Language Reference Manual

GLS™ FORTRAN

210–856001–001

AEG

MODCOMP

AEG

Language Reference Manual

GLS™ FORTRAN

MODCOMP

Property
of
LOGICAL DATA CORPORATION

RECEIVED DEC 2 2 1992

Manual History

Manual Order Number: 210-856001-001

Title: GLS™ FORTRAN Language Reference Manual

Revision Level	Date Issued	Description
000	11/89	Initial Issue.
001	12/90	Reissue. Compatible with Open Architecture Systems, and MAX-Based systems executing revision D.1 or later of the MAX 32 Operating System.

Contents subject to change without notice.

MODCOMP is a registered trademark of Modular Computer Systems, Inc.
REAL/IX, GLS, and MAX are trademarks of Modular Computer Systems, Inc.
DEC, VAX, and VMS are trademarks of Digital Equipment Corporation.
UNIX is a registered trademark of AT&T in the U.S. and other countries.

Portions of this document are based on or reprinted from copyrighted documents by permission of Green Hills Software, Inc.

Copyright © 1990, by Modular Computer Systems, Inc.
All Rights Reserved.
Printed in the United States of America.

PROPRIETARY NOTICE

THE INFORMATION AND DESIGNS DISCLOSED HEREIN WERE ORIGINATED BY AND ARE THE PROPERTY OF MODULAR COMPUTER SYSTEMS, INC. (MODCOMP). MODCOMP RESERVES ALL PATENT, PROPRIETARY DESIGN, MANUFACTURING, REPRODUCTION, USE, AND SALES RIGHTS THERETO, AND RIGHTS TO ANY ARTICLE DISCLOSED THEREIN, EXCEPT TO THE EXTENT RIGHTS ARE EXPRESSLY GRANTED TO OTHERS. THE FOREGOING DOES NOT APPLY TO VENDOR PROPRIETARY PARTS.

SPECIFICATIONS REMAIN SUBJECT TO CHANGE IN ORDER TO ALLOW THE INTRODUCTION OF DESIGN IMPROVEMENTS.

FOR GOVERNMENT USE THE FOLLOWING SHALL APPLY:

RESTRICTED RIGHTS LEGEND

USE, DUPLICATION, OR DISCLOSURE BY THE GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN RIGHTS IN DATA CLAUSES DOE 952.227-75, DOD 52.227-7013, AND NASA 18-52.227-74 (AS THEY APPLY TO APPROPRIATE AGENCIES).

MODULAR COMPUTER SYSTEMS, INC.
1650 WEST McNAB ROAD
P.O. BOX 6099
FORT LAUDERDALE, FL 33340-6099

THIS MANUAL IS SUPPLIED WITHOUT REPRESENTATION OR WARRANTY OF ANY KIND. MODULAR COMPUTER SYSTEMS, INC. THEREFORE ASSUMES NO RESPONSIBILITY AND SHALL HAVE NO LIABILITY OF ANY KIND ARISING FROM THE SUPPLY OR USE OF THIS PUBLICATION OR ANY MATERIAL CONTAINED HEREIN.

Preface

Audience
This manual is written for users of the General Language System (GLS) FORTRAN compiler. It assumes you have previous FORTRAN programming experience.

Subject
This manual describes the FORTRAN programming language supported on the GLS FORTRAN compiler.

Product Requirements
The GLS FORTRAN compiler runs on MAX-Based and Open Architecture Systems.

Special Symbols and Notation
The GLS FORTRAN Compiler supports a subset of the VAX/VMS™ language extensions. The double dagger symbol (††) identifies the supported VAX/VMS language extensions in this manual.

Related Publications
Refer to the following manuals for additional information.

Manual Number	Title
216-856004-REV	GLS Programming Guide for 97xx Systems
216-856005-REV	GLS Programming Guide for MAX 32 Systems

MODCOMP Service and Assistance
MODCOMP offers a variety of programs and services that demonstrate our commitment to customer satisfaction. Our Technical Education department provides comprehensive hands-on instruction either at our facilities or at customer-designated sites. Our worldwide field service organization is ready to give installation assistance, free service during the warranty period, and flexible service programs tailored to your requirements.

Questions, Problems, and Suggestions
Your MODCOMP sales and service representatives can help you with any questions, problems, or suggestions you may have regarding our products and services. In addition, for your convenience MODCOMP maintains toll-free telephone numbers at which we can be reached for questions, problems, and suggestions. Please feel free to use the following numbers:

- **For questions, sales information, or suggestions:** in the U.S. and Canada, 1-800-255-2066 (In countries outside the U.S. and Canada, please call your regional sales support office or 1-305-974-1380 extension 1800 worldwide.)

- **For service:** in Florida, 1-800-432-1405; in the U.S., 1-800-327-8928; in Canada, 1-416-890-0666 (In countries outside the U.S. and Canada, please call your regional serv-

ice/support office.)

- **For Technical Education information:** in the U.S., 1-305-977-1708 (In countries outside the U.S., please call your regional support office.)

For comments about documentation, please use the response form at the back of this manual.

Revision Summary

This manual was reviewed for compatibility with MAX-Based Systems. No technical changes were required for compatibility.

The following additions and corrections were made:

- A quick reference of FORTRAN statements, system subroutines, and built-in functions was added to form Appendix C
- The function IARGC was added
- The subroutine GETARG was added

TABLE OF CONTENTS

Page

Chapter 1 Introduction

Audience and Required Knowledge 1-1
Manual Organization 1-1
Related Manuals 1-2
Product Overview 1-2

Chapter 2 Statements and Syntax

Program Statements 2-1
  Lines 2-1
    Line Formats 2-2
Debugging Statements 2-3
Statement Order 2-4
Statement Syntax 2-4
  Character Set 2-5
  Symbolic Names 2-5
  Statement Labels 2-6

Chapter 3 Data Types

Overview 3-1
*n Data Size Qualifiers 3-1
Initialization in Type Declaration 3-2
Integer Data 3-3
Real Data 3-3
Double Precision Data 3-4
Complex Data 3-4
Logical Data 3-5
Character Data 3-5
Hollerith Data 3-6
Specifying a Data Type Implicitly 3-6

Chapter 4 Constants, Variables, Arrays, and Substrings

Constants 4-1
  Octal Integer Constants 4-2
  Octal and Hexadecimal Typeless Constants 4-2
    Octal Constants 4-3
    Hexadecimal Constants 4-3
  Radix-50 Constants 4-4
Variables 4-5
Arrays 4-6

Page

Chapter 4 Constants, Variables, Arrays, and Substrings [continued]

Array Declarators and Subscripts 4-6
One-Dimensional Arrays 4-8
Multidimensional Arrays 4-9
Adjustable Array Declarators 4-9
Assumed-Size Array Declarators 4-10
Substrings 4-11

Chapter 5 Expressions

Overview 5-1
Expression Types 5-1
Arithmetic Expressions 5-2
Multiple Operators 5-2
Valid Operands 5-3
Data Type Evaluation 5-3
Character Expressions 5-5
Valid Operands 5-5
Relational Expressions 5-6
Arithmetical 5-6
Character 5-7
Complex 5-7
Logical Expressions 5-8
Valid Operands 5-9
Integer Operands 5-9

Chapter 6 Program Structure

Main Program 6-2
Subprograms 6-2
Block Data 6-2
Procedures 6-4
Procedure Arguments 6-4
Subroutines 6-5
Functions 6-6
External Functions 6-7
Statement Functions 6-7
Alternate Return Specifiers 6-8

Chapter 7 FORTRAN I/O

Overview 7-1
Records 7-1
Files 7-1

Page

Chapter 7 FORTRAN I/O [continued]

I/O Units 7-2
Connection 7-2
Preconnection 7-3
Access Method 7-3
Properties of Files 7-3
Existence 7-3
Position 7-4
I/O Statements 7-5
Data Transfer Statements 7-5
File Positioning Statements 7-6
Auxiliary I/O Statements 7-6
Data Transfer 7-7
Formatted Transfer 7-7
Editing 7-8
Format Control 7-8
List-Directed Formatting 7-9
Unformatted Transfer 7-9

Chapter 8 Format Specification

Specifying Formats 8-1
General Form for Format Specifications 8-1
Character Format Specification 8-2
Format Control 8-2
Repeatable Edit Descriptors 8-3
Alphanumeric Editing 8-4
Numeric Editing 8-5
Floating-Point Editing, D and E 8-6
Floating-Point Editing, F 8-7
Floating-Point Editing, G 8-8
Complex Editing 8-10
Integer Editing 8-11
Octal Editing 8-12
Hexadecimal Editing 8-13
Logical Editing 8-15
Nonrepeatable Edit Descriptors 8-16
Apostrophe Descriptor 8-17
Hollerith Descriptor 8-17
Q Editing 8-17
Carriage Control Editing 8-18
Blank-Control Descriptors BN and BZ 8-19
Scale-Factor Descriptor kP 8-20
Sign-Control Descriptors S, SP, and SS 8-21

Page

Chapter 8 Format Specification [continued]

Position Descriptors Tc, TLc, TRc, and nX . . . 8-21
Line-Termination Descriptor / . . . 8-22
Conditional Line-Termination Descriptor . . . 8-23
List-Directed Formatting . . . 8-23
List-Directed Input . . . 8-24
List-Directed Output . . . 8-26

Chapter 9 Statements

Assignment Statements . . . 9-1
Control Statements . . . 9-2
Input/Output Statements . . . 9-3
Specification Statements . . . 9-4
Structural Statements . . . 9-5
ACCEPT Statement . . . 9-6
ASSIGN Statement . . . 9-7
Assignment Statement (Arithmetic) . . . 9-8
Assignment Statement (Character) . . . 9-9
BACKSPACE Statement . . . 9-10
BLOCK DATA Statement . . . 9-12
BYTE Statement . . . 9-13
CALL Statement . . . 9-14
CHARACTER Statement . . . 9-15
CLOSE Statement . . . 9-16
COMMON Statement . . . 9-18
COMPLEX Statement . . . 9-19
CONTINUE Statement . . . 9-20
DATA Statement . . . 9-21
DECODE Statement . . . 9-23
DIMENSION Statement . . . 9-24
DO Statement . . . 9-25
DO WHILE Statement . . . 9-27
DOUBLE PRECISION Statement . . . 9-28
ELSE Statement . . . 9-29
ELSE IF Statement . . . 9-30
ENCODE Statement . . . 9-31
END Statement . . . 9-32
END DO Statement . . . 9-33
END IF Statement . . . 9-34
ENDFILE Statement . . . 9-35
ENTRY Statement . . . 9-36
EQUIVALENCE Statement . . . 9-37
EXTERNAL Statement . . . 9-39

Page

Chapter 9 Statements [continued]

FORMAT Statement 9–40
FUNCTION Statement 9–41
GOTO Statement (Assigned) 9–42
GOTO Statement (Computed) 9–43
GOTO Statement (Unconditional) 9–44
IF Statement (Arithmetic) 9–45
IF Statement (Block) 9–46
IF Statement (Logical) 9–47
IMPLICIT Statement 9–48
INCLUDE Statement 9–49
INQUIRE Statement 9–50
INTEGER Statement 9–54
INTRINSIC Statement 9–55
Logical Assignment Statement 9–56
LOGICAL Statement 9–57
NAMELIST Statement 9–58
  NML NAMELIST Specifier 9–59
  NAMELIST–Directed I/O 9–60
  NAMELIST Record Format 9–60
  Prompting for NAMELIST Values 9–63
OPEN Statement 9–64
OPTIONS Statement 9–68
PARAMETER Statement 9–70
  Compile–Time Expressions 9–71
  Symbolic Names in Constant Expressions 9–72
PAUSE Statement 9–73
PRINT Statement 9–74
PROGRAM Statement 9–76
READ Statement 9–77
REAL Statement 9–79
RETURN Statement 9–80
REWIND Statement 9–81
SAVE Statement 9–82
STOP Statement 9–83
SUBROUTINE Statement 9–84
TYPE Statement 9–85
VIRTUAL Statement 9–86
VOLATILE Statement 9–87
WRITE Statement 9–88

Chapter 10 System Subroutines, Built–Ins, and Intrinsic Functions

System Subroutines 10–1

Page

Chapter 10 System Subroutines, Built-Ins, and Intrinsic Functions [continued]

DATE . . . 10-2
ERRSNS . . . 10-2
EXIT . . . 10-2
GETARG . . . 10-3
IARGC . . . 10-3
IDATE . . . 10-3
MVBITS . . . 10-4
RAN . . . 10-4
SECNDS . . . 10-5
TIME . . . 10-5
Built-In Functions . . . 10-6
%VAL . . . 10-6
%REF . . . 10-6
%LOC . . . 10-7
Intrinsic Functions . . . 10-8
ABS Function . . . 10-8
ACOS Function . . . 10-9
ACOSD Function . . . 10-9
AIMAG Function . . . 10-9
AINT Function . . . 10-10
AMAX0 Function . . . 10-10
AMIN0 Function . . . 10-11
ANINT Function . . . 10-11
ASIN Function . . . 10-11
ASIND Function . . . 10-12
ATAN Function . . . 10-12
ATAND Function . . . 10-12
ATAN2 Function . . . 10-13
ATAN2D Function . . . 10-13
BTEST Function . . . 10-13
CHAR Function . . . 10-14
CMPLX Function . . . 10-14
CONJG Function . . . 10-15
COS Function . . . 10-15
COSD Function . . . 10-15
COSH Function . . . 10-16
DBLE Function . . . 10-16
DCMPLX Function . . . 10-17
DFLOAT Function . . . 10-17
DIM Function . . . 10-18
DPROD Function . . . 10-18
DREAL Function . . . 10-18
EXP Function . . . 10-19

Page

Chapter 10 System Subroutines, Built-Ins, and Intrinsic Functions [continued]

FLOAT Function 10-19
IABS Function 10-19
IADDR Function 10-20
IAND Function 10-20
IBCLR Function 10-21
IBITS Function 10-21
IBSET Function 10-21
ICHAR Function 10-22
IDIM Function 10-22
IDINT Function 10-22
IDNINT Function 10-23
IEOR Function 10-23
IFIX Function 10-24
INDEX Function 10-24
INT Function 10-25
IOR Function 10-26
ISHFT Function 10-26
ISHFTC Function 10-27
ISIGN Function 10-27
LEN Function 10-27
LGE Function 10-28
LGT Function 10-28
LLE Function 10-29
LLT Function 10-29
LOG Function 10-30
LOG10 Function 10-30
MAX Function 10-30
MAX0 Function 10-31
MAX1 Function 10-31
MIN Function 10-31
MIN0 Function 10-32
MIN1 Function 10-32
MOD Function 10-32
NINT Function 10-33
NOT Function 10-33
REAL Function 10-34
SIGN Function 10-34
SIN Function 10-35
SIND Function 10-35
SINH Function 10-35
SQRT Function 10-36
TAN Function 10-36
TAND Function 10-36

Page

Chapter 10 System Subroutines, Built-Ins, and Intrinsic Functions [continued]

TANH Function 10-37
ZEXT Function 10-37

Chapter 11 Records, Structures, and Unions

Records 11-1
Structure Declarations 11-1
UNION Declarations 11-3
Using RECORDS and STRUCTURES 11-4
Aggregate Assignment Statement 11-5
Scalar Field References 11-5
Aggregate Field References in I/O Statements 11-5

Appendix A ASCII and Hexadecimal Conversions A-1

Appendix B VAX/VMS Language Extensions B-1

Appendix C Quick Reference C-1
GLS FORTRAN Statements C-1
System Subroutines C-5
Built-in Functions C-6

Glossary Glossary-1

Index Index-1

LIST OF FIGURES

		Page
6–1	GLS FORTRAN Program Structure	6–1

LIST OF TABLES

Page

2-1 Special Characters 2-5
4-1 Octal Integer Constants Usage 4-2
4-2 Octal Constants Usage 4-3
4-3 Hexadecimal Usage 4-3
4-4 Radix-50 Equivalents 4-4
4-5 Radix-50 Usage 4-5
5-1 Expression Summary 5-1
5-2 Arithmetic Expressions 5-2
5-3 Data Type Hierarchy 5-4
5-4 Relational Operators 5-6
5-5 Logical Operators 5-8
8-1 Repeatable Edit Descriptors 8-3
8-2 Storage Allocation For Data Types 8-4
8-3 Summary of Nonrepeatable Edit Descriptors 8-16
9-1 OPTION Values and Command Line Equivalents 9-69
10-1 System Subroutines 10-1

Chapter 1

Introduction

This chapter describes the intended audience, required product knowlege, manual organization, related manuals, product overview, and manual conventions.

Audience and Required Knowledge

This manual is written for programmers using the General Language System (GLS™) FORTRAN compiler. It assumes you have previous FORTRAN programming experience, and are familiar with either the REAL/IX™ Operating System on Open Architecture Systems, or the MAX 32™ Operating System on MAX-Based Systems.

Manual Organization

The manual is organized as follows:

Chapter 1 describes the intended audience and gives an overview of the FORTRAN language standards and conventions.

Chapter 2 describes the syntax of a GLS FORTRAN statement and the rules for ordering statements within a program.

Chapter 3 describes the data types and how to specify a data type implicitly.

Chapter 4 describes constants, variables, arrays, and substrings.

Chapter 5 describes arithmetic, character, relational, and logical expressions.

Chapter 6 describes the structure and program unit of an executable GLS FORTRAN program.

Chapter 7 describes the I/O system, records, files, I/O units, and data transfer.

Chapter 8 describes various methods of format specification.

Chapter 9 describes GLS FORTRAN statements.

Chapter 10 describes system subroutines, built-in functions, and intrinsic functions.

Chapter 11 describes records, structures, and unions.

Appendix A is an ASCII/Hexadecimal conversion chart.

Appendix B lists the VAX/VMS Language extensions.

Appendix C is a quick reference of GLS FORTRAN statements, system subroutines, and built-in functions.

Related Manuals

The following manuals contain related information.

GLS Programming Guide for 97xx Systems (216-856004-REV)
This manual describes generic compilation and how to invoke the GLS C, FORTRAN, and Pascal Compilers for 97xx systems.

GLS Programming Guide for MAX 32 Systems (216-856005-REV)
This manual describes generic compilation and how to invoke the GLS C, FORTRAN, and Pascal Compilers for MAX 32 systems.

Product Overview

The GLS FORTRAN compiler provides you with a code development environment for writing source programs in FORTRAN, interlanguage callability, global and local optimizations, and portability among MODCOMP hardware systems. The GLS FORTRAN compiler includes a set of FORTRAN library routines and functions. You can also write routines in FORTRAN and maintain them in object library files using the Library Update tool.

GLS FORTRAN implements the ANSI FORTRAN-77 (Full Language) Standard, ANSI X3.9-1978, and the Military Standard FORTRAN, as described in the document "FORTRAN, DOD Supplement to American National Standard X3.9-1978, MIL-STD-1753". GLS FORTRAN is compatible with the Berkeley 4.3BSD *f77* compiler and supports a subset of the VAX/VMS™ FORTRAN V4.6 compiler extensions. Refer to Appendix B for a list of the VAX/VMS extensions supported by the GLS FORTRAN Compiler.

The double dagger symbol (‡‡) identifies the VAX/VMS language extensions supported by the GLS FORTRAN Compiler. These extensions require the -X181 compile option to enable VAX/VMS FORTRAN compatibility. Refer to the GLS Programming Guide *for details about compiler options and default modes.*

GLS FORTRAN is validated by running the FORTRAN Compiler Validation System Version 2.0 (1978) from the U.S. Office of Software Development and the U.S. Department of Commerce, National Technical Information Service.

Chapter 2

Statements and Syntax

This chapter describes GLS FORTRAN program statements and their syntactical elements.

Program Statements

GLS FORTRAN program statements are comprised of syntactical items such as symbolic names, statement labels, constants, operators, and special characters. There are two types of statements:

- Executable

  These statements indicate a processing action, such as the PRINT or CONTINUE statement. They execute sequentially in the order placed in a program unit. Execution of an executable program begins with the first statement in the main program. Control statements, such as GOTO and CALL, transfer the execution sequence to a different point in the program. Statements that transfer the execution sequence are considered executable statements.

- Non-Executable

  These statements define and classify program units and specify entry points in subprograms, data formatting information, initial values, and execution characteristics for data.

Lines

Each GLS FORTRAN statement is written on one or more program lines. A program line is a sequence of columns numbered consecutively from left to right beginning with 1. A statement can span a maximum of 20 program lines, or 1320 character positions. There are three types of program lines:

- Initial lines

  This is the first line of a statement. If statements exceed column 72 of the initial line, the remainder of the statement can reside in continuation lines. A maximum of 19 continuation lines can be used to hold a statement.

- Continuation lines

  Continuation lines hold statement elements that exceeds the initial line; they begin in column 6 and can be any character (except a blank or zero). You must distinguish continuation lines from initial lines in a program.

- Comment lines

  Comment lines hold program notes for documenting your program and do not affect program execution. To specify comment lines you must enter a C or asterisk (*) in column one, or use an ††exclamation point (!) in a column. the other lines in a program with the letter C or an asterisk in column one.

Example

```
C       Standard FORTRAN  comment lines
*
         AA = A + B             !Compute AA as sum of A and B
         C = 'Hi there!'
!
!        The assignment statement above includes a trailing comment line
!        However, the exclamation point in the string 'Hi there!' is
!        within a quoted string and not interpreted as a comment
!        delimiter
```

Line Formats

GLS FORTRAN supports two types of line formatting: free-field and column-based. Free-field format allows you to enter program lines of unlimited length without regard to columns. Column-based format requires you to enter specific information in defined fields; it is supported for compatibility with programs written prior to free-field format.

Free-Field

Free-field format is indicated by a tab character in columns 1 through 72 of the initial line, or an ampersand character (&) in column 1 of a continuation line. For example,

```
10<tab>FORMAT('This statement is in free-field format')

&10<tab>FORMAT('This statement is in free-field format')
```

If the initial line of a statement is in free-field format, all continuation lines must be in free-field format.

††Another way to indicate a free-field format continuation line is by placing a <TAB> character in column one, followed by a number from 1 through 9. A single statement may include up to 99 continuation lines and the numbers can be used again. For example,

```
<tab>9<tab>FORMAT('This statement is in free-field format')
```

Column-Based

Column-based format program lines consist of 80 columns divided into four fields. Each field is reserved for the following information:

- Columns 1 through 5

  Statement label; identifies and references specific statements in a program.

- Column 6

  Continuation mark; indicates the line is a continuation of the preceding line.

- Columns 7 through 72

  Statement text.

- Columns 73 through 80

  Notes; the compiler ignores all characters in this field.

The following example shows a line in column-based format. Columns 1 through 5 of the first line contain a line number, 10000. Column 6, the continuation column, contains a blank, indicating the initial line of a statement. Columns 7 through 72 contain the initial line of a FORMAT statement.

Example

```
10000 FORMAT('This line contains no tab, it is in column-based format')
```

Debugging Statements

††Debugging statements are distinguished from standard FORTRAN source code by placing one of the following letters in column one: D, d, X, or x. They can include labels and continuations as shown below.

Example

```
C       The following code is compiled only when -X82 is used
C
D       DO 10 I = 1,10
X       J = (I**2) + (D/4)
D10     CONTINUE
```

Debugging statements are compiled as ordinary statements when the –X82 option is specified at compile time. The default operation (–Z82) treats debugging statements as comment lines.

Statement Order

The following rules apply to the order of statements and comments within a program unit:

- Comment lines can appear anywhere before the END statement.
- The PROGRAM statement can appear only as the first statement of a main program. The FUNCTION, SUBROUTINE, and BLOCK DATA statements can appear only as the first statement in a subprogram.
- FORMAT and ENTRY statements can appear anywhere before the END statement.
- PARAMETER statements can appear anywhere before DATA statements, statement function statements, and executable statements.
- IMPLICIT statements must appear before all other specification statements except PARAMETER statements and FORMAT statements.
- All other specification statements (COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, INTRINSIC, SAVE) must appear before any DATA statements.
- DATA statements can appear anywhere following the specification statements.

  Standard FORTRAN requires that all DATA statements appear before the first executable statement.

  ‡‡ When VAX/VMS compatibility is enabled, DATA statements can be used anywhere within a program unit.
- All statement function statements must appear before any executable statements.
- All executable statements must appear before the END statement.
- The END statement must be the last statement in a program unit.

Statement Syntax

This section describes syntax elements including the supported character set, symbolic names, and statement labels.

Character Set

GLS FORTRAN uses the standard ASCII character set, consisting of uppercase and lowercase letters (A through Z), digits (0 to 9), and a group of special characters. Letters and digits are collectively referred to as alphanumeric characters.

Table 2–1 lists the special characters:

Table 2–1. Special Characters

Character	Name	Character	Name
'	Apostrophe	=	Equal sign
*	Asterisk	(	Left parenthesis
	Blank	-	Minus sign
:	Colon	%	Percent sign
,	Comma	+	Plus sign
.	Decimal point	)	Right parenthesis
$	Dollar sign	/	Slash
&	Ampersand	_	Underscore
"	Quotation	!	Exclamation
<	Left bracket	>	Right bracket

GLS FORTRAN also recognizes the ASCII control characters that signify carriage returns, tabs, line feeds (also called new lines), and form feeds.

All characters conform to a collating order that defines a hierarchy for sorting character strings. The ASCII convention assigns a number to each character that determines the hierarchy. This enables GLS FORTRAN to compare two character strings, a character at a time. For a list of the collating sequence, refer to Appendix A.

Symbolic Names

A symbolic name identifies a program entity: either a **local entity** (constant, variable, array, dummy procedure, statement functions) or a **global entity** (main program, subroutine, block data subprograms, common blocks, and external functions). Symbolic names are one to six alphanumeric characters, where the first character is a letter. Other valid characters in symbol names are the dollar sign ($) and underscore (_). Symbol names must be unique except for common block and external function names defined globally and locally in the same program unit.

Symbolic names for a dummy argument for a statement function statement, and the DO variable, have smaller scopes (range of affect) than the program unit. Note the following differences:

- A symbolic name for a dummy argument for a statement function statement has a scope of that statement
- A symbolic name for the DO variable for an implied DO in a DATA statement has a scope of the implied DO list.

Statement Labels

Statement labels identify and reference individual statements in a program. Any statement can have a statement label except continuation lines. Executable statements must have statement labels to be referenced by other statements in your program. For example, the GOTO statement can only transfer execution to a labeled executable statement.

Statement labels are one to five digits, where at least one of the digits is nonzero. They can appear anywhere in columns 1 through 5 of the statement's initial line.

Statement labels must be unique as they have the scope of a program unit. Blank characters and leading zeros cannot be used for distinguishing different statement labels. For example, the following statement labels are considered identical in a GLS FORTRAN program:

```
Column  1  2  3  4  5
-------------------------
        7  7  7
        0  0  7  7  7
        0  7  7  7
              7  7  7
```

Chapter 3

Data Types

This chapter describes the supported data types and how to specify a data type implicitly.

Overview

Data can be a numeric value represented by a series of digits, or text represented by strings of characters. Any character in the ASCII character set can be used as text data.

GLS FORTRAN supports the following data types:

- **n*
- Integer
- Real
- Double precision
- Complex
- Logical
- Character
- Hollerith

*n Data Size Qualifiers

†† A symbol data type declaration may include a data type length specifier with the type and/or name. This size qualifier overrides any length that would otherwise be implied by the statement. Use the following syntax for **n* size qualifiers:

type [**n*] *symbol* [**n*] [,*symbol* [**n*]...]

type is any valid FORTRAN data type keyword and *n* specifies the data type length, preceded by an asterisk (*).

Example

```
C                  A, B and D are INTEGER*4
C                  C and E are INTEGER*2
C
      INTEGER*4    A(5), B, C*2, D, E*2(5)
C
      REAL         X*8, Y*8, Z*4
```

Initialization in Type Declaration

†† Variables and arrays can be initialized within a single type declaration statement, omitting the need for a separate DATA statement. Use the following syntax:

type symbol/value/ [[,]*symbol/value/*]...

type is any valid type declarator, *symbol* is a variable or array name, and *value* consists of one or more constants that will be assigned to *symbol* and are delimited by slashes (/).

Examples

In the following example, the value 3.1415 is assigned to the variable PI; the values 1, 2, and 3 are assigned to the array IBUF, and the array IARRAY is filled with zeros.

```
DIMENSION   IBUF(3)
REAL        PI/3.1415/
INTEGER     IBUF /1,2,3/, IARRAY(4) /4*0/
```

A CHARACTER variable or CHARACTER array that is one character in length can be initialized to a numeric constant within a type or DATA statement. The numeric constant must be a value from 0 to 255 decimal, specified as an integer, octal or hexadecimal value.

```
CHARACTER*1   CBUF/35/
CHARACTER*1   INBUF(4) /10, '4'O, 25, '5F'X/
```

Integer Data

An integer is any whole positive or negative number, or zero (0). You can write integers with a leading sign. If you omit the sign, the integer is considered to be positive.

A standard integer occupies four bytes and can represent values from −2147483648 through +2147483647. However, you can specify a 1- or 2-byte integer. For more information about specifying a 1- or 2-byte integer, refer to the INTEGER statement Chapter 9.

A constant is a value that does not change, for example: 1, 642, +25, 9287, −41, −73268, 0. For more information about constants, refer to Chapter 4.

Real Data

A real number is any number that can express a fractional component, an exponent, or both. Real numbers can be positive or negative, and can be expressed as a constant in the following forms:

- Real constant with or without an exponent
- Integer constant with an exponent

A real constant consists of an optional sign, an integer component, a decimal point, and a fractional component. Both the integer and fractional components are series of digits. The following are examples of real constants: 1.5, +82.7, .007, 375., −794.0, −.299999.

Exponential or scientific notation consists of the letter E followed by an optionally signed integer. The value of a real or integer constant with exponential notation is the product of the constant that precedes the E (the mantissa) and the power of 10 indicated by the integer that follows the E (the exponent).

The following are examples of real constants and integer constants with exponents.

```
      5.82E2  =  582.0
314159.00E-5  =  3.14159
    -.229E-3  =  -.000229
        11E5  =  1100000
       -5E-2  =  -.05
     -100E+8  =  -10000000000
```

Real numbers occupy four bytes of memory space. You can also specify real numbers that occupy eight bytes of memory. For more information about specifying an 8-byte real number, refer to the REAL statement in Chapter 9.

Double Precision Data

Double precision real numbers provide additional significant digits of accuracy for real numbers. They can be expressed as a real or integer constant with an exponent.

Double precision exponential notation consists of the letter D followed by an optionally signed integer. The value of a real or integer constant with exponential notation is the product of the constant that precedes the D (the mantissa) and the power of 10 indicated by the integer that follows the D (the exponent). The following are examples of real and integer constants with double precision exponential notation:

252.7D2	=	25270.0
.007D−1	=	.0007
883366.0D−4	=	88.3366
837612458378183D−8	=	8376124.58378183
−.0045D+3	=	−4.5
−69124820D−3	=	−69124.820

Double precision real numbers occupy eight bytes of memory space.

You can use the DOUBLE PRECISION statement or a REAL*8 statement to declare double precision numbers. For more information about declaring a double precision number, refer to the DOUBLE PRECISION statement and the REAL statement in Chapter 9.

Complex Data

A complex number is any number that can be expressed in the form A+B*i*. A and B are real numbers and the imaginary number *i* is equal to the square root of −1. In GLS FORTRAN, you can write a complex number as an ordered pair of integer or real constants separated with a comma and enclosed in parentheses. The first member of the pair is the real constant and the second member of the pair is the imaginary constant as follows:

(real-constant,imaginary-constant)

A complex number is stored as a pair of real values. Either constant can be positive, negative, or zero. The following are examples of complex numbers expressed as constants:

(0,2)	=	0+2*i* or 2*i*
(7.5,2.2)	=	7.5+2.2*i*
(−11,−3)	=	−11−3*i*
(−.6E3,.55)	=	−600+.55*i*
(945E−2,−41E−1)	=	9.45−4.1*i*

A complex number occupies eight consecutive bytes of memory space in a storage sequence. You can also specify a 16-byte complex number. For more information about specifying a 16-byte complex number, refer to the COMPLEX statement in Chapter 9.

Logical Data

Logical data enables a program to evaluate a statement as true or false. There are two logical constants: the words true and false, delimited with periods as follows:

.TRUE. or .true.
.FALSE. or .false.

Logical data occupies four bytes of memory space. However, you can specify logical data that occupies one or two bytes. For more information about specifying 1- or 2-byte logical data, refer to the LOGICAL statement in Chapter 9.

The actual value assigned to the logical operator TRUE may vary depending on operating system implementation.

FALSE is assigned a value of zero, and TRUE is assigned a non-zero value. Therefore, the actual value of TRUE could be any non-zero value. For example, on VAX/VMS a logical is true only when the low order bit is 1. On UNIX® operating systems a logical is true only when it is non-zero.

Use the LOGICAL statement to declare a symbolic name with the logical data type.

The BYTE statement is equivalent to LOGICAL*1 and can contain signed integers in the range −128 through +127.

Character Data

Character data enables a program to process text. It is represented by one or more characters, delimited with apostrophes, and is referred to as strings. GLS FORTRAN recognizes and differentiates between upper- and lowercase letters. The following are examples of strings:

'Please enter your password.'

'PERCENTAGE OF ERROR < 7%'

Any character in the GLS FORTRAN character set can be within a string, including the blank character and the apostrophe. To represent the apostrophe within a string, use two consecutive apostrophes, as shown in the following example:

'It''s two o''clock!'

Displays as: It's two o'clock!

The length of a string is the number of characters, including blanks, that appear between the apostrophes. Two consecutive apostrophes count as one character. The length of a character string must be 1 or greater. The following are examples of strings and their corresponding string length:

'Please enter your password.' (string length is 27)
'It''s two o''clock!' (string length is 17)

Character data occupies one byte per character.

Use the CHARACTER statement to declare a symbolic name as character data.

Hollerith Data

Hollerith data provides text processing capability and is considered an extension to the FORTRAN-77 standard. For compatibility with earlier versions of FORTRAN, GLS FORTRAN supports Hollerith data. (The FORTRAN-77 standard is the first version of the language to provide the character data type that is considered superior to the Hollerith form.)

Hollerith data, like character data, is a string of characters. You can use any character in the GLS FORTRAN character set within a Hollerith string, including blanks. A Hollerith constant consists of a nonzero, unsigned, integer constant (*n*), the letter H, and a string of contiguous characters (*c*) as shown in the following format specification.

*n*H*ccc*...*c*

(*n* characters after the H)

The following are examples of Hollerith constants:

16HToday's date is:
11HGRAND TOTAL
4HNaCl

To declare an integer, real, or logical symbolic name as Hollerith, use a DATA statement or READ statement.

Specifying a Data Type Implicitly

There are two ways to specify a data type:

- Explicitly

  To specify a data type explicitly, use a statement such as INTEGER or REAL

- **Implicitly**

To implicitly specify a data type, the first letter of the symbolic name determines the datatype. The default convention is: symbolic names beginning with letters I, J, K, L, M, or N are integer data types (for example, list, increment, multiple, kilo); symbolic names beginning with any other letter are real data types (for example, alpha, delta, total, variation).

