
```

8.7.3.2 Description

The *NOT*() function returns the bitwise-not result of the bit pattern contained in the input argument *M*, as shown in Table 8.19.

Table 8.19—Definition of Bitwise-Not

Bit in Argument *M*	Bit in Result
0	1
1	0

8.7.3.3 Errors

The *NOT*() function is always successful, and no return argument is specified to indicate an error.

8.8 System Date and Time

The following subroutine shall be provided to access the system clock based on the *TZ* environment variable (see POSIX.1 {2} 2.6).

8.8.1 Local Time

Subroutine: *PXFLOCALTIME*()

8.8.1.1 Synopsis

```
SUBROUTINE PXFLOCALTIME (ISECNDS, IATIME, IERROR)
INTEGER ISECNDS, IATIME(9), IERROR
```

8.8.1.2 Description

The *PXFLOCALTIME*() subroutine converts the time (in seconds since the epoch) in the *ISECNDS* argument to local date and time as described by the integer array *IATIME* as shown:

IATIME(1)= Seconds (0–61)
IATIME(2)= Minutes (0–59)
IATIME(3)= Hours (0–23)
IATIME(4)= Day of the month (0–31)
IATIME(5)= Month of the year (1–12)
IATIME(6)= Gregorian year (e.g., 1990)
IATIME(7)= Day of the week (0 = Sunday)
IATIME(8)= Day of the year (1–366)
IATIME(9)= Daylight savings flag (0 = standard, nonzero = daylight savings)

Table 8.20—Arguments for *PXFLOCALTIME()*

FORTRAN Argument	Intent	Notes
ISECNDS	IN	
IATIME	OUT	
IERROR	OUT	

If *IATIME* is not dimensioned to at least nine elements, the action performed by *PXFLOCALTIME*() is undefined.

Local time as returned by *PXFLOCALTIME*() is relative to the time zone defined by the current value of the *TZ* time-zone environment variable (see POSIX.1 {2} 2.7) or based on implementation-defined default time-zone information if *TZ* is absent from the environment. The environment variable *TZ* can be set using *PXFSETENV*() (see 4.6.1). The value of *TZ* shall be as defined by POSIX.1 {2} 8.1.1.

8.8.1.3 Errors

Upon successful completion of *PXFLOCALTIME*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFLOCALTIME*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL] The current value of the *TZ* environment variable is invalid.

8.9 Command-Line Arguments

The following subroutines shall be provided to access the arguments of the command that invoked the application.

8.9.1 Get Command-Line Argument

Subroutine: *PXFGETARG*()

8.9.1.1 Synopsis

```
SUBROUTINE PXFGETARG (M, BUF, ILEN, IERROR)
CHARACTER*(*) BUF
INTEGER M, ILEN, IERROR
```

8.9.1.2 Description

The *PXFGETARG*() subroutine examines the command used to invoke the executing program and places the *M*th command-line argument in the character string *BUF*. If *M* has a value of zero, the value of the argument returned is the command name. The significant length of *BUF* is returned in *ILEN*.

Table 8.21—Arguments for *PXFGETARG()*

FORTRAN Argument	Intent	Notes
M	IN	
BUF	OUT	
ILEN	OUT	
IERROR	OUT	

8.9.1.3 Errors

Upon successful completion of *PXFGETARG*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETARG*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.9.

[EINVAL]	The argument *M* is out of range.
[ETRUNC]	The declared length of the character argument *BUF* is insufficient to contain the string to be returned. (See 2.3.2.4.)

8.9.2 Index of Last Command-Line Argument

Function: *IPXFARGC*()

8.9.2.1 Synopsis

```
INTEGER FUNCTION IPXFARGC()
```

8.9.2.2 Description

The function *IPXFARGC*() returns the number of command-line arguments, *excluding* the command name, in the command used to invoke the executing program. A return value of zero indicates that there are no command-line arguments other than the command name itself.

8.9.2.3 Errors

The *IPXFARGC*() function is always successful, and no return argument is specified to indicate an error.

8.10 Character String Procedures

8.10.1 Length of a String Trimmed of Trailing Blanks

Function: *IPXFLENTRIM*()

8.10.1.1 Synopsis

```
INTEGER FUNCTION IPXFLENTRIM (STRING)
CHARACTER*(*) STRING
```

8.10.1.2 Description

The function *IPXFLENTRIM*() returns the index of the last nonblank character in the input argument *STRING*, or zero if all characters in *STRING* are blank characters.

8.10.1.3 Errors

The *IPXFLENTRIM*() function is always successful, and no return argument is specified to indicate an error.

8.11 Extended Range Integer Manipulation

8.11.1 Unsigned Comparison

Function: *PXFUCOMPARE*()

8.11.1.1 Synopsis

```
SUBROUTINE PXFUCOMPARE (I1, I2, ICMPR, IDIFF)
INTEGER I1, I2, ICMPR, IDIFF
```

8.11.1.2 Description

The subroutine *PXFUCOMPARE*() is used to determine the difference between two integer arguments representing unsigned (extended range; see 2.3.2.2) numbers.

Table 8.22—Arguments for *PXFUCOMPARE()*

FORTRAN Argument	Intent	Notes
I2	IN	
I2	IN	
ICMPR	OUT	
IDIFF	OUT	

The argument *ICMPR* indicates the relative value of the two unsigned numbers, as shown in Table 8.22.

Table 8.23—*ICMPR* Return Values

Value of ICMPR	Relation of I1 and I2
-1	I1 > I2
0	I1 = I2
1	I1 <I2

The argument *IDIFF* shall provide the absolute value of the difference of *I1* and *I2*.

8.11.1.3 Errors

The *PXFUCOMPARE*() subroutine is always successful, and no return argument is specified to indicate an error.

8.12 Process Termination

Process termination shall occur when the FORTRAN 77 STOP statement is executed or the FORTRAN 77 END statement in the main program is executed. This subsection describes the interactions of FORTRAN 77 process termination with procedures defined by POSIX.9. These interactions define behavior that is undefined or unspecified by FORTRAN 77 and do not modify or replace any behavior that is defined in FORTRAN 77.

8.12.1 Interactions of the FORTRAN 77 STOP Statement

The FORTRAN 77 STOP statement shall terminate the program with at least the consequences of *PXFFASTEXIT*() with a value for the *ISTATUS* argument. If the optional argument to STOP exists and is a string of digits, the termination consequences shall be as if these digits were interpreted as the integer value of the *ISTATUS* argument. Otherwise, the termination consequences shall be as if the *ISTATUS* argument was set to zero.

8.12.2 Interactions of the FORTRAN 77 END Statement

The execution of the FORTRAN 77 END statement in the main program shall terminate the program with at least the consequences of calling *PXFFASTEXIT*() with a value of zero for the status argument.

8.12.3 POSIX-Based Fortran Process Termination

Function: *PXFEXIT*()

8.12.3.1 Synopsis

```
SUBROUTINE PXFEXIT (ISTATUS)
INTEGER ISTATUS
```

8.12.3.2 Description

The *PXFEXIT*() subroutine shall provide the same FORTRAN 77 functionality as execution of the FORTRAN 77 END statement in the FORTRAN 77 main program and shall provide the same POSIX.1 {2} functionality as the POSIX.1 {2} function *_exit*() (see POSIX.1 {2} 3.2). There is no possible return value from *PXFEXIT*() and no *IERROR* argument is defined for *PXFEXIT*(). Arguments for *PXFEXIT*() correspond to the arguments for *_exit*(), as shown in Table 8.24.

Table 8.24—Arguments for *PXFEXIT()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
ISTATUS	status	IN	

9. System Databases

9.1 System Databases

9.2 Database Access

9.2.1 Group Database Access

Subroutines: *PXFGETGRGID*(), *PXFGETGRNAM*()

9.2.1.1 Synopsis

```
SUBROUTINE PXFGETGRGID (IGID, JGROUP, IERROR)
INTEGER IGID, JGROUP, IERROR

SUBROUTINE PXFGETGRNAM (NAME, ILEN, JGROUP, IERROR)
CHARACTER*(*) NAME
INTEGER ILEN, JGROUP, IERROR
```

9.2.1.2 Description

The *PXFGETGRGID*() and *PXFGETGRNAM*() subroutines shall provide the same functionality as the POSIX.1 {2} functions *getgrgid*() and *getgrnam*() (see POSIX.1 {2} 9.2). Arguments for *PXFGETGRGID*() and *PXFGETGRNAM*() correspond to the arguments for *getgrgid*() and *getgrnam*(), as shown in Table 9.1.

Table 9.1—Arguments for *PXFGETGRGID()* and *PXFGETGRNAM()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IGID	gid	IN	
NAME	name	IN	
ILEN	--	IN	Length of NAME; see 2.3.2.4
JGROUP	ret_value	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘group’,...); see 8.3.1.

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘group’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *group* structure as defined in POSIX.1 {2} 9.2. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JGROUP, COMPNAM, IVALUE, IERROR)
INTEGER JGROUP, IVALUE, IERROR

SUBROUTINE PXFSTRGET(JGROUP, COMPNAM, SVALUE, ILEN, IERROR)
INTEGER JGROUP, ILEN, IERROR
CHARACTER*(*) SVALUE

SUBROUTINE PXFESTRGET(JGROUP, COMPNAM, INDEX, SVALUE, ILEN, IERROR)
INTEGER JGROUP, INDEX, ILEN, IERROR
CHARACTER*(*) SVALUE
```

where *JGROUP* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 9.2.

Table 9.2—Components for *group* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
gr_name	‘gr_name’	PXFSTRGET
gr_gid	‘gr_gid’	PXFINTGET
--	‘gr_nmem’	PXFINTGET
gr_mem	‘gr_mem’	PXFESTRGET

The component *gr_mem* is an array of character strings that can only be accessed one element at a time. The number of elements in *gr_mem* is contained in the component *gr_nmem*, which is *not* a structure component defined by POSIX.1 {2}.

9.2.1.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *getgrgid*() and *getgrnam*() functions. Upon successful completion of *PXFGETGRGID*() and *PXFGETGRNAM*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETGRGID*() and *PXFGETGRNAM*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ENOENT] The requested entry could not be found.

9.2.2 User Database Access

Subroutines: *PXFGETPWUID*(), *PXFGETPWNAM*()

9.2.2.1 Synopsis

```
SUBROUTINE PXFGETPWUID (IUID, JPASSWD, IERROR)
INTEGER IUID, JPASSWD, IERROR

SUBROUTINE PXFGETPWNAM (NAME, ILEN, JPASSWD, IERROR)
CHARACTER*(*) NAME
INTEGER JPASSWD, ILEN, IERROR
```

9.2.2.2 Description

The subroutines *PXFGETPWUID*() and *PXFGETPWNAM*() shall provide the same functionality as the POSIX.1 {2} functions *getpwuid*() and *getpwnam*() (see POSIX.1 {2} 9.2.) Arguments for *PXFGETPWUID*() and *PXFGETPWNAM*() correspond to the arguments for *getpwuid*() and *getpwnam*(), as shown in Table 9.3.

Table 9.3—Arguments for *PXFGETPWUID()* and *PXFGETPWNAM()*

FORTRAN Argument	POSIX.1 Argument	Intent	Notes
IUID	uid	IN	
NAME	name	IN	
ILEN	--	IN	Length of NAME; see 2.3.2.4
JPASSWD	ret_value	IN	1.
IERROR	ret_value/errno	OUT	

1. Handle obtained from PXFSTRUCTCREATE (‘passwd’,...); see 8.3.1.

The *PXFSTRUCTCREATE*() subroutine (see 8.3.1) with the string ‘passwd’ given as the *STRUCTNAME* argument shall be used to obtain a handle for an instance of the *passwd* structure as defined in POSIX.1 {2} 9.2. Each component access shall require one of the following structure-component manipulation subroutines (see 8.3.2):

```
SUBROUTINE PXFINTGET(JPASSWD, COMPNAM, IVALUE, IERROR)
INTEGER JPASSWD, IVALUE, IERROR

SUBROUTINE PXFSTRGET(JPASSWD, COMPNAM, SVALUE, ILEN, IERROR)
INTEGER JPASSWD, ILEN, IERROR
CHARACTER*(*) SVALUE
```

where *JPASSWD* is a handle and *COMPNAM* is a character expression which evaluates to one of the component names shown in Table 9.4.

Table 9.4—Components for *passwd* Structure

POSIX.1 Component	COMPNAM	Structure Procedures Used to Access
pw_name	‘pw_name’	PXFSTRGET
pw_uid	‘pw_uid’	PXFINTGET
pw_gid	‘pw_gid’	PXFINTGET
pw_dir	‘pw_dir’	PXFSTRGET
pw_shell	‘pw_shell’	PXFSTRGET

9.2.2.3 Errors

POSIX.1 {2} does not specify any error conditions that are required to be detected for the *getpwuid*() and *getpwnam*() functions. Upon successful completion of *PXFGETPWUID*() and *PXFGETPWNAM*(), the argument *IERROR* shall be set to zero. If any of the following conditions occur, *PXFGETPWUID*() and *PXFGETPWNAM*() shall set the argument to the corresponding value. *IERROR* may be set to a nonzero value to indicate error conditions that are not specified by POSIX.1 {2} and POSIX.9.

[ENOENT] The requested entry could not be found.

10. Data Interchange Format

10.1 Archive/interchange File Format

The functionality described in this section in POSIX.1 {2} is outside the scope of this standard.

Annex A Rationale and Notes

(Informative)

The Annex summarizes the development of the FORTRAN 77 language binding to the ISO/IEC 9945-1: 1990 (IEEE Std 1003.1-1990). It presents the deliberations of the developers of this standard, discusses design considerations and alternatives, and provides notes of interest to application programmers and system implementors where appropriate.[3]

This rationale is modeled after the rationale accompanying the existing ISO/IEC 9945-1: 1990 standard, and, like that one, is organized such that the rationale follows the exact structure of the standard (with the exception of the introductory sections, which cover additional material).

A.1 General

The development of FORTRAN 77 Bindings to the (at the time proposed) POSIX.1 {2} standard was first discussed at the */usr/group/supercomputing* meeting in June 1987. The discussion was initiated by supercomputer users anxious to program in the POSIX environment. The first draft of this standard was presented in January 1988 to the FORTRAN subgroup. After further subgroup work, the proposal was accepted by the committee and officially forwarded by */usr/group* to the IEEE P1003 committee in September 1988.

In January 1989, the P1003.9 working group was formed and was charged with developing FORTRAN Bindings to POSIX. The document produced under */usr/group* was used as the base document, and subsequent work on the standard focused largely on integrating characteristics of the FORTRAN programming environment and common practice into the heavily UNIX®-oriented[4] POSIX environment. The group was comprised of a mixture of vendors and users, with a variety of FORTRAN and UNIX expertise.

Following the acceptance of this first FORTRAN bindings standard, the committee will continue work on developing FORTRAN bindings to other POSIX functional areas (e.g., IEEE P1003.2, P1003.4, etc.).

A.1.1 Scope

The areas declared as out of scope were done so for various reasons, but were primarily motivated by the desire to limit the work to a small enough area that could gain the consensus of the affected community and still be of value to that community. When a topic area was considered highly contentious, with nearly equal arguments both for including it and against including it, the developers of this standard generally chose to exclude it from this standard. As a result a number of “nice to have” or “convenient” features were not included. This approach was often softened by the belief that this binding, because of its use of FORTRAN 77 and the imminent emergence of Fortran 90, could be viewed as an interim standard. The coming Fortran 90 standard was a major reason for scope restriction (1), which avoided significant extensions to FORTRAN 77 in this binding. The expectation of a subsequent binding using Fortran 90 was a major factor in restriction (2), which avoids dealing with any part of Fortran 90 was a major factor in restriction (2), which avoids dealing with any part of Fortran 90 in this binding. A combination of limited development resources and lack of desire to “open Pandora’s box” caused restriction (3). Finally, the developers of this standard wished to minimize the impact on existing implementations of FORTRAN 77 by the installation of these bindings. A fourth scope limitation was in place throughout the development of this standard and well into balloting. That scope limitation was: “Possible side effects to the operating system by standard FORTRAN 77 language constructs (e.g., READ, WRITE, STOP), or such side effects by I/O routines specified in Section 8 of this standard which operate on files connected to FORTRAN 77 units (as opposed to POSIX.1 {2} file descriptors).” Significant ballot objections were raised to this scope limitation.

[3]The material in this annex is derived in part from copyrighted draft documents developed under the sponsorship of UniForum, as a part an ongoing program of that association to support the POSIX standards program efforts.
[4]UNIX is a trademark of Unix System Laboratories in the US and other countries..