Note that you cannot have explicit data types specified if you want to use implicit data type specification; explicit supercedes implicit.

To change the data type convention, use the IMPLICIT statement described in the "Statements" chapter.

††To override all implicit defaults and force them to be declared explicitly, refer to the IMPLICIT NONE statement described in Chapter 9.

Chapter 4

Constants, Variables, Arrays, and Substrings

This chapter describes elements specific to the FORTRAN-77 language.

Constants

A constant is a numeric, logical, or character value in a program that does not change during program execution.

For example, consider a program that calculates the area of a circle using the formula $A = \pi r^2$ where r is the radius of the circle, π is equal to 3.1415926, and A is the area. The value of π remains unchanged regardless of any other value in the calculation. Therefore, to translate the area formula to GLS FORTRAN, you can use the real constant 3.1415926 for π.

To associate a constant with a symbolic name, use the PARAMETER statement. This causes your program to substitute the constant value wherever the symbolic name appears in the program. The symbolic name associated with a constant cannot assume a different value during the execution of a program. For more information, refer to the PARAMETER statement in Chapter 9.

You can specify a logical, real, or integer variable as a binary, hexadecimal, or octal constant with the following format:

letter'string'

If *letter* is b, *string* is binary, and must be only ones and zeros. If *letter* is o, *string* is octal, with digits 0 – 7. If *letter* is z or x, *string* is hexadecimal, with digits 0 – 9, a – f, A – F. For example, the following statements initialize all three elements of `myarrary` to 10:

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

Octal Integer Constants

†† An octal integer constant is treated as an integer data type using the following syntax:

"*nnn*

where *nnn* is a string of digits from 0 to 7 inclusive, prefixed by a quotation mark.

Table 4-1 shows some sample valid and invalid octal integer constants.

Table 4-1. Octal Integer Constants Usage

Sequence	Valid?	Reason
"01237	Yes	Correct form
34560	No	Missing quotation mark
"123"	No	No trailing quotation marks allowed
'01234	No	Incorrect punctuation form
"13579	No	Invalid octal digit (9)

Octal and Hexadecimal Typeless Constants

†† Octal and hexadecimal constants can be used wherever numeric constants are allowed. A maximum of 128 bits (16 bytes) can be specified in octal or hexadecimal constants, allowing a maximum of 43 octal digits, or up to 32 hexadecimal digits. If more digits are specified than can be stored in the corresponding data type, the constant is truncated on the left. If the constant specified is less than the total storage for the corresponding data type, the value is zero-filled.

Octal and hexadecimal constants assume a data type based on use, and they have no previous implicit data type. Use the following syntax:

'*nnnnn*'O

or

'*nnnnn*'X

nnnnn represents a string of valid octal or hexadecimal digits, followed by the letter 'X' for hexadecimal constants or 'O' for octal constants.

Octal Constants

†† An octal constant consists of a string of valid octal digits delimited by apostrophes, followed by the letter 'O'. The letter 'O' may be either upper- or lowercase. Valid octal digits are the numbers 0 through 7, inclusive. Table 4–2 shows some sample valid and invalid octal sequences.

Table 4–2. Octal Constants Usage

Sequence	Valid	Reason
'01234567'o	Yes	Correct form
'255'O	Yes	Correct form
3456O	No	Missing apostrophes
'01234O'	No	Incorrect placement of apostrophes
'13579'o	No	Invalid octal digit (9)

Hexadecimal Constants

A hexadecimal constant consists of a string of valid hexadecimal digits delimited by apostrophes and followed by the letter 'X'. The letter 'X' can be specified in either upper- or lowercase.

Valid hexadecimal digits include numbers 0 through 9 and the letters A through F (or a through f). Hexadecimal letters can be specified in either upper- or lowercase.

Table 4–3 shows some sample valid and invalid hexadecimal sequences.

Table 4–3. Hexadecimal Usage

Sequence	Valid	Reason
'4FFF'X	Yes	Correct form
'4FFFX'	No	Incorrect placement of apostrophes
'offa'x	Yes	Correct form
offfX	No	Missing apostrophes

Radix–50 Constants

††Radix–50 encoding allows character data to be represented in packed form, storing up to 3 characters in 16 bits. Normal character storage allows a maximum of 2 characters per 16-bit word. Use the following syntax:

*n*R*cccccccccccc*

where *n* is a value from 1 to 12 that specifies the number of characters to the right of the letter *'R'*, and *cccccccccccc* represents an ASCII character string of *n* characters that will be converted to Radix–50 notation.

Radix–50 encoding is accomplished by assigning a subset of the ASCII character set Radix–50 values, then calculating a single 16-bit number using the formula

((rad1 * 40 + rad2) * 40 + rad3)
((rad1 * '0050'c + rad2) * '0050'c + rad3)

rad1, *rad2*, and *rad3* are Radix–50 values for the selected ASCII characters from left to right. For example, the Radix–50 constant 3RABC will be stored internally as decimal 1683.

The ASCII character subset and Radix–50 equivalents are listed in Table 4–4. Both Radix–50 and ASCII codes are shown as octal values. Note that the Radix–50 value 35 is not assigned. Note that upper and lowercase alphabetic characters are equivalent in Radix–50.

Table 4–4. Radix–50 Equivalents

Character	Octal		Decimal	
	ASCII	Radix–50	ASCII	Radix–50
space	40	0	32	0
A – Z	101 – 132	1 – 32	65 – 90	1 – 26
a – z	041 – 062	1 – 32	33 – 50	1 – 26
$	44	33	36	27
.	56	34	46	28
unassigned		35		29
0 – 9	60 – 71	36 – 47	48 – 57	30 – 39

Radix-50 constants may be used only in DATA statements. The data type of the variable determines the total number of bytes that may be stored for the specified constant. If the constant value is too large to be stored in the selected data type, the rightmost bytes are truncated. If the constant evaluates to a value smaller than the maximum storage for the selected data type, the constant is blank-filled from the right. Table 4-5 shows some sample valid and invalid Radix-50 sequences.

Table 4-5. Radix-50 Usage

Sequence	Valid	Reason
5RHELLO	Yes	Correct form
14R1234567890123 4	No	More than 12 characters specified
8RHI THERE	Yes	Correct (note that embedded spaces are allowed)
6RI_TST	No	Underscore is not a valid Radix-50 character

Variables

A variable is a symbolic name that is associated with a numeric, logical, or character value in a program. The symbolic name for the variable can assume a different value during the execution of a program.

Consider the program that calculates the area of a circle with the formula $A = \pi r^2$ where r is the radius of the circle, π is equal to 3.1415926, and A is the area. The value of π does not change and is represented with the real constant 3.1415926. However, the value of r varies for circles of different sizes. If you use a constant to represent r, the program would be limited to circles of one size. You would have to rewrite the program every time you wanted to calculate the area of a circle with a different radius. Instead, if r is a variable, the program can assign a new value to r for each new area calculation.

A variable must have an assigned value before you can reference its name in the program. When a reference to a variable name executes, the program uses the value that is currently assigned to the variable at that point in the execution of the program.

Variables assume values through assignment statements. For example, the integer variable `int` assumes the value 14 in the following arithmetic assignment statement:

```
int = 12 + 2
```

In the following arithmetic assignment statement, `int` assumes a value based on its current value:

```
int = int + 1
```

Assuming this assignment statement executes after the previous assignment statement in a program, the value of `int` changes from 14 to 15.

You can specify an initial value for a variable in a program using the DATA statement. For more information, refer to the DATA and BLOCK DATA statements in Chapter 9.

Arrays

An array is a sequence of variables that represent numeric, logical, or character values in a program. Each variable in the sequence is called an array element and can have a data type comprised of several bytes. Using arrays, you can reference a group of similar or related values with a symbolic name.

Arrays consist of one or more dimensions. Dimensions enable you to organize data according to various criteria defined in the context of your program. For example, a program designed to analyze political opinion poll information could organize data according to the voter's age, precinct, and political party affiliation. This program would use a 3-dimensional array.

Array Declarators and Subscripts

To declare an array in a program you must use an array declarator in a DIMENSION, COMMON, or type statement. An array declarator specifies a symbolic name to identify the array and a number of dimension declarators. The format for specifying an array declarator is:

symbolic name (*dimension declarator* [,*dimension declarator*]...)

Dimension declarators specify the number of elements in each array dimension that you declare. You set the number of elements with a lower- and upper-bound value. These values are called dimension bounds. The format for specifying a dimension declarator is:

[*lower-bound*:] *upper-bound*

Dimension bounds can be arithmetic constant or variable expressions that evaluate to integers. *lower-bound* can be negative, zero, or positive. If you do not specify *lower-bound*, a value of 1 is implied. *upper-bound* can be negative, zero, positive, or an asterisk indicating an assumed-size array declarator. A variable expression can be used as a dimension bound value and is referred to as an adjustable array declarator. Adjustable and assumed-size array declarators are described later in this chapter.

The number of dimension declarators you specify in an array declarator determines the number of dimensions for the array. An array in GLS FORTRAN can have a maximum of seven dimensions.

The size of an array is equal to the number of elements in the array. The number of elements is equal to the product of the dimension sizes specified in the array declarator.

The following is an example of an array declarator that declares an array with the symbolic name DISTANCES and one dimension declarator. DISTANCES is a 1-dimensional array with a lower bound of 10, an upper bound of 20, and 11 elements.

```
DISTANCES (10:20)
```

The next example is an array declarator that declares an array with the symbolic name AMPS and three dimension declarators. The dimension declarators do not include lower-bound specifications. Therefore, each dimension has an implied lower bound of 1. AMPS is a 3-dimensional array. The first two dimensions have an upper bound of 8 and the third dimension has an upper bound of 2. AMPS consists of 128 elements.

```
AMPS (8,8,2)
```

Each element in an array is a variable and can assume different values during program execution. A program can access the values in the array by referencing the element name in an expression. To reference an element name, specify the name of the array followed by a subscript. A subscript consists of one or more arithmetic constant expressions enclosed in parentheses. Subscript expressions must evaluate to integers. The format for specifying an array element is:

symbolic name (*subscript expression* [,*subscript expression*]...)

The number of subscript expressions you specify in an element name reference must match the number of dimension declarators specified in the array declarator.

You can reference an array name without a subscript in the following GLS FORTRAN statements. Unsubscripted array names can also be used as arguments in a reference to a subroutine or external function.

- CHARACTER
- COMMON
- COMPLEX
- DATA
- DOUBLE PRECISION
- ENTRY
- EQUIVALENCE
- FUNCTION

- Input/Output (I/O) statements
- INTEGER
- LOGICAL
- REAL
- SAVE
- SUBROUTINE

One–Dimensional Arrays

A one-dimensional array organizes data in linear form according to one criterion. For example, consider a program that calculates the average high meteorological temperature for one week. The program must store seven temperature readings, one for each day of the week, then calculate the average high. To store the temperatures, declare a one-dimensional integer array with the symbolic name `TEMPS` and enclose the number 7 in parentheses to specify an upper bound of seven elements. To declare the integer data type, use the array declarator in an INTEGER statement as follows:

```
INTEGER TEMPS(7)
```

Each element in `TEMPS` is a variable numbered 1 through 7 that can assume an assigned temperature value. You can assign initial values to array elements with the DATA statement. During program execution, you can assign values to array elements with assignment or input statements. The following table represents the structure of array `TEMPS` with assigned values.

	SUN	MON	TUE	WED	THU	FRI	SAT
TEMPS	63	60	55	68	72	73	64
	(1)	(2)	(3)	(4)	(5)	(6)	(7)

The program can access any temperature value in the `TEMPS` array for a calculation by referencing the array element name. To reference an element name, you specify the array name followed by a subscript. For example, `TEMPS(3)` refers to the third element in `TEMPS`, which has been assigned the value 55 degrees. The number 3 is a subscript expression. The number 3 including the parentheses constitutes the entire subscript. The number of subscript expressions you specify in an element name reference must match the number of dimensions specified in the array declarator.

Each array element in `TEMPS` is a variable that can take on new values. Therefore, at the end of each week you can assign new temperature readings to the corresponding array elements and execute the program to calculate the average high for the new week.

Multidimensional Arrays

A multidimensional array allows you to organize data according to more than one criterion. For example, suppose you want to expand the temperature program to calculate the average body temperature for a medical patient for one week using three temperature readings taken each day instead of one. There are two criteria upon which to organize the temperature values: the day of the week and the time of the day.

The program must store 21 temperature readings, three for each day of the week, then calculate the average. To store the temperature values, declare a 2-dimensional array with the symbolic name `TEMPS2`. The real data type for `TEMPS2` must be used because body temperatures for medical patients must be accurate to a tenth of a degree. To declare the real data type, use the array declarator in a REAL statement as follows:

```
REAL TEMPS2(7,3)
```

The two dimension declarators are enclosed in parentheses following the symbolic name `TEMPS2`. The first dimension has an upper bound of seven elements and the second dimension has an upper bound of three elements. Both dimensions have an implied lower bound of 1.

Use the DATA statement to assign initial values to the elements in `TEMPS2`. During program execution, you can assign values to array elements with assignment or input statements. The following table represents the structure of array `TEMPS2` with assigned values:

TEMPS2	SUN	MON	TUE	WED	THU	FRI	SAT	
6 A.M.	103.4	103.4	103.2	103.1	101.1	102.9	103.3	(1)
2 P.M.	103.0	102.3	100.4	99.7	99.1	98.8	98.7	(2)
10 P.M.	99.2	100.2	101.6	102.9	100.7	99.2	98.6	(3)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	

To reference an element in a two-dimensional array, specify two subscript expressions after the array name. For example, `TEMPS2(4,2)` refers to the fourth element in the first dimension and the second element in the second dimension – 99.7 degrees taken at 2 P.M. on Wednesday.

Adjustable Array Declarators

An adjustable array declarator is when you use a variable in a dimension bound specification. This enables program units to pass various sized arrays as arguments. The adjustable array declarator serves as a dummy array in a subroutine or function procedure. The reference to the procedure contains the actual array size specifications. The adjustable array declarator must specify the same number of dimensions as the actual argument array.

The following example shows an adjustable array declarator named ADJ used in a DIMENSION statement. The DIMENSION statement is in a subroutine named TEST. The adjustable array declarator name, ADJ, and the two variables used as dimension bound values, M and N, are dummy arguments for the TEST subroutine. M and N are the adjustable dimensions.

```
SUBROUTINE  TEST(ADJ,M,N)
    .
    .
DIMENSION  ADJ(M,N)
    .
    .
END
```

The following example declares two arrays, ACT1 and ACT2, and contains two calls to the TEST subroutine defined above. Each call passes a different array to TEST for processing. The first CALL statement passes the array name ACT1 and the two dimension declarators 5 and 10 as arguments. The second CALL statement passes the array name ACT2 and the dimension declarators 25 and 50 as actual arguments.

```
DIMENSION  ACT1(5,10)
DIMENSION  ACT2(25,50)
    .
    .
CALL TEST(ACT1,5,10)
    .
    .
CALL TEST(ACT2,25,50)
    .
    .
END
```

ACT1 and ACT2 are different sized arrays, but the adjustable array declarator, ADJ, enables the TEST subroutine to process both arrays one at a time.

Assumed–Size Array Declarators

Assumed-sized array declarators, like adjustable array declarators, serve as dummy arrays in a function or subroutine procedure. An assumed-size array declarator uses an asterisk (*) as an upper dimension bound for the last dimension declared in the array. The actual upper dimension bound passes to the procedure from the procedure reference. Dimension bound values in an assumed-size array declarator other than the upper bound of the last dimension can be integer constant or variable expressions.

The following example shows the assumed-size array declarator ASM used in a DIMENSION statement. The DIMENSION statement is in a function named CALC. In this example, the lower dimension bounds for both dimensions are integer constants. The upper bound for the first

dimension is the variable W. The upper bound for the second dimension is an asterisk. ASM serves as a dummy array within the CALC function. ASM and W are dummy arguments for CALC.

```
FUNCTION  CALC(ASM,W)
    .
    .
DIMENSION  ASM(1:W,1:*)
    .
    .
END
```

The following example contains a reference to the CALC function defined in the previous example. The reference passes the array ACT for processing. The reference to CALC passes ACT and the value 10 as arguments. The actual upper bound for the second dimension, 30, does not pass as an argument.

```
DIMENSION  ACT(10,30)
    .
    .
VALUE = CALC(ACT,10)
    .
    .
END
```

The assumed-size array declarator, ASM, assumes the size of the array, ACT, passed to the CALC function in the function reference.

Substrings

A substring is a contiguous portion of the space a character variable or character array element represents. Substring references enable you to manipulate segments of character strings in a program. A substring is the character data type.

Substring references have two forms: one for a character variable and one for a character array element. A substring reference for a character variable is specified in the following format:

variable name([*1st expression*]:[*2nd expression*])

The character positions a character variable represents are numbered from left to right, beginning with 1. *1st expression* specifies the first or leftmost character of the substring you want to reference. *2nd expression* specifies the last or rightmost character of the substring you want to reference. For example, the following substring reference specifies character positions 3 through 7 in the character variable materials:

```
materials(3:7)
```

`materials` can take on a variety of values during program execution. However, the substring reference always specifies character positions 3 through 7 regardless of the value of `materials` at any given time.

The format for a substring reference for a character array element is:

array name(*sub*[, *sub*]...) ([*1st expression*]:[*2nd expression*])

Like a substring reference for a character variable, the character positions a character array element represents are numbered from left to right, beginning with 1. *1st expression* in the substring reference specifies the first or leftmost character of the substring you want to reference. *2nd expression* specifies the last or rightmost character you want to reference. *sub* is the subscript expression. You can specify any number of subscript expressions in a substring reference. For example, the following substring reference specifies character positions 5 through 10 of an element in the 3-dimensional character array named `products`:

```
products (4,4,12) (5:10)
```

For both character variable and character array element references, *1st expression* must be greater than or equal to 1 and less than or equal to *2nd expression*. *2nd expression* must be less than or equal to the length of the variable or array element. Expressions that do not evaluate to integers convert to integers.

If you omit *1st expression*, a leftmost character position of 1 is implied. If you omit *2nd expression*, a rightmost character position equal to the length of the variable or array element is implied. To omit both expressions implies a reference to all the character positions in the variable or array element. If you omit both expressions, you must still specify the colon enclosed in parentheses. The following are examples of substring references.

```
versions(2:8)
items(5:)
fourth(3,9)(:11)
sysform(:)
```

Chapter 5

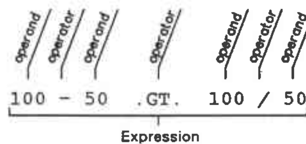
Expressions

This chapter describes expression types and their conventions.

Overview

An expression is a character sequence that specifies instructions for calculating a value. It can be a data item such as a constant or variable, or a combination of data items and operators. Operators are special characters that specify computations to perform using values given in the expression. Operands are the values an operator processes. All expressions represent or evaluate to a single value.

In the following expression, 100 and 50 are operands and – , .GT., and / are operators. The expression evaluates to true.



Expression Types

Table 5–1 lists the four GLS FORTRAN expression types, their set of operators, and the order in which operands are evaluated when combined in expressions that contain more than one kind of operator.

Table 5–1. Expression Summary

Expression	Operators	Evaluation Order
Arithmetic	+, –, /, *, **	1 (highest)
Character	//	2
Relational	.LT., .LE., .EQ., .NE., .GT., :GE.	3
Logical	.NOT., .AND., .OR., .XOR., EQV., NEQV.	4 (lowest)

Arithmetic Expressions

Arithmetic expressions represent numeric values. An arithmetic expression uses a special set of operators, operands, and parentheses to control the evaluation order of the operations specified in the expression. Valid arithmetic operators are show in Table 5–2.

The *, /, and ** operators work with two operands and are called binary operators. The + and – operators can work as binary operators or as unary operators that work on a single operand.

The standard rules of algebra are used to determine the evaluation order of two or more operators in arithmetic expressions.

Table 5–2. Arithmetic Expressions

Operator/Function		Example		Evaluation Order
+	Addition	OP1 + OP2 +OP1	(Add OP1 and OP2) (Identify OP1 as positive)	3 (lowest)
–	Subtraction	OP1 – OP2 –OP1	(Subtract OP2 from OP1) (Identify OP1 as negative)	3
*	Multiplication	OP1 * OP2	(Multiply OP1 by OP2)	2
/	Division	OP1 / OP2	(Divide OP1 by OP2)	2
**	Exponentiation	OP1 ** OP2	(Raise OP1 to the power OP2)	1 (highest)

Multiple Operators

When an expression contains two or more operators of equal precedence, such as * and /, operations are evaluated algebraically from left to right.

Exponentiation is evaluated from right to left. In the following expression, GLS FORTRAN first calculates OP2 raised to the power indicated by OP3, then calculates OP1 raised to the power indicated by the value that results from the first calculation.

```
OP1 ** OP2 ** OP3
```

To supercede the original hierarchy of evaluation of operators, you can add parentheses. Parentheses specify the part of the expression to evaluate first. The following examples show the evaluation order of multiple operators when parentheses are used.

```
25  +  15  -  10  +  12 = 42            (25  +  15)  -  (10  +  12) = 18
    ↑      ↑      ↑                          ↑       ↑        ↑
   1st    2nd    3rd                        1st     3rd      2nd

3  +  4  *  3  -  5  =  10              3  +  4  *  (3  -  5)  =  -5
   ↑     ↑     ↑                           ↑     ↑      ↑
  2nd   1st   3rd                         3rd   2nd    1st

100  -  10  **  2  *  3  =  -200        (100  -  10  **  2)  *  3  =  0
     ↑      ↑      ↑                          ↑      ↑       ↑
    3rd    1st    2nd                        2nd    1st     3rd

11  +  3  *  2  -  10  /  2  =  12      ((11  +  3)  *  2  -  10)  /  2  =  9
    ↑     ↑     ↑      ↑                      ↑      ↑     ↑       ↑
   3rd   1st   4th    2nd                    1st    2nd   3rd     4th
```

Valid Operands

The following operands are valid in an arithmetic expression:

- Unsigned numeric constants
- Symbolic name of an unsigned numeric constant
- Numeric variable reference
- Numeric array element reference
- Arithmetic function reference
- Arithmetic expression enclosed in parentheses

Data Type Evaluation

The data type of an arithmetic expression is determined by the data types of the operands in the expression. An arithmetic expression that contains operands of one data type evaluates to a value of that type. An arithmetic expression that contains operands of two or more different data types evaluates to the highest ranking type in the expression. Operands of low ranking data types automatically convert to the higher ranking types. The one exception to this hierarchy is that the combination of a REAL*8 (DOUBLE PRECISION) value with a COMPLEX*8 (COMPLEX) value causes both operands to be converted to COMPLEX*16 (DOUBLE COMPLEX).

Table 5–3 defines the hierarchy of data types for conversion within expressions:

Table 5–3. Data Type Hierarchy

Data Type	Rank
COMPLEX*16	1 (highest)
COMPLEX*8 (COMPLEX)	2
REAL*8 (DOUBLE PRECISION)	3
REAL*4 (REAL)	4
INTEGER*4	5
INTEGER*2 (INTEGER)	6
INTEGER*1	7
LOGICAL	8 (lowest)

According to this hierarchy, an arithmetic expression that consists of real and complex operands evaluates to a complex value. An arithmetic expression that consists of INTEGER*2 and INTEGER*4 operands evaluates to an INTEGER*4 value. The following rules apply to the conversion of data types within arithmetic expressions:

- In an expression that contains integer and real operands, the integer operands first receive a fractional component of 0 in the conversion to real. GLS FORTRAN evaluates the expression using real arithmetic. Consider the expression (10/5)*3.14. GLS FORTRAN first evaluates the integer division, (10/5), then converts the result of the division to real.

- To convert an operand of one real data type to a real data type with a higher precision, GLS FORTRAN uses the existing operand as the most significant portion of the higher precision data item and the least significant part of the data item becomes zero. GLS FORTRAN then evaluates the expression using the higher precision arithmetic.

- In an expression that contains complex and integer operands, the integer operands convert to real as described above. The converted real operand then serves as the real part of a complex number and the imaginary part becomes zero. GLS FORTRAN then evaluates the expression using complex arithmetic. The expression evaluates to a complex value.

- Any fractional component that results from a division of integers is truncated, not rounded. For example, the expression (1/2 + 1/2) is equal to 0, not 1. The expression (12/5) is equal to 2. The fractional components are truncated.

Character Expressions

Character expressions allow you to manipulate character strings. A character expression uses character operands and a special character operator. All character expressions evaluate to a single character string value.

The character operator consists of two slashes, //, and is called the concatenation operator. The concatenation operator joins two character operands together. For example, the following character expression evaluates to the character string value `'FORTRAN'`:

```
'FOR' // 'TRAN'
```

Valid Operands

The following operands are valid within a character expression:

- Character constant
- Symbolic name of a character constant
- Character variable reference
- Character array element reference
- Character substring reference
- Character expression enclosed in parentheses
- Character function reference

Using parentheses does not affect the value of a character expression. For example, the following character expressions are equivalent:

```
'PRESS' // 'ANY KEY' // 'TO CONTINUE'
('PRESS' // 'ANY KEY') // 'TO CONTINUE'
'PRESS' // ('ANY KEY' // 'TO CONTINUE')
```

The length of a character expression equals the sum of the lengths of the individual operands. The length value includes any spaces that are parts of an operand. Parentheses are not considered part of a character expression and are not included in the length value. For example, the following expression has a length of 19:

```
'ENTER' // ' "b/"YOUR "b/"PASSWORD'
```

Relational Expressions

A relational expression compares the values of two operands using the relational operators (shown in Table 5–4), and then evaluates to true or false. All relational operators have equal precedence.

Table 5–4. Relational Operators

Operator	Meaning
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

There are two types of relational expressions: arithmetical and character. A single relational expression can compare two arithmetical expressions, or two character expressions; it cannot compare an arithmetic expression with a character expression. Arithmetic and character operators are evaluated before relational operators.

Arithmetical

In an arithmetic relational expression, the arithmetic operands are evaluated first; then the resulting values are compared to determine if the relationship specified by the operator exists.

In the following example, the arithmetic expressions on either side of the .GT. operator are evaluated first. Then the entire expression is evaluated for validity. In this case, 50 is greater than 2 so the value of the relational expression is true.

```
100  -  50   .GT.    100 / 50
     ↑                   ↑
     50                  2
```

You can use parentheses to change the evaluation order of the arithmetic operands of a relational expression.

Character

In a character relational expression, character operands are evaluated first, according to the ASCII character collating sequence described in Appendix A. The length of the character operands is not significant for comparison. (If the two character operands have different lengths, the shorter operand is padded on the right with blank characters until the two strings are equal. The two strings are then compared a character at a time according to the ASCII collating sequence.) After sequence evaluation, the resulting values are compared to determine if the relationship specified by the operator exists.

In the following example, the character expressions on either side of the .LT. operator are evaluated. Because the string 'APPLE' has a length of 5, and 'APRICOT' a length of 7, 'APPLE' is padded on the right with two blank characters to make the strings equal in length. The two strings are then compared a character at a time. The third letters of each string are evaluated according to the collating sequence and P is less than R; therefore, the string 'APPLE' is less than the string 'APRICOT', and the relational expression is true.

```
'APP' // 'LE' .LT. 'AP' // 'RICOT'
```

Complex

Complex expressions are compared with the .EQ. and .NE. operators only. Two complex values are considered equal only if their corresponding real and imaginary parts are both equal.

A relational operator can compare two numeric expressions of different data types. However, prior to making the comparison, the value of the expression with the lower ranked data type is converted to the higher ranked data type. For example, when a REAL*8 and a COMPLEX*8 value are compared, the values are first converted to the type COMPLEX*16 before making the comparison.

Logical Expressions

Logical expressions compare logical values using the logical operators shown in Table 5–5. When a logical expression has two or more logical operators, they are evaluated in the order shown in table. The operands in a logical expression evaluate to true or false.

Table 5–5. Logical Operators

Operator	Meaning	Evaluation Order
.NOT.	Logical negation	1 (highest)
.AND.	Logical conjunction	2
.OR.	Logical inclusive disjunction	3
✝✝.XOR.	Equivalent to .NEQV	4 (lowest)
.EQV.	Logical equivalence	4
.NEQV.	Logical nonequivalence	4

The following examples show the use logical operators in expressions. OP1 denotes an operand for a unary operator or an operand to the left of a binary operator; OP2 denotes an operand to the right of a binary operator.

Examples

```
OP1 .AND OP2
```
The expression is true only if both OP1 and OP2 are true.

```
OP1 .OR. OP2
```
The expression is true if either OP1 or OP2, or both, are true.

```
OP1 .EQV. OP2
```
The expression is true only if both OP1 and OP2 have the same logical value: either true or false.

```
OP1 .NEQV. OP
```
The expression is true if OP1 is true and OP2 is false, or if OP2 is true and OP1 is false. The expression is false if both operands have the same value.

```
OP1 .XOR. OP2
```
✝✝The .XOR operator is equivalent to NEQV.y.

```
.NOT.OP1
```
The expression is true only if OP1 is false.

Parentheses are used to control the evaluation order of operations in a logical expression. In the first example, the operations are evaluated algebraically from left to right, and the expression evaluates to true. In the second example, parentheses are change the evaluation order, so the expression evaluates to false.

```
6*3+4 .LT. 25 .AND. 4 .LE. 8/2
6*(3+4) .LT 25 .AND. 4 .LE. 8/2
```

Two logical operators cannot appear consecutively within an expression unless the second operator is .NOT. The following is an example of a valid expression that evaluates to true.

```
3.14159 .LE. 10 .AND. .NOT. 10/5 .EQ. 3
```

Valid Operands

The following operands are valid forms within a logical expression:

- Arithmetic relational expression
- Character relational expression
- Logical constant (.TRUE. or .FALSE.)
- Symbolic name of a logical constant
- Logical variable reference
- Logical array element reference
- Logical function reference

Integer Operands

†† Logical operations can be performed on integer operands, and are carried out bit by bit on the internal values. Logical and integer operands can be used in combination; first the logical operand is converted to an integer value, then the logical operation is performed.

Similarly, logical variables and/or expressions can be used in an integer context within an expression. The logical expression is converted to an integer value before the overall expression is evaluated.

Value of A	Value of B	Operation	Result
0	255	C=A .AND. B	C= 0
0	255	C=.NOT. B	C=-256
43	-6	C=A .OR. B	C= -5

Chapter 6

Program Structure

This chapter describes the main program, the different classes of subprograms, and the relationships among program units that combine to form GLS FORTRAN executable programs.

An executable GLS FORTRAN program consists of one or more program units. A program unit is a logical, self-contained sequence of statements and optional comment lines that form a discrete part of a larger program. There are two kinds of program units: main programs and subprograms. Figure 6–1 shows the GLS FORTRAN program structure.

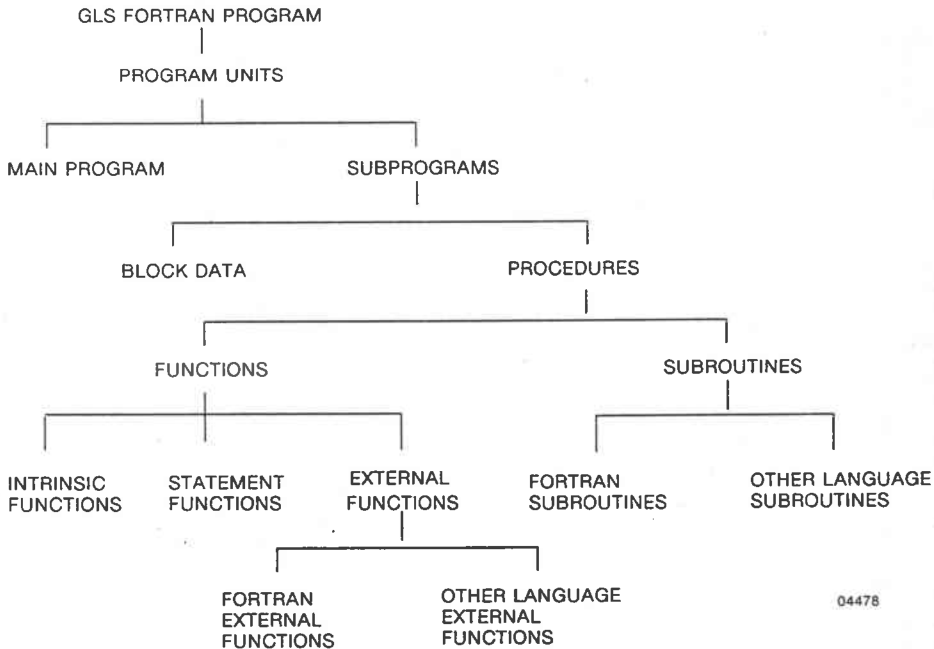


Figure 6–1. GLS FORTRAN Program Structure

Main Program

The main program serves as the center or base of all processing activity in an executable program. It receives control to begin execution. During execution, the main program can invoke a variety of subprograms that perform different tasks. Control returns to the main program to terminate execution except when a STOP statement executes. For more information about program termination, refer to the END and STOP statements in Chapter 9.

An executable GLS FORTRAN program can consist of a main program with no subprograms. Each executable program can have only one program unit defined as the main program. Subprograms cannot call or reference the main program and the main program cannot call or reference itself.

Use the PROGRAM statement to define a program unit as a main program. The PROGRAM statement is not required, but if used, it must be the first statement of the main program. The PROGRAM statement specifies a symbolic name for the main program as shown in the following syntax specification:

PROGRAM *symbolic name*

The main program can contain any of the GLS FORTRAN statements except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, and RETURN statements.

Subprograms

The term subprogram covers two kinds of program units: block data subprograms and procedures. Block data subprograms initialize variables and array elements, and cannot contain executable statements. Procedures contain executable statements that define a specific computing operation. Subprograms cannot call or reference the main program. An executable GLS FORTRAN program must contain only one main program but can contain any number of subprograms.

Block Data

A block data subprogram enables you to specify initial values for variables and array elements that are listed in named common blocks. Common blocks are storage areas that contain data that a number of program units can share. Any program unit that contains a definition of a given common block can use the data in that block. Use the COMMON statement to define common blocks.

Common blocks can be either named or unnamed. However, only variables and array elements in named common blocks can be initialized in a block data subprogram. You can initialize data from several named common blocks in one block data subprogram. Refer to the COMMON statement in Chapter 9 for more information on common storage area management.

You identify block data subprograms with the BLOCK DATA statement. The BLOCK DATA statement has the following form:

BLOCK DATA [*symbolic name*]

symbolic name is a global reference for the subprogram. The name is optional, but if used it must be unique. There cannot be more than one unnamed block data subprogram in an executable program. The BLOCK DATA statement must be the first statement in a block data subprogram.

A block data subprogram can contain only the following specification statements: COMMON, DIMENSION, DATA, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and type statements. You can also include comment lines. The last statement in a block data subprogram must be the END statement. An executable GLS FORTRAN program can contain any number of block data subprograms. Block data subprograms have the following form:

BLOCK DATA [*symbolic name*]

.
. *specification statements and comments*
.