FORTRAN I/O in a POSIX.1 {2} environment is a highly contentious issue. POSIX.1 {2} I/O has no concept of a record, a concept integral to FORTRAN I/O. While POSIX.1 {2} I/O has some special behavior concerning the line feed and null characters, this is not the same as FORTRAN records. The developers of this standard intentionally avoided (as much as possible) specifying the underlying implementation of these bindings. It recognized that existing implementations of FORTRAN I/O may need to exist in parallel with POSIX I/O, where FORTRAN records might have nothing to do with line feeds. However, due to ballot objections, the operating system side effects of "native" FORTRAN 77 constructs, especially I/O constructs, were specified, but the side effects were limited to those files that were explicitly created as POSIX-based FORTRAN I/O files. Even the POSIX I/O flag was originally included only after considerable debate, because guaranteeing the specified behavior is essential to constructing FORTRAN 77 utilities that are able to be piped with other traditional utilities (e.g., *cat, sort, grep,* etc.).

A.1.2 Normative References

In addition to the references used in the main body of this standard, the following standards or drafts under development are referred to in this appendix:

{A1} ISO/IEC 1539:1991, *Information technology-Programming languages-FORTRAN*.

{A2} ISO/IEC 9945-2:, *Information technology-Portable Operating System Interface (POSIX)-Part 2: Shell and Utilities*

{A3} P1003.6/D12, *Draft Security Interface Standards for POSIX*.[5]

{A4} MIL-STD-1753, *Military Standard: FORTRAN, DOD Supplement to American National Standard X3.9-1978.*

A.1.3 Conformance

A.1.3.1 Implementation Conformance

There is no additional rationale provided for this subclause.

A.1.3.2 Application Conformance

There is no additional rationale provided for this subclause.

A.1.3.3 Language-Dependent Services for the FORTRAN 77 Programming Language

There is no additional rationale provided for this subclause.

A.1.3.3.1 FORTRAN 77 Language Binding

Further information on issues and discussions related to FORTRAN 77 standard conformance is given in 2.3.2.1 and 2.3.2.4.

A.1.3.4 Other Language Related Specifications

The FORTRAN 77 standard defines the functions *CHAR*() and *ICHAR*() with respect to some implementation-defined collating sequence. For details see Section 3.1 of the FORTRAN 77 {3} standard, especially Section 3.1.5, concerning the collating sequence, and see Section 15.10 of the FORTRAN 77 {3} standard, note (5) Table 5, for the description of *CHAR*() and *ICHAR*() as mapping to this implementation-defined collating sequence. While this collating sequence must at least contain the 49 FORTRAN 77 characters in the order specified, special characters might, for example,

[5]To be approved and published.

come before the alphabet. Therefore, there is normally no guarantee in FORTRAN 77 that CHAR(0) equals the C-language character '\0'. CHAR(0) simply equals whatever is the first character in the collating sequence.

A.2 Terminology and General Requirements

A.2.1 Conventions

A.2.1.1 Typographical Conventions

There is no additional rationale provided for this subclause.

A.2.1.2 Namespace Conventions

There is no additional rationale provided for this subclause.

A.2.1.2.1 Subroutine Naming

There was a great deal of debate about the prefix to be used for procedure names. It was agreed that the prefix needed to be fairly unusual, to minimize naming conflicts with names in existing application code, and short, to maximize usability. In early drafts, the prefix *F77* was used. This convention was strenuously objected to by members of the ANSI Fortran 90 committee. (In fact, they voted unanimously against it.) The chosen prefix *PXF* is a shorthand notation for POSIX-FORTRAN, which represents the bridge between the two worlds. Since naming conventions are personal, aesthetic choices, it is unlikely that any prefix chosen will be considered acceptable to all. The *PXF* prefix, it is hoped, will be objectionable to the fewest.

A.2.1.2.2 Function Naming

There was some debate over whether or not to prefix functions returning an integer with the letter I and functions returning a logical with the letter L. Some thought a consistent use of the *PXF* prefix in all procedures was better, and some wanted to take into account the historical carelessness of FORTRAN programmers, who are used to letting variables and functions be undeclared and thus be declared implicitly. The ease-of-use argument won out for the I prefix, which implicitly declares the function to return an integer, but since there was no equivalent argument for logical functions (an L prefix implicitly causes an integer return), the L prefix was dropped.

A.2.2 Definitions

A.2.2.1 Terminology

There is no additional rationale provided for this subclause.

A.2.2.2 General Terms

The term *intent* is derived from the Fortran 90 standard, where it is a keyword used to describe the intended usage of an actual argument. In Fortran 90, an "intent out" argument guarantees that the variable declared will be set by the subprogram. This standard does not intend that usage, but only the looser, English usage; namely, what the argument is intended to be used for: passing arguments to or from the subprogram.

A.2.2.3 Abbreviations

There is no additional rationale provided for this subclause.

A.2.3 FORTRAN 77 Language Bindings Concepts

This section of the rationale is used to present many high-level objectives and design alternatives considered in the development of the FORTRAN 77 bindings. The topics and issues discussed here are those that have broad effects on the bindings specification. Full specifications and low-level technical details associated with the interfaces are provided in the appropriate sections elsewhere in the rationale.

A.2.3.0.1 Choice of Standards

A.2.3.0.1.1 FORTRAN 77 Versus Fortran 90

A fundamental issue addressed by the developers of this standard early in the standard-development process was the choice between using FORTRAN 77 or Fortran 90 as the base language for this standard. A summary of the issues related to both possibilities follows.

— *FORTRAN 77*. The work of the established FORTRAN community is predominantly based on the FORTRAN 77 language. Many FORTRAN 77 applications will be ported to POSIX-conforming systems, and new application code will continue to be written in FORTRAN 77 on such systems. In order to provide utility to the established community, it is necessary to work with FORTRAN 77. Furthermore, the size of the FORTRAN 77 community guarantees that it will continue to be an effective standard for the indefinite future.

— *Fortran 90*. The emerging Fortran 90 standard provides many language features that could be used effectively in developing a bindings standard. For example, the presence of structured data types will allow the use of more traditional (in the POSIX environment) data handling techniques. However, there are several drawbacks associated with using Fortran 90 at its inception, most notably its lack of presence in the FORTRAN community. Having just been finalized during the late stages of the FORTRAN 77 bindings development, but not formally approved as a standard by start of balloting of this standard, it will probably be several years until it is in widespread use.

NOTE — Although the decision was made to produce this FORTRAN 77 Binding first, the developers of this standard have already begun work on a Fortran 90 Binding. This future Binding will take advantage of the new features provided in Fortran 90, but is intended to coexist with this FORTRAN 77 binding, both as a standard and as an implementation.

A.2.3.0.1.2 POSIX Language Independence

In early 1990, the IEEE, in response to direction from ISO, mandated a formal change in the structure of the POSIX standards, namely, a shift towards a more dear division of work between *language-independent functional standards*, and *language bindings* to those functional standards. Using this structure, all functional standards are to be specified in a language-independent style and a language binding is always to be correlated to the appropriate functional standard. This division of work forces the functionality to be specified in a more abstract style and provides the language binding developers more freedom to develop a binding that is particularly appropriate for their language.

The most specific target of this division of work is the fact that all earlier POSIX work was specified using the C language, a convention that resulted in the dependence on C-Language-specific features in many areas. Included in this body of earlier work was the now obsolete IEEE Std 1003.1-1988, which was used as the functional basis for this FORTRAN 77 Binding. Because this new division of work within POSIX was so late in gaining momentum, the language-independent version of POSIX.1 {2} is scheduled to be balloted after this standard; therefore, the developers of the P1003.9 standard used the existing POSIX.1 {2} standard as the reference specification. The result of this decision is that many of the technical decisions the developers have made deal with the differences between the C and FORTRAN 77 languages. In fact, the issues confronted by the developers of this standard have led to extensive feedback to the developers of the language-independent specifications. The language-independent specifications should begin appearing in the future, and subsequent language bindings work will use these newer specifications as base reference standards.

A.2.3.0.2 Design Objectives

The primary goals behind the design of this standard were as follows:

1) *Standardization and System Independence*. In order to achieve complete portability, following existing standards as closely as possible was the foremost objective. More specifically, the ANSI X3.9-1978 FORTRAN {3} standard was the primary basis of the language requirements; with one exception (long identifier names; see 1.2.3.1 and A.2.3.0.4.1), there are no dependencies on language extensions.
2) *Consistency*. The FORTRAN 77 language bindings must present a consistent user and system interface. Additionally, this binding definition should be capable of serving as a model for future development of FORTRAN 77 bindings to other areas of POSIX functionality, and possibly for other programming languages binding to POSIX.
3) *Integration of FORTRAN 77 and UNIX*. This was of course the motivation for the development of these bindings: to allow effective FORTRAN 77 programming in a UNIX (POSIX) environment. Preserving the key elements as well as accepted or common practices of both of these environments was essential. Among the developers of this standard, this issue was often referred to as "FORTRAN-ness versus UNIX-ness."
4) *Run-Time Performance*. Performance is always a concern, but it was definitely not as crucial as the previous goals. In order to achieve the three primary goals, some performance efficiency may be sacrificed; however, the overall benefits of achieving the above objectives far outweigh the lost efficiency. Also, whenever possible, notes have been made in this rationale to describe possible implementation alternatives that may enhance performance. However, the developers of this standard did attempt to avoid precluding a performance-efficient implementation or extensions to provide efficiency.

A.2.3.0.3 Design Strategy

Consistency was a major goal throughout this standard. There is a direct correspondence between the POSIX.1 {2} (C language) bindings and these FORTRAN 77 language bindings. There is a FORTRAN 77 interface defined for every POSIX.1 {2} system call, plus a few additional procedures that are necessary for achieving the complete POSIX.1 {2} functionality. In practice, it is most likely that the initial FORTRAN 77 language bindings would be implemented as a set of interface procedures built on top of the existing C-language system calls.

In order to implement this design strategy of corresponding C/FORTRAN system interfaces, a convention had to be developed to differentiate the FORTRAN 77 procedures from the C system routines (the same names could not be used; see A.2.3.0.4.1). Therefore, the convention of prefixing the three characters PXF to the actual system routine name was used. With this design, using the system calls from FORTRAN 77 is very similar to using them from C; i.e., with a few exceptions, the calling sequences of the FORTRAN 77 and C versions are identical.

A.2.3.0.4 Extensions to and Deviations From the FORTRAN 77 Standard

In the early development of these FORTRAN 77 bindings, various technical proposals required extensions to and deviations from the ANSI X3.9-1978 FORTRAN {3} standard One such extension — the use of identifier names greater than six characters in length — was retained, but all others were discarded.The rationale for these decisions follows.

A.2.3.0.4.1 Length of Identifier Names

The ANSI X3.9-1978 FORTRAN {3} standard states that identifier names can contain only six characters. In practice, most implementations allow identifiers much longer than six characters, although some still require uniqueness within six or eight characters. However, most newer systems provide a higher limit for uniqueness; it was this precedent that the developers of this standard chose to follow.

This FORTRAN 77 bindings standard specifies identifiers containing up to 15 characters, requiring a maximum of 11 to determine uniqueness (this "worst case" occurs in the structure-handling routines; see 8.3). Rather than use this limit as the general requirement, the developers of this standard decided to adopt the 31-character limit that is common

among implementations and is specified by POSIX.1 {2} (see POSIX.1 {2} 1.3.5), the Fortran 90 {A1} standard, and the IEEE P1003.2 {A2} standard, the latter in its section concerning the execution environment of languages. This higher limit will provide sufficient flexibility for binding to other POSIX functional areas, as well as support for implementing system- and site-dependent extensions to POSIX in a consistent style.

The decision to require this extension to the FORTRAN 77 language faced much opposition in the early development of this standard. Several alternatives were suggested to avoid the need for the longer identifier names. These alternatives, and their rebuttals, are summarized below.

— *Choose names that fit within the six-character limit.* An encoding of all the bindings into six characters would result in a cryptic name space that would greatly decrease usability. It would also defeat the goal of having a consistent and direct name correspondence with the C-language bindings. Furthermore, the six-character limitation is thought to be based on old compilers and linkers from physically addressed machines with small (e.g., 64 Kbyte) address spaces. Most compilers and linkers already allow far more than six characters.
— *Choose names that fit within an eight-character limit.* It was argued that eight characters was a good compromise, since it is nearly a *de facto* standard minimum length for industry linkers. At an early stage of development, the developers of this standard proposed a set of conventions by which the names of the bindings interfaces (as of that time) could be converted into a set that was unique in eight characters. However, as the bindings matured, these conventions were soon inadequate. This scheme was abandoned as a result of its limited flexibility and extensibility.
— *Specify a preprocessor to convert the long binding names into a six- or eight-character encoding.* Specifying such a preprocessor would be difficult, and its presence would represent a substantial modification of the common FORTRAN programming environment. Of course, a vendor could choose to provide a preprocessor as a last resort, although it is thought that it would be strongly resisted by the user community. Another option would be for the compiler to recognize all binding interfaces as intrinsics, although this too has undesirable effects on the implementation.
— *Use the same interface names as the POSIX.1 {2} standard.* This alternative requires the implementation compiler and/or linker to be able to differentiate between source languages (e.g., by attaching a language identification to each symbol table entry). This scheme suffers from the potential flaw of impeding the linking of multiple-language programs. A practical drawback is the substantial implementation effort that this scheme might require.
— *Use the "single-entry-point" method.* In this scheme, all bindings would be accessed through a single common interface, [e.g., SYCALL('RENAME',...) instead of PXFRENAME(...)]. Besides introducing another standard deviation (variable-length argument lists; see A.2.3.0.4.2), it contained potential problems with program size, due to the static linking model common to UNIX systems, and might impair usability by hiding useful program development and debugging information.

A.2.3.0.4.2 Variable-length Argument Lists

Early drafts of this standard included instances of procedures requiring variable-length argument lists; however, such argument lists are in violation of the FORTRAN 77 standard. This was viewed as a significant deviation, as many compilers and linkers check the length of argument lists and issue warnings or errors for length mismatches. In order to accommodate variable-length argument lists, this useful diagnostic capability would have to be removed from those implementations.

Other options were examined, such as

1) Requiring the maximum number of arguments and passing null or zero values for the unused arguments, and
2) Specifying different versions of the same procedure to be used based on the number and/or type of actual arguments

However, each alternative has severe usability drawbacks. It was instead decided to modify or remove the specification of the nonconforming procedures to eliminate the problem. The routines that required the variable-length argument

lists were recognized as redundant, so they were removed with no loss of functionality in this standard. See 3.1.2 and A.3.1.2 for descriptions and further information about the specific procedures that caused this issue to be addressed.

A.2.3.0.4.3 Variable-Type Arguments

Early drafts of this standard specified a different implementation of the data abstraction concept that was used to access aggregate data (see 2.3.2, 8.3): rather than specify a different component-access routine for each different base type, a different access routine was specified for each unique structure. Using the older method, if a structure contains components of differing types, the type of the argument to the access routine will vary according to which component is being accessed. Other past proposals relied on a similar flexibility, including the "single entry point" alternative discussed in A.2.3.0.4.1. Early readings of the ANSI X3.9-1978 FORTRAN {3} standard seemed to reveal a lack of definition in the relevant areas, but an interpretation from the ANSI Fortran committee, X3J3, indicated that such generic-type behavior is in fact in violation of the standard. An excerpt from the interpretation follows:

> "There is no way in FORTRAN 77 that a user can provide the generic behavior of intrinsic functions. Therefore, a standard conforming set of language bindings to a set of supplied library functions requires type matching..."

While this generic-type functionality is available on many systems, the developers of this standard decided to modify the nonconforming areas of the bindings to remove the need for it. Consequently, the data abstraction implementation was modified as mentioned above.

A.2.3.0.4.4 Character Set Restrictions

Lowercase alphabetic characters are not strictly conformant to the standard, although their use is a very commonly implemented extension. Binding names in the proposal were previously shown in lowercase, more for aesthetic presentation than with an intent to require an implementation to support this extension. After encountering standard-based objections to this, all procedure names were changed to uppercase, since this is in agreement with the ANSI X3.9-1978 FORTRAN {3} standard.

Many implementations that support both cases fold the cases to one. Thus, alternatives that require a distinction to be recognized between uppercase and lowercase were not considered.

Another nonstandard convention in an early draft of these bindings was the use of the underscore character, which is not part of the character set of the FORTRAN 77 language standard. All of the function names began with *F77_*, but were later changed to use just *F77* (and then to *PXF*) as the prefix.

A.2.3.0.4.5 MIL-STD-1753 Extensions

The MIL-STD-1753 {A4} Extensions to the FORTRAN 77 standard provide additional functionality both in terms of language constructs and intrinsic routines; examples include a mechanism for inclusion of headers, and routines to perform bit-manipulation operations. Because this set of extensions is implemented on many systems, it was suggested that this standard either require them to be implemented fully or at least borrow portions of the functionality to meet specific needs. The examples given above (file inclusion, bit manipulation) were among the most obvious examples of functionality that might prove beneficial to the development of this standard. However, it was determined that these extensions are *not* implemented on a substantial number of systems; that MIL-STD-1753 {A4} requires these bit-manipulation procedures to be implemented as externals, and they are often intrinsics; and also that the functionality provided by the complete set of extensions was in fact not critical or highly desirable. Therefore, the decision *not* to require the MIL-STD {A4} Extensions was among the earliest actions of the developers of this standard.

Much of the debate in this area centered on the file inclusion mechanism; see A.2.3.1.1.2 for a technical discussion. A limited set of bit-manipulation operations are required by this standard, and those defined in this standard are functionally equivalent to those defined in MIL-STD-1753 {A4}; see 8.7 for their specification. No other constructs or routines from these Extensions are intentionally duplicated in this standard.

A.2.3.1 System Headers

The POSIX.1 {2} standard specifies many headers intended for inclusion within programs through the use of the preprocessor defined for the C language. These headers contain definitions of *symbolic constants and macros*. Because FORTRAN 77 does not provide equivalent functionality, alternate techniques were developed to provide the required functionality. The following sections introduce the chosen techniques, as well as several others that were considered.

A.2.3.1.1 Symbolic Constants

This standard defines additional interface routines that provide access to symbolic constants defined in POSIX.1 {2}. These routines are introduced in 2.3.1.1 and defined fully in 8.2. The following two clauses discuss two alternate techniques for providing the required functionality. Neither of these methods was ever considered to the point of actually being included in a draft of this standard, but the discussion provides valuable background material.

A.2.3.1.1.1 Proposed Use of a Preprocessor

Ideally, symbolic constants should be defined and used the same way they are in POSIX.1 {2} (i.e., in the C language). Unfortunately, a symbolic preprocessor scheme similar to (or identical to) that defined for the C language (*cpp*) is not implemented by most vendors, and the concept is foreign to most FORTRAN 77 programmers. Even if implemented, the FORTRAN 77 language imposes certain restrictions that limit the usability and usefulness of such a mechanism. Additional relevant information is given below:

- — Unlike C, symbolic names are case-*in*sensitive in FORTRAN 77. Thus, the C convention of defining constants in uppercase to easily distinguish them from real variable names would be of no help. This makes it difficult to define a set of largely invisible, yet readable symbolic constant names that are unlikely to clash with existing user variable names. (FORTRAN 77 programmers are also case-insensitive. Some only use uppercase. Some only use lowercase — a common ANSI extension. Some even mix cases.) Further, if any set of names chosen is different from the C binding names, a parallel set of headers would have to be maintained.
- — Variables need not be declared in FORTRAN 77. Thus, a programmer might accidentally use a common constant name while neglecting to include the correct file. The mistake would not only not be flagged at compile time, but would be difficult to track down at runtime! (For example, IF (IERROR .EQ. ENOENT)...) Misspelled names would also be a problem.
- — Spaces are not significant in FORTRAN 77. A preprocessor might have to be smart enough to parse the entire language in order to properly isolate tokens for substitution.
- — FORTRAN 77 source is line-oriented and limited to 72 characters per line. Textual substitution (e.g., as in *cpp*) would have to be cognizant of this restriction and replace *n*-character symbolic names with n-character numeric constants. This may be a problem if the symbolic name is short but the constant is long (e.g., a constant like HUGE or MAXVAL).
- — C programmers are used to debugging in an environment with a well-defined preprocessor, which is basically part of the language. Programmers who are less familiar with preprocessors may easily get confused when they ask the debugger for the value of a symbolic constant and the debugger does not know about it. This is especially true if language considerations, such as those above, make preprocessing more complex. (Take, for example, a C++ interpreter.)

It is clear that the existing C preprocessor, *cpp*, would not be fully capable, and the specification of an appropriate tool would be a difficult task. The developers of this standard decided that pursuing this approach would not be beneficial.

It should be noted that while the developers of this standard chose not to require any preprocessing mechanism, this decision should in no way be taken as an attempt to preclude the use of preprocessors. An implementation could choose to define its own preprocessing system that could replace all calls to the symbolic constant access routines with the appropriate values at compile time. Such preprocessors may be vendor- or site-dependent.

A.2.3.1.1.2 Proposed Use of the MIL-STD-1753 INCLUDE Mechanism

Another potentially attractive alternative is the INCLUDE statement as defined in the MIL-STD-1753 {A4} Extensions (but not the ANSI X3.9-1978 FORTRAN {3} standard used in conjunction with the FORTRAN 77 PARAMETER statement, which defines the equivalent of a constant. However, this approach has several disadvantages and was discarded by the developers of this standard. The rationale behind this decision is given below:

— Since there are no global variables in FORTRAN 77, header files must be included in *every* program unit (i.e., subroutine or function) that uses a constant, not just at the beginning of the compilation unit. This coordination problem is compounded by the capability of independent compilation of a program and any procedures and/or libraries it may use. This is very inconvenient for the programmer.
— Each PARAMETER statement in each include file is entirely processed by the compiler, not by a preprocessor. This is likely to increase compile time substantially.
— Since the syntax is different from C, even if the symbolic names chosen are the same as the C binding names, a parallel set of include/header files must be maintained.
— Constraints on statement ordering in FORTRAN 77 may restrict the contents of include files to just parameter statements (e.g., no function declarations) and may require precise positioning of the INCLUDE statement within each program unit.
— Again, the MIL-STD-1753 {A4} Extensions are not a part of FORTRAN 77, and are not implemented on all systems. Requiring them to be implemented would be an unreasonable burden on vendors who do not currently support them. (See A.2.3.0.4.5 for further discussion of the consideration of the MIL-STD-1753 {A4} Extensions.)

A.2.3.1.2 Macros

Another common C-language feature used in the POSIX.1 {2} standard is the macro capability. These macros reside in system headers and are accessed from application code in a manner similar to a standard function call. However, the C preprocessor performs macro substitution at compile time, thus eliminating the run-time overhead associated with a standard function call. FORTRAN 77 does not provide any equivalent feature, so two options were discussed.

The approach adopted by the developers of this standard is to specify all functionality, including that explicitly provided with macros in POSIX.1 {2}, through separate interfaces. Therefore, each macro specified in POSIX.1 {2} corresponds to a distinct routine in this standard.

The initial approach taken by the developers of this standard was to specify a generalized utility for accessing the functionality provided in macros in POSIX.1 {2}. This utility was a single interface routine that accepted the name of the desired macro and the required arguments and returned the appropriate result. This scheme was rejected for the following reasons:

— In order to accommodate the full set of macros specified in POSIX.1 {2}, the *PXFMACRO*() function required the use of a variable-length argument list. As discussed earlier, it was decided to eliminate variable-length argument lists (see A.2.3.0.4.2).
— The *PXFMACRO*() function also required the use of variable-type arguments, as the types of the arguments would have to vary according to the macro being specified. As discussed earlier, the developers of this standard decided to eliminate variable-type arguments (see A.2.3.0.4.3).
— In POSIX.1 {2}, the distinction between functions and macros is sometimes vague. This could create confusion within this standard due to uncertainty as to exactly which functionalities should be provided through the *PXFMACRO*() utility or as separate interfaces.

The separate-interfaces approach that was adopted eliminates the need to accept any of the deficiencies of the generalized approach.

Although this standard provides only interfaces corresponding to those macros specified in POSIX.1 {2}, it is intended that this same model could be used to provide a consistent FORTRAN 77 binding to additional system- or site-dependent macro functionalities.

A.2.3.2 Data Types

There is no additional rationale provided for this subclause.

A.2.3.2.1 Primitive Data Types

There is no additional rationale provided for this subclause.

A.2.3.2.2 Numeric Range of Integer Data

A potential problem results from the fact that FORTRAN 77 provides only one integer data type, whereas C provides several (*short, long, unsigned*). In all but a few cases, the FORTRAN 77 *INTEGER* is sufficient to accommodate the intended usage specified in POSIX.1 {2}. However, there are a few cases where it is likely that the lack of an unsigned integer in FORTRAN 77 may limit its ability to provide functionality equivalent to that provided by the C language in POSIX.1 {2}. The specific cases where this problem may arise are values related to time and file offsets. (Regarding time, it is less of a concern for those with units of seconds — they will not expire until about 2033 A.D. However, those measured in CLK_TCKs may expire substantially sooner than expected.) Technical details and a mechanism for dealing with this problem are described below.

Assuming that a FORTRAN 77 *INTEGER* is the same size as a C long (as is true on a large number of implementations), the FORTRAN 77 (signed) variable will be able to store values providing only half the range of the C (unsigned) variable. Actually, the FORTRAN 77 variable can in fact contain the same range, but cannot be conveniently or portably used (compared) beyond the signed integer range without great difficulty. Therefore, this standard provides a subroutine that provides comparison of two integer values that may contain extended-range (unsigned) values. This routine is specified in 8.11.

A.2.3.2.3 Aggregate Data Types

Another of the early fundamental decisions was to use the data abstraction technique in order to hide the complexities of managing aggregate data types from the FORTRAN 77 programmer. This decision led to the consideration of several specific proposals for structure access and manipulation procedures; see A.8.3 for discussion of these various alternatives for specific routines.

As with the other somewhat creative solutions devised, the decision to specify additional interfaces to implement the data abstraction model was not without considerable debate. The following alternate approaches were discussed at the earliest stage of development, but never seriously considered. They are provided here for additional technical detail and background:

— *Use no structure-access procedures*, just add all of the structure members to the argument list of the appropriate system procedures. The advantage is that no additional procedures are required, but the disadvantage is that there is no extensibility (e.g., structure members added or removed, addition of system-specific structure members). Furthermore, the affected *PXF* interface procedures become severely modified and are then inconsistent with the other *PXF* procedures, in terms of correspondence to the POSIX.1 {2} interface definitions.
— *Use the FORTRAN 77 EQUIVALENCE construct* with a local memory buffer to access the data stored in a system structure. Again, the advantage is the lack of additional structure-access procedures, but the drawbacks are severe: applications using this technique would be largely nonportable, as it requires intimate machine-level knowledge of data storage conventions (of course, such information would likely be implementation-dependent).

— *Use the FORTRAN 77 COMMON block construct* with a named COMMON for each structure. Again, the advantage is the lack of additional structure-access procedures, but the drawbacks are similar to the EQUIVALENCE construct and are again severe. Applications using this technique would also be largely nonportable due to implementation-dependent data storage conventions, and a nonstandard mechanism would be required to map the FORTRAN 77 COMMON block to the system structure. With the lack of either a "global" variable construct or a standard INCLUDE feature, the requirement for coordination of the existence of the COMMON block in every routine that needed it would be a problem. Further, any attempt at some form of INCLUDE would impose the variable names in the COMMON block on the program namespace, inviting conflict.

The most contentious issue was potential performance degradation resulting from the additional run-time overhead incurred by additional procedure calls for every structure access. While this may be a reasonable consideration for certain applications on certain systems, the developers of this standard felt that the programming model presented by the data abstraction technique is far superior to the alternatives. Furthermore, the cost of a library call is generally an order of magnitude less than the cost of a system call. Assuming that the structure-access procedures are implemented as library routines, the cost of their use is therefore very small relative to the cost of the associated system call.

In addition, the developers of this standard also hoped to define a construct that would easily deal with the concept of a NULL pointer to a structure, which exists throughout POSIX.1 {2}. Restricting valid handle values to nonzero values permitted reserving the handle value of zero as an equivalence to a NULL pointer, which intuitively matched the C construct.

See 8.3 and A.8.3 for specification and discussion of the issues related to the actual structure-access procedures.

A.2.3.2.4 Character Strings and String Manipulation

In C, character strings are terminated with the NULL character, which is defined to be '\0', but FORTRAN 77 strings are blank-padded and not NULL-terminated. It is the responsibility of the implementation to handle this difference where necessary (e.g., in a system that implements this standard on top of existing C bindings).

Due to the requirements of FORTRAN 77, the maximum length of an actual string argument is always known in a called procedure where it is a formal argument. Assignment of a sequence of characters to the string where the length of the sequence of characters is greater than the length of the string will result in truncation.

Strings (declared CHARACTER*(*) as dummy arguments) in FORTRAN 77 are fixed length and are blank padded. Because of these definitions, it is difficult to differentiate between a string that is *supposed* to contain trailing blanks and one that has simply been blank-padded according to the language definition. The remainder of this section is devoted to the discussion of this issue and the options considered for use in this standard.

The issue of significant trailing blanks provoked extended discussion among the developers of this standard. A problem arises because FORTRAN 77 defines character strings to be fixed length and blank padded, i.e., there are no variable-length strings. This causes difficulties when dealing with many string entities commonly used throughout POSIX.1 {2}, such as path/filenames and environment variables. For example, if a user writes the following code:

```
CHARACTER*14 C
C = 'foo'
CALL PXFOPEN(C,....)
```

is the name of the file that is opened 'foo' or 'fooΔΔΔΔΔΔΔΔΔΔ' (where Δ represents a blank character)? If the latter is the case, unusual filenames will abound on the system; potentially there will be strings differing only in the number of trailing blanks they contain, making it extremely difficult to distinguish between them. For example, another program may use

```
CHARACTER*12 C
```



```
C = 'foo'
CALL PXFOPEN(C,....)
```

This opens a *different* file than the previous program. Common utilities such as *ls* and *rm* would fare poorly, and users of FORTRAN 77 applications using this standard could be very unsatisfied.

The other option — ignoring any trailing blanks — seems more sensible but creates another difficulty. If a filename or environment variable value exists on the system that *does contain* a trailing blank, how does the FORTRAN 77 application access it? POSIX.1 {2} defines the portable filename character set so that it does not contain blanks, but admits the possibility (indeed the probability) of filenames containing nonstandard characters. This leads to a scenario where a FORTRAN 77 application might, for example, back up all the files on a storage device and be unable to work with those files (perhaps created by an errant C or shell application) that are named with filenames containing trailing blanks. Therefore, the developers of this standard deemed this option equally unacceptable.

In order to address this problem, a set of guidelines were determined to measure potential solutions:

— All functionality available from C must be available from FORTRAN 77 (provided the solution is not *too* unusable for the user or implementor).
— The user must be protected (if not prevented) from careless creation of filenames with trailing blanks.
— Performance is important. Solutions that require the string to be parsed in each call are unacceptable.

The committee explored several options covering the range of possibilities considered, including the second form that was retained:

— *Implicit but exact length*. For all routines that pass character strings, the user is required to pass the exact substring required by the subroutine. In the example above,

```
C = 'foo'
CALL PXFOPEN(C(1:3),....)
```

or more likely, using the function defined in Section 8:

```
CALL PXFOPEN(C(1:IPXFLENTRIM(C)),....)
```

Although passing explicit length arguments is available to FORTRAN 77 programmers, the developers of this standard considered this option unacceptable because:

1) It creates a high probability of the creation of unusual (trailing blank) filenames, especially by inexperienced users;
2) It is inaesthetic, difficult to use, and performs poorly (the first choice requires the user to save or recreate the length of each string; the latter requires the string to be scanned for the first nonblank in each call); and
3) It creates a burden on the user to work with an unusual corner case.

— *Explicit length passed*. All procedures that pass character strings require an additional length parameter. Thus, to link 'foo' to 'bar', the following code would be used:

```
C = 'foo'
D = 'bar'
CALL PXFLINK(C,3,D,3)
```

Besides having many of the same problems as the above implicit-length approach, this solution raised the objection that FORTRAN 77 was designed to avoid length passing and that FORTRAN 77 users would rebel at passing string lengths explicitly (as was required in FORTRAN 66).

— *"Global" variable*. A global variable, or context, is set to indicate whether or not trailing blanks are to be ignored, such as the following:

```
C = 'foo∆∏
CALL PXFTRAILINGBLANKSARESIGNIFICANT(.TRUE.)
CALL PXFOPEN(C,...)
```

This option still requires a length to be passed in case trailing blanks are significant, so

```
CALL PXFOPEN(C,4,...)
```

is the calling sequence. This option creates problems similar to those encountered in "explicit length passed".

— *Embedded escape characters*. Blanks are not significant unless preceded by an escape character (e.g., backslash). This requires all strings to be scanned for escape sequences and possibly translated by the system, which was deemed unacceptable. A combination of global variables and embedded escape characters was suggested: Escape characters are only special if the flag is set. This option was similarly rejected as being complex and unwieldy.

— *Remove trailing blanks*. Disallow trailing blanks except in contexts where absolutely necessary [e.g., *PXFREAD*()]. This solution differentiates between two character constructs (abstract data types): strings and character buffers. Each character usage is examined to determine which of these choices is appropriate (in some cases both are appropriate). For those where "string" is chosen, trailing blanks are ignored. It was suggested that all pathnames, login and group names, and the names of environment variables were likely candidates for the "string" category. As previously discussed, this makes such strings with trailing blanks inaccessible from FORTRAN 77. After seeing the drawbacks of the other choices, some developers of this standard thought that this option was relatively acceptable; after all, FORTRAN 77 applications would still be able to access any file that is "creatable" from FORTRAN 77. Besides, FORTRAN 77 potentially is able to create filenames with embedded NULL characters that are inaccessible from C, so there are other instances of definite incompatibilities in this area. However, this approach was eventually discarded also.