END

The following block data subprogram example defines three named common blocks for a program that calculates weather information. The example specifies a data type for each variable in the named common blocks and declares initial values for some of the variables.

```
      BLOCK DATA weather

C Define named common areas

      COMMON /time/day,hour,min  /location/zone,altitude
      COMMON /conditions/temp,humidity,pressure,wind,rain

C Declare data types for the variables

      INTEGER day,hour,min,zone,altitude,temp,humidity,wind
      REAL pressure
      LOGICAL rain

C Declare initial values for some of the variables

      DATA day/31/, hour/24/, zone/7/, altitude/5285/
      DATA temp/65/, pressure/29.92/, rain/.true./
      END
```

You must specify all the variables in the named common blocks using specification statements even if you do not declare initial values for all the variables with the DATA statement.

Procedures

A procedure is a subprogram that performs a specific task within an executable program. Unlike block data subprograms, procedures contain the executable statements that define the purpose of the program. Procedures structure programs into a series of routines that can execute repeatedly, in any order, to accomplish a larger task. An executable GLS FORTRAN program can contain any number of procedures.

A procedure receives execution control through a call or reference. Procedures can receive control from the main program or from another procedure. However, procedures cannot call or reference the main program. Procedures can share values through the use of arguments and common blocks.

There are two types of procedures in GLS FORTRAN: subroutines and functions. Subroutines and functions differ primarily in the method by which they are invoked during execution and in the result they produce. Functions are further classified into three categories: external, statement, and intrinsic. Subroutines and external functions are collectively referred to as external procedures.

Procedure Arguments

Arguments supply the values that a procedure requires to produce the desired result. A program unit, such as the main program, can invoke a procedure to perform a specific task using arguments to pass the values the procedure needs to complete the task. Both subroutine and function procedures can change the values of the arguments during execution. Therefore, values the procedure produces can return to the invoking program unit via the arguments. Arguments are sometimes called parameters.

There are two types of arguments: dummy or formal arguments and actual or calling arguments. Dummy arguments are used in the procedure definition to reserve a place and declare a data type for actual values the procedure requires to produce the desired result. Actual arguments are used in the procedure call or reference and are substituted for the dummy arguments during procedure execution.

Dummy arguments used in the procedure definition must correspond in number, order, and data type with the actual arguments in the procedure call or reference. Depending on the kind of procedure, a dummy argument can be a variable name, an array name, a dummy procedure name, or an alternate return specifier.

A variable name that serves as a dummy argument can be associated only with an actual argument that is a variable, a constant, a symbolic name of a constant, a function reference, an array element, a substring, or an expression.

An array name that serves as a dummy argument can be associated only with an actual argument that is an array, array element, or array element substring of matching data type.

Subroutines

A subroutine is an external procedure that performs a specific task within an executable GLS FORTRAN program. An external procedure is a program unit that is defined outside the program unit that invokes it. You can write external procedures using a programming language other than GLS FORTRAN, such as C or assembly language. For more information, refer to the *GLS Programming Guide*.

Use the SUBROUTINE statement to define a program unit as a subroutine. The SUBROUTINE statement must be the first statement in the subroutine. The SUBROUTINE statement specifies a symbolic name for the subroutine and a list of dummy arguments the subroutine requires.

The syntax for the SUBROUTINE statement is:

SUBROUTINE *symbolic name* [(*dummy* [, *dummy*] ...)]

where a dummy argument can be one of the following:

- Variable name
- Array name
- Dummy procedure name
- Asterisk (alternate return specifier)

A dummy procedure name enables actual procedure names to be passed as arguments.

A subroutine can receive control of execution from the main program or from another procedure. A subroutine cannot invoke itself. Execution control transfers to a subroutine through the CALL statement. The CALL statement must specify the symbolic name of the subroutine you want to invoke and a list of actual arguments the subroutine requires. Actual arguments specified in the CALL statement must correspond in number, order, and data type to the dummy arguments specified in the SUBROUTINE statement. For more information, refer to the CALL statement in Chapter 9.

An actual argument in a CALL statement can be one of the following:

- Expression
- Array name
- Intrinsic function name
- External procedure name
- Dummy procedure name
- Alternate return specifier using the statement label of an executable statement in the same program unit as the CALL statement

Functions

A function is a procedure that performs a specific task within an executable GLS FORTRAN program. Execution control transfers to a function through a reference. (You cannot use the CALL statement to invoke a function.) To reference a function means to use the function name within an expression. Functions can receive control of execution from the main program or from another procedure. A function cannot reference itself.

Unlike a subroutine, a function is specifically designed to return a single value to the program unit that contains the function reference. The function assigns this return value to the function name. Therefore, the return value becomes the value of the function. When the function name appears in an expression, the value of the function is used in the evaluation of the expression. The function name determines the data type for the return value.

There are three types of functions: external, statement, and intrinsic. To reference a function, you must specify the symbolic name and any actual arguments the function requires. Use the following format to reference a function:

symbolic name [(*actual* [, *actual*]...)]

Actual arguments used in the function reference must correspond in number, order, and data type with the dummy arguments in the function definition. Actual arguments in the function reference can be any one of the following:

- Expression
- Array name
- Intrinsic function name
- External procedure name
- Dummy procedure name

External Functions

An external function, like a subroutine, is a program unit that is defined outside the program unit that invokes it. You can write external functions in a language other than GLS FORTRAN, such as C or assembly language. For more information, refer to the *GLS Programming Guide.*

Use the FUNCTION statement to define a program unit as an external function. The FUNCTION statement must be the first statement in the external function. The FUNCTION statement specifies a symbolic name for the external function, a data type for the value the function returns, and a list of dummy arguments the function requires.

The syntax for the FUNCTION statement is:

type FUNCTION *symbolic name* [(*dummy* [, *dummy*] ...)]

where a dummy argument can be one of the following:

- Variable name
- Array name
- Dummy procedure name
- Asterisk (alternate return specifier)

Statement Functions

A statement function is a procedure that is completely defined in one statement. Unlike an external function, you can reference a statement function only within the program unit that contains the statement function definition. A statement function consists of a symbolic name to identify the function, a list of dummy arguments the function needs, and an expression:

symbolic name ([*dummy* [, *dummy*] ...]) = *expression*

A statement function is structured much like an assignment statement. The data type of the expression converts to the type of the symbolic name according to the rules for conversion described in the Arithmetic Assignment statement Chapter 9. The symbolic name is the statement function name.

The dummy arguments reserve a place and declare a data type for actual values the statement function requires to produce the desired result. Actual arguments are used in the statement function reference and are substituted for the dummy arguments during the actual procedure execution. The dummy arguments in a statement function are local to the statement function. You

can use the dummy argument names to represent other entities in the same program unit. (You cannot use the statement function name to represent another entity within the same program unit.)

Statement functions are referenced in an expression. Actual arguments specified in a statement function reference must correspond in number, order, and data type with the dummy arguments in the statement function definition.

Other functions can be referenced within the expression of a statement function. However, the functions you reference in a statement function expression must be defined before that statement function in the same program unit. The definition of a statement function and all references to that statement function must be in the same program unit.

Alternate Return Specifiers

A CALL statement transfers execution control to a subroutine. A RETURN or END statement transfers execution control from the subroutine back to the program unit that contains the CALL statement. A normal return from a subroutine is when execution control transfers to the first executable statement following the CALL statement in that calling program unit. You can, however, specify alternate return points from subroutines using alternate return specifiers. Alternate return specifiers operate through the passing of arguments between a calling program unit and a subroutine.

An alternate return specifier consists of an asterisk (*) that you specify as a dummy argument to a subroutine in a SUBROUTINE or ENTRY statement. The corresponding actual argument used in a CALL statement must be a statement label. The statement label identifies an executable statement that serves as an alternate return point for the subroutine. You can specify any number of alternate return specifiers for a subroutine.

The RETURN statement has an optional integer expression used to select one alternate return specifier from a series of specifiers. The integer expression indicates which asterisk or ††ampersand in the SUBROUTINE or ENTRY statement dummy argument list to use for the return. A valid integer expression must be greater than or equal to 1 and less than or equal to the number of asterisks or ampersands specified in the SUBROUTINE or ENTRY statement dummy argument list. (Each asterisk in the dummy argument list corresponds to an actual argument supplied in the CALL statement that invokes the subroutine.) The actual arguments must be statement labels that indicate the alternate return points.

For example, the following subroutine contains three RETURN statements. The first RETURN statement, statement 110, specifies a normal return from the subroutine because there is no specified integer expression. The second and third RETURN statements, statements 120 and 130, have integer expressions indicating alternate returns.

Statement 100 is an arithmetic IF statement that determines which of the three RETURN statements will execute. The SUBROUTINE statement contains two alternate return asterisks that serve as dummy arguments to the subroutine named `thrust`.

```
        SUBROUTINE thrust (var1, *, *, var2)
             .
             .
             .
100     IF (dat/val) 110, 120, 130
110     RETURN
120     RETURN 1
130     RETURN 2
```

If the expression in the arithmetic IF statement evaluates to less than zero, the first RETURN statement, statement 110, executes. The first RETURN statement specifies a normal return. If the expression evaluates to zero, the second RETURN statement, statement 120, executes. The second and third RETURN statements specify 1 and 2, indicating the first and second alternate return asterisk in the dummy argument list, respectively. If the expression evaluates to greater than zero, the third RETURN statement, statement 130, executes.

The following CALL statement calls the `thrust` subroutine. The second and third actual arguments in the CALL statement are statement labels that correspond to the two dummy argument asterisks in the SUBROUTINE statement.

```
    CALL thrust (2.86, *200, *300, 4.13)
```

If the second RETURN statement in the subroutine executes, execution control transfers to the statement identified with the label 200. This happens because the first dummy argument asterisk in the SUBROUTINE statement corresponds to the actual argument 200 in the CALL statement. If the third RETURN statement in the subroutine executes, execution control transfers to the statement identified with the label 300. This happens because the second dummy argument asterisk corresponds to the actual argument 300. Statements 200 and 300 exist in the calling program and are alternate return points selected on the basis of the arithmetic IF statement.

If the integer expression used in a RETURN statement is less than 1 or greater than the number of asterisks in the dummy argument list, a normal return from the subroutine is executed.

†† An ampersand can be used in place of an asterisk in an argument list to indicate an alternate return statement.

```
      CALL UPDATE (I, J, &100, K)
        ...
100     CONTINUE
```

is equivalent to:

```
      CALL UPDATE (I, J, *100, K)
        ...
100     CONTINUE
```

Chapter 7

FORTRAN I/O

This chapter describes the GLS FORTRAN input/output (I/O) system, records, files, I/O units, and data transfer.

Overview

GLS FORTRAN provides a device-independent I/O system for transferring data from one location to another within a processor's memory. It can also transfer data to and from processor memory and any external device such as a console, printer, or a storage medium such as a disk or magnetic tape.

Records

A record is a logically related set of data items. There are three kinds of records:

- Formatted

  A sequence of ASCII characters; the length of a formatted record is the number characters it contains, and depends on the number of characters written to the record when created; length can be zero and is measured in bytes; records can only be access with formatted I/O statements

- Unformatted

  A sequence of items having any combination of data types; the length of unformatted records is specified when created; length can be zero and is measured in bytes; records can only be access with unformatted I/O statements

- Endfile

  The last record in a file, written with the ENDFILE statement

Files

A file is a sequence of records. The records in a file are either all formatted or all unformatted. A file cannot contain both kinds of records.

Each record in a file is assigned a unique record number by the I/O system when it is created. There are two types of files:

- External

  External files contain data that can be transferred between internal storage and any external device. Also, external files can be permanently stored on external storage medium, such as a hard disk or magnetic tape.

- Internal

  Internal files are an area of internal storage. They are implemented as a character variable, a character array, or an element of a character array.

The physical size of a file is the number of records in the file multiplied by the length of record, measured in bytes.

The number of files that can be open concurrently depends on limits set by the operating system. For example, on REAL/IX 97xx systems this is a modifiable parameter. For detailed information, refer to the documentation supplied with your operating system.

I/O Units

An I/O unit is a logical or generic designation for a file. At any given time, an I/O unit can designate one of several different files.

Internally, the GLS FORTRAN I/O system uses I/O units when transferring data or manipulating files, so the I/O statements described later in this chapter generally refer to I/O units rather than names of files.

An I/O unit is designated by an unsigned integer from 0 through 99.

Connection

Connection defines the relationship between the I/O unit and a file. An I/O unit is connected when it refers to a specific file; it is disconnected when it does not refer to a specific file.

When an I/O unit is connected to a file, the file is also connected to the I/O unit. An I/O unit cannot be connected to more than one file at a time, nor can a file be connected to more than one I/O unit at a time.

All GLS FORTRAN I/O statements, except OPEN and CLOSE, operate on I/O units that are connected to files on a one-to-one basis. No data transfer can take place on a file unless it is connected to an I/O unit. The I/O unit/file connection must be made when the file is opened, or by preconnection.

Preconnection

An I/O unit is preconnected if by some means external to the program, it is connected to a specific file before the program begins to run. Each GLS FORTRAN program has three preconnected I/O units:

- Unit 0 – default error output (**stderr**)
- Unit 5 – default console input (**stdin**)
- Unit 6 – default console output (**stdout**)

Access Method

There are two methods for accessing a file:

- Sequential – records are accessed in the order in which they were created
- Direct – records are accessed in any order

You can access an external file using either access method, but you can only access an internal file sequentially.

If a file is connected for direct access, all the records must have the same length. A file connected for sequential access can have records with different lengths.

Properties of Files

Each file has an associated set of properties, some of which are determined by the OPEN statement at the time of connection to an I/O unit. This section discusses the properties of existence and position.

Existence

Existence is the potential an executable program has to access the file. The files a program can potentially access are said to exist for that program. The files a program cannot access do not exist. For example, a file could be protected for security reasons, or a file could be in use by another program. In such a case, the file is inaccessible to the program, and therefore does not exist.

The property of existence applies only to a file in relation to a particular executable program. It does not apply to the file's physical existence within the environment of an implementation.

The following list describes all possible combinations of connection and existence:

- A file can exist and be connected

  Example: a disk file that is being written to

- A file can exist and not be connected

  Example: a disk file that is not yet open

- A file can be connected but not exist

  Example: a newly created disk file before the first record is written

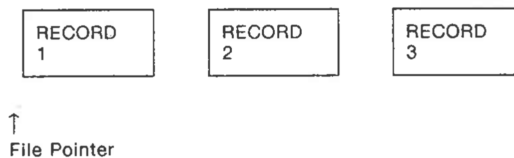
- A file can both not exist and not be connected

  Example: a disk file that was erased

Position

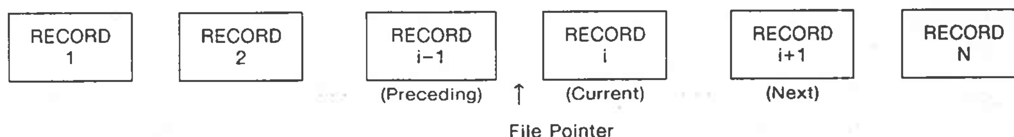
When a file is connected to an I/O unit, it has a position. The file position is determined by a file pointer. The file pointer is not a physical entity; rather, it is an internal reference the I/O system uses to keep track of the file position.

In the following diagram, the file is positioned at the initial point just before the first record.



In the following diagram, the file is positioned at the terminal point just after the last record.

In the following diagram, the file is positioned at a specific record, the current record. If the file is not positioned at or within a record, there is no current record.



If a file contains N records, and the file pointer points to record i, where 1 ≤ i ≤ N, then record i+1 is the next record, and record i−1 is the preceding record. If i=1, there is no preceding record. If i≤N or if N=0, there is no next record.

I/O Statements

This section describes I/O statements in the following categories: Data transfer, File positioning, and Auxiliary. Another category, Format statements, are described in Chapter 8.

Data Transfer Statements

Data transfer statements move data between internal (processor) storage and a file; they can reference both internal and external files. The data transfer statements are:

- READ
  Inputs data from a specific I/O unit
- WRITE
  Outputs data to a specific I/O unit
- PRINT
  Outputs data to the default output I/O unit

- ENCODE
  Translates data from external character form to internal binary representation

- DECODE
  Translates data from internal binary representation to external character form

If a READ or WRITE statement contains a format specifier, it is a formatted I/O statement. Otherwise, it is an unformatted I/O statement.

The ENCODE and DECODE statements transfer data between internal locations only. Both ENCODE and DECODE translate the data using a format specifier.

Using formatted READ and WRITE statements with internal files achieves the same results as ENCODE and DECODE. GLS FORTRAN supports ENCODE and DECODE for compatibility with older FORTRAN compilers.

File Positioning Statements

File positioning statements affect the position of the file pointer relative to a specific file; they cannot reference internal files. All three file positioning statements require that the file be connected for sequential access. The file positioning statements are:

- BACKSPACE
  Moves the file pointer to the start of the preceding record

- REWIND
  Moves the file pointer to the initial point of a file

- ENDFILE
  Marks the preceding record as the last record in a file

Auxiliary I/O Statements

Auxiliary statements manipulate the connection of I/O units to external devices and media, and inquire about the characteristics of a particular connection; they cannot reference internal files. The auxiliary I/O statements are:

- OPEN

  - Creates a file and connect it to an I/O unit

  - Creates a preconnected file

  - Connects an existing file to an I/O unit

- Changes the characteristics of an existing I/O unit/file connection

- CLOSE

  Disconnects a file from an I/O unit

- INQUIRE

  Returns information about the characteristics of a named file, or the connection of a file to a particular I/O unit

Data Transfer

Data transfer is the process of moving data between records and individual items specified in an I/O list. An I/O list is a list of data items whose values are transferred by a data transfer statement.

The I/O system performs the following steps each time it executes a data transfer statement:

1. Determines the direction of the data transfer.
2. Identifies the I/O unit.
3. Establishes the format (if one is specified).
4. Positions the file before the transfer.
5. Performs the transfer between the file and the I/O list.
6. Positions the file after the transfer.
7. Sets the I/O status specifier (if one is defined).

There are two types of data transfer: formatted and unformatted. I/O statements for formatted data requires formatted I/O statements; unformatted data must requires unformatted I/O statements. statement of the same type.

Formatted Transfer

During formatted data transfer, the I/O system transfers data between a file and the I/O list, and performs editing on the data. The current record and (optionally) other records are read from or written to.

Editing

The editing performed during the transfer is directed by a format specification. A format is a description of the arrangement or pattern that data has when it is read or written. The same data can be read or written with different formats to suit different situations.

A format specification is a list of items separated by commas and enclosed in parentheses. The list is called a format list, and contains items called edit descriptors. A format list can also contain other format lists.

There are two kinds of edit descriptors:

- Repeatable

  Repeatable edit descriptors control the editing of character, logical, and numeric data. Each item in the I/O list corresponds to a repeatable edit descriptor in the format list. Each repeatable edit descriptor can be preceded by an integer constant called a repeat factor, that tells the I/O system how many times to repeat the edit specified in the edit descriptor.

- Nonrepeatable

  Nonrepeatable edit descriptors are not associated with specific data items in the I/O list. Instead, they control such things as column position, spacing, sign control, blank control, and line termination.

Format Control

The interaction between the I/O list and the format specification is a dynamic process called format control.

Format control always proceeds from left to right, matching each item in the I/O list with the next repeatable edit descriptor. It executes nonrepeatable edit descriptors as they are encountered.

If an edit descriptor has a repeat factor, format control processes the I/O list as if it contained the specified number of consecutive items.

If the format list ends before reaching the end of the I/O list, format control reverts to the beginning of the last nested format list, if there is one. If there is none, format control reverts to the beginning of the format specification and again passes through the I/O list. Each time format control reverts, it accesses a new record.

List–Directed Formatting

List-directed formatting is an alternative method of formatted data transfer. It is specified by using an asterisk (*) as the format specification in an I/O statement. A FORMAT statement is not required.

When the I/O system executes a list-directed READ, WRITE, or PRINT statement, it begins a new record, and formats each input or output value using the data type and field width of the corresponding I/O list item to generate an equivalent edit descriptor.

For more information about the rules for writing valid format specifications, the actions of the various edit descriptors, and list-directed formatting, refer Chapter 8.

Unformatted Transfer

During unformatted data transfer, the I/O system transfers data between the current record of a file and the I/O list. The I/O system does not perform any editing of the data, and only one record is read from or written to.

Chapter 8

Format Specification

This chapter describes the format specification methods. It also describes how to use edit descriptors when performing formatted data transfer.

Specifying Formats

There are three ways to specify a format:

- Explicitly in a FORMAT statement
- Implicitly as a character variable, element of a character array, or any character expression that evalates to a valid format specification
- Implicitly as list-directed formatting

General Form for Format Specifications

A format specification has the following format:

([*format list*])

format list is a list of items enclosed in parentheses. The items in *format list* can be any of the following: [*r*] *repeatable edit descriptor*; *nonrepeatable edit descriptor*; or [*r*] *format list*.

repeatable edit descriptor and *nonrepeatable edit descriptor* are special character strings that describe the kind of editing being performed. *r* is a positive integer constant called the repeat factor. If *r* is not specified, the default value is 1.

format list can be empty only if the corresponding I/O list is also empty. If *format list* contains another (nested) *format list*, the nested list cannot be empty.

If the I/O list contains at least one item, *format list* must contain at least one repeatable edit descriptor.

Character Format Specification

A character format specification must be in the form of a valid format specification, starting at the leftmost character position. It is enclosed in parentheses, and any characters following the right parenthesis do not affect the format specification.

If a format specification is given as a character array name and the specification's length exceeds the length of the first array element, the specification becomes the concatenation of all the array elements in column-major order.

If the format specification is an array element, the specification's length cannot exceed the length of the array element.

Format Control

Format control is the interaction between the I/O list and the format specification. It always proceeds from left to right, matching each item in the I/O list with the next repeatable edit descriptor. (There is no match between I/O list items and nonrepeatable edit descriptors.) Format control executes nonrepeatable edit descriptors as they are encountered.

If an edit descriptor has a repeat factor *r*, format control processes the I/O list as if it contained *r* consecutive items.

If the format list ends before reaching the end of the I/O list, format control reverts to the beginning of the last nested format list, if there is one. If there is none, format control reverts to the beginning of the format specification and again passes through the I/O list. Each time format control reverts, it accesses a new record.

Repeatable Edit Descriptors

Repeatable edit descriptors control the editing of character, logical, and numeric data. Each item in the I/O list corresponds to a repeatable edit descriptor in the format list.

Each repeatable edit descriptor can be preceded by a repeat factor, that determines how many times to repeat the edit specified by the edit descriptor.

Repeatable edit descriptors consist of a letter and a number; the letter indicates the type of data to edit, and the number indicates the size of the data field.

In Table 8-1, A, D, E, F, G, I, L, ††O, and ††Z indicate the type of data to edit. *w* is a positive integer constant that indicates the number of characters in the field. *d* is an unsigned integer constant indicating the number of digits following the decimal point. *e* is a positive integer constant indicating the number of digits in the exponent.

Table 8–1. Repeatable Edit Descriptors

Syntax	Type of Descriptor
A[*w*]	Alphanumeric
D*w*.*d*	Floating-point
E*w*.*d*[Ee]	Floating-point
F*w*.*d*	Floating-point
G*w*.*d*[Ee]	Floating-point
I*w*	Integer
I*w*.*m*	Integer
L*w*	Logical
††O*w*[.*m*]	Octal descriptor
††Z*w*[.*m*]	Hexadecimal descriptor

The following section describes how to the use repeatable edit descriptors to edit alphanumeric, numeric, logical, ††octal, and ††hexadecimal data.

Alphanumeric Editing

The A[*w*] edit descriptor edits character or Hollerith data. If *w* is present, the field width is *w* characters. If *w* is not present, the field width is the length of the data item in the I/O list. The field width for non-chararacter data types is determined by the maximum storage length for that type. COMPLEX and DOUBLE COMPLEX values are stored as real number pairs and therefore require two format descriptors.

Storage is allocated based on the maximum number of characters that can be stored in that data type as shown in Table 8–2.

Table 8–2. Storage Allocation For Data Types

Data Type	Maximum Characters	Data Type	Maximum Characters
BYTE	1		
CHARACTER*n	*n*		
INTEGER*2	2	INTEGER*4	4
INTEGER	4		
LOGICAL*1	1	LOGICAL*2	2
LOGICAL*4	4		
REAL	4	DOUBLE PRECISION	8
REAL*4	4	REAL*8	8
COMPLEX	8	DOUBLE COMPLEX	16
COMPLEX*8	8	COMPLEX*16	16

The following rules apply to A[*w*] (*len* is the actual length of the item in the I/O list):

- On input, if $w \geq len$, the I/O system transfers the rightmost *len* characters. If $w < len$, the I/O system transfers *w* characters and they are left-justified, with $len-w$ trailing blanks.
- On output, if $w > len$, the I/O system transfers $w-len$ blanks, followed by *len* characters. If $w \leq len$, the I/O system transfers the leftmost *w* characters.

The following table shows examples of A[*w*] Editing.

	Data Type	Format	I/O List Item	Result
Input Processing	CHARACTER*8	A3	Math	Matƀƀƀƀƀ
	CHARACTER*8	A7	Math	Matƀƀƀƀƀ
	INTEGER*2	A5	Math	Ma
	REAL	A3	Math	Matƀ
Output Processing	CHARACTER*8	A3	Math	Mat
	CHARACTER*8	A7	Math	ƀƀƀMath

Numeric Editing

The D, E, F, G, I, ††O and ††Z edit descriptors edit numeric data. D, E, F, and G edit any floating-point data such as REAL*4, REAL*8, COMPLEX*8, or COMPLEX*16. I edits integer data. E and G produce floating-point numbers in scientific notation (on output only). ††O and Z are used for octal and hexadecimal numbers.

The following rules apply to numeric edit descriptors on input:

- The I/O system ignores leading blanks, except that a field of all blanks is treated as zero. Treatment of other blanks is determined by the BLANK specifier in the OPEN statement and the settings of the nonrepeatable edit descriptors BN and BZ.

- A decimal point in the input field overrides the placement of the decimal point specified by a D, E, F, or G edit descriptor. Also, the input field can contain more digits than the processor needs to approximate the value.

The following rules apply to numeric edit descriptors on output:

- All negative values are prefixed with a minus sign. A positive value or zero can have a plus sign as controlled by the S, SS, and SP nonrepeatable edit descriptors.

- The I/O system right justifies all numeric values, and when necessary, pads with blanks on the left.

- If the characters in the output field exceed the field width *w*, the I/O system produces a field of *w* asterisks (*).

Floating–Point Editing, D and E

The D*w*.*d* and E*w*.*d*[E*e*] edit descriptors describe a field whose width is *w* positions with a fractional part containing *d* digits (unless the scale factor $k > 1$), and an exponent of *e* digits. When using the D*w*.*d* and E*w*.*d*[E*e*] edit descriptors, the matching I/O list item must be a floating-point type.

The following rules apply to D*w*.*d* and E*w*.*d*[E*e*]:

- The input field is identical to that of the F*w*.*d* edit descriptor; *e* has no effect.
- If the scale factor *k*=0, the output field has the form [+−][0].*x*1*x*2*x*3...*xd exp*. *x*1*x*2*x*3...*xd* are the *d* most significant digits of the value after rounding, and *exp* is a decimal exponent. The form of *exp* depends on its absolute value as shown in the following table. D*w*.*d* and E*w*.*d* are not valid if $|exp| > 999$.

Edit Descriptor	Absolute Value of *exp*	Exponent Form
E*w*.*d*	|*exp*|≤99 99<|*exp*|≤999	E+z1z2 or 0+z1z2 +z1z2z3
E*w*.*d*[Ee]	|*exp*|≤(10**e)−1	E+z1z2z3...*ze*
D*w*.*d*	|*exp*|≤99	D+z1z2 or E+z1z2 or +0z1z2
	99<|*exp*|≤999	+z1z2z3 (*z* is a digit)

- The scale factor *k* also controls decimal normalization according to the rules listed below. For more information about *k*P, refer to the "Using Nonrepeatable Edit Descriptors" section in this chapter.
  - If $-d < k < 0$, then the output field has $|k|$ leading zeros and $d - |k|$ significant digits following the decimal point.
  - If $0 < k < d+2$, then the output field has *k* significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point.
  - No other values of *k* are valid.

The following table shows examples of D*w.d* and E*w.d*[E*e*] Editing.

	Format	I/O List Item	Result
On Input	D9.2	999999.99	9.9999999D+05
	D14.4	20583.4077D+03	2.05834077D+07
	E9.2	3.31587E2	3.31587E2
	E10.3	ƀƀ81.081E3	8.1081E4
On Output	D15.3	0.0181	ƀƀƀƀƀƀ0.181D−01
	D8.1	0	ƀ0.0D+00
	E10.2	1216641.731	ƀƀ0.12E+07
	E12.4	1216641.731	ƀƀ0.1217E+07

Floating−Point Editing, F

The F*w.d* edit descriptor describes a field whose width is *w* positions, with a fractional part containing *d* digits. When using F*w.d*, the I/O list item must be a floating-point type.

The following rules apply to F*w.d* on input:

- The field can contain an optional sign, followed by digits that can optionally contain a decimal point. If there is no decimal point, the I/O system interprets the rightmost *d* digits in the field as the fractional part. The input field can contain more digits than needed by the processor to approximate the value.

- The input field can be followed by an exponent expressed as a signed integer constant, or the character D or E followed by zero or more blanks, followed by an optionally signed integer constant.

The following rules apply to F*w.d* on output:

- The output field consists of any necessary blanks followed by digits containing a decimal point that represent the internal value rounded to *d* fractional digits and modified by the established scale factor (see the *k*P nonrepeatable edit descriptor).

- If the value is negative, the output field is prefixed with a minus sign. If the value is positive, the output field can have an optional plus sign.

- If the absolute value of the internal data is less than one, the output field can have an optional zero immediately to the left of the decimal point. The following table shows examples of F*w.d* editing.

	Format	I/O List Item	Result
On Input	F7.2	6671878	66718.78
	F9.5	-10.24E+2	-1024.0
On Output	F9.4	2.71828	ƀƀƀ2.7183
	F8.3	-98.87314	ƀ-98.873

Floating-Point Editing, G

The G*w.d*[E*e*] edit descriptors describe a field whose width is w positions with a fractional part containing *d* digits (unless the scale factor $k > 1$), and an exponent of *e* digits. When using the G*w.d*[E*e*] edit descriptor, the I/O list item must be a floating-point type.

The following rules apply to G*w.d*[E*e*]:

- The input field is identical to that of the F*w.d* edit descriptor; *e* has no effect.

- The output field depends on M, the magnitude of the I/O list item as follows:

  - If $M < 0.1$ or $M \geq 10^{**}d$, then the output field is identical to that produced by G*w.d*[E*e*] using the current scale factor *k*.

  - If $0.1 \leq M < 10^{**}d$, then M is inside the range that permits F*w.d* editing. In this case, the I/O system ignores the current scale factor *k*, and M produces an equivalent conversion as shown in the following table.

The following table shows G*w*.*d*[E*e*] conversion when 0.1,M<10***d*.

Value of M	Equivalent Conversion
0.1≤M<1	*F(w−n).d,n(b̸)*
1≤M<10	F(*w*−*n*).(*d*−1),*n*(b̸)
10**(*d*−2)≤M<10**(*d*−1)	F(*w*−*n*).1,*n*(b̸)
10**(*d*−1)≤M<10***d*	F(*w*−*n*).0,*n*(b̸)
n(b̸) = 4 blank spaces for G*w*.*d*	
n(b̸) = *e*+2 blank spaces for G*w*.*d*[Ee]	

The following table shows examples of G*w*.*d*[E*e*] editing.

	Format	I/O List Item	Result
On Input	L1	T	.TRUE.
	L3	b̸b̸T	.TRUE.
	L7	.FALSE.	.FALSE.
On Output	L4	.TRUE.	b̸b̸b̸T
	L1	.FALSE.	F

Complex Editing

A complex data value is represented as a pair of values: a real part and an imaginary part. Editing of complex data is accomplished by the successive interpretation of two D, E, F, or G edit descriptors. The first descriptor describes the real part, and the second descriptor describes the imaginary part.

The two edit descriptors can be different, and nonrepeatable edit descriptors can appear between any two successive D, E, F, or G edit descriptors.

The following table shows examples of complex editing.

	Format	I/O List Item	Result
On Input	2F9.3	28829809856777.765	288298.098, 56777.765
	D9.1,E9.3	999999.993.31587E2	9.9999999D+05, 3.31587E2
On Output	D8.1,D8.3	0.0, 3.14159	ƀ0.0D+00.314D+01
	2E9.2	283.2394, 0.129312	ƀ0.28E+03ƀ0.13E+00

Integer Editing

The I*w* and I*w.m* edit descriptors describe a field whose width is *w* positions. When using I*w* or I*w.m*, the I/O list must be of type INTEGER, and consist of at least one digit.

The following rules apply to I*w* and I*w.m*:

- The input field can be an optionally signed integer constant.
- For I*w*, the output field consists of an integer constant. If the value is positive, it can be prefixed with an optional plus sign. If the value is negative, it is prefixed by a minus sign.
- For I*w.m*, the output field is identical to that produced by I*w*, except that it must have at least *m* digits, and if necessary, have leading zeros. The value of *m* cannot exceed *w*. If *m* = 0 and the internal value of the I/O list item is zero, the output field consists of all blanks regardless of any sign control in effect.

The following tables shows examples of I*w* and I*w.m* editing.

	Format	I/O List Item	Result
On Input	I5	+128	128
	I5	-999	-999
	I5	b̸b̸b̸b̸0	0
On Output	I4	+128	b̸b̸128
	I4	-999	b̸-999
	I5.3	1	b̸b̸001
	I5.3	-1	b̸-001

Octal Editing

††The O*w*[.*m*] edit descriptor inputs or outputs data in octal format (base 8). It can be used with any data type. *w* specifies the number of output digits, zero-padded as necessary. If *w* is not specified, the default field width for that data type is used. The following table lists the default field widths for data types.

Data Type	Default	Data Type	Default
BYTE	7		
INTEGER*2	7	LOGICAL*2	7
INTEGER*4	12	LOGICAL*4	12
REAL*4	12	REAL*8	23

The following rules apply to data input using the O*w* and O*w*.*m* edit descriptors:

- *w* octal digits are transferred from the external field to the target I/O list element. The external field may contain only valid unsigned octal numbers (digits 0 – 7).
- A blank external field defaults to all zeros.
- Invalid characters in the external field results in an error.
- The *m* optional parameter is not valid on input, and is ignored.

The following rules apply to data output using the O*w* and O*w*.*m* edit descriptors:

- The first *w* digits of the unsigned octal representation of the corresponding I/O list element are transferred to the external field.
- If the resulting output is less than *w* digits, the external field is padded with spaces on the left.
- If the resulting output is larger than *w*, the external field is filled with *w* asterisks, indicating an overflow result.
- If *m* is specified and the resulting output is less than *m* digits in length, the external field is right-justified, zero-filled up to a total of *m* digits. Any remaining locations are blank-filled up to the total field length, *w*.