From all the above discussions, option (2) was the eventual choice, albeit in a slightly modified form. The modification is that the programmer can specify the length as zero when trailing blanks are to be ignored, such as (using the example from above):

```
CALL PXFLINK(C,0,D,0)
```

would link 'foo' to 'bar' and

```
CALL PXFLINK(C,4,D,0)
```

would link 'foob' to 'bar'. Although this option still puts the burden of the additional arguments on the user, it simplifies the situation and does allow for full functionality. As a side effect, since the most likely programming practice is for the length argument to be zero in all cases where there are no significant blanks, wherever the length argument is not zero highlights the likelihood that the value has a significant railing blank.

A.2.3.2.5 Pointers

The use of the *handle* abstraction to reference aggregate data (i.e., structures) and subroutines caused much debate among the developers of this standard. Some felt that using this abstraction essentially augmented the FORTRAN 77 language, while others countered that because it is only defined abstractly and not as a construct available for general use by programmers that it cannot be considered an extension. In terms of implementation, the topic of memory allocation was related: creating handles entails the allocation of memory dynamically (from the perspective of the program, that is; of course the system implementation could use a static block allocation). Again, the developers of this standard were split on the principle of whether the implicit specification of dynamic memory allocation was out of the scope of this standard. Eventually, consensus was reached that the handle abstraction scheme is the best solution: specifically, it allows full functionality and causes the programmer the least hardship. In addition, it is flexible and easily extensible, thereby allowing the easy integration of system- and site- dependent extensions to this standard.

C-language pointers are used throughout POSIX.1 {2}; however, FORTRAN 77 does not have a pointer data type. Many of the uses of pointers in C such as passing a pointer to a character string are functionally similar to the FORTRAN 77 method of passing by reference. Therefore, no explicit solution had to be devised for this language binding. However, the use of a NULL pointer in C cannot be duplicated in FORTRAN 77 because a NULL pointer cannot be distinguished from a valid pointer in the pass-by-reference FORTRAN 77 model. In cases where POSIX.1 {2} specifies functionality dependent on the use or detection of a NULL pointer, the behavior has been modified slightly in this binding.

Finally, by requiring a valid handle value to be nonzero, this abstraction scheme reserved the value of zero as an indication of a NULL pointer, which was an intuitive equivalence to a NULL pointer in the C language.

A.2.4 Error Numbers

In order to understand the motivation for the error reporting conventions specified by this standard, it is important to understand first the common usage of the *errno* mechanism in the POSIX.1 {2} environment. In the event of an error return from a system call, the programmer checks the current *errno* value against other possible *errno* values (i.e., those listed in POSIX.1 {2} as applicable to that system call) by using the appropriate symbolic constants. The value of *errno* is defined only when an error is returned from a system call.

In order to provide the equivalent functionality in this standard, the developers of this standard considered many alternatives. As with other areas, the main goals were to provide all necessary functionality in a style convenient for the FORTRAN 77 programmer. The following options were considered:

— *Specify an additional function to return the current errno value*. One additional interface was specified, *PXFERRNO*(), which took no arguments and returned the current value of *errno*. The common POSIX.1 {2} programming model could then be mimicked quite closely by FORTRAN 77 by simply putting an inline call to the *PXFERRNO*() function in all places where a C program would directly reference the *errno* variable, and using the *IPXFCONST*() function (see 8.2) in order to do comparisons to other error numbers represented by symbolic constants.
While this option was the accepted solution through several drafts of this standard, it was eventually discarded. A primary reason for its demise was the decision to specify all interfaces as subroutines; without a function return value to indicate success or failure (as in POSIX.1 {2}), the POSIX.1 {2} *errno* model is broken. To provide the basic functionality, the FORTRAN 77 subroutines then had to be specified with an additional argument to indicate success or failure; this additional argument then was easily adapted to provide the functionality of both indicating success or failure and returning the specific error value. Other less significant factors in the decision to abandon the conventional *errno* model included its implication of the existence of an underlying C binding implementation and its uncommon (to FORTRAN 77 programmers) programming model.
— *Specify an additional function to return the current string representation of the (symbolic constant for the) current errno value*. The function would return a string containing, for example, [ENOENT], which could then be used in string compares against the appropriate symbolic constant strings. This option was discarded as being generally undesirable (string manipulations, performance considerations), while not necessarily achieving either of the goals (functionality and usability). Furthermore, the string handling functionality was redundant after the *PXFCONST*() mechanism (see 8.2) was specified.
— *Specify an additional function to compare a passed-in string representation of a symbolic constant to the current errno value*. This option was discarded for reasons similar to those described in the previous item.
— *Use FORTRAN 77 COMMON to access the errno variable*. This option was discarded because, although the value of *errno* can be accessed, there is no comparable (i.e., direct) way to obtain other *errno* values that are stored in the system headers in order to do comparisons. After the *PXFCONST*() mechanism was conceived (see 8.2), the other *errno* values became accessible, but using FORTRAN 77 COMMON was still viewed as being inconvenient for the programmer and inconsistent with the overall language bindings design strategy.

Another consideration was whether or not an additional function should be provided to facilitate setting *errno* values [e.g., *PXFERRNOSET*()] from within FORTRAN 77 programs. Although this functionality is available in C (and often used in library code), no immediate use was found for it in these bindings, and therefore it was not included.

There are a number of functions in POSIX.1 {2} that are defined to be “always successful.” Despite this, the POSIX.9 equivalent procedure for some of these functions includes an argument for these procedures to return an error. In the C-language binding to POSIX.1 {2}, if either a vendor or another standard (e.g. POSIX.6) provides an extension that creates a possibility of an error, because of the errno construct the source code invocation of the C-binding function does not need to be changed. Since errno is not accessible in the FORTRAN 77 bindings, a new procedure with an added error argument would have to be defined to provide such an extension. Thus, FORTRAN 77 source code would

not be portable across systems that did or did not provide for such an error. The developers of this standard felt that an additional argument was not a significant burden to provide this portability and extensibility. Since most FORTRAN 77 binding procedures do have an error argument, also specifying an error argument on these procedures makes them more consistent with the rest of the FORTRAN 77 binding procedures. Finally, just because this error argument exists, an application that was Strictly Conforming to only POSIX.9 would be under no obligation to check for errors when using POSIX.9 procedures that had no POSIX.1 {2} or POSIX.9 errors defined.

However, there was an outstanding objection that this alters the semantics of the binding and unnecessarily burdens the programmer with having to code for errors, where none are possible. It was argued that the usability of these procedures is diminished by the addition of these error return values. These include *PXFALARM*(), *PXFGETUID*() *PXFGETEUId*() *PXFGETGID*(), *PXFGETEGID*(), *PXFGETPID*(), *PXFGETPPID*(), *PXFGETGRP*(), and *PXFUMASK*().

A.2.5 Primitive System Data Types

A.2.6 Environment Description

A.2.7 FORTRAN 77 Language Definitions

Just as the developers of this standard wished to avoid duplicating the POSIX.i {2} definitions, in the spirit of a “thin” binding the FORTRAN 77 definitions are not duplicated.

A.2.8 Numerical Limits

A.2.8.1 FORTRAN 77 Language Limits

There is no additional rationale provided for this subclause.

A.2.8.2 Minimum Values

There is no additional rationale provided for this subclause.

A.2.8.3 Run-Time Increasable Values

There is no additional rationale provided for this subclause.

A.2.9 Symbolic Constants

A.2.9.1 Constants for FORTRAN 77 I/O to STDIO_UNIT Translation

The specification of the constants for mapping FORTRAN 77 unit identifiers to POSIX.1 {2} *stdio* streams was viewed as a standardization of common practice to enhance portability. Early proposals suggested specifying exact values (5,6,0 for *stdin*, *stdout*, *stclerr*), but it was determined that this convention was not widespread enough to justify its standardization. Therefore, the compromise of specifying the range 0–9 was reached; as far as the developers of this standard were able to identify, this range accommodates the vast majority of existing implementations.

Recommended usage (to ensure portability) is therefore to use the defined constants to access the *stdio* streams and to use program-defined unit identifiers outside the specified ranges to avoid conflict with the preconnected units. (See A.8.5.2.2).

A.3 Process Primitives

A.3.1 Process Creation and Execution

A.3.1.1 Process Creation

There is no additional rationale provided for this subclause.

A.3.1.2 Execute a File

Early drafts of this standard contained all of the *exec* entry points that are included in POSIX.1 {2}, but the developers of this standard decided to eliminate the subroutines that use variable-length argument lists. (See A.2.4.0.3.2 for discussion of the decision to eliminate the use of variable-length argument lists.) The decision to eliminate these routines was made easier by the fact that the POSIX.1 {2} *exec* family is redundant; all of the functionality of the discarded functions is still available in the remaining functions. These subroutines were also eliminated because they require the construct of an external global variable (environ), which is a construct not directly available in FORTRAN 77.

The subroutines that were eliminated are: *PXFEXECL*(), *PXFEXECLE*(), and *PXFEXECLP*(), defined as follow:

```
SUBROUTINE PXFEXECL (PATH, ARG0, ARG1, ..., ARGN, PXFNULL() )
CHARACTER*(*) PATH, ARG0, ARG1, ..., ARGN

SUBROUTINE PXFEXECLE (PATH, ARG0, ARG1, ..., ARGN, PXFNULL() ENVP)
CHARACTER*(*)PATH, ARG0, ARG1, ..., ARGN, ENVP(*)

SUBROUTINE PXFEXECLP (FILE, ARG0, ARG1, ..., ARGN, PXFNUL() )
CHARACTER*(*) FILE, ARG0, ARG1, ..., ARGN
```

The POSIX.1 {2} versions of the remaining subroutines require the use of NULL-terminated argument arrays; however, the FORTRAN 77 versions use additional arguments to specify the number of elements in each array.

A.3.2 Process Termination

A.3.2.1 Wait for Process Termination

Due to difficulties discussed in POSIX.1 {2} 3.2, it is not possible to specify a NULL pointer for the *stat_loc* argument.

A.3.2.2 Terminate a Process

The underscore is not in the legal identifier character set in FORTRAN 77 and so is not used in the name. This is the only exception to the naming convention of prefixing *PXF* before the C equivalent.

In early drafts of this standard, the FORTRAN 77 language construct STOP was referenced rather than specifying *PXFFASTEXIT*(). The functionality is similar (i.e., it terminates the process), but STOP does not provide a standard, defined method for returning a status value to the system.

The function *PXFFASTEXIT*() is analogous to the POSIX.1 {2} function *_exit*(). The functionality of *_exit*() is required in order to recover from failed calls to any one of the *PXFEXEC*() subroutines. *PXFEXEC*() executes a new program without creating a new process. A new process is created by calling *PXFFORK*(). *PXFFORK*() creates a new process that is a copy of the current process, including all code and data. Generally, the copy of the code and data of the parent is soon replaced by a new program when the child calls *PXFEXEC*(). However, if the call to *PXFEXEC*() should fail, the child has no way to exit without risking modification of the open files of the parent program. Since the

buffers of the child process are copies of the parents', when *PXFEXIT*(), STOP, or END is executed the data in the buffers will be written to the file and the child will terminate. When the parent writes additional data or closes its files, the files will be updated with the parents' copy of the same buffers; therefore, the data will be duplicated in the files. In order to recover from a failed call to *PXFEXEC*(), the child process must be able to exit without flushing buffers, which is the functionality of *_exit*(). Without the functionality of *_exit*(), large programs will tend to avoid the use of any of the *PXFEXEC*() functions so that data written will not be corrupted, or they will devise some scheme to keep track of all units that are currently open and use *PXFFLUSH*() to dear all the buffers. The functionality of either *PXFEXIT*() is also needed so that a child process can flush its buffers, terminate normally, and return a status value to its parent.

A.3.3 Signals

See POSIX.1 {2} B.3.3 for description of the evolution of the *sigset_t* defined type.

A.3.3.1 Signal Concepts

There is no additional rationale provided for this subclause.

A.3.3.1.1 Signal Names

SIG_IGN and SIG_DFL are possible values of a subroutine handle that cannot match any other subroutine handle. This is also the case in POSIX.1 {2} (see POSIX.1 {2} 3.3.1.1).

A.3.3.1.2 Signal Generation and Delivery

There is no additional rationale provided for this subclause.

A.3.3.1.3 Signal Actions

Because many implementations will choose to implement the *IERROR* return value by building it on top of *errno*, which is inherently unreliable, *IERROR* must also be considered unreliable.

Consider the following hypothetical implementation of *PXFFORK*():

```
void f77fork(long *rtn_value, long *status)
{
  if ( (*rtn_value = fork()) == -1 )
    *status = errno;
  else
    *status = 0;
}
```

Since the return value of the POSIX.1 {2} function *fork*() is reliable, values of zero for *IERROR* are also reliable. However, since errno is not reliable, nonzero values of *IERROR* are not reliable. Given that most interfaces on UNIX systems are C interfaces, this standard did not prohibit POSIX.9 implementations layered on top of C. Requiring nonzero values of *IERROR* to be reliable would require most existing POSIX.1 {2} implementations to rewrite the system interfaces. In order to have this standard implemented in a timely fashion and as widely as possible, this requirement was not made.

A.3.3.1.4 Signal Effects on other Subroutines

For historical reasons, some implementations of errno may be unreliable. Implementations should note that reliability of error reporting may be required by future standards.

A.3.3.2 Send a Signal to a Process

There is no additional rationale provided for this subclause.

A.3.3.3 Manipulate Signal Sets

While *PXFSIGISMEMBER*() could have been defined as a LOGICAL function, POSIX.1 {2} does define the error [EINVAL] for the purpose of testing whether the signal number is valid or supported. Therefore, the construct of subroutine with two OUT arguments (the value and the error) seemed more appropriate.

A.3.3.4 Examine and Change Signal Action

There is no additional rationale provided for this subclause.

A.3.3.5 Examine and Change Blocked Signals

There is no additional rationale provided for this subclause.

A.3.3.6 Examine Pending Signals

There is no additional rationale provided for this subclause.

A.3.3.7 Wait for a Signal

There is no additional rationale provided for this subclause.

A.3.4 Timer Operations

A.3.4.1 Schedule Alarm

There is no additional rationale provided for this subclause.

A.3.4.2 Suspend Process Execution

There is no additional rationale provided for this subclause.

A.3.4.3 Delay Process Execution

There is no additional rationale provided for this subclause.

A.4 Process Environment

A.4.1 Process Identification

A.4.1.1 Get Process and Parent Process IDs

There is no additional rationale provided for this subclause.

A.4.2 User Identification

The existence of an error return argument was considered essential, even though POSIX.1 does not currently define any errors, since security enhancements are likely to provide errors in this topic area.

A.4.2.1 Get Real User, Effective User, Real Group, and Effective Group IDs

There is no additional rationale provided for this subclause.

A.4.2.2 Set User and Group IDs

There is no additional rationale provided for this subclause.

A.4.2.3 Get Supplementary Group IDs

The option of using one argument to combine the argument *ISETSIZE* with the argument *NGROUPS* was considered; however, it was discarded because it made it impossible for a constant to be passed as *ISETSIZE*. The error [EARRAYLEN] could have been defined, but the existing POSIX.1 {2} definition of error [EINVAL] for this function covers this condition.

A.4.2.4 Get User Name

It was necessary to provide arguments to this routine in order to accommodate the decision to eliminate character-string function returns, since two OUT arguments are usually required (the string and the significant length).

A.4.3 Process Groups

A.4.3.1 Get Process Group ID

There is no additional rationale provided for this subclause.

A.4.3.2 Create Session and Set Process Group ID

There is no additional rationale provided for this subclause.

A.4.3.3 Set Process Group ID for Job Control

There is no additional rationale provided for this subclause.

A.4.4 System Identification

A.4.4.1 Get System Name

The data items in the *utsname* structure are null-terminated character arrays in C, so it is the responsibility of the implementation to return them to the FORTRAN 77 caller as character arrays that conform to FORTRAN 77 (i.e., blank-padded, not null-terminated).

A.4.5 Time

A.4.5.1 Get System Time

It is not possible to specify a NULL pointer for the tloc argument. However, since this is a subroutine and not a function and the value is always and only stored in the *ITIME* argument, having a NULL flag to prevent storage is not meaningful.

A.4.5.2 Get Process Times

There is no additional rationale provided for this subclause.

A.4.6 Environment Variables

A.4.6.1 Environment Access

It was necessary to add arguments to this function and split the environment name and value because of the nature of FORTRAN 77 strings and the CHARACTER type. This is an example of the need for a mechanism for significant trailing blanks (see A.2.3.2.4). Both the environment name and the environment value might have a significant trailing blank. While POSIX.1 {2} states that they "should consist solely of characters from the portable filename character set", which does not include blank, it goes on to state that "other characters may be permitted by an implementation; applications shall tolerate the presence of such names." The splitting of the OUT arguments into two (the name and the value) has a side effect of requiring the underlying implementation of POSIX.9 to perform the parsing into those two values rather than the application. This lets the application avoid the issue of a name or a value containing an equals sign. The C-language binding of POSIX.1 {2} affords two mechanisms to access the environment variables. In addition to this function, an application could access the global variable *environ* directly. As FORTRAN 77 has no direct equivalent of a global variable, this procedure is the only FORTRAN 77 mechanism available to access the environment.

POSIX.1 {2} does not contain the functions *setenv()* or *clearenv()*. However, these functions are currently defined in the draft revision to POSIX.1 {2}. In addition, because of the capability of directly accessing the global variable *environ* mentioned above, the C-language binding already permits the capability of setting or clearing the environment even without these explicit routines. The developers of this standard decided that this useful functionality must be defined in POSIX.9, even if only in Section 8, but put it in Section 4 to match where it will be after POSIX.1 {2} is revised. The nature of the C language allows the POSIX.1 {2} interface to return a NULL value when there is no such variable in the environment list and to return a zero-length string when the variable is in the list but has no value assigned. In FORTRAN 77, these two conditions are not easily represented in one return argument. This standard specifies that if the variable is not in the list, the error condition [EINVAL] will be returned in *IERROR*, indicating that the value of the *NAME* argument is invalid. If the variable is in the list but has no value, the VALUE argument will be set to all blanks and the *LENVAL* argument will be set to zero, indicating that the value of the variable named is a zero-length string.

It is common to use shell utilities to create environment variables that have no values. This standard also allows the creation of such variables by calling *PXFSETENV*() with *NEW* set to all blanks and *LENNEW* set to zero.

A.4.7 Terminal Identification

A.4.7.1 Generate Terminal Pathname

The *PXFCTERMID()* subroutine provides the same interface as the POSIX.1 {2} function *ctermid()*, i.e., it returns a string that will refer to the controlling terminal if used with a pathname. The POSIX.1 {2} description provides a C interface with its rules of character array declaration and assignment.

POSIX.9 uses the FORTRAN 77 rules for character declaration and assignment. FORTRAN 77 requires that character entities are declared with an integer constant or an integer constant expression. They may be declared with an asterisk in a subprogram to indicate that the length of the dummy character argument is the length of the actual argument. This does not allow the use of L_ctermid for the length of the character argument. It also does not allow the implementation to store the name completely in the character argument if the name is longer than the declared length of the character argument.

The character argument is declared with a fixed-length that may or may not be large enough to hold the entire name returned by the *PXFCTERMID()* subroutine. FORTRAN 77 rules for assignment are used. That is, the name is truncated if the size of the name is larger than the size of the fixed-length character argument. If the name is less than the fixed length of the character argument, the name is left justified and filled with blanks. The interface provides an extra length argument that returns the size of the name returned by the function. This is helpful when truncation or blank fill occurs or to emulate the C-language construct of an empty string.

A.4.7.2 Determine Terminal Device Name

It was necessary to modify the calling sequence of the *PXFTTYNAME()* function in order to accommodate the decision to eliminate character-string function returns.

A.4.8 Configurable System Variables

POSIX.1 {2} includes access to the special symbol {CLK_TCK}but declares such access to be obsolescent. This standard does not provide such access.

A.4.8.1 Get Configurable System Variables

In a previous revision, it was documented that all of the system variables and constants shown that can be returned by *PXFSYSCONF()* are recognized by the *PXFCONST()* function. This is actually not the case: *PXFSYSCONF()* is for accessing *runtime-variable* system configuration variables. That is, the variable may vary from system to system, even of the same model from the same vendor (e.g., memory available for process). *PXFCONST()* is used only for variables that may differ from one vendor to the other but, once compiled in an application, will not change from one run to the next.

A.5 Files and Directories

A.5.1 Directories

A.5.1.1 Format of Directory Entries

There is no additional rationale provided for this subclause.

A.5.1.2 Directory Operations

Note that, since *IDIRID* could be a file descriptor, the value of zero is not reserved. Thus, an equivalent to the return of a NULL pointer is not defined. However, with the existence of the error [EEND], all cases that would have required a return of NULL produce a nonzero *IERROR* and are therefore identifiable.

A.5.2 Working Directory

A.5.2.1 Change Current Working Directory

There is no additional rationale provided for this subclause.

A.5.2.2 Get Working Directory Pathname

It was necessary to modify the calling sequence of this function in order to accommodate the decision to eliminate character-string function returns.

The *size* argument was eliminated from the calling sequence for *PXFGETCWD()* because it is redundant if it is assumed that the underlying implementation of POSIX.9 has access to the FORTRAN 77 declared length of the CHARACTER argument. (See A.2.3.3.1 further discussion.) The OUT argument *LEN* is explicitly set to zero in the presence of an error to match the explicit return of a NULL in the presence of an error in *getcwd()*.

A.5.3 General File Creation

A.5.3.1 Open a File

There is no additional rationale provided for this subclause.

A.5.3.2 Create a New File or Rewrite an Existing One

In previous revisions, *PXFCREAT()* was not present as it was deemed redundant with *PXFOPEN();* however, the operation performed by *PXFCREAT()* is very common, and the interface to *PXFOPEN()*is awkward enough that *PXFCREAT()* was put back in for usability reasons and to match its (redundant) existence in POSIX.1 {2}. The description was left minimal because it is redundant.

A.5.3.3 Set File Creation Mask

While *umask()* currently is always successful, the *IERROR* argument was included in anticipation of the possibility of returning an error if the process is not permitted to use a particular mask for some defined security reason.

A.5.3.4 Link to a File

The argument names changed from *PATH1* and *PATH2* to *EXISTING* and *NEW* to reflect the corresponding name changes from POSIX.1-1988 to POSIX.1-1990.

A.5.4 Special File Creation

A.5.4.1 Make a Directory

There is no additional rationale provided for this subclause.

A.5.4.2 Make a FIFO Special File

There is no additional rationale provided for this subclause.

A.5.5 File Removal

A.5.5.1 Remove Directory Entries

There is no additional rationale provided for this subclause.

A.5.5.2 Remove a Directory

There is no additional rationale provided for this subclause.

A.5.5.3 Rename a File

There is no additional rationale provided for this subclause.

A.5.6 File Characteristics

A.5.6.1 File Characteristics: Header and Data Structure

There is no additional rationale provided for this subclause.

A.5.6.1.1 File Types

The initial draft of this standard specified that the POSIX.1 {2} macros S_ISDIR*(m)*, S_ISCHR*(m)*, S_ISBLK*(m)*, S_ISREG*(m)*, and S_ISFIFO*(m)* shall all be recognized by the generalized macro usage utility *PXFMACRO()*. This utility was later discarded and the standard updated to reflect the decision to specify a distinct procedure corresponding to each POSIX.1 {2} macro. Because these file-related macros return exclusively true/false results, the FORTRAN 77 LOGICAL type is used to define the return value.

See A.2.3.1.2 for further discussion of decisions related to macros.

A.5.6.2 Get File Status

There is no additional rationale provided for this subclause.

A.5.6.3 Check File Accessibility

There is no additional rationale provided for this subclause.

A.5.6.4 Change File Modes

There is no additional rationale provided for this subclause.

A.5.6.5 Change Owner and Group of a File

There is no additional rationale provided for this subclause.

A.5.6.6 Set File Access and Modification Times

There is no additional rationale provided for this subclause.

A.5.7 Configurable Pathname Variables

A.5.7.1 Get Configurable Pathname Variables

There is no additional rationale provided for this subclause.

A.6 Input and Output Primitives

FORTRAN 77 contains an extensive list of I/O operations. These operations might conflict with use of the system interfaces listed in this section. Interactions of these procedures and FORTRAN 77 I/O are defined in 8.5.5.

No relationship of FORTRAN 77 files to the underlying POSIX operating system can be assumed except in three conditions:

1) The file was successfully opened with *PXFFDOPEN()*,
2) The file was opened for formatted or unformatted sequential access with FORTRAN 77 OPEN while the POSIX I/O flag was one (see 8.5),
3) A file opened by the mechanism of either (1) or (2) was inherited from a parent process.

While a given implementation may provide additional relationships, a Strictly Conforming Application cannot rely on them.

The results of mixing many of the I/O operations defined in this section and Section 8 with FORTRAN 77 I/O operations is implementation defined. For example, if a file descriptor that is associated with a FORTRAN 77 unit identifier is closed, subsequent FORTRAN 77 operations on the unit may cause the program to terminate abnormally.

A.6.1 Pipes

A.6.1.1 Create an Inter-Process Channel

There is no additional rationale provided for this subclause.

A.6.2 File Descriptor Manipulation

A.6.2.1 Duplicate an Open File Descriptor

There is no additional rationale provided for this subclause.

A.6.3 File Descriptor Deassignment

A.6.3.1 Close a File

There is no additional rationale provided for this subclause.

A.6.4 Input and Output

A.6.4.1 Read From a File

The *BUF* argument is specified as an array of characters instead of a CHARACTER*(*) in order to avoid blank filling. It is truly a buffer of characters, *not* a string. The option of using one argument to combine the in *NBYTE* argument with the out *NREAD* argument was considered; however, it was discarded because it made it impossible for a constant to be passed as *NBYTE*. As is typical in FORTRAN 77, specifying *NBYTE* greater than the dimensioned sixed of *BUF* is unsafe, and the results are undefined. C-language programmers should note that *BUF* is a FORTRAN 77 array and is therefore one-based.

A.6.4.2 Write to a File

The option of using one argument to combine the in *NBYTE* argument with the out *NWRITTEN* argument was considered; however, it was discarded because it made it impossible for a constant to be passed as *NBYTE*. (See also the discussion on the *BUF* argument in A.6.4.1).

A.6.5 Control Operations on Files

A.6.5.1 Data Definitions for File Control Operations

There is no additional rationale provided for this subclause.

A.6.5.2 File Control

Although the POSIX.1 {2} (C language) version, *fcntl(),* varies between two and three parameters, this standard requires that the third and fourth arguments always be present. Since these arguments may be either a (integer) handle for an instance of the *flock* structure or a “plain” integer, there is no conflict. Two arguments are required to avoid an IN/OUT argument that would not allow a constant to be used as the IN argument.

It was suggested that the interface for POSIX.1 {2} *fcntl()* was awkward and should not be propagated into POSIX.9. The developers of this standard chose to retain the interface to maintain a better name recognition for users of this new standard to leverage long-existing familiarity with the C-language interface.

A.6.5.3 Reposition Read/Write File Offset

The file offset is defined to be of type *off_t*, which is one of the Primitive System Data Types (see 2.5, and also POSIX.1 {2} 2.5). It is possible that this data type may be defined within the system (or, more specifically, in the C-language bindings) as being an unsigned integer, in which case its range will be greater than that of the FORTRAN 77 INTEGER. See also 2.3.2.2 for more information on handling unsigned quantities.

A.7 Device- and Class-Specific Functions

A.7.1 General Terminal Interface

A.7.1.1 Interface Characteristics

There is no additional rationale provided for this subclause.

A.7.1.2 Parameters That Can Be Set

There is no additional rationale provided for this subclause.

A.7.1.2.1 *termios* Structure

There is no additional rationale provided for this subclause.

A.7.1.2.2 Input Modes

There is no additional rationale provided for this subclause.

A.7.1.2.3 Output Modes

Although there is only one mask defined, the text still applies, since an implementation may support more that one mask for this field.

A.7.1.2.4 Control Modes

The mask *CSIZE* can be used to mask off the baud rate bits for the other control bits. The behavior of *PXFOPEN()* with respect to these control codes is no different than that of *open()*, defined in POSIX.1 {2}.

A.7.1.2.5 Local Modes

There is no additional rationale provided for this subclause.

A.7.1.2.6 Special Control Characters

In POSIX.1 {2} and this standard, the elements of the *c_cc* array are integral values. A FORTRAN 77 programmer could do the following to define the kill character as control-D (CHAR(4)):

```
C   create an instance of the structure
      CALL PXFSTRUCTCREATE('termios', JHANDLE, IERROR)
C   fill the components of the structure
```



```
      CALL PXFTCGETATTR(FILDES, JHANDLE, IERROR)
C   set a single element of the control_character component
      CALL PXFESETINT(JHANDLE, 'c_cc', IPXFCONST('VKILL'), 4, IERROR)
C   now make the change NOW
      CALL PXFTCSETATTR(FILDES, IPXFCONST('TCSANOW'), JHANDLE, IERROR)
```

Implementors of POSIX.9 should note that since the entire 'c_cc' array can be obtained by

```
      CALL PXFAINTGET(JHANDLE, 'c_cc', IAVALUE, IPXFCONST(' NCCS' ), IERROR)
```

and the array *IAVALUE* will be a FORTRAN 77 one-based array, the subscript values to be returned to the FORTRAN 77 application will be one greater than those returned to a C-language application. This is not an issue for the application writer since only the subscript names, not values, should be used. This difference could have been avoided by not providing access to the array as a whole, but the value of being able to store the entire array and then restore to an original condition seemed to outweigh this difference.

A.7.1.2.7 Baud Rate Values

There is no additional rationale provided for this subclause.

A.7.2 General Terminal Interface Control Subroutines

A.7.2.1 Get and Set State

There is no additional rationale provided for this subclause.

A.7.2.2 Line Control Subroutines

There is no additional rationale provided for this subclause.

A.7.2.3 Get Foreground Process Group ID

There is no additional rationale provided for this subclause.

A.7.2.4 Set Foreground Process Group ID

There is no additional rationale provided for this subclause.

A.8 FORTRAN 77 Language Library

A.8.1 FORTRAN 77 Intrinsics

A.8.2 System Symbolic Constant Access

A.8.2.1 Access and Verify Symbolic Constants

The following example illustrates the use of the *PXFCONST()* subroutine and the *IPXFCONST()* function for accessing symbolic constants. It uses the *PXFCHMOD()* system call, which changes the access permissions on a file, using *PXFCONST()* to obtain the mode specifier ('O_RDWR'). The value of the mode is then used in the *PXFCHMOD()* system call to change the access permissions on a file. Following the call, *IPXFCONST()* is used to obtain the unit identifier associated with the preconnected file identified by STDERR_UNIT (see 2.9.1) and, in case of an error, to obtain the values of two *errno* values (possible error conditions) for comparison with the error return from the system call.

```
      PROGRAM TEST
       ...
C   Make the call to PXFCONST()
      CALL PXFCONST('O_RDWR', IMODE, ISTAT)
      IF ( ISTAT .NE. 0 ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 102) 'Could not access constant!'
        ...
      END IF
C   Make the system call to PXFCHMOD().
C   If it fails, check a couple errno conditions.
      CALL PXFCHMOD('/tmp/testfile', 0, IMODE, ISTAT)
      IF ( ISTAT .NE. 0 ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 99) 'Call to PXFCHMOD failed!'
        IF ( ISTAT .EQ. IPXFCONST('ENOENT') ) THEN
          WRITE (IPXFCONST('STDERR_UNIT'), 99) 'errno = ENOENT'
        ELSE IF ( ISTAT .EQ. IPXFCONST('EPERM') ) THEN
          WRITE (IPXFCONST('STDERR_UNIT'), 99) 'errno = EPERM'
        END IF
99     FORMAT(1X,A)
        ...
      END IF
       ...
      END
```

As discussed in A.2.3.1.1, one of the earliest decisions was to specify additional procedures to provide access to the system symbolic constants. The evolution of the constant-access procedures is described in the following paragraphs.

During the early development of this standard, only one function was specified for accessing the constants; it was very similar to the current version of *IPXFCONST*(). The developers of this standard later realized the limitations of that one function, namely its inability to provide an acceptable error-reporting mechanism. More specifically, since the actual integers corresponding to the symbolic names in POSIX.1 {2} are not specified (and the list may grow in the future), it must be assumed that the range of valid constant values is the full range of integers possible on the machine. But the function must also be able to indicate an error if the passed string does not match a known symbolic name. Several methods of reporting/recording this error were considered:

- — Returning -2, noting that no known system constant has this value. The value -1 was not chosen because it is the return value for the system procedures (at this time, all system interfaces were specified as functions, not subroutines).
- — Adding an extra (return) argument to the argument list.
- — Specifying a symbolic name that was guaranteed to report an error.
- — Specifying an additional function that returns an implementation-defined value unique from all defined/valid constant values; this function could be used for comparisons against the value returned from the constant-access function to identify an error.