The following table shows examples of O*w* and O*w*.*m* output processing.

Format	Decimal Value	Result
O5	511	ƀƀ777
O2	511	**
O7	-1	ƀ177777 (assuming 2-byte integer)
O5.3	17	ƀƀ021
O5.5	17	00021
O5.0	0	ƀƀƀƀƀ

Hexadecimal Editing

††The Z*w*[.*m*] edit descriptor inputs or outputs data in hexadecimal format (base 16), and can be used with any data type. Valid hexadecimal digits are 0 – 9, a – f, A – F inclusive. Upper- and lowercase letters are equivalent. *w* specifies the number of hexadecimal digits to be transferred, and *m* optionally specifies the minimum number of output digits, zero padded if necessary. If *w* is not specified, the default field width for that data type is used, as shown in the table below.

Data Type	Default	Data Type	Default
BYTE	7		
INTEGER*2	7	LOGICAL*2	7
INTEGER*4	12	LOGICAL*4	12
REAL*4	12	REAL*8	23

The following rules apply to inputting data with the Z*w* and Z*w.m* edit descriptors:

- *w* hexadecimal digits are transferred from the external field to the target I/O list element. The external field may contain only valid unsigned octal hexadecimal digits (numbers 0 – 9, and letters a – f, A – F).
- A blank external field defaults to all zeros
- Invalid characters in the external field results in an error
- The *m* optional parameter is not valid on input, and is ignored

The following rules apply to outputting data with the Z*w* and Z*w.m* edit descriptors:

- The first *w* digits of the unsigned hexadecimal representation of the corresponding I/O list element are transferred to the external field
- If the resulting output is less than *w* digits, the external field is padded with spaces on the left
- If the resulting output is larger than *w*, the external field is filled with *w* asterisks, indicating an overflow result
- If *m* is specified and the resulting output is less than *m* digits in length, the external field is right-justified, zero-filled up to a total of *m* digits. Any remaining locations are blank-filled up to the total field length, *w*

The following tables show examples of Z*w* and Z*w.m* editing.

Input Processing		
Format	Input	Result
Z2	0FFF	0F
Z4	0FFF	0FFF
Z6	0FFF	000FFF

Output Processing		
Format	Decimal Internal Value	Result
Z2	4095	** (overflow)
Z6	4095	ƀƀƀFFF
Z6.4	4095	ƀƀ0FFF

Logical Editing

The L*w* edit descriptor describes a field whose width is *w* positions. When using L*w*, the I/O list item must be of type LOGICAL.

The following rules apply to L*w*:

- The input field can contain optionally contain leading blanks and a decimal point, followed by the character T for true or F for false. The logical constants .TRUE. and .FALSE. are also acceptable as input.
- The output field contains (*w*−1) leading blanks, followed by the character T if the value is true, or F if it is false

  The following table shows examples of L*w* editing.

	Format	I/O List Item	Result
On Input	L1	T	.TRUE.
	L3	ƀƀT	.TRUE.
	L7	.FALSE.	.FALSE.
On Output	L4	.TRUE.	ƀƀƀT
	L1	.FALSE.	F

Nonrepeatable Edit Descriptors

Nonrepeatable edit descriptors are not associated with specific data items in the I/O list. Instead, they control such things as column position, spacing, sign control, blank control, and line termination.

Table 8–3 lists the syntax of nonrepeatable edit descriptors; *c* is any printable ASCII character, *n* is a positive integer constant, and *k* is an optionally signed integer constant that represents a scale factor.

Table 8–3. Summary of Nonrepeatable Edit Descriptors

Syntax	Type of Descriptor
'*c*1,*c*2,...*cn*'	Apostrophe (literal string)
*n*H*c*1,*c*2,...*cn*	Hollerith string
BN	Blank control
BZ	Blank control
*k*P	Scale factor
S	Sign control
SP	Sign control
SS	Sign control
Tc	Position
TLc	Position
TRc	Position
*n*X	Position
/	Line termination
:	Conditional line termination
††Q	Read remainder of record
††$	Carriage control editing

The following subsections describe how to use each nonrepeatable edit descriptor.

Apostrophe Descriptor

The I/O system transfers literal character strings when they are enclosed in apostrophes. This is called apostrophe editing, and is valid only on output. The field width is length of the character string. An apostrophe inside the string must be written as two consecutive apostrophes.

The following table shows an example of apostrophe editing.

Format	I/O List Item	Output Result
3H	ABC	ABC
4H	IT'S	IT'S

Hollerith Descriptor

The *n*H edit descriptor is an alternative method for transferring literal character strings. *n*H causes the I/O system to transfer *n* characters following the H. Like apostrophe editing, it is valid only on output. If the character string contains a single apostrophe, it is counted as one character when specifying *n*.

The following table shows examples of Hollerith Editing.

Format	I/O List Item	Output Result
3H	ABC	ABC
4H	IT'S	IT'S

Q Editing

††The Q edit descriptor is used to read the remaining number of characters from the input record. It can be used to validate an input record length, or to clear out the input stream after all required data has been processed. The Q descriptor has no parameters associated with it, and is specified in a FORMAT statement by the single letter 'Q'.

The Q descriptor returns the number of characters read from the input stream. If Q is the first descriptor in a FORMAT statement, the actual input record length will be returned. The Q descriptor can be used only with INTEGER or LOGICAL data type list elements.

Example:

```
                READ (4,100) BUF
                IF (BUF.GT.100) THEN
                        STOP 'Record too long'
                ELSE
                        BACKSPACE 4
                        READ (4,300) STRINGA, STRINGB, STRINGC
                END IF
     100        FORMAT(Q)
     200        FORMAT(40A1,20A1,20A1)
                   ...
```

BACKSPACE is necessary because the first READ statement advanced the record pointer.

Carriage Control Editing

††The dollar sign ($) edit descriptor is used only in output formatting. It is used to change the default action specified by the first character of the record.

If the first character is a space, the $ causes the carriage return to be suppressed. This is useful for terminal I/O activity where you wish to display a user prompt, and accept input from the same line. If the first character in the record is plus sign (+), the $ descriptor causes the output to begin at the end of the previous line, effectively appending the new record to the previous one.

The $ descriptor has no effect if the first character of the record is either 0 or 1.

Example

```
                TYPE 1000
     1000       FORMAT('bJob Number?b'$)
                ACCEPT 1010, JOBNUM
     1000       FORMAT(14)
```

When you execute the previous statements it produces the following output:

```
   Job Number?
```

The user's response (for example, 1234) can then be accepted from the same line, as shown below:

```
   Job Number?  1234
```

Blank–Control Descriptors BN and BZ

BN and BZ control the interpretation of blanks (other than leading blanks) on input with D, E, F, G, and I editing only; they have no effect on output.

Prior to executing any formatted I/O statement, the BLANK specifier currently in effect for the unit determines the interpretation of blanks.

When the I/O system encounters BN in a format specification, it ignores all blank characters in any succeeding input fields. Ignoring blanks has the same effect as removing blanks, right justifying the field, and replacing the blanks as leading blanks.

When the I/O system encounters BZ in a format specification, it treats all blank characters in succeeding input fields as zeros.

Once specified, BN and BZ remain in effect until changed explicitly, or the I/O statement finishes executing.

The following table shows examples of BN and BZ editing.

Format	I/O List Item	Input Result
BN	12b̸34b̸	1234
BZ	12b̸34b̸	120340

Scale–Factor Descriptor kP

The *k*P edit descriptor establishes a scale factor when using D, E, F, or G editing.

Prior to executing an I/O statement, the scale factor is zero. Once set with the *k*P descriptor, the value of *k* remains in effect until changed with another *k*P descriptor, or the I/O statement finishes executing.

The scale factor *k* produces the following effects on input:

- *k* has no effect with D, E, F, and G editing if there is an exponent in the field.
- With D, E, F, and G editing, the externally represented number equals the internal representation multiplied by 10**k.

The scale factor *k* produces the following effects on output:

- With D and E editing, the mantissa is multiplied by 10**k, which moves the decimal point *k* positions to the right (or left, if negative), and the exponent is reduced by *k*.
- With F editing, the externally represented number equals the internal representation multiplied by 10**k.
- With G editing, *k* is ignored unless the output value is outside the range of F editing. If E editing is required, *k* has the same effect as described for E editing.

When kP immediately follows a D, E, F, or G edit descriptor, a comma is not required between items.

The following table illustrates the effect of a scale factor when used with floating-point edit descriptors.

	Format	I/O List Item	Result
On Input	2PF10.4	ƀƀ125.63ƀƀ	1.2563
On Output	2PF10.2	104.12345	10412.345
	2PD15.3	0.0181	18.10D–03

Sign–Control Descriptors S, SP, and SS

The S, SS, and SP edit descriptors control the optional plus sign character in numeric output fields. S, SS, and SP affect the D, E, F, G and I edit descriptors on output only; they have no effect on input.

When executing any formatted I/O statement, the I/O system normally has the option to produce a plus sign in numeric output fields. An SP edit descriptor directs the I/O system to always produce a plus sign in any subsequent position that normally contains an optional plus sign.

An SS edit descriptor directs the I/O system to always suppress a plus sign in any subsequent position that normally contains an optional plus sign.

An S edit descriptor restores to the I/O system the option of producing a plus sign in numeric output fields.

The following table shows examples of editing with Sign-Control Descriptors.

Format	I/O List Item	Output Result
SS	5	5
SP	5	+5

Position Descriptors Tc, TLc, TRc, and nX

The T*c*, TL*c*, TR*c*, and *n*X edit descriptors determine the position at which the I/O system transfers the next character to or from a record.

T*c* specifies that transfer of the next character to or from a record occurs at the *c*th position

TR*c* specifies that transfer of the next character to or from a record occurs *c* positions to the right of the current position

TL*c* specifies that transfer of the next character to or from a record occurs *c* positions to the left of the current position. If the current position is less than or equal to *c*, TL*c* transfers the next character at position one of the current record.

*n*X specifies that transfer of the next character to or from a record occurs at *n* positions to the right of the current record

The following rules apply to the position descriptors on input:

- T can specify a position in either direction from the current position. This allows the I/O system to process part of a record more than once, possibly with different editing.
- *n*X can specify a position beyond the last position in a record if no characters are transferred from such a position.

The following rules apply to the position descriptors on output:

- When the I/O system transfers characters to positions at or following the position specified by T*c*, TR*c*, TL*c*, or *n*X, any positions that are skipped and not previously filled are padded with blanks.
- A T*c*, TR*c*, TL*c*, or *n*X edit descriptor cannot replace an existing character within a record, but they can affect position such that subsequent editing causes a replacement.

Line–Termination Descriptor /

The line-termination descriptor / indicates the end of data transfer on the current record.

The following rules apply to the line-termination descriptor on input:

- If the file is connected for sequential access, the I/O system skips the rest of the current record and positions the file at the initial point of the next record, which then becomes the current record.
- If the file is positioned at the initial point of a record, the I/O system skips the entire record.

The following rules apply to the line-termination descriptor on output:

- If the file is connected for sequential access, the I/O system creates a new record, which then becomes the current and last record in the file.
- If the file is connected for direct access, the I/O system increments the current record number by one and positions the file at the initial point of the new record, which then becomes the current record.
- The I/O system can output an empty record. If the file is connected for direct access or is an internal file, the record contains blanks.

A comma is not required before or after the / and any I/O list items.

The following example illustrates the line-termination descriptor:

Format	Output Result
`(1X,'ABC'//1X,'DEF')`	`b̸ABC`
	`b̸`
	`b̸`
	`b̸DEF`

Conditional Line–Termination Descriptor

The conditional line-termination descriptor (:) terminates format control if there are no more items in the I/O list. If there are remaining items in the I/O list, the I/O system ignores the : descriptor.

The following example illustrates the conditional line-termination descriptor:

Print Statement	Output Result
`PRINT 10,5` `10 FORMAT (IX,'I=',I2,'b̸J=',I2)`	`I=b̸5b̸J=`
`PRINT 20,6` `20 FORMAT (IX,'I=',I2,:'b̸J=',I2)`	`I=b̸6`

List–Directed Formatting

List-directed formatting is specified by an asterisk (*) as the format specification in an I/O statement. An explicit FORMAT statement is not required.

A list-directed file is an external file containing list-directed records. A list-directed record is a sequence of characters that are either values or value separators.

A value can be one of the following:

- Constant
- Null value
- *r*c* or *r**. *r* is a positive integer constant repeat factor and *c* is a character value. *r*c* is equivalent to *r* successive occurrences of *c*; *r** is equivalent to *r* successive null values. Neither form can contain any embedded blanks, other than those within *c*.

A value separator can be one of the following:

- Comma, optionally preceded or followed by one or more contiguous blanks
- Slash (/), optionally preceded or followed by one or more contiguous blanks
- One or more contiguous blanks between two constants, or following the last constant

The following rules apply to list-directed formatting:

- The I/O system treats blanks as separators, so embedded blanks are allowed only inside character strings
- The I/O system treats the end of a record as a blank, except inside a character string
- The end of a record following any separator with or without any intervening blanks does not imply a null value
- There are two ways to specify a null value:
  - *r**
  - By having no characters precede the first value separator, or appear between the successive value separators

List–Directed Input

When the I/O system executes a list-directed READ statement, it begins a new record and formats each input value using the data type and field width of the corresponding I/O list item to generate an equivalent edit descriptor.

The following rules apply to list-directed input:

- When the I/O system encounters a null value while executing a list-directed input statement, the null value does not affect the corresponding I/O list item. The item retains its value, or if it is undefined, it remains undefined.
- A null value cannot appear as the real or imaginary part of a complex constant, but a null value can represent a whole complex constant
- When the I/O system encounters a slash (/) as a value separator while executing a list-directed input statement, it stops executing the statement at that point and treats any subsequent items in the I/O list as null values

- If the I/O list item is of type CHARACTER*n, the input value must be a nonempty character string enclosed in single apostrophes. Commas, blanks, and slashes (/) are all valid inside character-string constants. An apostrophe inside a character string must be written as two consecutive apostrophes.

  A character-string constant can continue from the end of one record to the beginning of the next for as many records as are needed. The end of a record does not cause a blank character to appear in the constant.

  If w is the field width and the input value is CHARACTER*n, the I/O system transfers characters as follows:

  - if $w \leq n$, transfers the leftmost w characters
  - if $w > n$, transfers the leftmost n characters and pads the remaining $w-n$ positions with blanks

  This is the same effect as assigning the I/O list item in an ordinary assignment statement

- If the I/O list is of type COMPLEX*8 or COMPLEX*16, the input value consists of a pair of numeric input fields separated by a comma and enclosed in parentheses. The first field contains the real part of the complex constant, and the second field contains the imaginary part. Each field can be preceded or followed by one or more contiguous blanks.

- If the I/O list item is of type LOGICAL, the input value cannot contain any commas or slashes (/) embedded among the optional characters after the T or F

- If the I/O list item is of type REAL*4 or REAL*8, the input value has the form of a numeric input field suitable for F editing. That is, it has no fractional digits unless a decimal point appears in the field.

The following table summarizes the correspondence between the data type of the I/O list item and the equivalent edit descriptors:

Data Type of Input Item	Equivalent Edit Descriptor
CHARACTER*n	Aw if $w \leq n$ An,($w-n$)X if $w>n$
COMPLEX*8 or COMPLEX*16	(Fw.0,Fw.0)
LOGICAL	Lw
REAL*4 or REAL*8	Fw.0

List–Directed Output

When the I/O system executes a list-directed WRITE or PRINT statement, it begins a new record and formats each output value using the data type and field width of the corresponding I/O list item to generate an equivalent edit descriptor.

The following rules apply to list-directed output:

- Each output record begins with a blank to provide carriage control when printing
- Output values are separated by one or more blanks
- The I/O system treats blanks as separators, so embedded blanks are allowed only inside character strings
- The I/O system treats the end of a record as a blank, except inside a character string
- When the I/O system outputs a CHARACTER**n* constant, the constant is not enclosed in single apostrophes, or preceded or followed by a value separator. The I/O system can insert a blank for carriage control if a record begins with the continuation of a character constant from the preceding record.
- When the I/O system outputs a COMPLEX**n* constant, the constant is enclosed in parentheses with a comma separating the real and imaginary parts, which are edited according the rules for REAL**k* values where $k = n/2$
- The I/O system outputs an INTEGER**n* value using the I*w* edit descriptor
- The I/O system outputs a LOGICAL constant using T for the value true or F for the value false
- When the I/O system outputs a REAL**n* constant, the constant is represented using G format

The following table summarizes the correspondence between the data type of the I/O list item and the equivalent edit descriptors:

Data Type	Output Format
LOGICAL	I5
INTEGER*2	I7
INTEGER*4	I12
REAL*4	1PG15.7
REAL*8	1PG25.16
COMPLEX*8	1X,'(',1PG14.7,',', 1PG14.7,')'
COMPLEX*16	1X,'(',1PG25.16,',',1PG25.16,',)'
CHARACTER**n*	1X, A*n*

Chapter 9

Statements

This chapter describes GLS FORTRAN statements. Statements are first described in their functional categories: assignment, control, I/O, specification, and structural. Next they are described individually, in alphabetical order. Statement descriptions contain a statement definition, syntax, and an example.

Assignment Statements

Assignment statements assign values to variables, arrays, array elements, and substrings. It evaluates an expression and assigns the result of the evaluation. FORTRAN has the following types of assignment statements:

- Arithmetic
- ASSIGN
- Character
- Data
- Logical

Control Statements

Control statements specify changes to the sequential flow of statement execution to a point within the same or different program unit. Some control statements change the flow of execution depending on a condition that is determined at a point in the flow of execution. Other control statements transfer the flow of execution every time a particular control statement executes, regardless of any condition. The following are the control statements:

- Arithmetic IF
- Assigned GOTO
- Block IF
- CALL
- Computed GOTO
- CONTINUE
- DO
- DO WHILE
- ELSE
- ELSE IF
- END
- END DO
- END IF
- INCLUDE
- Logical IF
- OPTIONS
- PAUSE
- RETURN
- STOP
- Unconditional GOTO

Input/Output Statements

I/O statements transfer data from one storage location to another within a processor, or between a processor and an external device or storage medium. I/O statements also can be used to position the internal file pointer of a specific file. The following are I/O statements:

- ††ACCEPT
- BACKSPACE
- CLOSE
- DECODE
- ENCODE
- ENDFILE
- FORMAT
- INQUIRE
- OPEN
- PRINT
- READ
- REWIND
- ††TYPE
- WRITE

Specification Statements

Specification statements define data types, establish the interpretation and use of symbolic names, and control the management of storage. Specification statements are nonexecutable and appear most often at the beginning of a program unit, before the executable statements. The following arespecification statements:

- Data type
  - BYTE
  - CHARACTER
  - COMPLEX
  - DOUBLE PRECISION
  - INTEGER
  - LOGICAL
  - REAL
- General
  - COMMON
  - DIMENSION
  - EQUIVALENCE
  - EXTERNAL
  - IMPLICIT
  - ††IMPLICIT NONE
  - INTRINSIC
  - ††NAMELIST
  - PARAMETER
  - SAVE
  - ††VIRTUAL
  - ††VOLATILE

Structural Statements

Structural statements define the different kinds of program units that make up a GLS FORTRAN program. A program unit is a logical, self-contained sequence of statements and optional comment lines that form a discrete part of the larger program. All GLS FORTRAN programs consist of one or more program units. The following are structural statements:

- BLOCK DATA
- ENTRY
- FUNCTION
- PROGRAM
- SUBROUTINE

ACCEPT Statement

††The ACCEPT statement transfers data from standard input to the variable or list of variables specified. All input is from the implicit unit (standard input).

Syntax

ACCEPT *format spec* [,*iolist*]
ACCEPT * [,*iolist*]
ACCEPT *group*

format spec	is a numeric format specifier
*	implies list-directed input
group	is a NAMELIST group specifier
iolist	is a list of elements, separated by commas, to be input

Example

```
            CHARACTER*10 NAMEA, NAMEB, NAMEC
            NAMELIST   /MYGROUP/ A,B,C
            ACCEPT 1000, NAMEA, NAMEB
            ACCEPT *, NAMEC
            ACCEPT MYGROUP
            ...
1000        FORMAT (2A10)
```

ASSIGN Statement

Use the ASSIGN statement to assign a label to an integer variable. This enables the variable to be used as a transfer specification in an assigned GOTO statement, or as a format specifier in a formatted I/O statement. The statement must be in the same program unit as the ASSIGN statement.

The ASSIGN statement must execute before any statements containing a reference to the assigned variable name. Once the variable is assigned a label, you cannot specify arithmetic operations.

Syntax

ASSIGN *label* TO *symbolic name*

label — is one to five digits. (At least one digit must be nonzero.) For more information about statement labels, refer to the "Syntax" chapter.

symbolic name — is the symbolic name of an integer variable. Once the variable becomes defined for reference as a statement label, it becomes undefined for use as an integer variable.

Examples

The following example assigns label `300` to the integer variable `trans`.

```
INTEGER  trans
ASSIGN  300  TO  trans
```

You can redefine `trans` in another assignment statement. The following statement returns `trans` to its status as an integer variable. After this statement executes, you cannot use `trans` in an assigned GOTO statement.

```
trans = 2149
```

Assignment Statement (Arithmetic)

Use an arithmetic assignment statement to assign the value of an arithmetic expression to an arithmetic variable or array element.

If the entity on the left of the equal sign has the same data type as the expression on the right, the statement assigns the value directly. If the data types differ, the value of the expression converts to the data type of the entity on the left before the assignment takes place.

Syntax

symbolic name = *arithmetic exp*

symbolic name specifies the name of an arithmetic variable or array element

arithmetic exp is the arithmetic expression. For more information about arithmetic expressions, refer to the "Expressions" chapter.

Examples

The following example assigns the real constant `6.02E23` to `avogadro`.

```
REAL  avogadro
avogadro = 6.02E23
```

The following example evaluates an expression, converts the result of the expression to an integer, and assigns the integer value to the first element in `ionic`.

```
INTEGER  ionic(25)
ionic(1) = 28.43/14.32**(-3)
```

Assignment Statement (Character)

Use a character assignment statement to assign the value of a character expression to a character variable, array element, or substring.

Assigning a value to a character substring does not affect any characters in the character variable or array element that are outside the substring reference. Any character positions in a character entity that are outside the substring reference remain unchanged whether or not the positions were defined or undefined.

If the length of the character expression is less than the length of the character entity, the statement pads the expression on the right with blank characters. If the length of the character expression is greater than the length of the character entity, the statement truncates the expression from the right.

Syntax

symbolic name = *character exp*

symbolic name is a name of the character variable, array element, or substring

character exp is the character expression to assign to *symbolic name*

Examples

The following example assigns the character constant `Sirius` to the character variable `starname`.

```
CHARACTER  starname*12
starname = 'Sirius'
```

The following example evaluates a character expression, then assigns the result to the character array element `compound(5)`.

```
CHARACTER*8  compound(25)
compound(5) = 'Na' // 'Cl'
```

The following example assigns the character constant `mix67` to the substring `var1(2:6)`. The character positions in `var1` that are outside of the substring reference remain unchanged after the assignment takes place.

```
CHARACTER  var1*7
var1(2:6) = 'mix67'
```

BACKSPACE Statement

Use the BACKSPACE statement to move the file pointer to the beginning of the preceding record. If the file has no preceding record, the I/O system ignores the BACKSPACE statement. If the preceding record is an endfile record, the BACKSPACE statement moves the file pointer to the beginning of the endfile record.

Following is a list of restrictions for the BACKSPACE statement:

- The BACKSPACE statement cannot reference an internal file.
- The file must be connected for sequential access.
- You cannot use the BACKSPACE statement with a record that was written using list-directed formatting.

If there is an error when executing the BACKSPACE statement, the I/O system takes the following actions:

1. Terminates the BACKSPACE operation.
2. Specifies the file position as undefined. The only valid statements you can execute are CLOSE, REWIND, INQUIRE, or BACKSPACE.
3. If IOSTAT is specified, sets *status*.
4. Transfers control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

BACKSPACE { *unit* | ([UNIT]=*unit* [,IOSTAT=*status*] [,ERR=*errlabel*]) }

unit — is an integer from 0 through 99 that specifies an external I/O unit

status — is an integer variable or array element where the I/O system posts the outcome of the backspace operation as follows: 0 means the backspace operation was successful, 1 or any number greater than 1 means an error occurred.

errlabel — is the label of an executable statement to which control passes if there is an error during execution of the BACKSPACE statement. The labeled statement must be in the same program unit as the BACKSPACE statement.

Example

The following example backspaces unit 2, returns status to indicate if the backspacing occurs error-free, and if there is an error, transfers control to the statement associated with label 999.

```
BACKSPACE(2,IOSTAT=errorflag,ERR=999)
```

BLOCK DATA Statement

Use the BLOCK DATA statement to identify a program unit as a block data subprogram. A block data subprogram is a nonexecutable program unit that enables you to specify initial values for variables and array elements in named common blocks. For more information about block data subprograms, refer to the "Program Structure" chapter.

Syntax

BLOCK DATA [*symbolic name*]

symbolic name specifies the symbolic name of the block data subprogram. If *symbolic name* is specified, it must be unique. It cannot be the same symbolic name as the main program, an external procedure, a common block, or another block data subprogram within the same executable program.

Example

The following example identifies the program unit `initial` as a block data subprogram.

```
BLOCK DATA initial
```

BYTE Statement

††The BYTE data type statement is equivalent to LOGICAL*1 and can be used to specify a logical data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions. The BYTE data type can contain signed integers in the range −128 through +127.

Syntax

BYTE *symbolic name* [, *symbolic name*]...

symbolic name is a name of the character variable, array element, or substring

Example

```
LOGICAL*1 var1, var2, array(10)
```

is equivalent to

```
BYTE var1, var2, array(10)
```

CALL Statement

Use the CALL statement to transfer control to a subroutine. The CALL statement specifies the symbolic name of the subroutine you want to invoke and a list of actual arguments to pass to the subroutine.

Actual arguments specified in the CALL statement must correspond in number, order, and data type to the dummy arguments specified in the subroutine. An actual argument can be one of the following:

- Expression
- Array name
- Intrinsic function
- External procedure name
- Dummy procedure name
- Alternate return specifier. You must use the statement label of an executable statement in the same program unit as the CALL statement.

Syntax

CALL *symbolic name* [(*argument* [, *argument*] ...)]

symbolic name is the name of the subroutine to invoke

argument is an actual argument to pass to *symbolic name*

Example

The following example calls the subroutine graph and passes three actual arguments.

```
CALL graph (vertical, horizontal, number)
```

CHARACTER Statement

Use the CHARACTER data type statement to specify the character data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions.

Syntax

CHARACTER[**number* | (*)] *name* [**number* | (*)] [, *name* [**number* | (*)]] ...

**number*	is the number of characters in the character item specified by *name*. The default is one.
(*)	indicates that a dummy argument assumes the length specification from the corresponding actual argument or that a function name obtains its length specification from the function reference. If you use an asterisk enclosed in parentheses as a length specifier for the symbolic name of a character constant, the symbolic name assumes the actual length of the constant it represents.
name	is the symbolic name of a constant, variable, array, external function, or statement function

Example

The following example specifies that `var1`, `var2`, and `array` are character type data. `var1` contains 15 characters, and `var2` and each element in `array` contain 5 characters.

```
CHARACTER*5  var1*15, var2*, array(10)
```

CLOSE Statement

Use the CLOSE statement to disconnect a file from an I/O unit. The CLOSE statement need not occur in the same program I/O unit as its corresponding OPEN statement. If the specified file does not exist, the I/O system ignores the CLOSE statement.

After the I/O unit is disconnected, use an OPEN statement to connect it to the same file or a different file in the same program. If the CLOSE statement disconnects a file, a subsequent OPEN statement can connect it to the same or different I/O unit if the file still exists.

When a program terminates normally, the I/O system closes all connected I/O units and deletes all files with STATUS = 'SCRATCH'.

If an error occurs during the execution of the CLOSE statement, the I/O system takes the following actions:

1. Terminates the CLOSE operation.
2. Specifies the file position as undefined, unless the error is an end-of-file condition. In that case, the file pointer points just past the endfile record, and the only valid statements you can execute are BACKSPACE, REWIND, and INQUIRE.
3. If IOSTAT is specified, sets *iostatus*.
4. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

CLOSE([UNIT=]*unit number* [†† ,DISPOSE=*disposition*] [,STATUS=*status*] [,ERR=*errlabel*]
[,IOSTAT=*iostatus*])

unit — is an integer from 0 through 99 that specifies an external I/O unit. The UNIT= keyword is optional if and only if *unit number* is the first item in the specified list.

†† *disposition* — is a character string expression that specifies whether the file should be saved or deleted. To save the file, specify 'KEEP' or 'SAVE'. To delete the file, specify 'DELETE', 'PRINT/DELETE', or 'SUBMIT/DELETE' (the print or submit operation is not performed). The default is 'KEEP'. If DISPOSE = 'KEEP' (or 'SAVE'), the file continues to exist after the I/O system executes the CLOSE statement. If DISPOSE = 'DELETE' (or 'PRINT/DELETE' or 'SUBMIT/DELETE'), the file does not exist after the I/O system executes the CLOSE statement. (The keyword DISPOSE can be abbreviated to DISP.) Disposition exceptions occur if you try to save a scratch file or delete a read-only file. In these cases, the conflicting *disposition* is ignored.

status — is a character string expression (partially duplicating the ††DISPOSE= specifier) that specifies whether the file should be saved or deleted. To save the file, specify 'KEEP'. To delete the file, specify 'DELETE'. The default is 'KEEP'. If the file is a scratch file, the default is 'DELETE'. If STATUS = 'KEEP', the file continues to exist after the I/O system executes the CLOSE statement. If STATUS = 'DELETE', the file does not exist after the I/O system executes the CLOSE statement.

errlabel — is the label of an executable statement to which control passes if there is an error when processing the CLOSE statement. The labeled statement must be in the same I/O unit as the CLOSE statement.

iostatus — is an integer variable or array element where the I/O system posts the outcome of the close operation as follows: 0 means the close operation was successful; 1 or any number greater than 1 means an error occurred; −1 means an end-of-file condition occurred.

Example

The following example closes I/O unit 3 and saves the file. The status of the close operation is posted in `errorflag`, and if the operation fails, control passes to the statement at label 999.

```
CLOSE(3,IOSTAT=errorflag,ERR=999,STATUS='KEEP')
```

COMMON Statement

Use the COMMON statement to define common blocks. Common blocks are contiguous areas of storage containing data that a number of program units can share. When you declare common blocks of the same name in different program units, the blocks share the same storage space when the program units are combined into an executable program.

COMMON statements are frequently used in a block data subprogram. A block data subprogram enables you to specify initial values for variables and array elements that are listed in named common blocks. For more information about block data subprograms, refer to the "Program Structure" chapter.

Syntax

COMMON [/*symbolic name*/] *common list* [[,] / [*symbolic name*] / *common list*] ...

symbolic name is the name of a common block. If you do not specify *symbolic name*, all items in the corresponding *common list* are in an unnamed common block. (A program can have only one unnamed common block.) If you omit the first symbolic name in a COMMON statement that specifies more than one common block, the slashes are optional. Otherwise, slashes are required, with or without a symbolic name, to delimit one common block specification from another.

common list is a list of variable names, array names, and array declarators that represent values in each common block you specify. The items in *common list* are contained in the common block identified by the symbolic name that precedes the list. You must delimit all the items in *common list* with commas. The slashes delimit each symbolic name/common list pair you specify in the COMMON statement. (You can also use commas as secondary delimiters.)

Example

The following example defines the common block `atomic`, which contains `var1`, `var2`, and `array1`.

```
COMMON  /atomic/var1,var2,array1
```

COMPLEX Statement

Use the COMPLEX data type statement to specify the complex data type for symbolic names that represent complex constants, variables, arrays, external functions, and statement functions. For more information about complex numbers, refer to the "Data Types" chapter.

Syntax

COMPLEX[**number*] *symbolic name* [, *symbolic name*] ...

**number*	specifies either an 8- or 16-byte complex value. The default is 8 bytes.
symbolic name	is the symbolic name of a constant, variable, array, external function, or statement function

Examples

The following examples are equivalent – each variable and element in `array1` occupy 8 bytes.

```
COMPLEX  var1, var2, array1(10)
COMPLEX*8  var1, var2, array1(10)
```

CONTINUE Statement

Use the CONTINUE statement to transfer control to the next executable statement in the program. The CONTINUE statement can be the terminal statement for a DO loop that would otherwise end incorrectly with a control statement, such as an arithmetic IF, ELSE IF, or RETURN statement.

Syntax

CONTINUE

Example

The following example shows a DO loop that would end incorrectly with an arithmetic IF statement.

```
            DO 211 var1 =  1, 5
                        .
                        .
                        .
            IF (tax/2 + 4.25) 300, 400, 500
   211      CONTINUE
```

DATA Statement

Use the DATA statement to assign initial values to variables, arrays, array elements, and substrings. An initial value is assigned to a program entity at the beginning of program execution. An entity that is not assigned an initial value is undefined.

DATA statements are nonexecutable. You can use a DATA statement anywhere in a program unit after the specification statements.

An entity in a name list and the corresponding constant in a constant list must be the same data type. If the data types differ, the data type of the constant converts to the type of the name list entity.

When the length of a character entity in a name list is greater than the corresponding character constant, the DATA statement initializes the extra characters in the entity with blanks. When the length of an entity in a name list is less than the corresponding character constant, the DATA statement ignores the extra characters in the entity.

Syntax

DATA *name list* / *constant list* / [[,] *name list* / *constant list* /] ...

name list
: is one or more variable names, array names, array element names, substring names, or implied DO loops. You must separate the names in the *name list* with commas. You cannot use the names of functions, entities in blank common, or dummy arguments. (You can use the names of entities in named common blocks in *name list* if the DATA statement is in a block data subprogram.) If you specify an unsubscripted array name in *name list*, you must specify a constant for each element of the array in *constant list*.

  Use an implied DO loop to initialize a portion of an array as follows:

  (*do list*, *variable* = *initial*,*limit* [,*increment*])

  do list specifies the array elements to initialize. *variable* assumes each iteration value in the range specified by *initial* and *limit*. *variable* must be an integer. The iteration count specified with *initial* and *limit* must be positive. *increment* specifies an iteration step. The default for *increment* is 1.

constant list
: is constants, symbolic names for constants, and constants preceded by a factor. A constant preceded by a factor specifies multiple, successive appearances of the constant in *constant list* and is specified as follows: *factor* * *constant*. *factor* must be a nonzero, unsigned integer constant or the symbolic name of such a constant. *constant* is a zero, a signed or unsigned constant, or a symbolic name. You must delimit each *constant list* with slashes and separate the constants with commas. The number of constants in *constant list* must be the same as the number of entities specified in the preceding *name list*.