Each of these options was discarded for various reasons, and the discussions generated here were significant in the development of the current family of constant-access procedures.

The current family of constant-access procedures appeared midway through the development of this standard and was conceived as the following:

- — A function that returns the value of the symbolic constant but provides no error checking *[IPXFCONST()]*. This function is easy to embed in expressions and subroutine calls where the programmer does not wish to utilize any error checking. Also, implementations that provide an intelligent preprocessor may do error checking during preprocessing/compilation. (The leading *I* in the function name was deemed necessary to

avoid problems on implementations with loose type checking and was a nod towards usability; now it need not be declared.)

— A function that verifies that the argument is the name of a symbolic constant *[PXFISCONST()]*. This function provides error checking for the cautious programmer, but also provides a capability somewhat similar to the conditional compilation available in the C language. This capability can prove useful in inquiring about the presence of various features at run-time (including *sysconf* variables) and possibly at compile-time (if the implementation supports an intelligent preprocessor).

— A subroutine that returns the value of the symbolic constant *and* provides error checking *[PXFCONST()]*. This subroutine interface is more awkward but more robust. Note that it essentially combines the functionality of the other two functions, but can both return the constant value and provide error checking (using separate arguments) in only one procedure call.

Although all constants defined in POSIX.1 {2} are integral, specific implementations and/or future standards may require constants of other types. It is recommended that the family of names corresponding to *PXFREALCONST()* and *PXFSTRCONST()* be reserved for use by implementations that require nonintegral typed constants. Specifically, a family of constant-access routines analogous to the current set could be defined, with the appropriate type name *(REALCONST* or *STRCONST)* being substituted for *CONST* in the current procedure names.

A contentious issue was the potential performance degradation resulting from the additional run-time overhead incurred by the additional procedure calls for every constant access. While this may be a reasonable consideration for certain applications on certain systems, it was felt that there was no adequate solution to the problems of accessing the constants that did not involve additional procedure-call overhead. Furthermore, the cost of a library call is generally an order of magnitude *less* than the cost of a system call. Assuming that the constant-access procedures are implemented as library routines, the cost of using them is therefore very small relative to the cost of the associated system call. If performance is a critical issue, an implementation may still choose to implement an intelligent preprocessor that replaces instances of calls to the constant-access procedures with the appropriate constant values (thereby removing the run-time overhead). Of course, such an implementation should also provide error-checking during the preprocessing.

A.8.3 Structure Creation and Manipulation

A.8.3.1 Structure Creation

To reference a given structure type, a FORTRAN 77 string (trailing blanks ignored) containing the structure name, in lowercase, is used, e.g.:

```
      CALL PXFSTRUCTCREATE('utimbuf',JHANDLE,IERROR)
```

A.8.3.2 Structure-Component Manipulation

The following is an example of using one of the structure-component access routines, specifically *PXFINTGET()* with the *PXFSTAT()* system call.

```
      PROGRAM TEST
      INTEGER STHAND, ISTAT, ISIZE, IERROR
C   Allocate an instance if a stat structure
      CALL PXFSTRUCTCREATE( 'stat', STHAND, IERROR)
       ...
C   Make the system call to PXFSTAT()
      CALL PXFSTAT('/etc/passwd', 0, STHAND, ISTAT)
C   Obtain the value stored in the st_size component
      CALL PXFINTGET(STHAND, 'st_size', ISIZE, ISTAT)
      IF ( ISTAT .NE. 0 ) THEN
        WRITE (IPXFCONST('STDERR_UNIT), 102) 'Could not access component!'
```

```
  ...
END IF
WRITE (*,101) 'Number of bytes in file: ', ISIZE
 ...
END
```

Using all of the TYPEs defined in Table 8.3 with the list of procedure names in 8.3.2.1, this standard defines 42 different procedures for structure-component manipulation. However, only some of the procedures are actually used in this standard; the others should be reserved for possible use in future standards and also used for implementation-defined structures and components.

While the data abstraction model for accessing and manipulating aggregate data from FORTRAN 77 was accepted at the earliest stage of development of this standard, the selection and specification of the procedures for component manipulation generated must debate. Several options proposed earlier are discussed in the following paragraphs:

— Specify component-manipulation procedures on a per-system-interface basis; i.e., for each system interface procedure, the necessary component-manipulation procedures are specified. This method was used in several early drafts of this standard and was specified as follows:

```
FUNCTION PXF<SYS_ROUTINE_NAME>GET (MEMBER_NAME_VALUE)
CHARACTER*(*) MEMBER_NAME
TYPE VALUE
FUNCTION PXF<SYS_ROUTINE_NAME>SET (MEMBER_NAME_VALUE)
CHARACTER*(*) MEMBER_NAME
TYPE VALUE
```

where TYPE varies according to the *MEMBER_NAME*.

The primary weakness was that this method required procedures with actual arguments that could differ in type from call to call. This was determined to be a deviation from the FORTRAN 77 standard (see A.2.3.0.4.3) and led to the evolution of the current set of procedures (which are specified on a *per-type* basis rather than per-function). Another drawback is that this method is not easily extensible without additional potentially complex specifications, for example, situations where one system interface utilizes multiple instances of the same structure or different structures with identical member names.

— Specify a single structure-access procedure that takes three arguments: the name (type) of the structure, the name of the desired field, and the value to be loaded (or the variable to be returned). This is a more generalized solution, but still suffers from the variable-type arguments problem mentioned above. Additionally, the performance of an implementation of this method might be quite poor due to the multiple string lookups required for every invocation.

— Specify one structure-access procedure *per structure* that requires (simultaneously) arguments representing all members of the structure. This method presents the advantage that all members of a structure may be loaded or extracted with only one procedure call, but suffers severely in terms of extensibility; if a member is added to the structure (either by another standard or in a particular implementation), the access procedure argument list would be inappropriate.

Performance implications were frequently discussed, both with respect to the general model of requiring a procedure call for every structure-component access, and also with respect to the implementation of the various access models discussed previously. See A.2.3.2.3 for a discussion of the general procedure-call overhead issue. Regarding implementation, it is interesting to note that a variation of the model finally accepted was considered much earlier but discarded largely because of performance concerns. However, the introduction of the *handle* mechanism encourages much more efficient implementation than the earlier variation (which required parsing multiple strings in each call).

A.8.3.3 Structure Deletion

When finished with an instance of a structure, the structure should be deleted to return resources to the system. For example:


```
       ...
      CALL PXFSTAT('/etc/passwd', O, STHAND, ISTAT)
       ...
      CALL PXFINTGET(STHAND, 'st_size', ISIZE, ISTAT)
       ...

C   Delete the stat structure when done with it.
      CALL PXFSTRUCTFREE(STHAND, ISTAT)
      IF ( ISTAT .NE. O ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 102) 'Stat structure handle not
deleted!'
       ...
      END IF
      WRITE (*,101) 'Number of bytes in file: ', ISIZE
       ...
```

A.8.3.4 Structure Copy

It is sometimes useful to keep several instances of a structure. The *PXFSTRUCTCOPY()* subroutine can be used to maintain identical or similar instances. In the following example, the behavior of the terminal driver is temporarily modified (see 7.2.1 for information on the terminal interface control subroutines used here).

```
       ...
      INTEGER OLDTS, NEWTS, ISTAT, NOECHO, CLFLAG, IFD
       ...
      CALL PXFSTRUCTCREATE('termios', OLDTS, ISTAT)
       ...
      CALL PXFSTRUCTCREATE('termios', NEWTS, ISTAT)
       ...
      CALL PXFTGETATTR(IFD, OLDTS, ISTAT)
       ...

C   Copy the contents to the current terminal settings to the new
C   structure and modify the contents slightly, thus changing only
C   one terminal characteristic.
      CALL PXFSTRUCTCOPY(OLDTS, NEWTS, ISTAT)
      IF ( ISTAT .NE. O ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 10)
    +      'Error copying termios structure'
        STOP
      END IF

C   Disable terminal echo.
      NOECHO = NOT(IPXFCONST('ECHO'))
      CALL PXFINTGET(OLDTS, 'c_lflag', CLFLAG, ISTAT)
      CLFLAG = IAND(NOECHO, CLFLAG)
      CALL PXFINTSET(NEWTS, 'c_lflag', CLFLAG, ISTAT)
      CALL PXFTCSETATTR(IFD, IPXFCONST('TCSANOW'), NEWTS, ISTAT)
       ...

C   When it's time to exit restore the 'old' terminal driver settings.
      CALL PXFTCSETATTR(IFD, IPXFCONST('TCSANOW'), OLDTS, ISTAT)
       ...
```

A.8.4 Subroutine-Handle Manipulation

A.8.4.1 Save and Reference Subroutine Handle

Without these subroutines, there is no way to obtain the value of an element of a structure that is a pointer to a function and subsequently call that function. This is a requirement for the *sigaction* structure.

The text of the standard states that *PXFCALLSUBHANDLE()* is called with (and, in 3.3.4, that the *sa_handler* component of the *JSIGACT* structure shall be) a subroutine handle obtained from a previous call to *PXFGETSUBHANDLE()* or *PXFSIGACTION()*. Implementors should note that, if an implementation for subroutine handles is other than a pointer to a function, process initialization code (e.g. the “MAIN” code that calls the FORTRAN program) or calls to the kernel *sigaction()* functionality from another language may cause confusion between the handler representation in internal kernel tables and the representation that is manipulated by the application. These differences must be appropriately translated by the bindings implementation.

The following program uses *PXFGETSUBHANDLE()* to set up a Control-C trap.

```
      PROGRAM TEST

      INTEGER ISTAT, HANDLE, SIGACT
      CHARACTER*80 ALINE

      EXTERNAL CTRLCH

      CALL PXFSTRUCTCREATE(SIGACT, 'sigaction', ISTAT)
      IF ( ISTAT .NE. 0 ) STOP

C   Get the handle for the subroutine CTRLCH.

      CALL PXFGETSUBHANDLE(CTRLCH, HANDLE, ISTAT)
      IF ( ISTAT .NE. 0 ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 10)
      +        'Error getting handle for subroutine CTRLCH'
        STOP
      END IF

C   Test the handle by calling it once.

      CALL PXFCALLSUBHANDLE(HANDLE, 0, ISTAT)
      IF ( ISTAT .NE. 0 ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 10)
    +        'Error calling handle for subroutine CTRLCH'
        STOP
      END IF

C   Now pass the handle to the system to use as a Control-C handler.

      CALL PXFINTSET(SIGACT, 'sa_handler', HANDLE, ISTAT)
      CALL PXFINTSET(SIGACT, 'sa_mask', 0, ISTAT)
      CALL PXFINTSET(SIGACT, 'sa_flags', 0, ISTAT)

      CALL PXFSIGACTION(IPXFCONST('SIGINT'), SIGACT, 0, ISTAT)
      IF ( ISTAT .NE. 0 ) STOP
```



```
C   Block on the controlling terminal just to test the interrupt.
      READ (*,10) ALINE
10      FORMAT (A)
      END

C   The handler subroutine.
      SUBROUTINE CTRLCH(ARG)
      INTEGER ARG
      WRITE (*,10) 'Control C was pressed'
      RETURN
      END
```

A.8.5 External Unit and File Descriptor Interaction

There are three different ways of referring to external files in POSIX.9. They can be referred to by an external unit identifier, by a file descriptor, or by both an external unit identifier and a file descriptor. This section attempts to clarify the difference between a unit identifier and a file descriptor.

A unit identifier is provided in FORTRAN 77 to refer to a FORTRAN 77 file. It is used in READ, WRITE, and other I/O statements to perform operations on files. There is a direct correlation between a specific unit and a specific file.

A file descriptor is provided by the POSIX system to refer to a file. All open files on a POSIX system have one or more associated file descriptors. For each open file, the POSIX system keeps a file description. The file description is used by the system to access the file. It tells the POSIX system the position of the file pointer, in addition to other important file attributes. Each process has its own file table. The file table contains pointers to file descriptions. The file descriptor is an index into the file table. When a process is created, it receives a copy of the file table of its parent; hence, it receives the pointers to the descriptions for all of the open files of the parent. File table entries may be manipulated by using *PXFFCNTRL()*. In summary, a file descriptor is kept by a process and is an integer value that is associated with a file description that is kept by the system.

POSIX.9 defines some of the interactions of units and file descriptors and provides interfaces to manipulate file descriptors and units.

A FORTRAN 77 file can be opened with a unit. The FORTRAN 77 OPEN statement can be used to open either a non-POSIX FORTRAN 77 file or a POSIX-based FORTRAN 77 file. A subroutine *PXFPOSIXIO()* (see 8.5.1.1) is provided to determine the current setting of the global POSIX I/O flag and to change it to the required setting. If the value of the flag is zero, then the file created is not required to be accessed as if it contained newline delimited records and the unit is not required to be connected to a file descriptor. If the flag is set to one, the unit will be connected to a file descriptor and formatted files will be accessed as newline delimited records.

A FORTRAN 77 file can be opened with both a unit and a file descriptor. POSIX.9 provides a call to *PXFFDOPEN()* (see 8.5.3) to connect an external unit to a file descriptor. Both the unit and the file descriptor are supplied as input arguments to the subroutine. The NEWLINE=YES/NO string in the *ACCESS* string argument indicates whether the file will be accessed as if it contains newline delimited records. If the value is YES or the string is omitted, the file will be accessed as if it contained newline delimited records. If the value is NO, the file is not required to be accessed as if it contained newline delimited records. Most POSIX.2 utilities will not execute correctly on files without newline delimited records.

Once the connection between the unit and the file descriptor has been established, that association may be verified by calling *PXFFILENO()* (see 8.5.2.1) to return the file descriptor to which the unit is connected.

The procedures *PXFFDOPEN()* and *PXFFILENO()* provide access to file descriptors and allow the use of FORTRAN 77 I/O on I/O channels that have no file name. Such channels are created by *PXFPIPE()* and *PXFFORK()*. This allows one to use FORTRAN 77 READ and WRITE to communicate on pipes or inherited file descriptors.

In general, text files (see POSIX.2 {A2}) should be written using POSIX-based FORTRAN 77 I/O in order to assure interoperability with other POSIX programs and utilities. One may choose not to use POSIX-based FORTRAN 77 I/O in order to take advantage of implementation-defined features or performance options not defined by this standard. POSIX-based FORTRAN I/O applies also to unformatted sequential access files, thereby allowing unformatted FORTRAN I/O across interprocess communication channels. In addition, such files are inherited by a child process after a call to *PXFFORK*().

The ability to perform system level I/O using *PXFLSEEK*(), *PXFREAD*(), and *PXFWRITE*() in addition to FORTRAN 77 I/O to the same open file is intentionally left undefined by this standard. If byte access is required on a connected unit, the procedures *PXFFFSEEK*(), *PXFFGETC*(), and *PXFFPUTC*() should be used. Note that it is a requirement for strictly conforming applications to insure that system level I/O and FORTRAN level I/O is not performed on the same file. Note that this restriction is limited to system level I/O subroutines that can affect the file offset, namely, *PXFLSEEK*(), *PXFREAD*(), and *PXFWRITE* therefore, *PXFFCNTRL*() and *PXFFSTAT*() may be used to determine or set file attributes such as protections and locking.

Finally, a POSIX.9 file can be opened without involving a FORTRAN 77 unit by invoking *PXFOPEN()* (see 5.3.1.2). It provides the same functionality as the POSIX.1 {2} function *open*(). In this case, a file descriptor is used to access the file. Because of the possibility of breaking existing FORTRAN applications, this standard does not specify default setting of the POSIXIO flag. Therefore, the subroutine *PXFPOSIXIO*() should be called to set the POSIX I/O flag to a value of one, if any of the following POSIX behaviors are required:

1) Interoperability with POSIX system utilities (defined in the forthcoming POSIX.2 standard — e.g., *grep, cat, sort*);
2) Ability to perform FORTRAN formatted or unformatted sequential I/O over pipes or fifos;
3) Ability to inherit FORTRAN files open for formatted or unformatted sequential access. It is intended that the FORTRAN binding to POSIX.2 shall specify that, for the compiler “fort77,” the default setting of the *POSIXIO* flag shall be one.

Some questions that have been asked about the interaction of external unit identifiers and file descriptors are:

> Can unformatted sequential or direct access I/O be done to a file opened using *PXFFDOPEN*()?
> Files can be opened for unformatted sequential access by *PXFFDOPEN*(), but not for direct access I/O.
> Can *PXFFDOPEN()* be used in conjunction with *PXFFILENO*() to have two different unit numbers connected to the same file use the same file with the same file descriptor?
> Yes this can occur. It is up to the application or applications to coordinate usage of two or more units connected to the same file (see 8.5.5).
> Can section 6.4 and 6.5 procedures be applied to file descriptors obtained with *PXFFILENO*()? If so, what is the interaction?
> For sequential access files, the procedures in 6.4 and 6.5 may be applied to file descriptors obtained with *PXFFILENO*(), as specified in 8.5.5. For direct access files, the following operations are not defined: *PXFLSEEK*(), *PXFREAD*(), and *PXFWRITE*().

A.8.5.1 POSIX-Based FORTRAN I/O

Since FORTRAN 77 does not define record control information, it is possible for a FORTRAN 77 program to create text files that can not be used by other POSIX utilities and would not be portable to other POSIX systems. This subroutine allows the user to specify a POSIX-compatible record structure. This is intended to insure that FORTRAN programs could interoperate with other POSIX programs and utilities (defined in the forthcoming POSIX.2 standard — e.g., *grep, cat*). In addition, the application programmer must know the record structure in order to use *PXFFSEEK*() and the other stream I/O subroutines effectively.

During balloting of this standard, one alternative that was discussed was to permit the *PXFPOSIXIO*() and *PXFFDOPEN*() procedures to return the error [ENOSYS]. This subtle change would permit the actual implementation of POSIX side effects on existing FORTRAN 77 language I/O statements to be optional, i.e., an implementation could

always return [ENOSYS] from any call to either procedure. While an application could portably test whether such side effects existed, no application could portably rely on such side effects. If an application required POSIX side effects, the application would be required to use the POSIX I/O procedures for all I/O It was determined that this optionality would reduce consensus; thus, this error was not defined for these procedures.

There was concern that FORTRAN 77 files might not be byte oriented. By specifying byte access routines on formatted files that are open for POSIX-based FORTRAN I/O, this standard is requiring that such files be byte oriented.

It was noted that on some implementations the POSIX I/O system and a record manager may coexist. On such systems, certain FORTRAN 77 applications may perform better if the record manager is used. By providing a subroutine that selects POSIX-based FORTRAN I/O, an implementation may provide access to the record manager, if one exists.

Since this standard does not define the default record structure, all applications concerned with data portability, data interoperability, and stream I/O access either should set the POSIX I/O flag to one before performing any FORTRAN 77 I/O operation or should only use the POSIX.9 file descriptor I/O primitives. Since all other record structures are implementation defined, the setting of the POSIX I/O flag to zero cannot be relied upon to give any portable result.

The POSIX I/O flag was changed from a logical two-state variable to a multistate integer variable in order to accomodate a request for more flexibility within the flag, e.g., to specify stream I/O separately from POSIX records. Values other than true or false are now available to implementations to allow extensions.

Therefore, with an appropriate setting of the POSIX I/O flag POSIX.9 conforming (but not strictly conforming) applications can create and process files for any number of implementation-defined record structures. With the introduction of more values than two, an error return was added to indicate when an attempt was made to set a value other than the two defined by POSIX.1 {2} (0 and 1), but that value was not defined on this implementation. A POSIX.9 conforming implementation must define zero and one since zero is an unspecified default and one is the state required for POSIX.9 strictly conforming portability. Note that an implementation that only provides POSIX I/O has the trivial case where zero means the same as one, but both are still defined. Adding the error return also allows other errors associated with other values for *NEW*. While the current value of the POSIX I/O flag may be considered a global flag, the setting for a given file is a property of each specific connection since the property for that connection does not change if or when the flag changes. The flag could be explicitly set or changed prior to each specific open but, once open, the property of that file remains unchanged. Making a global flag simply eliminates the need to define it for each open when a series are desired to have the same property.

The FORTRAN 77 standard {3} (Section 12.9.5.2.3) discusses the action called “printing,” which might be done to a formatted record. While POSIX-based FORTRAN I/O files are defined to contain formatted records, this document does not specify whether any action (such as directing a record to a specific file like STDOUT) constitutes FORTRAN 77 “printing” in a POSIX-based environment. Those implementations surveyed have no “automatic” POSIX-based actions that constitute “printing.” Some implementations have a utility program (*asa*) that converts a file containing newline-terminated records to a new set of newline-terminated records with the first character of each record removed and containing appropriate additional newlines, carriage-returns, formfeeds, or other ASCII codes. The conversion performed by this utility is defined as equivalent to the FORTRAN 77 action of “printing.” If some action on an implementation constitutes “printing” as defined by FORTRAN 77, it is expected that such an implementation will document such action. It is expected that future standards developers will deal with utilities when the Fortran binding to POSIX.2 is addressed. The developing draft of POSIX.2 already includes the *asa* utility.

A.8.5.2 Map a Unit to a File Descriptor

This subroutine returns the file descriptor associated with a connected unit. Initially it was required that all files opened with the FORTRAN 77 OPEN statement must have an associated file descriptor. While extensions to the FORTRAN 77 OPEN statement would have been permitted to bypass POSIX I/O in order to access implementation-defined I/O systems for improved performance or functionality, there were objections to requiring implementations to build

FORTRAN 77 I/O on top of POSIX I/O The compromise position reached was to require only POSIX-based FORTRAN I/O (see 8.5) to have an associated file descriptor.

A child process will inherit units connected to file descriptors. If a child attempts to use a unit connected by its parent that is not connected to a file descriptor, the results are unspecified.

The units obtained through STDIN_UNIT and STDOUT_UNIT are preconnected input and output files, respectively. This standard does not specify that these preconnected units are the same as the processor-determined external units specified by asterisk on READ or WRITE statements. FORTRAN 77 does not require that the actual unit numbers for these processor-determined external units be retrievable by a Fortran application program. Therefore, they need not be valid unit numbers. Some current implementations of FORTRAN 77 place these units outside the range of units available to a Fortran application program so the program can access more units with fewer restrictions on the number of units available.

A.8.5.3 Open a Unit

The developers of this standard considered several alternative methods to achieve this functionality. Altering the FORTRAN 77 OPEN statement was ruled out as being beyond the scope of the standard. Also discussed was providing a mapping routine that associated a connected unit to a different file descriptor. This would require that a file be opened to obtain the connected unit first, and then the file would be have to be closed (and possibly deleted) by the mapping routine. It would also create a problem with some of the FORTRAN 77 OPEN keywords, especially *STATUS*, *IOSTAT* and *ERR*. *STATUS* and *ERR* could be defined to have no meaning during the mapping call, but access to *STATUS* would be required. In addition, the OPEN keywords have been extended by many implementations. By defining a subroutine and the keywords that will connect a unit to a file descriptor, these performance and keyword problems were eliminated. For example, a program may create a file that contains checkpoint information, and the parent and child processes may then both write into the file (see 8.5). Such behavior is illustrated in the following example:

```
      PROGRAM PARENT

      CHARACTER*10 ARGS():1),ARG1
      INTEGER LENARGS():1)

C   Be sure the unit is connected to a file descriptor by setting
C   the POSIXIO flag to 1

      CALL PXFPOSIXIO(1, IOLD, IERROR)
      ARGS())(1:8) = 'childpgm'
      LENARGS()) = 8

      OPEN(UNIT=11, FILE='pgm.log', ACCESS='SEQUENTIAL',
    +   STATUS='NEW', FORM='FOMATTED')

C   Get the file descriptor associated with the unit to pass to
C   the child program

      CALL PXFFILENO(11, IFD, IERROR)
      IF ( IERROR .NE. 0 ) STOP 'Error getting file descriptor'
      WRITE(UNIT=ARGS(1), FMT=10) IFD
10     FORMAT(15)
      LENARGS(1) = 5

C   Now create a new process and exec a new image

      CALL PXFFFLUSH(11, IERROR)
```



```
      CALL PXFFORK(IPID, IERROR)
      IF ( IERROR .NE. 0 ) STOP

      IF ( IPID .EQ. O ) THEN
        CALL PXFEXECV('./childpgm', 10, ARGS, LENARGS, 2, IERROR)
        CALL PXFFASTEXIT(-1)
      END IF

C   The parent may do other work or wait...

      CLOSE(11)
      END

      PROGRAM CHILD
      CHARACTER ARGFMT*5, ARG1*100

C   The child program reads its argument list and then connects to
C   the file descriptor passed as the first argument...

      IF (IPXFARGC() .NE. 1 ) CALL PXFEXIT(-1)
      CALL PXFGETARG(1, ARG1, LENARG1, IERROR)

      READ(UNIT=ARGFMT, FMT=10) LENARG1
10     FORMAT('(I',15,')')
      READ(UNIT=ARG1, FMT=ARGFMT) IFD

      CALL PXFFDOPEN(IFD, 14, 'STATUS=OLD, POSIXIO=YES', IERROR)
      IF (IERROR .NE. 0) CALL PXFEXIT(-1)

C   Now the child can write to the file...
      WRITE(14,20) 'Child complete.'
20     FORMAT (A)
      CLOSE(14)
      CALL PXFEXIT(O)
      END
```

During balloting of this standard, one alternative that was discussed was to permit the implementation of the functionality of *PXFFDOPEN()* to be optional (see A.8.5.1).

A.8.5.4 Flush Output

If FORTRAN I/O is to be resumed by a child process on a file opened by the parent using the OPEN statement, *PXFFFLUSH*() must be called before the child is created in order to flush the I/O buffers of the parent. In the example below, a child process is created that writes to a unit that the parent may have been using. The parent waits for the child to complete and then may resume writing to the file. The results of this program would be unpredictable if the parent did not flush the buffers before creating the child. When the child closes its connection to the file, the connection made by the parent to the file remains. Notice that the child performs FORTRAN 77 I/O on the file by directly using the external unit without need to identify or use file descriptors. This is permitted since *PXFFORK*() will duplicate the parents' connection to the file, and the connected file descriptor will be inherited. Once a call to one of the *PXFEXEC*() subroutines is made, the connected unit is destroyed, but the file descriptor is preserved (unless the FD_CLOEXEC flag is set on the file). Therefore, after a call to one of the *PXFEXEC*() subroutines is made, *PXFFDOPEN*() must be called to establish the connection of a unit to the inherited file descriptor.

```
      PROGRAM SHARE
```

```
      CALL PXFPOSIXIO(1, IOLD, IERROR)

      OPEN (UNIT=11, FILE='share.me', ACCESS=' SEQUENTIAL',
     +  STATUS=' NEW', FORM='FORMATTED')

      WRITE(11,10) 'THIS IS THE PARENT TALKING'
      CALL PXFFFLUSH(11, IERROR)
      IF ( IERROR .NE. 0 ) STOP 'Error flushing output!'
      CALL PXFFORK(IPID, IERROR)
      IF ( IERROR .NE. 0 ) STOP 'Error during fork!'
      IF ( IPID .EQ. 0 ) THEN
        WRITE(11,10) 'THIS IS THE CHILD TALKING'
        CLOSE(11)
        CALL PXFEXIT(0)
      ELSE
        CALL PXFWAIT(ISTAT, IPID, IERROR)
      END IF
      WRITE(11,10) 'THIS IS THE PARENT SAYING GOOD-BYE'
      CLOSE(11)
10       FORMAT (A)
      END
```

A.8.5.5 FORTRAN Language Input/Output Statements

All of the interactions defined by this standard only apply to POSIX-based FORTRAN I/O files. As much as possible, these specifications both reflect what is the intuitive relationship of the FORTRAN 77 construct and an underlying POSIX system, as well as reflect a number of current implementations.

A.8.5.5.1 Interactions of FORTRAN I/O Statements

This standard does not define the operations of *PXFREAD()*, *PXFWRITE*(), or *PXFLSEEK*() on file descriptors that are connected to direct access files. This allows implementations to provide special optimizations while allowing *PXFSTAT*() and *PXFFCNTL*() to be used on a file.

POSIX.9 defines two methods that can result in the same file being connected to two different unit. After a *PXFFORK*(), the I/O buffers of the parent will be duplicated in the child. If any of those buffers contain unwritten data, there is the danger of duplicating that data in the file. The duplication of data may be avoided by flushing the data before performing *PXFFORK*() or by performing *PXFEXEC*() immediately after the *PXFFORK()*. If *PXFFORK*() should fail, *PXFFASTEXIT*() may be used to terminate the process without writing the buffered data.

A.8.5.5.2 Interactions With FORTRAN 77 OPEN Statement

The setting of the mode to

```
      IOR(IPXFCONST('S_IRUSR'),IOR(IPXFCONST('S_IWUSR'),
     + IOR(IPXFCONST('S_IRGRP'),IOR(IPXFCONST ('S_IWGRP'),
     + IOR (IPXFCONST('S_IROTH'),IPXFCONST('S_IWOTH'))))))
```

ensures that the umask value of the user will be used to determine the file permissions of newly created files.

A.8.5.5.3 Interactions With FORTRAN 77 INQUIRE Statement

POSIX.1 {2} does not define a reliable method of determining the absolute pathname of a file. Each open must do a "get working directory" call to try to get this at the time of the open.

A.8.5.5.4 Interactions With FORTRAN 77 CLOSE Statement

There is no additional rationale provided for this subclause.

A.8.5.5.5 Interactions With FORTRAN 77 READ Statement

For all reading and writing on a unit, the read or write must not fail due to an interrupt. This is not to say that an underlying POSIX read cannot fail due to an interrupt.

When reading the terminal device file associated with the controlling terminal, the behavior of READ statements may be altered by setting the mode of the controlling terminal. An example arises in user terminal input: Since POSIX-based FORTRAN I/0 records are terminated by the newline character, it is important not to put the terminal in a mode that will suppress the transmission of newline. For example, if the terminal is in noncanonical mode (see 7.1) and INLCR is set (map NL to CR), then there is no way for the application to receive a newline from the controlling terminal. A READ statement at this point may cause the application to hang. Although setting the controlling terminal to noncanonical mode with INLCR not set will allow the newline to be sent, on most keyboards the only way to send the newline is by pressing control-J. Applications should set ICRNL (map CR to NL) whenever noncanonical mode is entered. Also, take care that IGNCR (ignore CR) is not set. This allows the read operation to complete when the carriage return key is pressed at the keyboard of the controlling terminal.

A.8.5.5.6 Interactions With FORTRAN 77 WRITE Statement

There is no additional rationale provided for this subclause.

A.8.5.5.7 Interactions With FORTRAN 77 BACKSPACE and REWIND Statements

A question was asked about the following program:

```
      CHARACTER ONELINE*7, CHAR

      CALL POSIXIO(1, DUMMY, IERROR)
      OPEN (UNIT=14, FILE='TEMP.TXT', STATUS='NEW')
      WRITE (14,10) 'ABCDEF'
      WRITE (14,10) 'JKLMNO'
      WRITE (14,10) 'STUVWX'
      CLOSE (14)
      OPEN (UNIT=14, FILE='TEMP.TXT', STATUS='OLD')
      READ (14,10) ONELINE
      CALL PXFFGETC (14, CHAR)
      BACKSPACE(14)
      READ (14,10) ONELINE
      WRITE (*,10) ONELINE
10     FORMAT (A)
      END
```

Which line will be written? The BACKSPACE will move the file position to the beginning of the preceding record. Section 8.5.5.1 defines what the preceding record will be after the call to *PXFFGETC();* therefore, the answer is:

JKLMNO with *CHAR* containing the letter J. This is also the intuitive answer.

A.8.5.5.8 Interactions With FORTRAN 77 ENDFILE Statement

There is no additional rationale provided for this subclause.

A.8.6 Stream I/O

The subroutines defined in this section are based on common-practice extensions to many FORTRAN 77 compilers and libraries available on UNIX-based systems today. The specifications here match as closely as possible those in common usage. However, the syntax has been changed so that these subroutines are consistent with the rest of the binding (i.e., names prefixed with *PXF*). Therefore, there is little likelihood of conflict between these subroutines and the common vendor extensions. These subroutines provide functionality that is not available with the I/O facilities of FORTRAN 77 (i.e., the ability to access a file a byte at a time); such functionality has many applications in a POSIX environment (e.g., screen prompting, building of filter programs). These subroutines provide functionality that is not provided by Fortran 90 since the Fortran 90 {A1} language standard can only provide access to bytes within a record whereas these procedures can access bytes outside of a FORTRAN record. Such functionality is not possible to specify in a language standard that may be implemented on a wide variety of operating systems. It is only possible when the scope is restricted to a specific operating system, such as the scope of this standard.

Mixing stream I/O and FORTRAN 77 record I/O was a concern. The model of mixing chosen was taken from existing implementations. The model specifies that calls to *PXFGETC()*, *PXFFGETC()*, *PXFPUTC()*, *PXFFPUTC()*, and/or *PXFFSEEK()* may be intermixed with FORTRAN 77 READ and/or WRITE statements to the same connected unit. There was much concern that this would not be portable and would be difficult for some architectures to implement. There was also concern that the existing implementations were in conflict with the FORTRAN 77 standard. As a result, the developers of this standard sought an official interpretation and guidance from the ANSI FORTRAN committee (X3J3), with the feedback indicating that this model of mixing stream I/O with FORTRAN 77 I/O was not in conflict with the FORTRAN 77 standard.