Examples

The following example declares initial values for two variables, an array, and a substring.

```
INTEGER array1(3)
REAL var1
LOGICAL var2
CHARACTER subst*5
DATA var1, var2 /1.02E3,.TRUE./ array1,subst /3*100,'total'/
```

The following example converts the real constant assigned to `intvar` to the integer 3. It then converts the integer constant 128 assigned to `realvar` to the real number 128.0.

```
INTEGER intvar
REAL realvar
DATA intvar, realvar /3.14159,128/
```

The following example uses an implied DO loop to initialize elements 10 through 20 of a 50-element integer array. The statement initializes a total of 11 elements with the constant 100.

```
INTEGER array(50)
DATA (array(i), i = 10, 20)/11*100/
```

The following example uses an implied DO loop to initialize every other element from (1,1) through (100,199) in a 20,000-element real array. The statement initializes a total of 10,000 elements with the real value 3.14159.

```
REAL array(100,200)
DATA ((array(k,m), k=1, 100), m=1, 200,2)/10000*3.14159/
```

DECODE Statement

Use the DECODE statement to transfer data from external character form to internal binary representation, using a format specification. DECODE is functionally equivalent to using a READ statement with formatted records on an internal file connected for sequential access.

If there is an error during execution of the DECODE statement, the I/O system takes the following actions:

1. Terminates the DECODE operation.
2. If IOSTAT is specified, sets *status*.
3. Passes control to the statement associate with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

DECODE (*char*,*format*,*loc* [,IOSTAT=*status*] [,ERR=*errlabel*]) [*transfer list*]

char	is the number of characters to translate
format	is a format identifier that controls the editing of the data during the transfer. For more information, refer to the "Format Specification" chapter.
loc	is the name of a variable, an array, or an array element that contains the characters to translate. If *loc* is an array, the I/O system processes the elements in column-major order.
status	is an integer variable or array element where the I/O system posts the outcome of the data transfer as follows: 0 means the transfer was successful, 1 or any number greater than 1 means an error occurred, −1 means an end-of-string condition occurred.
errlabel	is the label of an executable statement to which control passes if there is an error when executing the DECODE statement. The labeled statement must be in the same program unit as the DECODE statement.
transfer list	is the list of variables that receive the data after translation

Example

The following example transfers data from `block1` to `ivar1`, `ivar2`, and `ivar3`. The number of characters transferred is specified by `ichar`. The format for the transfer is specified by the format statement.

```
DECODE (ichar, '100', block1) ivar1, ivar2, ivar3
```

DIMENSION Statement

Use a DIMENSION statement to declare arrays in a program unit. A DIMENSION statement can declare more than one array.

An array in GLS FORTRAN can have a maximum of seven dimensions. The size of an array is equal to the number of elements in the array. The number of elements is equal to the product of the dimension sizes. For more information about arrays, refer to the "Constants, Variables, Arrays, and Substrings" chapter.

Syntax

DIMENSION *symbolic name* (*dim* [,*dim*]...) [,*symbolic name* (*dim* [,*dim*]...)] ...

symbolic name is the name of the array

dim specifies the number of elements in the array in the following format:

[*lower bound*:]*upper bound*

lower bound can be negative, zero, or positive. The default is 1. *upper bound* can be negative, zero, positive, or an asterisk indicating an assumed-size array declarator. *lower bound* and *upper bound* can be an arithmetic constant or a variable expression that evaluates to an integer. A variable expression used as a dimension bound is called an adjustable array declarator.

Examples

The following example declares the 2-dimensional array `values`. The first dimension has a lower and upper bound of 12. The second dimension has an implied lower bound of 1 and an upper bound of 5. `values` contains 5 elements.

```
DIMENSION  values (-12:12, 5)
```

The following example declares the arrays `arr1`, `arr2`, and `arr3`. The arrays are 1-dimensional with a lower bound of 0 and an upper bound of 19. Each array contains 20 elements.

```
DIMENSION  arr1(0:19), arr2(0:19), arr3(0:19)
```

DO Statement

Use the DO statement to specify a block of statements for repetitious execution the specified number of times. The statement block is referred to as a DO loop and is indicated by the DO statement. The last (or terminal) statement in the DO loop is specified by a label in the DO statement.

You can nest DO loops; however, the nested (inner) DO loop must be within the host (outer) DO loop. Nested DO loops can use the same terminal statement.

If you use a DO statement within the statement block of a block IF, ELSE IF, or ELSE statement, the corresponding DO loop must be within that statement block. If you use a block IF statement within a DO loop, the corresponding END IF statement must be within the DO loop.

Syntax

DO *label* [,] *variable* = *exp1*, *exp2* [,*exp3*]

label — is the label of the terminal statement. This statement cannot be one of the following: unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO.

variable — is a variable that determines how many times the DO loop executes. *variable* can be integer, real, or double precision. The value of *variable* at any given time during execution of the DO loop depends on the values of *exp1*, *exp2*, and *exp3*.

exp1 — is the initial value of *variable*. *exp1* can be an integer, real, or double precision expression.

exp2 — is the limit value of *variable*. *exp2* can be an integer, real, or double precision expression.

exp3 — specifies how much to increment *variable* after each execution of the DO loop. The default is 1.

Examples

The following example specifies 10 iterations of a DO loop. Because an increment value is not specified, the default of 1 is used. The statement associated with label 130 is the terminal statement for the loop.

```
DO 130 var = 1, 10
```

The following example specifies 50 iterations of a DO loop. The increment value is 2.

```
DO 2100 var1 = 1, 100, 2
```

The following example specifies eight iterations of a DO loop. The increment value is −2.

```
DO 623 var2 = 16, 1, -2
```

The following example demonstrates the use of expressions.

```
DO 599 var3 = value1*2+1, value2-var4/2, incr/2
```

††The range of a DO loop is extended if it contains a statement, *s1*, that transfers control outside the loop, and, after executing one or more statements, control is returned to the loop via statement *s2*. The range of the DO loop is extended to include all executable statements outside the initial loop between *s1* and *s2*.

Extended-range DO loops are subject to the following rules:

- Transfers into the range of a DO statement may be made only from within the extended range of that DO statement.
- No statement in the extended range of a DO statement may alter the control variable of the DO statement.

Example

```
C        Valid extended DO loop
C
         DO 10, i = 1,10
         ...
         GO TO 200
10       CONTINUE
         ...
200      j = j +1
         ...
         GO TO 10
         ...
```

DO WHILE Statement

††The DO WHILE statement is similar to the DO statement, except that DO WHILE executes a loop while a logical expression remains true, rather than for a fixed number of iterations.

Before each execution of a DO WHILE loop, *expression* is evaluated. If the logical expression is true, the statements within the loop are executed. If the logical expression is false, control is transferred to the first executable statement following *label*.

Syntax

DO [*label* [,]] WHILE (*expression*)

label	identifies the terminating statement
expression	is a logical expression delimited by parentheses

If the optional *label* is not supplied, use END DO as the terminating statement for the loop. As with the DO statement, *label* must be a valid DO loop terminator, and may not be a GOTO, IF, RETURN, STOP, END, or DO statement.

Example

```
INTEGER A
A = 0
DO WHILE (A .LT. 10)
   A = A + 1
END DO
```

DOUBLE PRECISION Statement

Use the DOUBLE PRECISION data type statement to specify the double precision data type for symbolic names that represent real number constants, variables, arrays, external functions, and statement functions. In GLS FORTRAN, a double precision real number occupies eight bytes. (Using the DOUBLE PRECISION statement is equivalent to specifying 8-byte real values with a REAL*8 statement.)

Syntax

DOUBLE PRECISION *symbolic name* [, *symbolic name*] ...

symbolic name is the symbolic name of a real number constant, variable, array, external function or statement function

Example

The following example specifies two double precision variables and a 10-element double precision array. Each variable and element in the array occupies eight bytes.

```
DOUBLE PRECISION  var1, var2, array(10)
```

ELSE Statement

Use an ELSE statement to specify a statement block for conditional execution within a block IF construct. An ELSE statement does not contain a logical expression to determine whether or not control transfers to the statement block. A statement block that corresponds to an ELSE statement executes only if no preceding statement block in the block IF construct executes.

An ELSE statement and its corresponding statement block must follow a block IF or ELSE IF statement and their corresponding statement block. An ELSE or ELSE IF statement cannot follow an ELSE statement in a block IF construct.

Syntax

```
.
.
ELSE
.
.
block
.
.
```

block is one or more statements that conditionally execute within a block IF construct

Example

The following example has a statement block that corresponds to the ELSE statement in the block IF construct. This block executes because the logical expression in the opening block IF statement evaluates to false.

```
largenum = 712
smallnum  = 4
IF (largenum .LT. smallnum) THEN
              factor = smallnum/2 - 2.612
              unit5 = val1 + val2 * factor
ELSE
              ASSIGN 2001 TO standard
              GOTO standard (2001)
END IF
```

ELSE IF Statement

Use an ELSE IF statement to specify a statement block for conditional execution within a block IF construct. An ELSE IF statement contains a logical expression to determine whether or not control transfers to the specified statement block. (A block IF construct can contain any number of ELSE IF statements.)

An ELSE IF statement and its corresponding statement block must follow a block IF statement and its corresponding statement block.

Syntax

.
.
ELSE IF (*logical exp*) THEN
.
.
[*block*]
.
.

logical exp — is the logical expression. If *logical exp* is true, control transfers to *block*. If *logical exp* is false, control transfers to the next ELSE IF or ELSE statement, or the first executable statement following the END IF statement in the construct.

block — is one or more statements for conditional execution

Example

The following example has two ELSE IF statements in the block IF construct. The logical expression in the opening block IF statement is false so control passes to the first ELSE IF statement. The logical expression in the first ELSE IF statement is also false, causing control to pass to the second ELSE IF statement. The logical expression in the second ELSE IF statement is true. Therefore, the statement block specified after the second ELSE IF statement executes.

```
small num = 56000
largenum = 56000
IF (smallnum .GT. largenum) THEN
              STOP
ELSE IF (smallnum .LT. largenum) THEN
              total = largenum - smallnum
ELSE IF (smallnum .EQ. largenum) THEN
              ASSIGN 533 TO equality
              GOTO equality (133, 333, 533)
END IF
```

ENCODE Statement

The ENCODE statement transfers data from internal binary representation to external character form, using a format specification. ENCODE is functionally equivalent to using a WRITE statement with formatted records on an internal file connected for sequential access.

If there is an error during execution of the ENCODE statement, the I/O system takes the following actions:

1. Terminates the ENCODE operation.
2. If IOSTAT is specified, sets *status*.
3. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

ENCODE (*char*,*format*,*loc* [,IOSTAT=*status*] [,ERR=*errlabel*]) [*transfer list*]

char — is an integer that specifies the number of characters to translate to external form

format — is a format identifier that controls the editing of the data during the transfer. For more information, refer to the "Format Specification" chapter.

loc — is the name of a variable, array, or array element that contains the characters after translation to external form. If *loc* is an array, ENCODE processes the elements in column-major order.

status — is an integer variable or array element where the I/O system posts the outcome of the data transfer as follows: 0 means the transfer was successful, 1 or any number greater than 1 means an error occurred.

errlabel — is the label of an executable statement to which control passes if there is an error when executing the ENCODE statement. The labeled statement must be in the same program unit as the ENCODE statement.

transfer list — is a list of variables to translate to an internal binary representation

Example

The following example transfers three characters from `arr1` to `var1` in the format specified by the format statement.

```
ENCODE (3,100,arr1) var1
```

END Statement

Use the END statement to indicate the end of a program unit. The END statement must be the last statement in a program unit.

The END statement in a main program causes program execution to terminate. The END statement in a subprogram causes control to return to the main program. (GLS FORTRAN treats an END statement in a subprogram like a RETURN statement.)

Syntax

```
END
```

END DO Statement

††The END DO statement is used to terminate a DO or DO WHILE loop. END DO is an alternative terminator if a terminating statement label was not supplied with a DO or DO WHILE statement. The END DO statement may also be used in conjunction with a terminating statement label specification, just as a CONTINUE statement is used.

Syntax

END DO

Example

```
        DO WHILE (i .LT. 10)
            ...
            k = i + 1
         END DO
```

or

```
        DO J = 1, 10
            ...
10      END DO
```

END IF Statement

Use an END IF statement to indicate the end of a block IF construct. Each block IF statement must have a corresponding END IF statement.

Syntax

END IF

Example

The following example shows an END IF statement in a block IF construct.

```
IF (smallnum .GT. largenum) THEN
              STOP
ELSE IF (smallnum .LT. largenum) THEN
              total = largenum - smallnum
ELSE IF (smallnum .EQ. largenum) THEN
              ASSIGN 533 TO equality
              GOTO equality (133, 333, 533)
END IF
```

ENDFILE Statement

Use an ENDFILE statement to write an endfile record as the next record of the file. The file must be connected for sequential access. After executing the ENDFILE statement, the I/O system cannot process any further data transfer statements until it executes a BACKSPACE or REWIND statement.

If an error occurs while processing the ENDFILE statement, the I/O system takes the following actions:

1. Terminates the ENDFILE operation.

2. Specifies the file position as undefined. The only valid statements you can execute are CLOSE, REWIND, BACKSPACE, or INQUIRE.

3. If IOSTAT is specified, sets *status*.

4. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

ENDFILE {*unit* | ([UNIT=]*unit* [,IOSTAT=*status*] [,ERR=*errlabel*]) }

unit is an integer from 0 through 99 that specifies an external I/O unit

status is an integer variable or array element where the I/O system posts the outcome of the ENDFILE statement as follows: 0 means the endfile was written successfully, 1 or any number greater than 1 means an error occurred.

errlabel is the label of an executable statement to which control passes if there is an error when executing the ENDFILE statement. The labeled statement must be in the same program unit as the ENDFILE statement.

Example

The following example writes an endfile record to unit 4. If there is an error while the statement executes, the I/O system posts status in `errorflag` and passes control to the statement associated with label `999`.

```
ENDFILE(4,IOSTAT=errorflag,ERR=999)
```

ENTRY Statement

Use an ENTRY statement to specify an entry point in a procedure. An entry point enables execution of the procedure to begin with an executable statement other than the first executable statement in the procedure. A procedure can contain more than one ENTRY statement.

You can place an ENTRY statement anywhere after the FUNCTION statement in an external function procedure or after the SUBROUTINE statement in a subroutine procedure. However, you cannot use an ENTRY statement between a block IF statement and the corresponding END IF statement, or between a DO statement and the last statement of the DO loop.

Use the CALL statement to reference an entry point in a subroutine. To reference an entry point in a function, use the entry point name in an expression. The dummy argument list in an ENTRY statement does not have to match the dummy argument list in the SUBROUTINE or FUNCTION statement at the beginning of the procedure. However, the actual arguments specified in an entry point reference must correspond in number, order, and data type with the dummy arguments specified in the ENTRY statement.

Syntax

ENTRY *symbolic name* [(*dummy* [, *dummy*] ...)]

symbolic name is the symbolic name of the entry point in the procedure

dummy is an argument that holds a place and specifies a data type for an actual argument specified in the procedure reference

Example

The following example specifies the entry point `enter2` and the corresponding dummy arguments `pressure` and `volume`.

```
ENTRY enter2 (pressure, volume)
```

EQUIVALENCE Statement

Use the EQUIVALENCE statement to enable two or more program entities to share the same memory space. You can specify program entities of different data types in an EQUIVALENCE statement. For example, if you specify an integer and a complex variable in one item list, the integer variable shares storage with the real portion of the complex variable. No data type conversion takes place.

Syntax

EQUIVALENCE (*item list*) [, (*item list*)] ...

item list is a list of two or more variable names, array names, array element names, or character substrings. The items must have the same starting address in memory, even if the length of the items differs.

Examples

The following example specifies that DOUBVAR and the first two elements of the integer array INTARR occupy the same storage units.

```
INTEGER*2  INTARR(5)
DOUBLE PRECISION  DOUBVAR
EQUIVALENCE  (INTARR(1),DOUBVAR)
```

The following example specifies that the first five characters of two character variables share the same storage units.

```
CHARACTER  CODE*5, ZONE*12
EQUIVALENCE  (CODE,ZONE)
```

†† A single subscript may be used in an EQUIVALENCE statement to identify an element of a multidimensional array. The linear element number may then be used to reference the array elements.

The linear element numbers for array A as defined in the following statements are shown in the table on the next page.

```
DIMENSION B(6)
DIMENSION A(2,3)
EQUIVALENCE (A(2),B(2))
```

Linear Element	Array A Element	Array B Element
1	A(1,1)	B(1)
2	A(2,1)	B(2)
3	A(1,2)	B(3)
4	A(2,2)	B(4)
5	A(1,3)	B(5)
6	A(2,3)	B(6)

EXTERNAL Statement

Use the EXTERNAL statement to specify external and dummy procedure names for use as actual arguments.

Syntax

EXTERNAL *symbolic name* [, *symbolic name*] ...

symbolic name is the symbolic name of an external or dummy procedure. *symbolic name* can be specified only once in a program unit and it cannot represent an intrinsic function. To specify intrinsic functions as actual arguments, use the INTRINSIC statement. For more information, refer to the INTRINSIC statement later in this chapter.

Example

The following example specifies `alpha`, `bravo`, and `delta` as external procedures.

```
EXTERNAL  alpha, bravo, delta
```

FORMAT Statement

Use the FORMAT statement to define a format specification. The FORMAT statement must appear in the same program unit where it is referenced. For more information about format specifications, refer to the Chapter 8.

Syntax

label FORMAT ([*r*] *edit descriptor* [, [*r*] *edit descriptor* ...])

label is the statement label

r is the repeat factor. The default is 1.

edit descriptor is a character string that describes the kind of editing being performed. One or more edit descriptors comprise a format list. This list can be empty if the corresponding I/O list is empty. A format list can contain another format list, however, the nested list cannot be empty.

Example

The following example prints ABC and DEF on two separate lines.

```
10 FORMAT (1X,'ABC'//1X,'DEF')
```

FUNCTION Statement

Use the FUNCTION statement to define a program unit as an external function. An external function is a program unit defined outside the program unit that invokes it.

An external function can receive control of execution from the main program or from another procedure. Control transfers to an external function through a reference in an expression. For more information about external functions, refer to the "Program Structure" chapter.

You can write external functions using a programming language other than GLS FORTRAN, such as C or assembly language. For more information about using external functions in C, refer to the *GLS Programming Guide*.

Syntax

type FUNCTION *symbolic name* (*dummy* [, *dummy*] ...)

type — is the data type of the value that the function returns

symbolic name — is the symbolic name of the external function. *symbolic name* must be used as a variable name within the function procedure.

dummy — is a dummy argument that holds a place and specifies a data type for the actual arguments supplied in the function reference. *dummy* can be a variable name, array name, dummy procedure name, or an asterisk (alternate return specifier).

Example

The following example defines the program unit `reset` as an external function.

```
INTEGER FUNCTION reset(arg1, arg2, arg3)
```

GOTO Statement (Assigned)

Use an assigned GOTO statement to transfer control to an executable statement identified with a variable. The value assigned to the variable must represent the statement label of the executable statement that is to receive control. The executable statement the variable identifies must be in the same program unit as the assigned GOTO statement. You assign a statement label to an integer variable using the ASSIGN statement. For more information, refer to the ASSIGN statement earlier in this chapter.

You can transfer control to different executable statements using multiple ASSIGN statements. An assigned GOTO statement and any related ASSIGN statements must be in the same program unit.

Syntax

GOTO *variable name* [[,] (*list*)]

variable name is the name of a labeled integer variable. If a program unit contains more than one ASSIGN statement for *variable name*, the most recently executed ASSIGN statement determines the value of *variable name*.

list contains the labels to which the assigned GOTO statement can transfer control. If you specify *list*, your program compares the assigned value with the values in *list*. If the program does not find the assigned value in *list*, the program interrupts execution and issues an error message. If you specify *list*, you must include all valid statement labels.

Examples

The following example transfers control to the executable statement associated with label 810.

```
ASSIGN  810  TO  inert
GOTO inert
```

The following example transfers control to the executable statement associated with label 464.

```
ASSIGN 464 TO kalend
GOTO  kalend (312, 1012, 464, 2000)
```

GOTO Statement (Computed)

Use a computed GOTO statement to transfer control to a statement in a list of executable statements based on the value of an arithmetic expression. (The executable statements must be in same program unit as the computed GOTO statement.) The value of the arithmetic expression specifies a number that corresponds to the position of a statement label in the list of executable statements.

Syntax

GOTO (*list*) [,] *arith exp*

list — is a list of statement labels of executable statements. The labels must be separated with commas.

arith exp — is an arithmetic expression that specifies a number that corresponds to the position of a statement label in *list*. If *arith exp* is less than one or greater than the number of labels in *list*, control transfers to the first executable statement following the computed GOTO statement.

Examples

The following example selects one of four statement labels for the transfer of control depending on the value of num.

```
GOTO (54, 166, 418, 500), num
```

The following example selects one of six statement labels for the transfer of control depending on the value of velocity/t1-t2.

```
GOTO (250, 500, 1000, 2000, 2500, 3000) velocity/t1-t2
```

GOTO Statement (Unconditional)

Use an unconditional GOTO statement to transfer control to an executable statement. The executable statement must be in the same program unit as the unconditional GOTO statement.

Syntax

GOTO *label*

label is one to five digits associated with an executable statement. For more information about statement labels, refer to Chapter 2.

Example

The following example transfers control to the statement associated with label 2189.

```
GOTO  2189
```

IF Statement (Arithmetic)

Use an arithmetic IF statement to transfer control to one of three executable statements. The executable statements must be in same program unit as the arithmetic IF statement.

The value of an arithmetic expression determines which of the three statements receives control. If the arithmetic expression evaluates to a number less than zero, control transfers to the first statement. If the arithmetic expression evaluates to a number equal to zero, control transfers to the second statement. If the arithmetic expression evaluates to a number greater than zero, control transfers to the third statement.

All three labels are required, but they do not have to identify three different statements.

Syntax

IF (*arithmetic exp*) *label1*, *label2*, *label3*

arithmetic exp — specifies the arithmetic expression

label1 — is the label of the statement to which control passes if *arithmetic exp* is less than zero

label2 — is the label of the statement to which control passes if *arithmetic exp* is equal to zero

label3 — is the label of the statement to which control passes if *arithmetic exp* is greater than zero

Example

The following example transfers control to one of the following statements: statement `7514` if `num` is greater than zero, statement `3500` if `num` is equal to zero, or statement `2000` if `num` is less than zero.

```
IF (num)  2000, 3500, 7514
```

IF Statement (Block)

Use the block IF statement to indicate the beginning of a block IF construct. A block IF construct is a block of statements that execute if the logical expression is true. If the logical expression is false, control transfers to the first executable statement following the END IF. (Each block IF statement you specify must have a corresponding END IF statement to identify the end of a block IF construct.)

Syntax

IF (*logical exp*) THEN

.

.

statements

.

.

END IF

logical exp is a logical expression

statements is a block of statements that execute depending on the value of *logical exp*. A statement block can be empty.

Example

The following example shows a block IF construct that contains an expression that evaluates to true. Because the expression is true, the statement block executes.

```
smallnum = 100/2
largenum = 100*2
IF (smallnum .LT. largenum) THEN
   pi = 3.14159
   radius = smallnum
   area = pi * radius ** 2
END IF
```

IF Statement (Logical)

Use the logical IF statement to conditionally execute a GLS FORTRAN statement.

Syntax

IF (*logical exp*) *statement*

logical exp — is a logical expression. If *logical exp* is true, control transfers to *statement*. If *logical exp* is false, control transfers to the first executable statement following the logical IF statement.

statement — is any complete, executable FORTRAN statement, except any of the block IF statements, DO, END DO, or another logical IF statement

Example

The following example shows a logical expression that evaluates to true. Therefore, the STOP statement specified in the logical IF statement executes.

```
largenum = 427
smallnum = 8.602
IF (largenum .GT. smallnum) STOP
```

IMPLICIT Statement

Use the IMPLICIT statement to change the data type of the specified letters from the implicit data type convention. (The convention assigns the integer data type to symbolic names that begin with the letters I to N and the real data type to symbolic names that begin with any other letter.)

A program unit can contain more than one IMPLICIT statement. However, IMPLICIT statements must precede all other specification statements (except PARAMETER statements) in a program unit.

†† The IMPLICIT NONE statement is used to override all implicit defaults, forcing all symbolic names to be declared explicitly. No other IMPLICIT statements are allowed in conjunction with IMPLICIT NONE. The –undefined compile option performs the equivalent functions. Refer to your *GLS Programming Guide* for details about compiler options and defaults.

Syntax

IMPLICIT *type* [**length*] (*letter*) [, (*letter*)] ...
†† IMPLICIT NONE

type — is one of the following data types: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, or HOLLERITH.

length — represents the length for the CHARACTER data type. *length* can be an unsigned integer constant or integer constant expression enclosed in parentheses. The default is 1.

letter — is one or more letters. You can specify a range of letters by specifying the first and last letters in the range separated with a hyphen sign.

Examples

The following example specifies that all symbolic names beginning with the letters A, B, C, D, or E have an implied data type of INTEGER*2.

```
IMPLICIT  INTEGER*2(A, B, C, D, E)
```

The following example specifies that symbolic names beginning with letters A to M and S to Z are CHARACTER data type and have a length of 10.

```
IMPLICIT  CHARACTER*10(A-M, S-Z)
```

INCLUDE Statement

††The INCLUDE statement directs the compiler to interpret the contents of the specified INCLUDE file as if it were a part of the program body.

The −I*string* compile time option allows the user to specify a default directory location to be searched for INCLUDE files. Refer to the *GLS Programming Guide* for details about FORTRAN INCLUDE compiler options and default modes.

Syntax

INCLUDE *file-spec*

file-spec is the name of the included file. No default filename extension is assumed. The INCLUDE file must not start with a continuation line, although it can contain other INCLUDE statements.

Examples

```
FILEA.F:
          COMMON      /PCOM/I(10),J(10)
PROG.F:
          INCLUDE 'FILEA.F'
          DO 10 inum = 1,10
          j(inum) = i(inum) + 1
10        CONTINUE
          END
```

Compiling PROG.F results in the following compiled statements:

```
          COMMON /PCOM/I(10),J(10)
          DO 10 inum = 1,10
          j = i(inum) + 1
10        CONTINUE
          END
```

INQUIRE Statement

Use the INQUIRE statement to obtain information about a file (inquire-by-file) or an I/O unit (inquire-by-unit). As shown below, the syntax for inquire-by-file requires the name of the file and the syntax for inquire-by-unit requires the external I/O unit number.

Any variable or array element that becomes defined or undefined by being used as a specifier in an INQUIRE statement cannot be referenced by another specifier in the same INQUIRE statement.

If an error occurs during execution of the INQUIRE statement, the I/O system takes the following actions:

1. Terminates the INQUIRE operation.

2. Specifies the file position as undefined, unless the error is an end-of-file condition. In that case, the I/O system positions the file pointer just past the endfile record, and the only statements you can execute are CLOSE, BACKSPACE, and REWIND.

3. If IOSTAT is specified, sets *iostat*. All other specifiers become undefined.

4. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

INQUIRE({FILE='*filename*' | [UNIT=]*unit*} [,ACCESS=*acc*] [,BLANK=*bl*]
[†† ,CARRIAGECONTROL=*car*] [,DIRECT=*dir*] [,ERR=*errlabel*]
[,EXIST=*exstat*] [,FORM=*formtype*] [,FORMATTED=*form*] [,IOSTAT=*iostat*]
[,NAME=*fn*] [,NAMED=*namestat*] [,NEXTREC=*next*] [,NUMBER=*num*]
[,OPENED=*opstat*] [†† ,ORGANIZATION=*org*] [,RECL=*reclen*]
[,SEQUENTIAL=*seq*] [,UNFORMATTED=*unf*])

filename — is the name of the file being inquired about. *filename* does not have to exist or be connected.

unit — is an integer from 0 through 99 that specifies an external I/O unit. The UNIT= keyword is optional if and only if *unit* is the first parameter specified in the INQUIRE statement.

acc — is a character variable or element of a character array that the I/O system assigns as 'SEQUENTIAL' if the file is connected for sequential access, or as 'DIRECT' if the file is connected for direct access. If *unit* is specified, it must exist and be connected to a file; otherwise *acc* remains unchanged or †† the I/O system returns *acc* as 'UNKNOWN'.

bl is a character variable or element of a character array that the I/O system assigns as 'NULL' for null blank control, or as 'ZERO' for zero blank control. The file must be connected for formatted I/O; †† otherwise the I/O system specifies *bl* as 'UNKNOWN'. For more information, refer to the OPEN statement.

††*car* is a character variable that the I/O system assigns as 'FORTRAN' if the file uses FORTRAN carriage control, as 'LIST' if the file has implied carriage control, or as 'NONE' if the file has no implicit carriage control. If the I/O system cannot determine the carriage control attributes, it specifies *car* as 'UNKNOWN'.

dir is a character variable or element of a character array that the I/O system assigns as 'YES' or 'NO' depending on whether direct access is allowed for the file being inquired about. If the I/O system cannot determine whether direct access is allowed for the file, it specifies *dir* as 'UNKNOWN'. *dir* remains unchanged if *exstat* or *opstat* is .FALSE..

errlabel is the label of an executable statement to which control passes if an error occurs during execution of the INQUIRE statement. The labeled statement must be in the same program unit as the INQUIRE statement.

exstat is a logical variable or element of a logical array that the I/O system assigns as .TRUE. or .FALSE. depending on whether *filename* or *unit* exists. When the I/O system executes the INQUIRE statement, *exstat* is assigned a value unless an error condition occurs.

formtype is a character variable or element of a character array that the I/O system assigns as 'FORMATTED' if the file being inquired about is connected for formatted I/O, or as 'UNFORMATTED' if the file is connected for unformatted I/O. If *exstat* or *opstat* is .FALSE., *formtype* remains unchanged or †† the I/O system returns *formtype* as 'UNKNOWN'.

form is a character variable or element of a character array that the I/O system assigns as 'YES' or 'NO' depending on whether the file being inquired about can contain formatted records. If the I/O system cannot determine whether the file can contain formatted records, it specifies *form* as 'UNKNOWN'. *form* remains unchanged if *exstat* or *opstat* is .FALSE..

iostat is an integer or element of an integer array that indicates the outcome of the inquire operation as follows: 0 means the inquire operation was successful; 1 or any number greater than 1 means an error occurred; −1 means an end-of-file was encountered.

fn is a character variable or element of a character array that the I/O system assigns the current name of the file being inquired about. *fn* remains unchanged if *exstat* or *opstat* are .FALSE. or if the file has no name.

namestat — is a logical variable or element of a logical array the I/O system assigns as .TRUE. if the file connected to *unit* has a name. Otherwise the I/O system specifies *namestat* as .FALSE.. *namestat* remains unchanged if *unit* does not exist or is not defined.

next — is an integer variable or element of an integer array the I/O system specifies as *n*+1, where *n* is the number of the last record the I/O system has read from or written to in the file being inquired about. If the file is connected but the I/O system has not yet read or written any records, the I/O system specifies *next* as 1. The I/O system specifies *next* as 0 if the file is not connected for direct access or the file position is undefined because of a previous error.

num — is an integer or element of an integer array to which the I/O system assigns the number of the I/O unit currently connected to *filename*. If no I/O unit is connected to *filename*, *num* remains unchanged.

opstat — is a logical variable or element of a logical array that the I/O system assigns as .TRUE. if *filename* is connected to an I/O unit or if *unit* is connected to a file. Otherwise the I/O system specifies *opstat* as .FALSE.. When the I/O system executes the INQUIRE statement, *opstat* is assigned a value unless an error condition occurs. If *opstat* is .TRUE., the following specifiers can be defined: *acc*, *bl*, *next*, *reclen*, and *formtype*.

†† *org* — is a character variable or array element that the I/O system assigns as 'SEQUENTIAL' or 'RELATIVE' depending on file organization. If the I/O system cannot determine the file organization, it specifies *org* as 'UNKNOWN'.

reclen — is an integer variable or element of an integer array that the I/O system assigns the current value for the record length in the file being inquired about. *reclen* is the number of bytes, or †† if the file is connected for unformatted I/O, *reclen* is the number of 32-bit words. If the file is not connected, or is not connected for direct access, *reclen* is set to 0. *reclen* remains unchanged if the file is not connected for direct access, or if *exstat* or *opstat* is .FALSE..

seq — is a character variable or element of a character array that the I/O system assigns as 'YES' or 'NO' depending upon whether the file can be accessed sequentially. If the I/O system cannot determine whether sequential access is allowed for the file, it specifies *seq* as 'UNKNOWN'. *seq* remains unchanged if *exstat* or *opstat* is .FALSE..

unf — is a character variable or element of a character array that the I/O system assigns as 'YES' or 'NO' depending on whether the file being inquired about can contain unformatted records. If the I/O system cannot determine whether the file can contain unformatted records, it specifies *unf* as 'UNKNOWN'. *unf* remains unchanged if *exstat* or *opstat* is .FALSE..

Examples

The following example returns information about logical unit 3, if it exists. The filename is returned in the variable `fname` and the open status (.TRUE. or .FALSE.) in the variable `estat`.

```
INQUIRE (3, NAME=fname, OPENED=estat)
```

The following example returns information about the file `tmp.dat`. The logical unit number, if any, is returned in the variable `fileunit`.

```
INQUIRE (FILE='tmp.dat', NUMBER=fileunit)
```

INTEGER Statement

Use the INTEGER data type statement to specify an integer data type for symbolic names that represent integer constants, variables, arrays, external functions, and statement functions.

Syntax

INTEGER[**number*] *symbolic name* [, *symbolic name*] ...

**number*	is the number of bytes that each integer value occupies in memory. You can specify 1-, 2-, or 4-byte integers. A 2-byte integer can represent values from −65536 through 65535. A 1-byte integer can represent values from −128 through 127. The default is 4 bytes.
symbolic name	is the name of an integer constant, variable, array, external function, or statement function

Example

The following example specifies the integer data type for `array1`. Each variable and element in `array1` occupies two bytes.

```
INTEGER*2  var1, var2, array1(10)
```

INTRINSIC Statement

Use the INTRINSIC statement to specify intrinsic function names for use as actual arguments. For more information about intrinsic functions, refer to Chapter 10.

Syntax

INTRINSIC *symbolic name* [, *symbolic name*] ...

symbolic name is the symbolic name for an intrinsic function

Example

The following example declares `exp`, `tan`, and `sqrt` as representing intrinsic functions.

```
INTRINSIC  exp, tan, sqrt
```

Logical Assignment Statement

Use a logical assignment statement to assign the value of a logical expression to a logical variable or array element. A logical expression must evaluate to a logical value, either true or false. For more information about logical expressions, refer to Chapter 5.

Syntax

symbolic name = *logical exp*

symbolic name is the symbolic name of a logical variable or array element

logical exp is a logical expression

Examples

The following example assigns the logical constant `.false.` to the logical variable `switch`.

```
switch = .false.
```

The following example evaluates a logical expression, then assigns the result to the logical variable `prnout`.

```
prnout = var1/var7 .GT. 128 .AND. var2/var8 .LT. 128
```

LOGICAL Statement

Use the LOGICAL data type statement to specify the logical data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions.

Syntax

LOGICAL[**number*] *symbolic name* [, *symbolic name*] ...