The developers of this standard also asked for guidance from X3J3 on the issue of mixing PXFGETC/PXFPUTC and READ/WRITE on the same formatted sequential file during a single OPEN of a device. X3J3 indicated that it did not take a vote on the issue, but that a survey of FORTRAN 77 implementors who were present at the meeting indicated that several of them provide this feature. It is generally disliked because its behavior can be erratic. Generally, X3J3 expressed no support for mixing these I/0 methods to the same file. At a later meeting of X3J3, a survey taken of ten implementors indicated that four would approve of the mixture, four would disapprove, and two abstained. Most developers of this standard felt that the feature would benefit the users of this standard.

In order to ensure that these routines could be used portably, the operation of the stream I/O subroutines was limited to formatted sequential files that are open for POSIX-based FORTRAN I/O (see 8.5). The behavior of the stream I/0 subroutines is undefined for files that are not opened for POSIX-based FORTRAN I/O. This restriction effectively limits the use of these subroutines to formatted sequential files.

When a call to one of the stream I/0 subroutines is followed by a FORTRAN 77 I/O statement, the record accessible to the FORTRAN 77 statement begins with the byte following the byte processed by the stream I/0 subroutine (see 8.6.3.2). Conversely, if a stream I/O call is made following a FORTRAN 77 I/O statement, the byte processed would be the next byte after the record processed by the FORTRAN 77 statement. The following program and sample output illustrates the mixing behavior defined in 8.5.5.

```
      PROGRAM IO

      CHARACTER CH*1, STRING*20

C   Set the POSIXIO flag to 1 so that mixing can occur predictably.

      CALL PXFPOSIXIO(1, IOLD, IERROR)
```



```
C   Write to standard out using both kinds of I/O

      PRINT 10, 'Hello,'
      CALL PXFPUTC(' ', IERROR)
      CALL PXFPUTC('W', IERROR)
      CALL PXFPUTC('o', IERROR)
      PRINT 10, 'rld!'
10     FORMAT (A)

C   Read from a file using both kinds of I/O.

      OPEN(UNIT=14, FILE='temp.txt', STATUS='OLD')
      DO 80 I=1, 5
        CALL PXFFGETC(14, CH, IERROR)
        CALL PXFPUTC(CH, IERROR)
80     CONTINUE

C   Mark the transition from Stream I/O to FORTRAN I/O with an X.

      CALL PXFPUTC('X', IERROR)
      READ(14, 10) STRING
      PRINT 10, STRING

C   Go back to the beginning of the file and do partial reads.
C   Notice that FORTRAN 77 always reads a complete record.

      CALL PXFSEEK(14, O, IPXFCONST('FSEEK_BEGIN'), IERROR)
      READ(14,30) STRING
30     FORMAT (A3)
      PRINT 10, STRING

      DO 90 I=1, 4
        CALL PXFFGETC(14, CH, IERROR)
        CALL PXFPUTC(CH, IERROR)
90     CONTINUE

C   Read the end of record character. This is something you
C   cannot do with FORTRAN 77 or Fortran 90 as it now stands.

      CALL PXFFGETC(14, CH, IERROR)
      PRINT 40, ICHAR(CH)
40     FORMAT (14)

      END
      If the file temp.txt contains the data:
      Line 1
      Text
      last

      The output from the program will be the following:
      Hello,
       World!
      Line X1
      Lin
```

```
      Text 10
```

In the example above, the character 10 is the newline character, which is the end of record. However, newline-delimited can only be assumed to be the definition of the record structures for POSIX-based FORTRAN 77 I/O files (see 8.5). Therefore, as stated in 8.6, portable use of the stream I/O procedures can only be assured when used with such files.

A.8.6.1 Modify a File Position

The *PXFFSEEK()* subroutine may be used on formatted POSIX-based FORTRAN I/O files (see 8.5). While the IUNIT argument "shall refer to an open unit," this might not be the case, hence the error value. Further, it is intentionally left unspecified whether performing this subroutine performs an implicit connection of a file to the unit for some unit numbers.

In the following code fragment, lines in a data file are accessed according to the byte offset stored in an array.

```
C   Seek to the starting field within the current line

80     CALL PXFFSEEK(IUNIT, RELPOS(IPTR),
    + IPXFCONST('FSEEK_CURRENT'), ISTAT)
      IF ( ISTAT .EQ. IPXFCONST('EEND') GOTO 90
      IF ( ISTAT .NE. 0 ) THEN
        WRITE (IPXFCONST('STDERR_UNIT'), 10) 'Error during PXFFSEEK'
        STOP
      END IF
10       FORMAT (A)

C   now read a record beginning at this location on the line

      READ (UNIT=IUNIT, FMT=10, END=90) LINE

      PRINT 10, LINE

      GOTO 80
90     CONTINUE
```

A.8.6.2 Read a File Position

The *PXFFTELL()* subroutine is used to obtain the byte offset in a file that is open for POSIX-based FORTRAN I/O. It may used in conjunction with *PXFFSEEK()* to return to specific byte locations within a file. While the *IUNIT* argument "shall refer to an open unit," this might not be the case, hence the error value. Further, it is intentionally left unspecified whether performing this subroutine performs an implicit connection of a file to the unit for some unit numbers. The following code fragment reads a large data file containing DNA sequences and references. *PXFFTELL()* is used to store the byte offset of each DNA sequence.

...

```
C   Read a line

      READ (UNIT=IUNIT, FMT=10, END=110) LINE

C   If the line begins with >>> it is a sequence

      IF ( LINE(1:3) .EQ. '>>>' ) THEN
        CALL PXFFTELL(IUNIT, IOFF, ISTAT)
```



```
        IF ( ISTAT .NE. 0 ) THEN
          WRITE (IPXFCONST('STDERR_UNIT'), 10) 'Error during PXFFTELL'
10       FORMAT (A)
          STOP
        END IF

C   Store the offset in a file for later use by PXFFSEEK

        WRITE (IUNIT2, 20) LINE(4:), IOFF
20       FORMAT (A,I6)
      END IF

C   Repeat until all the lines have been read
       ...
```

A.8.6.3 Get a Character

The *PXFGETC()* subroutine reads data from a file a byte at a time. It is useful for constructing menus, filter programs, or system utilities. The following code fragment waits for a single key to be pressed at the keyboard. The controlling terminal must be in noncanonical mode in order for this code to function properly (see 7.1).

```
      PRINT 10, 'Press any key when ready.'
      Call PXFGETC(CH, ISTAT)
```

A.8.6.4 Write a Character

The subroutines *PXFFPUTC()* and *PXFFGETC()* may be used together to create menu prompts, filter programs, or system utilities. The following is an example of a filter program that converts a file with carriage-return-delimited lines to a file with newline-delimited lines.

```
      PROGRAM CRTOLF

      INTEGER I, INUNT, OUTUNT, ISTAT, ILEN
      CHARACTER*256 PGM, OPT, INFILE, OUTFIL
      CHARACTER*1 CH

      CALL PXFPOSIXIO(1, IOLD, IERROR)
      INUNT = IPXFCONST('STDIN_UNIT')
      OUTUNT = IPXFCONST('STDOUT_UNIT')

C   Get the file names from the command line. If they are
C   missing use standard in and standard out.
C   No OPEN is required for either standard input or standard output.
C   Note that a more robust program would probably check for errors on OPEN.

      IF (IPXFARGC() .GT. 0 ) THEN
        CALL PXFGETARG(1, OPT, ILEN, ISTAT)
        IF (OPT(1:ILEN) .NE. '-' ) THEN
          INFILE(1:ILEN) = OPT(1:ILEN)
          INUNT = 14
          OPEN(UNIT=INUNT, FILE=INFILE, STATUS='OLD',
    +          ACCESS='SEQUENTIAL', FORM='FORMATTED')
        END IF
      END IF
```

```
      IF (IPXFARGC() .EQ. 2 ) THEN
        CALL PXFGETARG(2, OPT, ILEN, ISTAT)
        IF ( OPT(1:ILEN) .NE. '-' ) THEN
          OUTFIL(1:ILEN) = OPT(1:ILEN)
          OUTUNT = 15
          OPEN(UNIT=OUTUNT, FILE=OUTFIL, STATUS='UNKNOWN',
     +          ACCESS='SEQUENTIAL', FORM=,FORMATTED')
        END IF
      END IF
      IF (IPXFARGC() .GT. 2 ) THEN
        CALL PXFGETARG(0, PGM, ILEN, ISTAT)
        PRINT 10, 'USAGE: ', PGM(1:ILEN), ' [infile] [outfile]'
        STOP
      END IF

C   This is where the actual work of the program begins.
C   The input is byte filtered to the output until the input is
C   exhausted.

50     CALL PXFFGETC(INUNT, CH, ISTAT)
        IF ( ISTAT .EQ. IPXFCONST('EEND') ) GOTO 60
        IF ( ISTAT .NE. 0 ) STOP 'PXFFGETC ERROR'
        IF ( CH .EQ. CHAR(13) ) CH = CHAR(10)

        CALL PXFFPUTC(OUTUNT, CH, ISTAT)
        IF ( ISTAT .NE. 0 ) STOP 'PXFFPUTC ERROR'

      GOTO 50
60     CONTINUE
      CLOSE(INUNT)
      CLOSE(OUTUNT)
      END
```

A.8.7 Bit Field Manipulation

These functions are functionally identical to those of the same name in the MIL-STD-1753 {A4}, a common extension to FORTRAN 77. (See A.2.3.0.4.5 for further discussion of MIL-STD-1753 {A4} Extensions.) While the MIL-STD {A4} requires this type of function to be external, the developers of this standard intentionally avoided specifying whether these functions were to be implemented as externals or intrinsics. Only this set of bit-manipulation functions was specified in this standard because they were deemed minimally sufficient to access all the available functionality provided in POSIX.1 {2}.

A.8.7.1 Inclusive OR

There is no additional rationale provided for this subclause.

A.8.7.2 Logical AND

There is no additional rationale provided for this subclause.

A.8.7.3 Bitwise NOT

There is no additional rationale provided for this subclause.

A.8.8 System Date and Time

A.8.8.1 Local Time

The *PXFLOCALTIME()* subroutine converts time in seconds since the epoch to local time. The current time and date can be retrieved from the system by calling *PXFTIME()*. It can then be converted into local time by calling *PXFLOCALTIME()*. The *PXFSTAT()* subroutine will return file times in seconds since the epoch, which can also be converted to local time using *PXFLOCALTIME()*. Although a number of time procedures exist as standard practice, the developers of this standard chose to introduce this new procedure since the return value of year was changed to provide the Gregorian year rather than simply the year of the century. This full value permits monotonic increase across the century change, which is almost upon us. Further, Gregorian year is returned in both the *DATE* and the *VALUES(1)* arguments of the Fortran 90 {A1} intrinsic subroutine *DATE_AND_TIME()*.

A subroutine that returned a character string describing current time was discarded in order to avoid the inherent problems of internationalization of such a string. It was felt that this one subroutine provided minimal but complete functionality.

While the normal range of the value of seconds is 0–59, the range is extended to 0–61 to be able to handle the cases of “leap seconds.”

A.8.9 Command-Line Arguments

Although a large number of existing implementations already have the procedures *GETARG()* and *IARGC()* defined, the developers of this standard chose to specify new procedures in order to increase their robustness. The argument list returned by *PXFGETARG()* is zero-based, i.e., argument number zero is the command. It was argued that FORTRAN 77 programmers are more accustomed to one-based indexing, and that because the array passed to the *PXFEXECV()* subroutine would be, by default, one based, specifying *PXFGETARG()* to be zero-based would be confusing. However, the most common usage of *PXFGETARG()* will likely be to read the arguments of the command now executing. Since the command name is argument number zero, the list of arguments to the command are effectively one based. In addition, all of the current implementations surveyed are zero based.

The following program demonstrates the usage of the *PXFGETARG()* and *IPXFARGC()* subroutines; it simply displays the command-line arguments that were passed to the current program.

```
      PROGRAM ARGS

      INTEGER I, STATUS, ILEN
      CHARACTER*128 ARG, PGMNAM

      INTEGER IPXFARGC

C   Complain if no arguments are passed.

      IF ( IPXFARGC() .EQ. 0 ) THEN
        CALL PXFGETARG(0, PGMNAM, ILEN, STATUS)
       WRITE (IPXFCONST (‘STDERR’),20) ‘usage: ’, PGMNAM( 1:ILEN), ‘arg1 [arg2]
...’
        STOP
      END IF

      WRITE (*,10) ‘The number of arguments = ’, IPXFARGC()
10     FORMAT (A,14)

      DO I=1, IPXFARGC()
```

```
        CALL PXFGETARG(I, ARG, ILEN, STATUS)
        WRITE (*,20) ARG(1:ILEN)
20     FORMAT (A)
      END DO

      END
```

A.8.9.1 Get Command-Line Argument

Arguments were added to *PXFGETARG()* to specify the returned length of the string (to handle the significant trailing blanks issue), and to permit an error return.

A.8.9.2 Index of Last Command-Line Argument

As explained above, the array of command-line arguments is zero based (unlike a FORTRAN 77 array, which is one based).

A.8.10 Character String Procedure

A.8.10.1 Length of a String Trimmed of Trailing Blanks

Because of the fixed-declaration characteristic of FORTRAN 77 character variables, and the need to determine the actual length, minus the trailing blanks, of the string stored in that variable, this function was added. Although the FORTRAN 77 standard INDEX*function()*could information, the developers of this standard felt that usability and portability would be improved by including this special case function. While the name LNBLNK is current common practice, the specific name was chosen to reflect the *TRIM()* intrinsic function in Fortran 90 {A1}. In addition, the use of the PXF prefix made it clearer that this subroutine is only expected to be needed for a FORTRAN 77 binding.

A.8.11 Extended Range Integer Manipulation

As discussed in 2.3.2.2, POSIX.1 {2} makes use of the *unsigned integer* data type available in the C language. Because there is no primitive type available in FORTRAN 77 that is guaranteed to provide equivalent numeric range, the developers of this standard decided it was necessary to provide a portable means for manipulating these extended range integers. It was *not* intended to specify a new primitive type for use in FORTRAN 77 applications, or to provide utilities to support general-purpose functionality (e.g., arithmetic operations). Rather, it was agreed that the ability to compare two extended range integers was sufficient to support common or intended usage of these values in the POSIX.1 {2} environment. See A.2.3.2.2 for relevant technical details.

A.8.11.1 Unsigned Comparison

There is no additional rationale provided for this subclause.

A.8.12 Process Termination

No interactions for the PAUSE statement are specified since it has been identified as obsolete by Fortran 90 {A1}.

A.8.12.1 Interactions of the FORTRAN 77 STOP Statement

Note that the FORTRAN 77 standard does not allow for a negative value, so

```
      STOP -1
```

is not standard conforming. The equivalent


```
STOP 255
```

must be used to obtain the equivalent functionality.

A.8.12.2 Interactions of the FORTRAN 77 END Statement

There is no additional rationale provided for this subclause.

A.8.12.3 POSIX-Based Fortran Process Termination

Originally, the FORTRAN 77 language construct STOP was referenced rather than specifying *PXFEXIT()*. The functionality is similar (i.e., it terminates the process), but STOP does not provide a method for returning a status value to the system. Also, *PXFEXIT()* has clearly defined cleanup responsibilities that need not be met by a given implementation when STOP is executed.

A.9 System Databases

A.9.1 System Databases

A.9.2 Database Access

A.9.2.1 Group Database Access

Note that the group structure differs slightly from the POSIX.1 {2} specification because POSIX.1 {2} specified the *gr_mem* field as a "null-terminated vector of pointers to the individual member names" that "may point to static data that is overwritten in each call." Such a value would be difficult to represent and manipulate using the structure access and manipulation subroutines in this standard. As a result, an array of character strings is used, each containing an individual member name. Implementors should note that the length of this array is bound only by the number of user names allowed in a group. Tricks with dynamic storage in the *PXF<TYPE>GET()* subroutines may be required if this bound is unspecified.

Group names may contain significant trailing blanks. Thus, a length argument is required and provided.

A.9.2.2 User Database Access

There is no additional rationale provided for this subclause.

A.10 Data Interchange Format

A.10.1 Archive/interchange File Format

There is no additional rationale provided for this clause.

Acknowledgments

The developers of this standard wish to thank the following organizations for donating significant computer, printing, and editing resources to the production of this standard: UniForum (formerly/usr/group) and Hewlett Packard Co.

Also, the developers of this standard wish to thank the organizations employing the members of the working group and the balloting group for both covering the expenses related to attending and participating in meetings and for donating the time required both in and out of meetings for this effort.

UniForum

Cray Research

Genetic Computing Group

Hewlett-Packard Company

IBM Corporation

Lawrence Livermore National Laboratory

Sandia National Laboratories

San Diego Supercomputer Center