**number*	is the number of bytes that each logical value occupies in memory. You can specify 1-, 2-, or 4-byte logical variables. The default is 4 bytes.
symbolic name	is the symbolic name of a constant, variable, array, external function, or statement function

Example

The following example assigns the logical data type to `array2`. Each variable and element in `array2` occupies one byte.

```
LOGICAL*1  var1, var2, array2(10)
```

NAMELIST Statement

††Use the NAMELIST statement to group together a list of variables and/or arrays under a single unique symbolic *group* name. *group* can then be used to reference all or part of the list in an I/O statement.

The order in which the list elements are specified in the NAMELIST statement defines the output order in NAMELIST-directed I/O. Elements can be of any data type, either explicit or implicit, but cannot consist of individual array elements, substrings, records, or dummy arguments. However, NAMELIST-directed input can be used to assign values to substrings, array elements, and/or records.

NAMELIST-directed I/O can be used only for list elements previously defined with the NAMELIST statement. However, a list element that is defined in a NAMELIST statement need not be referenced elsewhere.

Syntax

NAMELIST /*group*/ *namelist* [[,]/*group*/ *namelist* ...]

group is a unique symbolic name

namelist is a list of variables and/or array names, separated by commas, that are to be referenced by *group*

Example

In the following example, `payroll` in the first `NAMELIST` statement associates the variables `name`, `empnum`, `date`, and `hours`. Several data types can be grouped together under a single *group*. Note that the variables `name` and `empnum` are associated under two *groups*, `payroll` and `vacation`.

```
CHARACTER*20         name
NAMELIST /payroll/   name,empnum,date,hours
NAMELIST /vacation/  name,empnum,vacdays
```

NML NAMELIST Specifier

The NAMELIST specifier is used in the control list of a READ or WRITE statement to indicate that NAMELIST-directed I/O is to be performed on the specified *group*. The NML keyword is optional if and only if both of the following conditions are met:

- The first parameter of the I/O control list is a logical unit number without the optional UNIT= keyword.
- The NAMELIST parameter directly follows the unit number in the list.

You cannot use a NAMELIST specifier in a statement that contains a FORMAT specifier. For more information about control list specifiers, refer to the READ and WRITE statements later in this chapter.

Syntax

[NML=]*group*

Example

The following example illustrates valid and invalid uses of the NAMELIST specifier.

```
C   Valid formats for NAMELIST-directed I/O:
C
    READ (5,payroll)
    WRITE (UNIT=2, NML=vacation)
C
C   Invalid formats for NAMELIST-directed I/O:
C
    READ (UNIT=5,payroll)    !Must use NML= if UNIT= is used
    WRITE (5,100) vacation   !NAMELIST not allowed with FORMAT specifiers
```

NAMELIST–Directed I/O

The NAMELIST-directed READ statement reads records sequentially until the specified *group* name is encountered, translates NAMELIST data into internal format according to the corresponding data types of the data, and assigns the translated data to the specified NAMELIST elements.

The NAMELIST-directed WRITE statement takes the internal data for the NAMELIST elements, translates it from internal format to the appropriate data types, and writes the translated data to external records for a sequential file. Every NAMELIST element is written out along with its associated value, one *element-value* pair per record. The format of the output data in a NAMELIST-directed WRITE is compatible with the format required by a NAMELIST-directed READ or ACCEPT statement. The NAMELIST *elements* are written out in the order they occur in the NAMELIST statement.

NAMELIST Record Format

NAMELIST data records are enclosed within a pair of dollar sign ($) or ampersand (&) special characters. The starting record must have either a dollar sign or an ampersand in the first nonblank column of the record, followed by the NAMELIST *group* specifier. The NAMELIST data is terminated by a closing dollar sign or ampersand, optionally followed by the keyword END. You can use either a pair of dollar signs or a pair of ampersands, but you cannot mix a dollar sign and an ampersand in a single data record set. Values are assigned to NAMELIST elements in the form *element=value*.

Syntax

$*group element=value* [*element=value* ...] $ [END]

group	is the group defined in a NAMELIST statement
element	is an element in the NAMELIST-defined *group*
value	is the assigned value for that *element*

The *value* assigned to an *element* may consist of one or more constants. Constants can be integer, real, logical, complex, or character values. If the data type of the specified constant does not match the data type of the corresponding NAMELIST *element*, conversion is performed following standard rules for arithmetic assignments. Conversion between numeric and character data types is not permitted.

You cannot use a symbol defined in a PARAMETER statement as a constant in a NAMELIST *element=value* assignment. Multiple occurrences of a single constant value can be represented in the form *n**c* (*n* is the number of occurrences of the constant value *c*). Multiple null values can be specified in the form *n** (*n* is an integer number indicating the number of nulls to be supplied).

If a list of values is to be assigned to an array, the first value specified in the list will be stored in the first array element, and so forth. You do not need to assign every array element; however, the total number of values specified cannot exceed the array length. Consecutive commas within a value list indicate null assignments. If an array element (rather than array name) is specified, the corresponding values are assigned beginning with that element.

You do not need to assign a value to every element in a NAMELIST *group*. If the value is to remain unchanged, you can omit the element name from the record.

Input records for NAMELIST-directed I/O are subject to the following rules:

- The NAMELIST *group* name cannot contain spaces or tabs and must be specified at the beginning of the record
- The *value* assigned to a NAMELIST *element* cannot contain spaces or tabs, except within the parentheses of a subscript or substring
- *element-value* pairs cannot span records; both the NAMELIST *element* and its associated *value* must be contained in a single record
- Constants used in *value* assignments must be specified in standard FORTRAN format. For example, a complex constant takes the form of a real or integer number pair, separated by a comma and enclosed in parentheses. Only leading and/or trailing spaces are allowed; spaces cannot appear within a numeric constant.
- Logical constants can be assigned a logical TRUE or FALSE value with one of the following special symbols:
  - TRUE `.TRUE.`, `T`, `.T`, `t`, or `.t`
  - FALSE `.FALSE.`, `F`, `.F`, `f`, or `.f`
- Character constants must be enclosed in apostrophes. An apostrophe within a character constant can by specified by two consecutive apostrophes ('').
- Hollerith, octal, and hexadecimal constants are not allowed in NAMELIST records
- A sequence of constants can be separated by spaces, tabs, or commas. Two consecutive commas specifies a null value, indicating that the corresponding array value should remain unchanged. A null value can be used to represent an entire complex constant, but not for just one of the values in a complex pair.
- *element-value* pairs can be separated by spaces, tabs, or commas. Consecutive commas, which would imply a null assignment for an unspecified variable, are not allowed. Any number of spaces or tabs may be placed on either side of the equal sign (=) in an *element-value* pair assignment.

Examples

The input file records below illustrate the NAMELIST record format. This example shows the use of the dollar sign pair in the first record and the ampersand pair in the second record. This example also shows that input *element-value* pairs for a record can be specified on one or more lines.

```
$vacation name='Smith', rate=1.2 accrued=20.5 $END
```

or

```
&vacation
name='Jones'
rate=1.2
accrued=40.5
&
```

The following program reads vacation file records from logical unit 2, calculates the vacation accrued at the monthly rate, and writes the updated records to logical unit 3.

```
C      MONTHLY VACATION UPDATE PROGRAM
C
       NAMELIST /vacation/ name,rate,accrued
       CHARACTER*15 name
       REAL*4       rate,accrued,weekly,deduct
       INTEGER      emp_no
       ...
       READ (2,vacation)
       ...
       accrued = accrued + rate
       WRITE (3,vacation)
       ...
```

The following record was written out using the first example input record and the example program above.

```
$VACATION
NAME ='Smith          ',
RATE =  1.200000   ,
ACCRUED =  21.70000
$END
```

Prompting for NAMELIST Values

In an interactive session, you can prompt a program that is executing a NAMELIST-directed READ statement for the NAMELIST elements the program will accept. To prompt the program, enter a dollar sign ($) or ampersand (&) followed by the *group* name and one or more spaces, then a question mark followed by a carriage return. For example, if the program is executing the statement

```
READ (5,vacation)
```

and waits for input, you can enter

```
$vacation  ?
```

and the program display the variables as follows:

```
$VACATION
NAME
RATE
ACCRUED
$END
```

OPEN Statement

Use the OPEN statement to:

- Create a file and connect it to an I/O unit
- Create a preconnected file
- Connect an existing file to an I/O unit
- Change the characteristics of an existing I/O unit-to-file connection

If an error occurs while processing the OPEN statement, the I/O system takes the following actions:

1. Terminates the OPEN operation.
2. Specifies the file position as undefined. The only valid statements you can execute are CLOSE, BACKSPACE, REWIND, and INQUIRE.
3. If IOSTAT is specified, sets *iostatus*.
4. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Note that if the file is connected to an I/O unit, the only specifier that can be changed with the OPEN statement is *bl*. The file position is not affected.

The OPEN statement must contain a unit number; all other specifiers are optional, but there can be only one of each.

Syntax

OPEN({ [UNIT]=*unit*} [,ACCESS=*acc*] [†† ,ASSOCIATEVARIABLE=*var*] [,BLANK=*bl*]
[†† ,CARRIAGECONTROL=*car*] [†† ,DISPOSE=*disposition*] [,ERR=*errlabel*]
[,FILE=*name*] [,FORM=*formtype*] [,IOSTAT=*iostatus*] [†† ,ORGANIZATION=*org*]
[,RECL=*reclen*] [,STATUS=*status*] [†† ,USEROPEN=*routine*])

unit is an integer from 0 through 99 that specifies an external I/O unit. The UNIT= keyword is optional if and only if *unit* is the first parameter specified in the OPEN statement.

acc specifies the access method as 'SEQUENTIAL', 'DIRECT', or ††'APPEND'. The default is 'SEQUENTIAL'. You cannot specify RECL=*reclen* when the access method is 'SEQUENTIAL'. If you specify 'DIRECT', you must also specify RECL=*reclen*. ††If you specify 'APPEND', the file is accessed sequentially after the last record in the file.

If the file already exists, the access method must match the file's characteristics. If the file does not exist, the OPEN statement creates it with the specified access method.

⁑ *var*	is an INTEGER*4 variable that reflects the record number of the next sequential record in a direct access file (valid for direct access files only)
bl	specifies how the I/O system treats blank characters in formatted records. If you specify 'NULL', the I/O system ignores blank (20h) characters in numeric, formatted input fields. The only exception is a field of blank characters that has a value of zero (30h). If you specify 'ZERO', the I/O system treats all blank characters, except leading blank characters, as zeros. The default is 'NULL'. *bl* is not valid with unformatted records. If the file is connected to an I/O unit, the only specifier that can be changed with the OPEN statement is *bl*. The file position is not affected.
⁑ *car*	is one of the following: 'FORTRAN', 'LIST', or 'NONE'. The default is 'FORTRAN' if formatted files are used; otherwise the default is 'NONE'. If you specify 'FORTRAN', normal FORTRAN interpretation of the initial character is used when printing a file. If you specify 'LIST', the file is printed with single spaces between records. If you specify 'NONE', no implied carriage control is used when the file is printed.
⁑ *disposition*	specifies the disposition of a file as 'KEEP', 'SAVE', 'PRINT/DELETE', 'SUBMIT/DELETE', or 'DELETE' when the associated logical unit is closed. The default is 'KEEP'. If you specify 'KEEP' or 'SAVE' (which are equivalent), the file is retained after the close operation. If you specify 'PRINT/DELETE', 'SUBMIT/DELETE', or 'DELETE' (which are equivalent; the print or submit operation is not performed), the file is deleted after the close operation. The DISPOSE= keyword can be abbreviated to DISP=. Disposition exceptions occur if you try to retain a scratch file or delete a read-only file. In these cases, the conflicting *disposition* is ignored.
errlabel	is the label of an executable statement to which control passes if an error occurs when executing the OPEN statement. The labeled statement must be in the same program unit as the OPEN statement.
name	is the name of the file to be connected to the specified I/O unit. *name* must be in the operating system's file structure. If you omit *name* and the I/O unit is not preconnected, the I/O system connects the I/O unit to the file fort.*n* or ⁑ FOR*nnn*.DAT, where *n* or *nnn* is the number of the I/O unit.
formtype	specifies the file as 'FORMATTED' or 'UNFORMATTED'. The default is 'UNFORMATTED' for direct access files or 'FORMATTED' for sequential access files.

iostatus — **is an integer variable or array element where the I/O system posts the outcome of the open operation as follows: 0 means the open operation was successful, 1 or any number greater than 1 means an error occurred.**

††*org* — specifies the organization of the file as 'SEQUENTIAL' or 'RELATIVE'. If the file does not already exist, the default is 'SEQUENTIAL'. If the file exists and you do not specify *org*, the I/O system uses the organization of the existing file. If you specify *org* for an existing file, it must have the same value as that file.

reclen — is a positive integer that indicates the length of each record in the file. The file must be connected for direct access. *reclen* is the number of bytes, or ††if the records are unformatted, *reclen* is the number of 32-bit words.

If the file is connected for direct access, *reclen* must be specified. If the file is connected for sequential access, *reclen* must not be specified.

status — specifies the status of the file as 'NEW', 'OLD', 'SCRATCH', or 'UNKNOWN'. The default is 'UNKNOWN'. 'NEW' and 'OLD' are valid only if you specify *name* or if the file is preconnected. If you specify 'SCRATCH', the I/O system connects the specified I/O unit to a temporary file. The connection lasts until you execute a CLOSE statement or the program terminates. If you specify 'SCRATCH', you must not specify FILE=*name*. If you specify 'UNKNOWN' and the file exists, the I/O system connects the file. If you specify 'UNKNOWN' and the file does not exist, the I/O system first creates the file, then connects it. ††The keyword TYPE= is equivalent to STATUS=.

††*routine* — is the symbolic name of a user-supplied replacement for the run-time OPEN routine. The GLS FORTRAN Compiler accepts the USEROPEN= keyword and the run-time library supports it; however, its use implies the existence of a user-supplied replacement for the run-time OPEN routine. The FORTRAN OPEN routine has its own calling conventions and is not intended to be a replacement for any other such run-time routine. User replacement of this routine is a significant undertaking, which, at a minimum, requires licensed access to the library source.

Examples

The following example opens logical unit 3.

```
OPEN(3)
```

The following example opens logical unit 3 and connects it to a temporary file.

```
OPEN(UNIT=3, STATUS='SCRATCH')
```

The following example opens unit 8 for direct access. The file name is `overdues.dat`. The result of the open process is returned in the variable `errorflag` and, if an error occurs, control passes to the statement labeled `999`.

```
OPEN(8,FILE='overdues.dat',IOSTAT=errorflag,ERR=999,ACCESS='DIRECT')
```

OPTIONS Statement

Use the OPTIONS statementto override a subset of FORTRAN command line options in effect for a given program unit.

Only one OPTIONS statement is allowed per program unit, and must precede every other statement in the program unit, including PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements. The OPTIONS qualifiers override (or confirm) qualifiers specified on the FORTRAN command line. However, they remain in effect only to the end of the program unit that they begin, so an OPTIONS statement is needed at the beginning of every program unit in which you wish to override the command line or default qualifiers.

Syntax

OPTIONS *option* [,*option*]

option is a valid OPTIONS string. One or more *option* values may be specified on the OPTIONS line, separated by commas.

Example

The statement

```
OPTIONS /CHECK=BOUNDS
```

causes address references for arrays to be checked.

Table 9–1 lists the valid option values, along with their command line equivalents.

Table 9–1. OPTION Values and Command Line Equivalents

OPTIONS String	Cmd Line Equivalent	Summary Description
/CHECK=ALL	none	Overflow, underflow, bounds, checking
/CHECK=NONE	default	No run–time checks performed
/CHECK=OVERFLOW	n/a	Ignored
/CHECK=NOOVERFLOW	n/a	Ignored
/CHECK=UNDERFLOW	n/a	Ignored
/CHECK=NOUNDERFLOW	n/a	Ignored
/CHECK=BOUNDS	–bounds, –X57	Check array bounds
/CHECK=NOBOUNDS	none	Do not check array bounds
/G_FLOATING	n/a	Ignored
/NOG_FLOATING	n/a	Ignored
/I4	none	Default operation (INTEGER*4 and LOGICAL*4)
/NOI4	–i2	Use INTEGER*2 and LOGICAL*2 as defaults
/F77	none	Default operation
/NOF77	onetrip	One iteration DO loops
/NOCHECK	none	Ignored

PARAMETER Statement

Use the PARAMETER statement to assign a symbolic name to a constant, compile-time constant expression, or another symbolic name. Once you assign a symbolic name to a constant using the PARAMETER statement, you can use the symbolic name in place of the actual constant in the program unit.

In the first syntax format given below, one or more PARAMETER pairs can be assigned within a single PARAMETER statement. Each pair is delimited by commas, and the entire statement is enclosed in parentheses. This format requires the data type of the *symbolic name* to be defined, either implicitly or explicitly, before the constant is referenced in the PARAMETER statement.

†† The second syntax format given below is used when the *symbolic name* does not have a specific type that has been defined in a type declaration. The type of the *symbolic name* is taken from the type of the *expression*.

Syntax

PARAMETER (*symbolic name* = *expression* [,*symbolic name* = *expression*] ...)
†† PARAMETER *symbolic name* = *expression*

symbolic name is the symbolic name to assign. A symbolic name must be of the same data type as the expression it is being assigned to. For example, a symbolic name of real data type real can be assigned only to a real constant.

expression is an expression.

Compile–Time Expressions

In the PARAMETER statement, *expression* can be a logical, character, or arithmetic expression, subject to the following rules:

- A logical expression is a compile-time constant expression if:
  - Each operand is a constant; the symbolic name of a constant; one of the intrinsic functions IAND, IEOR, IOR, ISHFT, LGE, LLE, or LLT with constant arguments; or another compile-time constant expression.
  - Each operand is of a logical or an integer data type
  - Each operator is Boolean or a relational operator
- A character expression is a compile-time expression if:
  - Each operand is a constant; the symbolic name of a constant; the function CHAR with a constant argument; or another compile-time constant expression.
  - Each operand is of character data type
  - Each operator is the concatenation operator //
- An arithmetic expression is a compile-time constant expression if:
  - Each operand is a constant; the symbolic name of a constant; one of the intrinsic functions ABS, AIMAG, CMPLX, CONJG, DIM, DPROD, ICHAR, MAX, MIN, MOD, or NINT with constant arguments; or another compile-time constant expression.
  - Each operand is of an integer, real, or complex data type
  - Each operator is a +, −, *, /, or ** operator. (The exponentiation operator ** is evaluated at compile time only if the exponent is of integer data type.)

Symbolic Names in Constant Expressions

A symbolic name equated to a constant will assume a data type based on explicit type declarations preceding the PARAMETER statement, or by using the default conventions for determining implicit data types.

Symbolic constants are subject to the following rules:

- If the symbolic name is used as the length specifier in a CHARACTER statement, it must be enclosed in parentheses
- The symbolic name of a constant cannot appear as part of another constant, except that it can appear as either the real or the imaginary part of a complex constant
- A constant can be defined in a PARAMETER statement only once within a program or subroutine

Examples

The first example uses the first syntax format of the PARAMETER statement to define a variety of constants. This format requires the data type to be defined, implicitly or explicitly, before the constant is referenced in a PARAMETER statement.

```
C      Note that ONE and ZERO are implicitly REAL*4
C      This format requires all data types to be predefined

       REAL*8     E, ESQ
       COMPLEX*8  I
       PARAMETER  (namelength=10)
       CHARACTER*(namelength) lastname
       PARAMETER  (one=1.0,zero=0.0)
       PARAMETER  (I=(zero,one))
       PARAMETER  (E=2.7182818284D0)
       PARAMETER  (ESQ=E*E)
```

†† The second example uses the alternative format of the PARAMETER statement.

```
C      Alternative format of PARAMETER statement
C
C      Data types must NOT be predefined

       PARAMETER  lastname="Karamazov "
       PARAMETER  one=1.0,zero=0.0
       PARAMETER  I=(zero,one)
       PARAMETER  E=2.7182818284D0
       PARAMETER  (esq=E*E)
```

PAUSE Statement

Use the PAUSE statement to temporarily suspend program execution.

Syntax

PAUSE [*message*]

message is a character constant or a string of up to five digits. If you specify *message*, the PAUSE statement displays the message on the console screen, suspends program execution, and waits for user response from the console.

Example

The following example displays the message `TYPE ANY KEY TO CONTINUE` before suspending execution.

```
PAUSE 'TYPE ANY KEY TO CONTINUE'
```

PRINT Statement

Use the PRINT statement to transfer formatted data to the default output I/O unit. The PRINT statement is functionally equivalent to using a WRITE statement with formatted records.

When executing the PRINT statement, the I/O system does not print the first character in a formatted record. Instead, the I/O system uses this character to determine the vertical spacing to use before printing the remainder of the record. The PRINT statement then prints any remaining characters in the record on one line, beginning at the lefthand margin.

The following table lists the amount of vertical spacing determined by the first character.

Character	Hex Value	Vertical Space Output Before Printing Formatted Record
Blank	20h	Advance one line
0	30h	Advance two lines
1	31h	Advance to first line of next page
+	2Bh	No advance

Syntax

PRINT *format* [,*output list*]

format is the format specification. For more information, refer to the "Format Specification" chapter.

output list is a list of the items, separated by commas, to be written. *output list* can be one of the following:

- Variable name
- Array name. If you use an array name, the I/O system writes the array in column-major order.
- Array element name
- Expression containing one of the following data types: CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL
- Implied DO loop. The PRINT statement can write a range of subscripted array elements specified as follows:

  (*input list*,*var* = *e1*,*e2* [,*e3*])

input list is the array to be written. *var* is an integer, real, or double precision variable whose value controls the looping. *e1* is the initial value of *var*. *e2* is the limit value of *var*. *e3* is the increment value of *var*. The default for *e1* is 1. *output list* can contain nested implied DO loops. For more information about implied DO loops, refer to the DO statement earlier in this chapter.

Example

The following example prints name and score.

```
PRINT 35, name,score
```

PROGRAM Statement

Use the PROGRAM statement to define a program unit as the main program. The main program is the program unit that receives control at the beginning of execution. During execution, the main program can invoke subprograms to perform different tasks. Control returns to the main program to terminate execution unless you use a STOP statement in a subprogram.

The PROGRAM statement is not required, but if used, it must be the first statement of the main program. For more information, refer to Chapter 6.

Syntax

PROGRAM *symbolic name*

symbolic name is the symbolic name of the main program

Example

The following example specifies a main program with the symbolic name `main`.

```
PROGRAM main
```

READ Statement

Use the READ statement to transfer data from a specific I/O unit to internal processor storage. You can use the READ statement to reference both internal and external files. The READ statement can transfer both formatted and unformatted records.

The I/O system takes the following actions if there is an error or end-of-file condition while processing the READ statement:

1. Terminates the READ operation.

2. If an error is encountered, the I/O system specifies the file position as undefined. If an end-of-file condition is encountered, the I/O system positions the file pointer just past the endfile record, and the only statements you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.

3. If IOSTAT is specified, sets *iostat* to 1 for an error or −1 for an end-of-file condition.

4. If there is an error or end-of-file condition, the I/O system passes control to the statement identified by *errlabel* or *endlabel*, respectively. If ERR or END is not specified, a run-time error occurs.

The READ statement must contain an I/O unit specifier; all other specifiers are optional, but there can be only one of each. In the form READ *format* [*inputlist*], the I/O unit is the default input unit.

Syntax

READ ({ [UNIT=]*unit*} [[FMT=]*format* | [†† NML=]*group*] [,REC=*recnum*] [,IOSTAT=*iostat*]
  [,ERR=*errlabel*] [,END=*endlabel*]) [*input list*]
READ *format* [*input list*]

unit — is an external I/O unit. *unit* can be an unsigned integer from 0 through 99, an asterisk (*), or an internal file. An asterisk specifies the default input I/O unit. If you do not use UNIT =, *unit* must be the first specifier. If *unit* specifies an internal file, *format* must be specified (it cannot be an asterisk) and *recnum* cannot be specified.

format — is a format identifier that controls the editing of the data during the transfer. If *format* is an asterisk (*), specifying list-directed formatting, *recnum* cannot be specified. For more information, refer to the "Format Specification" chapter. If you do not use the FMT= keyword, then you cannot use the UNIT= keyword and *format* must be specified immediately after *unit*.

†† *group* — is the symbolic name of a group of associated elements defined in a NAMELIST statement and indicates that NAMELIST-directed I/O is to be performed on the specified *group*. You cannot use *group* in a READ statement that contains *format*. If you do not use the NML= keyword, then you cannot use the UNIT= keyword

	and *group* must be specified immediately after *unit*. For more information, refer to the NAMELIST statement earlier in this chapter.
recnum	is a nonzero integer expression that specifies the number of the record to read. *recnum* must be specified if the file is connected for direct access.
iostat	is an integer variable or array element where the I/O system assigns *iostat* the outcome of the data transfer as follows: 0 means the transfer was successful, 1 or any number greater than 1 means an error occurred, −1 means an end-of-file condition occurred.
errlabel	is the label of an executable statement to which control passes if there is an error during execution of the READ statement. The labeled statement must be in the same program unit as the READ statement.
endlabel	is the label of an executable statement to which control passes when there is an end-of-file condition. The labeled statement must be in the same program unit as the READ statement and the file must be connected for sequential access.
input list	is a list of items that receive the input data. These items can be names of variables, arrays or array elements. (The I/O system reads arrays in column-major order.) To specify a range of subscripted array elements for *input list*, use an implied DO list as follows: (*input list*,*var* = *e1*,*e2* [,*e3*]) *var* is the control variable, *e1* is the initial value, *e2* is the limit value, and *e3* is the increment value. *input list* can contain other implied DO lists. For more information about implied DO lists, refer to the DO statement earlier in this chapter.

Examples

The following example reads from unit 10 in format 200 to `name` and `score`.

```
READ(10,200) name,score
```

The following example reads record 5 from unit 3 to `altitude`. The outcome of the data transfer is reported in `errorflag`. If there is an error, control passes to the statement associated with label 200. If there is an end-of-file condition, control passes to the statement associated with label 999.

```
READ(3,REC=05,IOSTAT=errorflag,ERR=200,END=999) altitude
```

†† The following example, using the *'n* form of relative record specification, reads record 7 from a direct access file on logical unit 10.

```
READ (10'7)A
```

REAL Statement

Use the REAL data type statement to specify the real data type for symbolic names that represent real constants, variables, arrays, external functions, and statement functions.

Syntax

REAL[**number*] *symbolic name*[, *symbolic name*] ...

**number*	is the number of bytes that each real number occupies in memory. You can specify 4- or 8-byte real values. (Specifying an 8-byte real values with the REAL*8 type statement is equivalent to using the DOUBLE PRECISION type statement.) The default is 4 bytes.
symbolic name	is the symbolic name of a real constant, variable, array, external function, or statement function

Example

The following example specifies `var1, var2,` and `array3` as the real data type with 8-byte real values.

```
REAL*8  var1, var2, array3(10)
```

RETURN Statement

Use a RETURN statement to terminate execution of a procedure and to transfer control back to the program unit that referenced the procedure.

When a RETURN statement executes in a procedure, control passes to the program unit that called the procedure. If the procedure is a function, the value of the function must be defined before the RETURN statement executes.

When the RETURN statement executes, the program entities in the procedure become undefined except for the following:

- Entities in a blank (unnamed) common block
- Entities that are initially defined but do not become redefined or undefined in the procedure
- Entities in a named common block that appear in the procedure and in at least one other program unit that references the procedure

To retain the definition status of a program entity, use the SAVE statement.

The RETURN statement is not required to terminate a procedure. The END statement declares the physical end of a procedure subprogram. (An END statement used in a procedure has the same effect as a RETURN statement.)

Syntax

RETURN [*integer exp*]

integer exp — indicates which alternate return asterisk in the dummy argument list of the SUBROUTINE or ENTRY statement to use for the return. *integer exp* must be greater than or equal to one and less than or equal to the number of asterisks specified in the dummy argument list. Each asterisk in the dummy argument list corresponds to an actual argument supplied in the CALL statement that invokes the subroutine. Actual arguments are statement numbers that indicate the alternate return points. For more information about alternate return specifiers, refer to the "Program Structure" chapter.

Example

The following example returns control to the program unit.

```
largenum = 30
smallnum = 15
IF (smallnum .LT. largenum) RETURN
```

REWIND Statement

Use the REWIND statement to move the file pointer to the initial point of the file. (The file must be connected for sequential access and cannot be an internal file.)

If an error occurs during execution of the REWIND statement, the I/O system takes the following actions:

1. Terminates the REWIND operation.
2. Specifies the file position as undefined, and the only valid statements you can execute are CLOSE, BACKSPACE, REWIND, and INQUIRE.
3. If IOSTAT is specifies, sets *iostatus*.
4. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

REWIND {*unit* | ([UNIT=] *unit* [,IOSTAT=*iostatus*] [,ERR=*errlabel*]) }

unit	is an integer from 0 through 99 that specifies an external I/O unit
iostatus	is an integer variable or array element where the I/O system posts the outcome of the rewind operation as follows: 0 means the rewind operation was successful, 1 or any number greater than 1 means an error occurred.
errlabel	is the label of an executable statement to which control passes if there is an error during execution of the REWIND statement. The labeled statement must be in the same program unit as the REWIND statement.

Example

The following example rewinds external unit 4. `errorflag` will contain the status of the operation and if there are any errors, control passes to the statement associated with label `999`.

```
REWIND(4,IOSTAT=errorflag,ERR=999)
```

SAVE Statement

Use the SAVE statement to retain the definition status of a program entity following the execution of a RETURN or END statement. The RETURN and END statement in a procedure cause the program entities in that procedure to become undefined except for the following:

- Entities in a blank (unnamed) common block
- Entities that are initially defined but do not become redefined or undefined in the procedure
- Entities in a named common block that appear in the procedure and in at least one other program unit that references the procedure

Entities specified in SAVE statements for one program unit do not become undefined when a RETURN or END statement executes in that program unit. However, if the entities are in a common block, they may become undefined in another program unit.

Syntax

SAVE [*symbolic name* [, *symbolic name*] ...]

symbolic name is the symbolic name of a variable, array, or named common block. (You cannot use the names of procedures, entities within a common block, or dummy arguments.) The common block name must be enclosed in forward slashes as follows: */common block/*. If *symbolic name* is not specified, all program entities in the program units that contain the SAVE statement are saved.

Example

The following example saves the definition status of `var1`, `array1`, and `/block/` following the execution of a RETURN or END statement in the program unit.

```
SAVE  var1, array1, /block/
```

STOP Statement

Use the STOP statement to terminate program execution.

Syntax

STOP [*message*]

message is a character constant or a string of up to five digits. If you specify *message*, the STOP statement displays the message on the console screen, terminates program execution, and returns control to the operating system.

Example

The following example displays the message END OF PROGRAM and then terminates execution.

```
STOP 'END OF PROGRAM'
```

SUBROUTINE Statement

Use the SUBROUTINE statement to define a program unit as a subroutine. A subroutine is an external procedure that is defined outside the program unit that invokes it.

A subroutine can receive control of execution from the main program or from another procedure. Execution control transfers to a subroutine through the CALL statement. A subroutine cannot invoke itself. For more information, refer to Chapter 6.

You can write subroutines using a programming language other than GLS FORTRAN, such as C or assembly language. For more information about using C subroutines, refer to *GLS Programming Guide*.

Syntax

SUBROUTINE *symbolic name* [(*dummy* [, *dummy*] ...)]

symbolic name is the symbolic name of the program unit

dummy is a dummy argument that reserves a place and specifies a data type for the actual arguments supplied in the subroutine call. *dummy* can be one of the following: variable name, array name, dummy procedure name, or an asterisk (alternate return specifier).

Example

The following example defines the program unit `calculations` as a subroutine.

```
SUBROUTINE calculations (arg1, arg2, arg3)
```

TYPE Statement

††The TYPE statement transfers data from internal (processor) storage to standard output. It is the same as the formatted sequential WRITE statement, except that it cannot write to any unit other than standard output.

Syntax

TYPE *format spec* [,*iolist*]
TYPE * [,*iolist*]
TYPE *group*

format spec	is a numeric format specifier
*	implies list-directed output
group	is a NAMELIST group specifier
iolist	is a list of elements to be output

Example

In the following example, the first `TYPE` statement writes character data from array `name`. The second `TYPE` statement writes the values of I, J, and K. The third `TYPE` statement writes the NAMELIST values of `mylist`. All output is to the implicit unit (standard output).

```
      CHARACTER*12   name(5)
      NAMELIST /mylist/ A,B,C,D
      TYPE *,name(1)
      TYPE 1000,I,J,K
      TYPE mylist
1000  FORMAT (3I5)
```

VIRTUAL Statement

††The VIRTUAL statement is equivalent to the DIMENSION statement. It has been included for compatibility with PDP-11 FORTRAN.

VOLATILE Statement

††The VOLATILE statement is used to specify variables, arrays, and common blocks that will not be subject to certain compiler optimizations. This allows storage to be retained for these items if they are declared but not referenced in the program body. For any array name or common block name specified in the statement, the entire array or common block becomes volatile.

Syntax

VOLATILE *nlist*

nlist is a list of one or more variable names, named common blocks, or array names in any combination. Each item in the list is separated from the succeeding item (if any) by a comma. The name of a common block is preceded and followed by a slash (/).

Example

In the following example the common block `STOCK` and the integer variable `TEMP` are volatile.

```
PROGRAM TEST
INTEGER A,B,C,D,TEMP
COMMON /STOCK/A,B,C
. . .
VOLATILE /STOCK/TEMP
. . .
```

WRITE Statement

Use the WRITE statement to transfer data from internal processor storage to a specific I/O unit. You can use the WRITE statement to reference both internal and external files. The WRITE statement can transfer both formatted and unformatted records.

If there is an error during execution of the WRITE statement, the I/O system takes the following actions:

1. Terminates the WRITE operation.
2. Specifies the file position as undefined, and the only statements you can execute are CLOSE and INQUIRE.
3. If IOSTAT is specified, sets *iostat*.
4. Passes control to the statement associated with *errlabel*. If ERR is not specified, a run-time error occurs.

Syntax

WRITE([UNIT=] *unit* [, [FMT=] *format* | ☨☨[NML=]*group*] [,REC=*recnum*]
[,IOSTAT=*iostat*] [,ERR=*errlabel*]) [*output list*]

unit — is the external I/O unit. *unit* can be an unsigned integer from 0 through 99, an asterisk (*), or an internal file. The asterisk specifies the default output I/O unit. If you do not use UNIT=, then *unit* must be the first specifier. If *unit* designates an internal file, *format* must be specified (it cannot be an asterisk) and *recnum* cannot be specified.

format — is a format identifier that controls the editing of the data during the transfer. For more information, refer to the "Format Specification" chapter. If you do not use FMT=, then you cannot use UNIT= and *format* must be specified immediately after *unit*. If *format* is an asterisk (*), specifying list-directed formatting, *num* cannot be specified.

☨☨*group* — is the symbolic name of a group of associated elements defined in a NAMELIST statement and indicates that NAMELIST-directed I/O is to be performed on the specified *group*. You cannot use *group* in a WRITE statement that contains *format*. If you do not use the NML= keyword, then you cannot use the UNIT= keyword and *group* must be specified immediately after *unit*. For more information, refer to the NAMELIST statement earlier in this chapter.

recnum — is a nonzero integer expression that specifies the number of the record to write. *recnum* must be specified if the file is connected for direct access.

iostat is an integer variable or array element where the I/O system posts the outcome of the write operation as follows: 0 means the write operation was successful, 1 or any number greater than 1 means an error occurred.

errlabel is the label of an executable statement to which control passes if there is an error during execution of the WRITE statement. The labeled statement must be in the same program unit as the WRITE statement.

output list is the list of items to be written. *output list* can be the name of a variable, array, array element, or an expression. If you use an array name, the I/O system writes the array in column-major order. To write a range of subscripted array elements from *output list*, use an implied DO list as follows:

(*output list*,*var* = *e1*,*e2* [,*e3*])

var is the control variable, *e1* is the initial value, *e2* is the limit value, and *e3* is the increment value. *output list* can contain other implied DO lists. For more information about implied DO lists, refer to the DO statement earlier in this chapter.

Example

The following example transfers `name` and `score` to I/O unit 10. If there is an error during the transfer, the I/O system sets `errorflag` and passes control to the statement associated with label 350.

```
WRITE(10,200,IOSTAT=errorflag,ERR=350) name,score
```

Chapter 10

System Subroutines, Built–Ins, and Intrinsic Functions

This chapter identifies and describes the ††system subroutines, ††built-in functions, and intrinsic functions supported by the GLS FORTRAN Compiler.

System Subroutines

††FORTRAN supplies subroutines that can be called in the same way as a user-written subroutine. Table 10–1 is a summary of system subroutines. Detailed descriptions for each routine follows. Note that all integer values describes in the following sections must be INTEGER*4.

Table 10–1. System Subroutines

Subroutine	Description
DATE	Returns system date as a character string
ERRSNS	Returns information about most recent runtime error
EXIT	Terminates program, closes files, and exits to the operating system
IDATE	Returns integer values for month, day, and year
GETARG	Gets arguments from the command line
IARGC	Returns an integer corresponding to number of command line arguments specified when program was invoked
MVBITS	Copies a bit pattern from one location to another
RAN	Returns a pseudo–random number between 0.0 and 1.0 inclusive
SECNDS	Returns difference between supplied value and current system time
TIME	Returns system time as a character string

DATE

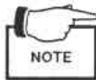
The DATE subroutine takes no arguments and returns the current system date as a 9-character string value. *datebuf* is a 9-byte variable that stores the system date. The *datebuf* variable can be a character string, substring, array, or array element. The date returned is in the *dd–mmm–yy* format, where *dd* is the 2-digit day, *mmm* is a 3-letter month abbreviation, and *yy* is the last two digits of the current year. For example, the string "03–JUN–88" is returned if the current system date is June 3rd, 1988.

Syntax

CALL DATE (*datebuf*)

ERRSNS

The ERRSNS subroutine returns information about the most recent run-time error detected. *iernum* returns a 1 if an error occurs or a 0 if no error has occurred since the previous call or start of execution; and *iunit* contains the unit number on which the last error occurred. The parameters *iosts* and *iostv* are unchanged and are included only for compatibility with VAX/VMS FORTRAN. The parameter *icondval* is set to zero.



In future releases of the GLS FORTRAN compiler, ierrnum will return specific error numbers, rather than a 1. Therefore, programmers should ensure that code does not depend on a fixed value of 1, but rather a value of 1 or greater.

Syntax

CALL ERRSNS (*iernum,iosts, iostv,iunit,icondval*)

EXIT

The EXIT subroutine takes an optional status return code argument and is called to perform a "clean" exit, closing all open files, terminating program execution, and returning control to the operating system. *istatus* is an optional integer argument used to return status code information when the program terminates. Refer to the *GLS Programming Guide* for return status code conventions.

Syntax

CALL EXIT [(*istatus*)]

GETARG

The GETARG subroutine gets arguments from the command line. GETARG takes two arguments. The first argument, *k*, is an integer indicating which command line argument is desired. The second argument, *s*, is a character string where the command line argument is to be placed. The length of *s* should be larger than the longest expected command line argument. GETARG reads the command line argument specified by *k* into the string specified by *s*. If *s* is shorter than the corresponding command line argument, the rightmost characters of the command line argument are truncated. If *s* is longer than the corresponding command line argument, *s* is padded on the right with blanks. It is illegal to attempt to access GETARG with *k* greater than IARGC () -1. Calling GETARG with a *k* of zero returns the command name.

Syntax

GETARG (*k*,*s*)

IARGC

The IARGC subroutine takes no arguments, and returns an integer corresponding to the number of command line arguments specified when the program is invoked. Because the number of command line arguments includes the command name, IARGC is always greater than 0.

Syntax

IARGC ( )

IDATE

The IDATE subroutine takes no arguments and returns the current system date as three integer values, representing month, day, and year. *imon* contains the month number, *iday* contains the day, and *iyear* contains the integer value for the last two digits of the current system year. For example, a system date of March 27th, 1990 would return *imon* = 3, *iday* = 27, and *iyear* = 90.

Syntax

CALL IDATE (*imon*,*iday*,*iyear*)

MVBITS

The MVBITS subroutine copies data from one integer variable to another, allowing the user to specify starting bit positions and number of bits to copy. *src* is the variable source of the bit field to be copied, *sstart* is the starting bit position within *src*, and *len* is the total number of bits to be copied to the destination variable *dst*. The designated bits of *src* are copied to *dst* starting at bit location *dstart*. All MVBITS parameters are of data type INTEGER*4. Bit locations are determined from right (bit 0) to left (bit 31). The values for (*sstart+len*) and (*dstart+len*) must be less than 32.

Syntax

CALL MVBITS (*src,sstart,len,dst,dstart*)

Example

```
      INTEGER*4 isrc,idst
C     isrc initialized to 0111111111111111
C     idst initialized to 0000000000000000
      isrc = '77777'O
      idst = '00000'O
C     copy last 4 bits from isrc to idst
      CALL MVBITS (isrc,3,4,idst,3)
C     idst now contains 0000000001111000
```

RAN

The RAN function takes a single seed argument and returns a pseudo-random number between 0.0 and 1.0 inclusive. Successive calls to RAN produce a uniformly distributed set of numbers. *iseed* is an INTEGER*4 variable used as in initial seed value for the random number generator. RAN alters the value of *iseed* using the formula

iseed = 69069 * *iseed* + 1 (mod 2**32)

so that subsequent calls to RAN return a different *ranval* number. As a general rule, different *iseed* values should be used for each execution so that distinct sets of random values are obtained. RAN calculates a value for *ranval* by taking the high order 24 bits of *iseed* and converting that to a floating-point number.

Syntax

ranval = RAN (*iseed*)

SECNDS

The SECNDS function accepts a single REAL*4 argument and returns the difference between the supplied value and the current system time (represented as seconds since midnight). SECNDS is useful for calculating elapsed time during program execution. SECNDS is accurate to the resolution of the system clock. *delta* is a REAL*4 variable that contains the difference between the current system and the user-supplied value *t*. Successive calls return the elapsed time, in seconds, since the previous call.

Syntax

delta = SECNDS (*t*)

Example

```
      REAL*4 a, b
      a = SECNDS(0.0)
      TYPE 10
10    FORMAT(" Enter carriage return",$)
      ACCEPT 15
15    FORMAT(A)
      b = SECNDS(a)
      TYPE 20,b
20    FORMAT(" That took ", F4.2," seconds")
      END
```

TIME

The TIME subroutine takes no arguments and returns the current system time in hours, minutes, and seconds. *timebuf* is an 8-byte variable that receives the system time, in 24-hour format, as an ASCII string. The format of *timebuf* is hh:mm:ss, where *hh* is a 2-digit hour, *mm* is a 2-digit minute, and *ss* is a 2-digit value for seconds. For example, a call to TIME 1 minute and 30 seconds after noon returns 12:01:30.

Syntax

CALL TIME (*timebuf*)

Built–In Functions

††The built-in functions described in this section are provided to facilitate communications between FORTRAN and non-FORTRAN subprograms. These functions are:

- %VAL
- %REF
- %DESCR
- %LOC

The first three functions – %VAL, %REF, and %DESCR – are used to pass arguments in forms other than standard FORTRAN. They must be used only within an actual CALL statement or function argument list. The fourth function listed – %LOC – is provided to obtain the internal storage address of an element.

%VAL

The %VAL function is used to pass arguments by value. *arg* is to be passed as a 32-bit value. If *argument* is shorter than 32 bits, the value is sign-extended to the full 32 bits. Refer to the ZEXT function later in this chapter if zero extension is required. Only INTEGER, REAL*4, and LOGICAL data types can be passed by value. Array names and procedure values must be passed by reference.

Syntax

%VAL(*argument*)

%REF

The %REF function is used to pass arguments by reference. This is the default method for argument passing and can be used with any data type.

Syntax

%REF(*argument*)

%LOC

The %LOC function returns the internal address of the storage element *argument* as an INTEGER*4 value. *argument* is an array name, memory reference, aggregate reference, or external procedure name.

Syntax

%LOC(*argument*)

Intrinsic Functions

This section presents the intrinsic functions in alphabetical order according to the generic function name, except the following functions, which are listed according to the specific function name (these functions do not have a generic name): AIMAG, CHAR, DIMAG, DPROD, DREAL, ICHAR, INDEX, LEN, LGE, LGT, LLE, LLT.

Functions can be called by generic name (except those listed above) or by specific functiön name. Using the specific name further identifies the function to GLS FORTRAN and enforces data typing.

Each function description contains a description of the function, the syntax, and a a table that lists the specific function name, the data type for the arguments, and the type of the return value. For more information about functions, refer to the "Functions" section in Chapter 6.

ABS Function

The ABS function returns the absolute value of an integer, real, or complex number.

Syntax

return value = ABS(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††IIABS	INTEGER*2	INTEGER*2
††JIABS	INTEGER*4	INTEGER*4
ABS	REAL*4	REAL*4
DABS	REAL*8	REAL*8
CABS	COMPLEX*8	REAL*4
CDABS	COMPLEX*16	REAL*8

††If you do not use the −X181 compile option, the CDABS function must by replaced by ZABS.

ACOS Function

The ACOS function returns the trigonometric arccosine of a real number expressed in radians.

Syntax

return value = ACOS(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ACOS	REAL*4	REAL*4
DACOS	REAL*8	REAL*8

ACOSD Function

†† The ACOSD function returns the trigonometric arccosine of a real number expressed in degrees.

Syntax

return value = ACOSD(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† ACOSD	REAL*4	REAL*4
†† DACOSD	REAL*8	REAL*8

AIMAG Function

The AIMAG function returns the imaginary part of a complex number with a real data type.

Syntax

return value = AIMAG(*argument*)

Specific Name	Type of *argument*	Type of *return value*
AIMAG	COMPLEX*8	REAL*4
†† DIMAG	COMPLEX*16	REAL*8

AINT Function

The AINT function truncates a real number to an integer but maintains the original data type specification. If the number you want to truncate is an integer, the AINT function returns that integer.

If the number you want to truncate is a real number with an absolute value less than 1, the AINT function returns 0. If the number you want to truncate is a real number with an absolute value greater than 1, the AINT function truncates the largest integer that does not exceed the value of the original number and returns it as a real (REAL*4 or REAL*8).

Syntax

return value = AINT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
AINT	REAL*4	REAL*4
DINT	REAL*8	REAL*8

AMAX0 Function

The AMAX0 function returns the largest value from a list of integers and converts the data type to real. All arguments specified must be of the same data type.

Syntax

return value = AMAX0(*argument1*,*argument2* [,*argument*...])

Specific Name	Type of *argument*	Type of *return value*
†† AIMAX0	INTEGER*2	REAL*4
†† AJMAX0	INTEGER*4	REAL*4

AMIN0 Function

The AMIN0 function determines the smallest value from a list of integers and converts the data type to real. All arguments specified must be of the same data type.

Syntax

return value = AMIN0(*argument1*,*argument2* [,*argument...*])

Specific Name	Type of *argument*	Type of *return value*
†† AIMIN0	INTEGER*2	REAL*4
†† AJMIN0	INTEGER*4	REAL*4

ANINT Function

The ANINT function translates a real number to the nearest whole number value and maintains the original data type. If the number you want to translate is an integer, the AINT function returns that integer.

Syntax

return value = ANINT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ANINT	REAL*4	REAL*4
DNINT	REAL*8	REAL*8

ASIN Function

The ASIN function returns the trigonometric arcsine of a real number expressed in radians.

Syntax

return value = ASIN(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ASIN	REAL*4	REAL*4
DASIN	REAL*8	REAL*8

ASIND Function

†† The ASIND function returns the trigonometric arcsine of a real number expressed in degrees.

Syntax

return value = ASIND(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† ASIND	REAL*4	REAL*4
†† DASIND	REAL*8	REAL*8

ATAN Function

The ATAN function returns the trigonometric arctangent of a real number expressed in radians.

Syntax

return value = ATAN(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ATAN	REAL*4	REAL*4
DATAN	REAL*8	REAL*8

ATAND Function

†† The ATAND function returns the trigonometric arctangent of a real number expressed in degrees.

Syntax

return value = ATAND(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† ATAND	REAL*4	REAL*4
†† DATAND	REAL*8	REAL*8

ATAN2 Function

The ATAN2 function returns the trigonometric arctangent of a quotient expressed in radians. *argument1* is the dividend, and *argument2* is the divisor.

Syntax

return value = ATAN2(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
ATAN2	REAL*4	REAL*4
DATAN2	REAL*8	REAL*8

ATAN2D Function

††The ATAN2D function returns the trigonometric arctangent of a quotient expressed in degrees. *argument1* is the dividend, and *argument2* is the divisor.

Syntax

return value = ATAN2D(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††ATAN2D	REAL*4	REAL*4
††DATAN2D	REAL*8	REAL*8

BTEST Function

††The BTEST function tests the specified bit location in the supplied bit pattern and returns a logical TRUE if the bit is set (one) or FALSE if the bit is clear (zero). Bit number *ibit* is checked in the integer bit pattern *buf*. If this bit is set to 1, a logical TRUE is returned; otherwise a logical FALSE is returned.

Syntax

return value = BTEST(*buf*,*ibit*)

Specific Name	Type of *argument*	Type of *return value*
††BITEST	INTEGER*2	LOGICAL*2
††BJTEST	INTEGER*4	LOGICAL*4

CHAR Function

The CHAR function converts an integer to the corresponding ASCII character representation. *argument* is from 0 through 127.

Syntax

return value = CHAR(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††CHAR	LOGICAL*1	CHARACTER
††CHAR	INTEGER*2	CHARACTER
CHAR	INTEGER*4	CHARACTER

CMPLX Function

The CMPLX function converts an integer or real number to a complex number (COMPLEX*8). If you use CMPLX with one argument, the function uses the argument for the real portion of the complex value. The imaginary portion becomes 0.

If you use CMPLX with two arguments, the function uses *argument1* for the real portion of the complex value and *argument2* for the imaginary part. Both arguments must be of the same data type.

Syntax

return value = CMPLX(*argument1*[,*argument2*])

Specific Name	Type of *argument*	Type of *return value*
††–	INTEGER*2	COMPLEX*8
–	INTEGER*4	COMPLEX*8
–	REAL*4	COMPLEX*8
–	REAL*8	COMPLEX*8
–	COMPLEX*8	COMPLEX*8
††–	COMPLEX*16	COMPLEX*8

CONJG Function

The CONJG function returns the conjugate of a complex number.

Syntax

return value = CONJG(*argument*)

Specific Name	Type of *argument*	Type of *return value*
CONJG	COMPLEX*8	COMPLEX*8
††DCONJG	COMPLEX*16	COMPLEX*16

COS Function

The COS function returns the trigonometric cosine of a real or complex number expressed in radians.

Syntax

return value = COS(*argument*)

Specific Name	Type of *argument*	Type of *return value*
COS	REAL*4	REAL*4
DCOS	REAL*8	REAL*8
CCOS	COMPLEX*8	COMPLEX*8
††CDCOS	COMPLEX*16	COMPLEX*16

††If you do not use the –X181 compile option, the CDCOS function must by replaced by ZCOS.

COSD Function

††The COSD function returns the trigonometric cosine of a real or complex number expressed in degrees.

Syntax

return value = COSD(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††COSD	REAL*4	REAL*4
††DCOSD	REAL*8	REAL*8

COSH Function

The COSH function returns the trigonometric hyperbolic cosine of a real number.

Syntax

return value = COSH(*argument*)

Specific Name	**Type of *argument***	**Type of *return value***
COSH	REAL*4	REAL*4
DCOSH	REAL*8	REAL*8

DBLE Function

The DBLE function converts an integer, real, or complex number to a double precision real number. If the number you want to convert is already a double precision real number, the DBLE function simply returns that double precision number.

For a complex number, the DBLE function ignores the imaginary portion and returns the real portion converted to double precision.

Syntax

return value = DBLE(*argument*)

Specific Name	**Type of *argument***	**Type of *return value***
†† –	INTEGER*2	REAL*8
–	INTEGER*4	REAL*8
DBLE	REAL*4	REAL*8
–	REAL*8	REAL*8
–	COMPLEX*8	REAL*8
†† –	COMPLEX*16	REAL*8

DCMPLX Function

††The DCMPLX function converts an integer or real number to a complex number (COMPLEX*16). If you use DCMPLX with one argument, the function uses the argument for the real portion of the complex value. The imaginary portion becomes 0.

If you use DCMPLX with two arguments, the function uses *argument1* for the real portion of the complex value and *argument2* for the imaginary part. Both arguments must be of the same data type.

Syntax

return value = DCMPLX(*argument1*[,*argument2*])

Specific Name	Type of *argument*	Type of *return value*
†† –	INTEGER*2	COMPLEX*16
†† –	INTEGER*4	COMPLEX*16
†† –	REAL*4	COMPLEX*16
†† –	REAL*8	COMPLEX*16
†† –	COMPLEX*8	COMPLEX*16
†† –	COMPLEX*16	COMPLEX*16

DFLOAT Function

††The DFLOAT function converts an integer number to the REAL*8 data type.

Syntax

return value = DFLOAT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††DFLOTI	INTEGER*2	REAL*8
††DFLOTJ	INTEGER*4	REAL*8

DIM Function

The DIM function returns the difference between two integers or real numbers, if that difference is a positive value. If *argument1* is greater than *argument2*, DIM returns the positive difference. If *argument1* is less than *argument2*, DIM returns 0.

Syntax

return value = DIM(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IIDIM	INTEGER*2	INTEGER*2
††JIDIM	INTEGER*4	INTEGER*4
DIM	REAL*4	REAL*4
DDIM	REAL*8	REAL*8

DPROD Function

The DPROD function returns the product of two real number factors as a double precision real number.

Syntax

return value = DPROD(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
DPROD	REAL*4	REAL*8

DREAL Function

††The DREAL function ignores the imaginary portion of a COMPLEX*16 number and returns the real portion as a REAL*8 number.

Syntax

return value = DREAL(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††DREAL	COMPLEX*16	REAL*8

EXP Function

The EXP function returns the constant *e* raised to a specified real or complex exponent. *argument* is the specified exponent.

Syntax

return value = EXP(*argument*)

Specific Name	Type of *argument*	Type of *return value*
EXP	REAL*4	REAL*4
DEXP	REAL*8	REAL*8
CEXP	COMPLEX*8	COMPLEX*8
†† CDEXP	COMPLEX*16	COMPLEX*16

†† If you do not use the −X181 compile option, the CDEXP function must by replaced by ZEXP.

FLOAT Function

The FLOAT function converts an integer number to the REAL*4 data type.

Syntax

return value = FLOAT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† FLOATI	INTEGER*2	REAL*4
†† FLOATJ	INTEGER*4	REAL*4

IABS Function

The IABS function returns the absolute value of an integer number.

Syntax

return value = IABS(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† IIABS	INTEGER*2	INTEGER*2
†† IJABS	INTEGER*4	INTEGER*4

IADDR Function

The IADDR function returns the address of the specified argument.

Syntax

return value = IADDR(*argument*)

Specific Name	Type of *argument*	Type of *return value*
IADDR	all types	INTEGER*4

IAND Function

††The IAND function performs a logical AND of two integer arguments and returns an integer result.

Syntax

return value = IAND(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IIAND	INTEGER*2	INTEGER*2
††JIAND	INTEGER*4	INTEGER*4

††If you do not use the –X181 compile option, the IAND function must by replaced by AND.

Example

The following two lines of code are functionally equivalent:

return value = *argument1*.AND.*argument2*
return value = IAND(*argument1*,*argument2*)

IBCLR Function

††The IBCLR function clears (sets to 0) the specified bit location and returns the integer value of the supplied bit pattern with the specified bit cleared. Bit number *ibit* is cleared in the integer bit pattern *buf*.

Syntax

return value = IBCLR(*buf,ibit*)

Specific Name	Type of *argument*	Type of *return value*
††IIBCLR	INTEGER*2	INTEGER*2
††JIBCLR	INTEGER*4	INTEGER*4

IBITS Function

††The IBITS function returns a bit field, specified by starting bit and length, from an integer bit pattern. *buf* is the bit pattern from which *len* number of bits are extracted starting at bit location *start*.

Syntax

return value = IBITS(*buf,start,len*)

Specific Name	Type of *argument*	Type of *return value*
††IIBITS	INTEGER*2	INTEGER*4
††JIBITS	INTEGER*4	INTEGER*4

IBSET Function

††The IBSET function sets to 1 the specified bit location and returns the integer value of the supplied bit pattern with the specified bit set. Bit number *ibit* is set in the integer bit pattern *buf*.

Syntax

return value = IBSET(*buf,ibit*)

Specific Name	Type of *argument*	Type of *return value*
††IIBSET	INTEGER*2	INTEGER*2
††JIBSET	INTEGER*4	INTEGER*4

ICHAR Function

The ICHAR function converts an ASCII character to its corresponding decimal integer value. *argument* must have a length of 1.

Syntax

return value = ICHAR(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ICHAR	CHARACTER	INTEGER*4

IDIM Function

The IDIM function returns the difference between two integers numbers, if that difference is a positive value. If *argument1* is greater than *argument2*, IDIM returns the positive difference. If *argument1* is less than *argument2*, IDIM returns 0.

Syntax

return value = IDIM(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IIDIM	INTEGER*2	INTEGER*2
††JIDIM	INTEGER*4	INTEGER*4

IDINT Function

The IDINT function converts a real number to an integer. If the number you want to convert is a real number with an absolute value less than 1, the IDINT function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the IDINT function returns the largest integer that does not exceed the value of the original number.

Syntax

return value = IDINT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††IIDINT	REAL*8	INTEGER*2
††JIDINT	REAL*8	INTEGER*4

IDNINT Function

The IDNINT function converts a real number to the nearest integer.

Syntax

return value = IDNINT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ⱡⱡ IIDNNT	REAL*8	INTEGER*2
ⱡⱡ JIDNNT	REAL*8	INTEGER*4

IEOR Function

ⱡⱡ The IEOR function performs an exclusive OR of two integer arguments and returns an integer result.

Syntax

return value = IEOR(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
ⱡⱡ IIEOR	INTEGER*2	INTEGER*2
ⱡⱡ JIEOR	INTEGER*4	INTEGER*4

Example

The following two lines of code are functionally equivalent:

return value = *argument1*.XOR.*argument2*
return value = IEOR(*argument1*,*argument2*)

IFIX Function

The IFIX function converts a real number to an integer. If the number you want to convert is a real number with an absolute value less than 1, the IFIX function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the IFIX function returns the largest integer that does not exceed the value of the original number.

Syntax

return value = IFIX(*argument*)

Specific Name	Type of *argument*	Type of *return value*
++IIFIX	REAL*4	INTEGER*2
++JIFIX	REAL*4	INTEGER*4

INDEX Function

The INDEX function returns an integer value that indicates the starting position of a substring within a string.

Syntax

return value = INDEX(*string,substring*)

Specific Name	Type of *argument*	Type of *return value*
INDEX	CHARACTER	INTEGER*4

INT Function

The INT function converts a real or complex number to an integer. If the number you want to convert is already an integer, the INT function returns that integer.

If the number you want to convert is a real number with an absolute value less than 1, the INT function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the INT function returns the largest integer that does not exceed the value of the original number.

If the number you want to convert is a complex number, the INT function applies the same rules as for real numbers to the real portion of the complex number. The INT function ignores the imaginary portion of a complex number.

Syntax

return value = INT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
–	REAL*4	INTEGER*4
–	REAL*8	INTEGER*4
†† IINT	REAL*4	INTEGER*2
†† JINT	REAL*4	INTEGER*4
†† IIDINT	REAL*8	INTEGER*2
†† JIDINT	REAL*8	INTEGER*4
†† –	COMPLEX*8	INTEGER*2
–	COMPLEX*8	INTEGER*4
†† –	COMPLEX*16	INTEGER*2
†† –	COMPLEX*16	INTEGER*4

IOR Function

††The IOR function performs an inclusive OR of two integer arguments and returns an integer result.

Syntax

return value = IOR(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IIOR	INTEGER*2	INTEGER*2
††JIOR	INTEGER*4	INTEGER*4

††If you do not use the −X181 compile option, the IOR function must by replaced by OR.

Example

The following two lines of code are functionally equivalent:

return value = *argument1*.OR.*argument2*
return value = IOR(*argument1*,*argument2*)

ISHFT Function

††The ISHFT function performs a linear shift, either right or left, of a bit pattern. Bits shifted out either end are discarded, and zero values are added on one end as old values are shifted out the other end. *buf* is the integer value to be shifted, and the absolute value of *number* indicates the number of bits to be shifted. A positive value for *number* indicates a shift to the left is to be performed; a negative value indicates a shift to the right.

Syntax

return value = ISHFT(*buf*,*number*)

Specific Name	Type of *argument*	Type of *return value*
††IISHFT	INTEGER*2	INTEGER*2
††JISHFT	INTEGER*4	INTEGER*4

ISHFTC Function

††The ISHFTC function performs a linear shift, either right or left, of a bit pattern. Bits shifted out either end are shifted in at the other end. The rightmost *field* bits of *buf* are shifted *number* times, where *number* is taken as an absolute value. A positive value for *number* indicates a shift to the left is to be performed; a negative value indicates a shift to the right.

Syntax

return value = ISHFTC(*buf*,*number*,*field*)

Specific Name	Type of *argument*	Type of *return value*
††IISHFTC	INTEGER*2	INTEGER*4
††JISHFTC	INTEGER*4	INTEGER*4

ISIGN Function

The ISIGN function transfers the sign of *argument2* to *argument1*. ISIGN returns the absolute value of *argument1* if *argument2* is greater than or equal to 0, or the negative of *argument1* if *argument2* is less than 0.

Syntax

return value = ISIGN(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IISIGN	INTEGER*2	INTEGER*2
††JISIGN	INTEGER*4	INTEGER*4

LEN Function

The LEN function returns an integer value that indicates the length of a specified character entity.

Syntax

return value = LEN(*argument*)

Specific Name	Type of *argument*	Type of *return value*
LEN	CHARACTER	INTEGER*4

LGE Function

The LGE function compares two strings according to the rules of the ASCII character collating sequence. LGE returns a logical TRUE if *argument1* equals or follows *argument2* in the collating sequence. Otherwise, LGE returns a logical FALSE.

If the string arguments are unequal in length, the LGE function pads the shorter string on the right with blanks.

Syntax

return value = LGE(*argument1*,*argument2*)

Specific Name	**Type of** ***argument***	**Type of** ***return value***
LGE	CHARACTER	LOGICAL*4

LGT Function

The LGT function compares two strings according to the rules of the ASCII character collating sequence. LGT returns a logical TRUE if *argument1* follows *argument2* in the collating sequence. Otherwise, LGT returns a logical FALSE.

If the string arguments are unequal in length, the LGT function pads the shorter string on the right with blanks.

Syntax

return value = LGT(*argument1*,*argument2*)

Specific Name	**Type of** ***argument***	**Type of** ***return value***
LGT	CHARACTER	LOGICAL*4

LLE Function

The LLE function compares two strings according to the rules of the ASCII character collating sequence. LLE returns a logical TRUE if *argument1* equals or precedes *argument2* in the collating sequence. Otherwise, LLE returns a logical FALSE.

If the string arguments are unequal in length, the LLE function pads the shorter string on the right with blanks.

Syntax

return value = LLE(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
LLE	CHARACTER	LOGICAL*4

LLT Function

The LLT function compares two strings according to the rules of the ASCII character collating sequence. LLT returns a logical TRUE if *argument1* precedes *argument2* in the collating sequence. Otherwise, LLT returns a logical FALSE.

If the string arguments are unequal in length, the LLT function pads the shorter string on the right with blanks.

Syntax

return value = LLT(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
LLT	CHARACTER	LOGICAL*4

LOG Function

The LOG function returns the natural logarithm of a real or complex number.

Syntax

return value = LOG(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ALOG	REAL*4	REAL*4
DLOG	REAL*8	REAL*8
CLOG	COMPLEX*8	COMPLEX*8
††CDLOG	COMPLEX*16	COMPLEX*16

††If you do not use the −X181 compile option, the CDLOG function must by replaced by ZLOG.

LOG10 Function

The LOG10 function returns the base 10 logarithm of a real number.

Syntax

return value = LOG10(*argument*)

Specific Name	Type of *argument*	Type of *return value*
ALOG10	REAL*4	REAL*4
DLOG10	REAL*8	REAL*8

MAX Function

The MAX function returns the largest value from a list of integer or real numbers. All arguments must be of the same data type.

Syntax

return value = MAX(*argument1*,*argument2* [,*argumentm*...])

Specific Name	Type of *argument*	Type of *return value*
††IMAX0	INTEGER*2	INTEGER*2
††JMAX0	INTEGER*4	INTEGER*4
AMAX1	REAL*4	REAL*4
DMAX1	REAL*8	REAL*8

MAX0 Function

The MAX0 function returns the largest value from a list of integer numbers. All arguments must be of the same data type.

Syntax

return value = MAX0(*argument1*,*argument2* [,*argumentn*...])

Specific Name	Type of *argument*	Type of *return value*
†† IMAX0	INTEGER*2	INTEGER*2
†† JMAX0	INTEGER*4	INTEGER*4

MAX1 Function

The MAX1 function returns the largest value from a list of real numbers and converts the data type to integer. All arguments must be of the same data type.

Syntax

return value = MAX1(*argument1*,*argument2* [,*argumentn*...])

Specific Name	Type of *argument*	Type of *return value*
†† IMAX1	REAL*4	INTEGER*2
†† JMAX1	REAL*4	INTEGER*4

MIN Function

The MIN function returns the smallest value from a list of integer or real numbers. All arguments must be of the same data type.

Syntax

return value = MIN(*argument1*,*argument2* [,*argumentn*...])

Specific Name	Type of *argument*	Type of *return value*
†† IMIN0	INTEGER*2	INTEGER*2
†† JMIN0	INTEGER*4	INTEGER*4
AMIN1	REAL*4	REAL*4
DMIN1	REAL*8	REAL*8

MIN0 Function

The MIN0 function returns the smallest value from a list of integer numbers. All arguments must be of the same data type.

Syntax

return value = MIN0(*argument1*,*argument2* [,*argumentn*...])

Specific Name	Type of *argument*	Type of *return value*
††IMIN0	INTEGER*2	INTEGER*2
††JMIN0	INTEGER*4	INTEGER*4

MIN1 Function

The MIN1 function returns the smallest value from a list of real numbers and converts the data type to integer. All arguments must be of the same data type.

Syntax

return value = MIN1(*argument1*,*argument2* [,*argumentn*...])

Specific Name	Type of *argument*	Type of *return value*
††IMIN1	REAL*4	INTEGER*2
††JMIN1	REAL*4	INTEGER*4

MOD Function

Syntax

The MOD function returns the remainder from an integer or real number division. MOD divides *argument1* by *argument2* and returns the remainder.

return value = MOD(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IMOD	INTEGER*2	INTEGER*2
††JMOD	INTEGER*4	INTEGER*4
AMOD	REAL*4	REAL*4
DMOD	REAL*8	REAL*8

NINT Function

The NINT function converts a real number to the nearest integer value. Note that the NINT function converts the data type to integer.

Syntax

return value = NINT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† ININT	REAL*4	INTEGER*2
†† JNINT	REAL*4	INTEGER*4
†† IIDNNT	REAL*8	INTEGER*2
†† JIDNNT	REAL*8	INTEGER*4

NOT Function

†† The NOT function returns the bit complement of an integer argument.

Syntax

return value = NOT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† INOT	INTEGER*2	INTEGER*2
†† JNOT	INTEGER*4	INTEGER*4

Example

The following two lines of code are functionally equivalent:

return value = .NOT.*argument*
return value = NOT(*argument*)

REAL Function

The REAL function converts an integer, real, or complex number to the REAL*4 data type. If the number you want to convert is already a REAL*4, the REAL function returns that number.

For a complex number, the REAL function ignores the imaginary portion and returns the real portion.

Syntax

return value = REAL(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††FLOATI	INTEGER*2	REAL*4
††FLOATJ	INTEGER*4	REAL*4
–	INTEGER*4	REAL*4
–	REAL*4	REAL*4
SNGL	REAL*8	REAL*4
–	COMPLEX*8	REAL*4
††–	COMPLEX*16	REAL*4

SIGN Function

The SIGN function transfers the sign of *argument2* to *argument1*. SIGN returns the absolute value of *argument1* if *argument2* is greater than or equal to 0, or the negative of *argument1* if *argument2* is less than 0.

Syntax

return value = SIGN(*argument1*,*argument2*)

Specific Name	Type of *argument*	Type of *return value*
††IISIGN	INTEGER*2	INTEGER*2
††JISIGN	INTEGER*4	INTEGER*4
SIGN	REAL*4	REAL*4
DSIGN	REAL*8	REAL*8

SIN Function

The SIN function returns the trigonometric sine of a real or complex number expressed in radians.

Syntax

return value = SIN(*argument*)

Specific Name	Type of *argument*	Type of *return value*
SIN	REAL*4	REAL*4
DSIN	REAL*8	REAL*8
CSIN	COMPLEX*8	COMPLEX*8
†† CDSIN	COMPLEX*16	COMPLEX*16

†† If you do not use the −X181 compile option, the CDSIN function must by replaced by ZSIN.

SIND Function

†† The SIND function returns the trigonometric sine of a real or complex number expressed in degrees.

Syntax

return value = SIND(*argument*)

Specific Name	Type of *argument*	Type of *return value*
†† SIND	REAL*4	REAL*4
†† DSIND	REAL*8	REAL*8

SINH Function

The SINH function returns the trigonometric hyperbolic sine of a real number.

Syntax

return value = SINH(*argument*)

Specific Name	Type of *argument*	Type of *return value*
SINH	REAL*4	REAL*4
DSINH	REAL*8	REAL*8

SQRT Function

The SQRT function returns the square root of a real or complex number.

Syntax

return value = SQRT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
SQRT	REAL*4	REAL*4
DSQRT	REAL*8	REAL*8
CSQRT	COMPLEX*8	COMPLEX*8
††CDSQRT	COMPLEX*16	COMPLEX*16

††If you do not use the –X181 compile option, the CDSQRT function must by replaced by ZSQRT.

TAN Function

The TAN function returns the trigonometric tangent of a real number expressed in radians.

Syntax

return value = TAN(*argument*)

Specific Name	Type of *argument*	Type of *return value*
TAN	REAL*4	REAL*4
DTAN	REAL*8	REAL*8

TAND Function

††The TAND function returns the trigonometric tangent of a real number expressed in degrees.

Syntax

return value = TAND(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††TAND	REAL*4	REAL*4
††DTAND	REAL*8	REAL*8

TANH Function

The TANH function returns the trigonometric hyperbolic tangent of a real number.

Syntax

return value = TANH(*argument*)

Specific Name	Type of *argument*	Type of *return value*
TANH	REAL*4	REAL*4
DTANH	REAL*8	REAL*8

ZEXT Function

††The ZEXT function copies logical or integer data types to integer and zero extends (fills) the upper bytes with zeros when the data type of *argument* is smaller than the data type of the *return value*.

Syntax

return value = ZEXT(*argument*)

Specific Name	Type of *argument*	Type of *return value*
††IZEXT	LOGICAL*1	INTEGER*2
††IZEXT	LOGICAL*2	INTEGER*2
††IZEXT	INTEGER*2	INTEGER*2
††JZEXT	LOGICAL*1	INTEGER*4
††JZEXT	LOGICAL*2	INTEGER*4
††JZEXT	LOGICAL*4	INTEGER*4
††JZEXT	INTEGER*2	INTEGER*4
††JZEXT	INTEGER*4	INTEGER*4

Chapter 11

Records, Structures, and Unions

This chapter describes records, structures, and unions.

Records

††Records are data structures that are like arrays except that they allow you to group together disparate but related items. For example, you can group time and temperatures, or names and account balances, and so forth. Unlike arrays, the disparate items may even have different data types. The format or structure of a record is defined via a STRUCTURE statement.

Note that "record" in this context bears no relation to a "record", an external file, you can read or write to.

Structure Declarations

A structure declaration defines the form of a record, as a blueprint defines the form of a building. Just as several buildings may be built with different materials from the same blueprint, so several different records may be defined with the same structure but with different names and contents.

Syntax

```
STRUCTURE /structure-name/
            [data type declarations]
            [PARAMETER statements]
            [sub-structure and mapped common statements]
            [union declarations]
END STRUCTURE
```

structure-name is a symbolic name used to reference the structure

One or more structure definition statements (*data type declarations*, *PARAMETER statements*, *sub-structure and mapped common statements*, or *union declarations)* may be specified within the STRUCTURE block and may appear in any order.

Data type declaration statements are allowed in STRUCTURE statements, subject to the following rules:

- All field names must be explicitly typed. Implicit typing, or typing via an IMPLICIT statement has no effect within a STRUCTURE.
- Any valid FORTRAN data type can be used
- Array dimensions, if any, must be specified within the type statement. No DIMENSION statements are allowed.
- Adjustable size arrays and passed length CHARACTER declarations are not allowed
- Field names within a single structure level must be unique, however names used in a substructure may duplicate those specified in an outer structural level

Examples

```
STRUCTURE /ADDR_STRUCT/
     CHARACTER*30    STREET_ADDR
     CHARACTER*15   CITY
     CHARACTER*2     STATE
     INTEGER        ZIP_CODE
END STRUCTURE

STRUCTURE /STUDENT_FILE/
     CHARACTER*25    NAME
     CHARACTER*15   MAJOR
     RECORD      /ADDR_STRUCT/   ADDRESS
END STRUCTURE
```

UNION Declarations

A union declaration allows you to define more than one set of fields within a STRUCTURE that can be used to reference a single data area. Unlike an EQUIVALENCE statement, union declarations allow a single data space to be used alternately. When fields of one map declarations are referenced, the other map declaration values become undefined.

Syntax

```
UNION
          MAP
                    field-declaration
                    [field-declaration]
                    ...
          END MAP

          MAP
                    ...
          END MAP

          [MAP
                    ...
          END MAP]
          ...
END UNION
```

field-declaration is any valid STRUCTURE field definition statement.

Examples

The following example defines the structure DATA_BUFFER, which can alternately contain payroll data or vacation information. A structure of this form might be used to alternately read data from an employee payroll file, or from the vacation file.

```
STRUCTURE /DATA_BUFFER/
     CHARACTER*20   LAST_NAME
     CHARACTER*15   FIRST_NAME
     INTEGER    EMPL_NBR
     UNION
            MAP
                  INTEGER  HOURLY_WAGE
                  INTEGER  YTD_EARNING
            END MAP
            MAP
                  INTEGER  VAC_RATE
                  INTEGER  VAC_DAYS
            END MAP
     END UNION
END STRUCTURE
```

Using RECORDS and STRUCTURES

Records consist of one or more fields, which are either ordinary variables or array elements or substructures themselves. Fields may be referenced either individually, or as an aggregate entity involving the entire record.

Records may be referenced by aggregate fields and/or scalar fields. An aggregate field reference refers to a single record or sub-structure. A scalar field reference refers to an individual variable or array.

Syntax

record_name[.*aggr-name*[.*aggr-name*...]][*scalar-field*]

record_name is the name used in a RECORD statement to identify a record.

aggr-name is the name of a field that is a substructure (a record or a nested structure declaration) within the record structure specified by the record name.

scalar-field is the name of a typed data item defined within the structure declaration.

Because all periods are used to delimit fields in record references, you should not define field names that, when set off with periods, signify relational operators (.EQ., .XOR.), logical constants (.TRUE., .FALSE.), or logical operators (.AND., .NOT., .OR.).

Aggregate Assignment Statement

Aggregate record assignments may be made when the aggregates to the left and right of the equal sign have the same structure. Using the previous example, one could write:

```
GRADUATE(5).NAME            =UNDERGRAD(47).NAME
GRADUATE(5).MAJOR           =UNDERGRAD(47).MAJOR
GRADUATE(5).ADDRESS         =UNDERGRAD(47).ADDRESS
```

or simply

```
GRADUATE(5)                 =UNDERGRAD(47)
```

Scalar Field References

A scalar field reference refers to a single, typed data item and may be treated like an ordinary reference to a FORTRAN variable or array element. The type conversion rules for these references are the same as the rules for variables and array elements. Scalar field references may be used in statement functions and executable statements. However, they may not be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements.

Aggregate Field References in I/O Statements

Aggregate field references may be used in unformatted I/O statements (one I/O record is written no matter how many aggregate and array name references there are in the I/O list) – but they may not be used in formatted or NAMELIST I/O statements.

Appendix A

ASCII and Hexadecimal Conversions

DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX
0	NUL	00	32	SP	20	64	@	40	96	`	60
1	SOH	01	33	!	21	65	A	41	97	a	61
2	STX	02	34	"	22	66	B	42	98	b	62
3	ETX	03	35	#	23	67	C	43	99	c	63
4	EOT	04	36	$	24	68	D	44	100	d	64
5	ENQ	05	37	%	25	69	E	45	101	e	65
6	ACK	06	38	&	26	70	F	46	102	f	66
7	BEL	07	39	'	27	71	G	47	103	g	67
8	BS	08	40	(	28	72	H	48	104	h	68
9	HT	09	41	)	29	73	I	49	105	i	69
10	LF	0A	42	*	2A	74	J	4A	106	j	6A
11	VT	0B	43	+	2B	75	K	4B	107	k	6B
12	FF	0C	44	,	2C	76	L	4C	108	l	6C
13	CR	0D	45	−	2D	77	M	4D	109	m	6D
14	SO	0E	46	.	2E	78	N	4E	110	n	6E
15	SI	0F	47	/	2F	79	O	4F	111	o	6F
16	DLE	10	48	0	30	80	P	50	112	p	70
17	DC1	11	49	1	31	81	Q	51	113	q	71
18	DC2	12	50	2	32	82	R	52	114	r	72
19	DC3	13	51	3	33	83	S	53	115	s	73
20	DC4	14	52	4	34	84	T	54	116	t	74
21	NAK	15	53	5	35	85	U	55	117	u	75
22	SYN	16	54	6	36	86	V	56	118	v	76
23	ETB	17	55	7	37	87	W	57	119	w	77
24	CAN	18	56	8	38	88	X	58	120	x	78
25	EM	19	57	9	39	89	Y	59	121	y	79
26	SUB	1A	58	:	3A	90	Z	5A	122	z	7A
27	ESC	1B	59	;	3B	91	[	5B	123	{	7B
28	FS	1C	60	<	3C	92		5C	124	|	7C
29	GS	1D	61	=	3D	93	]	5D	125	}	7D
30	RS	1E	62	>	3E	94	^	5E	126	~	7E
31	US	1F	63	?	3F	95	_	5F	127	DEL	7F

Appendix B

VAX/VMS Language Extensions

Line Formatting Extensions

Comments may be indicated by C, !, or * in column 1
Comments may be indicated by ! in the statement field
D in column 1 is a debug statement indicator
Continuation convention extension: **tab** character followed by any of the characters **1..9**
Allow up to 99 continuation lines
INCLUDE statement syntax: INCLUDE *pathname*
Not supported for INCLUDE statement:
  Include files in text libraries
  /LIST and /NOLIST options

Lexical Extensions

$, _ in identifiers
Identifiers up to 31 characters long
'*nnn*'O and '*nnn*'X octal and hexadecimal integer constants
Radix–50 constants
Hollerith typeless constants

Declaration Extensions

BYTE data type
DATA statement in any place in program unit
IMPLICIT NONE statement
VOLATILE statement
VIRTUAL statement
Extended PARAMETER statement with additional operators and functions
**n* size qualifier on function names and identifier declarations
Declare multifield records with STRUCTURE statement (no initialization allowed)
Single subscript in EQUIVALENCE of multidimensional array

Initialization Extensions

Initialization in type declaration statements
Initialize CHARACTER variables with integer values
Initialize static and COMMON storage to 0 if not specified (UNIX only)

Expression Extensions

%VAL(), %REF(), %LOC(), accept and ignore %DESCR
Logical type allowed in integer context in expressions
Logical operators apply (bitwise) to integers
Use non-integer type expression in integer context: convert to integer

Built-In Subroutine and Function Extensions

Degree-style trigonometric functions
System subroutines: DATE, IDATE, ERRSNS, EXIT, SECNDS, TIME, RAN, MVBITS

Statement Extensions

DO WHILE statement
END DO statement
Extended range DO loops
Ampersand (&) or asterisk (*) for alternative return
OPTIONS statement with full VAX syntax. Ignore all options except /I4, /NOI4, /D_LINES

Input/Output

NAMELIST
ACCEPT statement
TYPE statement
ENCODE/DECODE statements
Accept full VAX/VMS syntax for OPEN and CLOSE statements, but ignore most options
[1] IBM ('*n*) form of relative record specification

Format Extensions

O (octal) and **Z** (hexadecimal) edit descriptors, including the form **O***n.n*
H edit descriptor on input
Q edit descriptor
$ edit descriptor
Default field widths for IOZLFEDGA edit descriptors match data type
$ and **\0** carriage control – mapped to UNIX carriage control

---

[1] Not a VAX/VMS extension, but supported by the GLS FORTRAN Compiler when the -X181 compile option is set.

Intrinsic Functions

ACOSD
AIMAX0
AIMIN0
AJMAX0
AJMIN0
ASIND
ATAN2D
ATAND
BITEST
BJTEST
BTEST
CDABS
CDCOS
CDEXP
CDLOG
CDSIN
CDSQRT
COSD
DACOSD
DASIND
DATAN2D
DATAND
DCMPLX
DCONJG
DCOSD
DFLOAT
DFLOTI
DFLOTJ
DIMAG
DREAL
DSIND
DTAND
FLOATI
FLOATJ
IAND
IBCLR
IBITS
IBSET
IEOR
IIABS
IIAND
IIBCLR
IIBITS
IIBSET
IIDIM
IIDINT
IIDNNT
IIEOR
IIFIX
IINT
IIOR
IISHFT
IISHFTC
IISIGN
IMAX0
IMAX1
IMIN0
IMIN1
IMOD
ININT
INOT
IOR
ISHFT
ISHFTC
IZEXT
JIABS
JIAND
JIBCLR
JIBITS
JIBSET
JIDIM
JIDINT
JIDNNT
JIEOR
JIFIX
JINT
JIOR
JISHFT
JISHFTC
JISIGN
JMAX0
JMAX1
JMIN0
JMIN1
JMOD
JNINT
JNOT
JZEXT
NOT
SIND
TAND
ZEXT

Compiler Complexity Equal to or Better Than VAX/VMS FORTRAN

20 DO and IF statement nesting
255 Arguments in a CALL or function reference
250 Named COMMON blocks
8 Format group nesting
500 Labels in computed GOTO
40 Parentheses in nested expressions
10 Include file nesting
99 Continuation lines (with compiler option)
132 Source line length in characters
31 Identifier length
2000 Constant length, character, and Hollerith
12 Constant length, Radix-50
7 Array dimensions
250 Number of names in a NAMELIST group

Unimplemented VAX/VMS FORTRAN Extensions

REAL*16, COMPLEX*32
Indexed files
Expressions in FORMAT statements (F<i+j> <k-l>)
DELETE statement for relative files
REWRITE statements for relative files
Initialization of STRUCTUREs
DEFINE FILE statement
FIND statement

Appendix C

Quick Reference

This appendix contains a quick reference of GLS FORTRAN statements, system subroutines, and built-in functions.

GLS FORTRAN Statements

<table>
<tr><th>Statement</th><th>Syntax</th></tr>
<tr><td>ACCEPT</td><td>ACCEPT format spec [,iolist]
ACCEPT * [,iolist]
ACCEPT group</td></tr>
<tr><td>ASSIGN</td><td>ASSIGN label TO symbolic name</td></tr>
<tr><td>Assigment (Arithmetic)
(Character)
(Logical)</td><td>symbolic name = arithmetic exp
symbolic name = character exp
symbolic name = logical exp</td></tr>
<tr><td>BACKSPACE</td><td>BACKSPACE {unit | ([UNIT]=unit [,IOSTAT=status] [,ERR=errlabel])}</td></tr>
<tr><td>BLOCK DATA</td><td>BLOCK DATA [symbolic name]</td></tr>
<tr><td>BYTE</td><td>BYTE symbolic name [,symbolic name]...</td></tr>
<tr><td>CALL</td><td>CALL symbolic name [(argument [,argument] ...)]</td></tr>
<tr><td>CHARACTER</td><td>CHARACTER[*number | (*)] name [*number | (*)] [,name [*number | (*)]] ...</td></tr>
<tr><td>CLOSE</td><td>CLOSE([UNIT=]unit number [,DISPOSE=disposition]
[,STATUS=status] [,ERR=errlabel]
[,IOSTAT=iostatus])</td></tr>
<tr><td>COMMON</td><td>COMMON [/symbolic name/] common list [[,] / [symbolic name] / common list] ...</td></tr>
<tr><td>COMPLEX</td><td>COMPLEX[*number] symbolic name [,symbolic name] ...</td></tr>
<tr><td>CONTINUE</td><td>CONTINUE</td></tr>
<tr><td>DATA</td><td>DATA name list / constant list / [[,] name list / constant list /] ...</td></tr>
<tr><td>DECODE</td><td>DECODE (char,format,loc [,IOSTAT=status] [,ERR=errlabel]) [transfer list]</td></tr>
</table>

(cont.)

Statement	Syntax
DIMENSION	DIMENSION *symbolic name* (*dim* [,*dim*]...) [,*symbolic name* (*dim* [,*dim*]...)] ...
DO	DO *label* [,] *variable* = *exp1*, *exp2* [,*exp3*]
DO WHILE	DO [*label* [,]] WHILE (*expression*)
DOUBLE PRECISION	DOUBLE PRECISION *symbolic name* [,*symbolic name*] ...
ELSE	ELSE..*block*..*block*
ELSE IF	ELSE IF (*logical exp*) THEN
ENCODE	ENCODE (*char*,*format*,*loc* [,IOSTAT=*status*] [,ERR=*errlabel*]) [*transfer list*]
END	END
END DO	END DO
END IF	END IF
ENDFILE	ENDFILE {*unit* | ([UNIT=]*unit* [,IOSTAT=*status*] [,ERR=*errlabel*]) }
ENTRY	ENTRY *symbolic name* [(*dummy* [,*dummy*] ...)]
EQUIVALENCE	EQUIVALENCE (*item list*) [, (*item list*)] ...
EXTERNAL	EXTERNAL *symbolic name* [,*symbolic name*] ...
FORMAT	*label* FORMAT ([*r*] *edit descriptor* [, [*r*] *edit descriptor* ...])
FUNCTION	*type* FUNCTION *symbolic name* (*dummy* [,*dummy*] ...)
GOTO (Assigned) (Computed) (Unconditional)	GOTO *variable name* [[,] (*list*)] GOTO (*list*) [,] *arith exp* GOTO *label*

IF (Arithmetic)
(Block)
(Logical)
	IF (*arithmetic exp*) *label1*, *label2*, *label3*
	IF (*logical exp*) THEN...*statements* END IF
	IF (*logical exp*) *statement*

IMPLICIT	IMPLICIT *type* [**length*] (*letter*) [, (*letter*)] ... IMPLICIT NONE
INCLUDE	INCLUDE *file-spec*

(cont.)

Statement	Syntax
INQUIRE	INQUIRE({FILE=*'filename'* | [UNIT=]*unit*} [,ACCESS=*acc*] [,BLANK=*bl*] [,CARRIAGECONTROL=*car*] [,DIRECT=*dir*] [,ERR=*errlabel*] [,EXIST=*exstat*] [,FORM=*formtype*] [,FORMATTED=*form*] [,IOSTAT=*iostat*] [,NAME=*fn*] [,NAMED=*namestat*] [,NEXTREC=*next*] [,NUMBER=*num*] [,OPENED=*opstat*] [,ORGANIZATION=*org*] [,RECL=*reclen*] [,SEQUENTIAL=*seq*] [,UNFORMATTED=*unf*])
INTEGER	INTEGER[**number*] *symbolic name* [,*symbolic name*]...
INTRINSIC	INTRINSIC *symbolic name* [,*symbolic name*]...
LOGICAL	LOGICAL[*number] *symbolic name* [,*symbolic name*]...
NAMELIST	NAMELIST */group/ namelist* [[,]*/group/ namelist*
OPEN	OPEN({ [UNIT]=*unit*} [,ACCESS=*acc*] [,ASSOCIATEVARIABLE=*var*] [,BLANK=*bl*] [,CARRIAGECONTROL=*car*] [,DISPOSE=*disposition*] [,ERR=*errlabel*] [,FILE=*name*] [,FORM=*formtype*] [,IOSTAT=*iostatus*] [,ORGANIZATION=*org*] [,RECL=*reclen*] [,STATUS=*status*] [,USEROPEN=*routine*])
OPTIONS	OPTIONS *option* [,*option*]
PARAMETER	PARAMETER (*symbolic name* = *expression* [,*symbolic* = *expression*] ...) PARAMETER *symbolic name* = *expression*
PAUSE	PAUSE [*message*]
PRINT	PRINT *format* [,*output list*]
PROGRAM	PROGRAM *symbolic name*
READ	READ ({ [UNIT=]*unit*} [[FMT=]*format* | [NML=]*group*] [,REC=*recnum*] [,IOSTAT=*iostat*] [,ERR=*errlabel*] [,END=*endlabel*]) [*input list*] READ format [*input list*]
REAL	REAL[**number*] *symbolic name* [,*symbolic name*]...
RETURN	RETURN [*integer exp*]
REWIND	REWIND {*unit* | ([UNIT=]*unit* [,IOSTAT=*iostatus*] [,ERR=*errlabel*]) }
SAVE	SAVE [*symbolic name* [,*symbolic name*]...]
STOP	STOP [*message*]
SUBROUTINE	SUBROUTINE *symbolic name* [(*dummy* [,*dummy*] ...)]

(cont.)

Statement	Syntax
TYPE	TYPE *format spec* [,*iolist*] TYPE * [,*iolist*] TYPE *group*
VIRTUAL	VIRTUAL
VOLATILE	VOLATILE *nlist*
WRITE	WRITE ([UNIT=] *unit* [, [FMT=] *format* | [NML=]*group*] [,REC=*recnum*] [,IOSTAT=*iostat*] [,ERR=*errlabel*]) [*output list*]

System Subroutines

Subroutine	Syntax	Description
DATE	CALL DATE (*datebuf*)	Returns system date as a character string.
ERRSNS	CALL ERRSNS (*ierrnum,iosts,iostv,iunit,icondval*)	Returns information about most recent runtime error.
EXIT	CALL EXIT [(*istatus*)]	Terminates program, closes files, and exits to operating system.
IDATE	CALL IDATE (*imon,iday,iyear*)	Returns integer values for month, day, and year
MVBITS	CALL MVBITS (*src,sstart,len,dst,dstart*)	Copies a bit pattern from one location to another.
RAN	*ranvalu* = RAN (*iseed*)	Returns a pseudo-random number between 0.0 and 1.0 inclusive.
SECNDS	*delta* = SECNDS (*t*)	Returns difference between supplied value and current system time.
TIME	CALL TIME (*timebuf*)	Returns system time as a character string.

Built–in Functions

Function	Syntax	Description
%VAL	%VAL(*argument*)	Passes arguments by value.
%REF	%REF(*argument*)	Passes arguments by reference.
%LOC	%LOC(*argument*)	Returns the internal address of the storage element *argument* as an INTEGER*4 value.

Glossary

The FORTRAN–77 standard defines the concepts and terminology specific to the language. This glossary presents the GLS FORTRAN terms.

–A–

array
Sequence of data items collectively identified with one unique symbolic name and a data type.

array elements
Individual data items that form an array. To reference a particular element in an array, specify the array name with a subscript. The subscript value is an integer expression that determines which element is referenced.

array declarator
Symbolic name and number of dimensions in an array. The number of dimensions determines the number and configuration of array elements.

association
Enables data to be identified by different symbolic names within the same program unit or in different program units within the same executable program. There are four forms of association: common, equivalence, argument, and entry.

–B–

block data subprogram
Nonexecutable program unit used to provide initial values for variables and array elements in named common blocks. A block data subprogram has a BLOCK DATA statement as its first statement.

–C–

character storage unit
Amount of storage required to hold one character of data. GLS FORTRAN uses one byte of storage per character.

comment line
Character sequence within the program code, used to provide program documentation. A comment line does not affect an executable program in any way. All comment lines start with the letter C or an asterisk.

constant
Program entity that has an unchanging value during the execution of a program. Constants can be arithmetic constants, logical constants, or character constants.

continuation line
Used to contain portions of a GLS FORTRAN statement that exceed the 72 character columns available in the initial line for statement syntactic items. A statement can have up to nineteen continuation lines.

–D–

defined
Definition status of a program entity. A defined entity has a value that does not change until the entity is redefined with a different value. An entity must be defined before it can be referenced.

definition status
Defined or undefined condition of a syntactic entity.

dummy argument
Symbolic name or an asterisk (*) used in the argument list of a procedure. Symbolic name dummy arguments hold a place for actual arguments passed to the procedure in the procedure reference. Symbolic name dummy arguments can be variables, arrays, array elements, functions, or subroutines, and must correspond to the number, type, and sequence of actual arguments in the procedure reference. An asterisk dummy argument indicates that the corresponding actual argument in a subroutine reference is an alternate return specifier for the subroutine.

–E–

entity
Generic term that refers to any language or program element, such as a program unit, a procedure, a variable, or an array. The term syntactic entity or syntactic item refers to individual elements that make up statements and expressions, such as statement labels, keywords, symbolic names, constants, operators, and special characters.

executable program
Program units that consists of a main program and any number, including zero, of subprograms and external procedures. An executable program cannot have more than one main program.

executable statement
Any statement that specifies some processing action, such as a GOTO or RETURN statement.

external procedure
Subroutine or external function specified outside of the program unit that calls or references it. External procedures can be written in another language for use in a GLS FORTRAN program.

–F–

function subprogram
Executable procedure that can be referenced in an expression. A function subprogram returns a value to the expression that references it. There are three categories of functions: intrinsic, statement, and external.

–I–

initial line
First line of a statement. If the statement exceeds the initial line, you can use up to nineteen continuation lines.

initially defined
Definition status of an entity. An entity is initially defined if it is assigned a value in a DATA statement.

–K–

keyword
Sequence of letters that identify a statement, intrinsic function, or statement separator. Examples are DIMENSION, CONTINUE, ABS, SQRT, THEN, and TO.

–L–

list
Nonempty sequence of syntactic entities separated by commas. The entities in the list are called list items.

–M–

main program
Program unit that receives control from the operating system to begin execution of a GLS FORTRAN program. Main programs can execute isolated from any other program unit or can call and reference subprograms during execution. However, you cannot reference the main program from a subprogram. There can be only one main program in an executable GLS FORTRAN program.

—N—

nonexecutable statement
Statements that classify and define program units, specify entry points in subprograms, specify editing information, and specify initial values and execution characteristics for data.

numeric storage unit
Amount of storage required to hold an integer, real, or logical numeric value. A double precision or complex numeric value uses two numeric storage units in a storage sequence. The storage unit establishes a means of referring to data storage without implying a specific storage technology.

—P—

procedure, procedure subprogram
Subroutine and function subprograms. There are three categories of function subprograms: intrinsic functions, statement functions, and external functions. The term external procedure refers to subroutines and external functions only. You can write external procedures in another programming language for use in a GLS FORTRAN program.

program unit
Sequence of GLS FORTRAN statements and optional comment lines. A program unit is either a main program or subprogram.

—R—

reference
Applies to syntactic entities and function procedures. To reference a syntactic entity means to use the name of an entity in a statement that requires the value of that particular entity for execution within the program context. To reference a function procedure means to use the name of a function in an expression or statement that requires that particular function operation for execution within the program context.

—S—

scope
Extent to which a given symbolic name or statement label can affect a program. For example, a statement label has the scope of a program unit. A statement label in one program unit does not affect any other program unit.

sequence
Set of elements ordered by a one-to-one correspondence with the numbers 1, 2, 3, through n. The number of elements in a sequence is n. An empty sequence contains no elements.

statement
Sequence of syntactic items. Except for assignment and statement function statements, all statements begin with a keyword. Statements are written in one or more lines.

statement label
Sequence of one to five digits. One of the digits must be nonzero. Statement labels are used to identify specific statements in a program.

subprogram
Program unit that is called or referenced from either the main program or another subprogram. There are two classes of subprograms: block data and procedures.

subroutine subprogram
External procedure. An external procedure is specified outside of the program unit that calls it. A subroutine is referenced with the CALL statement.

substring
Contiguous sequence of characters that represents a portion of a character datum. A character datum is a string of one or more characters. Substrings are identified with a substring name. The substring name is used to define and reference the substring.

symbolic name
A sequence of alphanumeric characters. The first character in a symbolic name must be a letter. Symbolic names can identify constants, variables, arrays, main programs, subprograms, common blocks, and dummy procedures. Character sequences that serve within the program context as format edit descriptors and keywords are not considered symbolic names.

syntactic item, syntactic entity
Generic terms that refer to individual elements that make up statements and expressions, such as statement labels, keywords, symbolic names, constants, operators, and special characters. See entity for more information.

–V–

variable
Entity that can assume a changing value during program execution through implicit or explicit redefinition. A variable has both a name and a type.

VAX/VMS
The trademark of a proprietary computer system from Digital Equipment Corporation.

INDEX

arguments, 6-4
  actual, 6-4, 6-6
  calling, 6-4
  dummy, 6-4, 6-5
  formal, 6-4
array declarators, 4-6
  adjustable, 4-9
  assumed-sized, 4-10
arrays, 4-6, 4-7
  array declarator, 4-6
  declare an array, 4-6
  dimension declarators, 4-6
  multidimensional, 4-9
  one-dimensional, 4-8
assignment statements, 9-1
  arithmetic, 9-8
  ASSIGN, 9-7
  character, 9-9
  DATA, 9-21
  LOGICAL, 9-57
  logical, 9-56
auxiliary I/O statements, 7-6
  CLOSE, 7-7
  INQUIRE, 7-7
  OPEN, 7-6

built-in functions, 10-6, 10-7
  %LOC, 10-7
  %REF, 10-6
  %VAL, 10-6

carriage control, 9-51
character, 5-7
comment lines, 2-4
complex, 5-7
constant, 4-1
  hexadecimal, 4-2, 4-3
  octal, 4-2, 4-3
  Radix-50, 4-4
continuation lines, 2-6
control statements, 9-2
  arithmetic IF, 9-45
  assigned GOTO, 9-42
  block IF, 9-46
  CALL, 9-14
  computed GOTO, 9-43
  CONTINUE, 9-20
  DO, 9-25
  DO WHILE, 9-27
  ELSE, 9-29
  ELSE IF, 9-30
  END, 9-32
  END DO, 9-33
  END IF, 9-34
  INCLUDE, 9-49
  logical IF, 9-47
  OPTIONS, 9-68
  PAUSE, 9-73
  RETURN, 9-80
  STOP, 9-83
  unconditional GOTO, 9-44

data transfer, 7-7
  editing performed during the transfer, 7-8
  formatted, 7-7
  unformatted, 7-9
data transfer statements, 7-5
  DECODE, 7-6
  ENCODE, 7-6
  PRINT, 7-5
  READ, 7-5
  WRITE, 7-5
data types
  character, 3-1, 3-5
  complex, 3-1, 3-4
  double precision, 3-1, 3-4
  Hollerith, 3-1, 3-6
  integer, 3-1
  logical, 3-1, 3-5
  real, 3-1, 3-3
  size qualifier, 3-1
  specify a data type, 3-6
dimension declarators, 4-6

edit descriptors
  $, 8-18
edit descriptors, 8-3
  A, D, E, F, G, I, and L, 8-3

INDEX [continued]

edit descriptors *[continued]*
- A[w], 8-4
- conditional line-termination descriptor :, 8-23
- D, E, F, and G, 8-10
- D, E, F, G, and I, 8-5
- Dw.d and Ew.d[Ee], 8-6
- Fw.d, 8-7, 8-8
- Gw.d[Ee], 8-8
- Iw and Iw.m, 8-11
- kP, 8-20
- line-termination descriptor /, 8-22
- Lw, 8-15
- nH, 8-17
- nonrepeatable, 8-10, 8-16
- numeric, 8-5
- O and Z, 8-3, 8-5
- Ow[.m], 8-12
- repeatable, 8-3
- S, SS, and SP, 8-21
- Tc, TLc, TRc, and nX, 8-21
- Zw[.m], 8-13

expressions, 5-1, 5-4
- arithmetic, 5-3
- character, 5-5
- logical, 5-8
- precedence, 5-2
- relational, 5-6

extended-range DO loop, 9-26
external procedure, 6-5

fields
- aggregate, 11-4
- scalar, 11-4

file, 7-1
- accessing, 7-3

file disposition, 9-65
file organization, 9-52
file positioning statements, 7-6
- BACKSPACE, 7-6
- ENDFILE, 7-6
- REWIND, 7-6

floating-point, 8-6
format specification, 8-1
- character, 8-2
- editing of character, logical, and numeric data, 8-3
- format, 8-1
- format control, 8-2
- list-directed formatting, 8-23
- list-directed input, 8-24
- list-directed output, 8-26
- repeatable edit descriptors, 8-3

functions, 6-6
- external, 6-7
- statement, 6-7

hexadecimal constant, 4-2, 4-3

I/O statements, 7-5, 9-3
- ACCEPT, 9-6
- auxiliary I/O, 7-6
- BACKSPACE, 9-10
- CLOSE, 9-16
- DECODE, 9-23
- ENCODE, 9-31
- ENDFILE, 9-35
- file positioning, 7-6
- FORMAT, 9-40
- INQUIRE, 9-50
- OPEN, 9-64
- PRINT, 9-74
- READ, 9-77
- REWIND, 9-81
- TYPE, 9-85
- WRITE, 9-88

I/O unit, 7-2
intrinsic functions, 10-8 *to* 10-37
- ABS function, 10-8
- ACOS function, 10-9
- ACOSD function, 10-9
- AIMAG function, 10-9
- AINT function, 10-10
- AMAX0 function, 10-10
- AMIN0 function, 10-11
- ANINT function, 10-11
- ASIN function, 10-11
- ASIND function, 10-12

INDEX [continued]

intrinsic functions *[continued]*
- ATAN function, 10–12
- ATAN2 function, 10–13
- ATAN2D function, 10–13
- ATAND function, 10–12
- BTEST function, 10–13
- CHAR function, 10–14
- CMPLX function, 10–14
- CONJG function, 10–15
- COS function, 10–15
- COSD function, 10–15
- COSH function, 10–16
- DBLE function, 10–16
- DCMPLX function, 10–17
- DFLOAT function, 10–17
- DIM function, 10–18
- DPROD function, 10–18
- DREAL function, 10–18
- EXP function, 10–19
- FLOAT function, 10–19
- IABS function, 10–19
- IADDR function, 10–20
- IAND function, 10–20
- IBCLR function, 10–21
- IBITS function, 10–21
- IBSET function, 10–21
- ICHAR function, 10–22
- IDIM function, 10–22
- IDINT function, 10–22
- IDNINT function, 10–23
- IEOR function, 10–23
- IFIX function, 10–24
- INDEX function, 10–24
- INT function, 10–25
- IOR function, 10–26
- ISHFT function, 10–26
- ISHFTC function, 10–27
- ISIGN function, 10–27
- LEN function, 10–27
- LGE function, 10–28
- LGT function, 10–28
- LLE function, 10–29
- LLT function, 10–29
- LOG function, 10–30
- LOG10 function, 10–30
- MAX function, 10–30
- MAX0 function, 10–31
- MAX1 function, 10–31
- MIN function, 10–31
- MIN0 function, 10–32
- MIN1 function, 10–32
- MOD function, 10–32
- NINT function, 10–33
- NOT function, 10–33
- REAL function, 10–34
- SIGN function, 10–34
- SIN function, 10–35
- SIND function, 10–35
- SINH function, 10–35
- SQRT function, 10–36
- TAN function, 10–36
- TAND function, 10–36
- TANH function, 10–37
- ZEXT function, 10–37

NAMELIST record format, 9–60
NAMELIST-directed I/O, 9–58 *to* 9–63
nonrepeatable edit descriptors, 8–16

octal constant, 4–2, 4–3
OPEN routine
- user-supplied, 9–66

procedures, 6–4
program lines, 2–1
- column-based format, 2–2, 2–3
- comment lines, 2–2
- continuation lines, 2–1
- free-field format, 2–2
- initial line, 2–1

program unit, 6–4
program units, 6–1, 6–2
- block data subprograms, 6–2, 6–3
- procedures, 6–2

Radix-50, 4–4, 4–5
record, 7–1

INDEX [continued]

special characters, 2-5
specification statements, 9-4
  BYTE, 9-13
  CHARACTER, 9-15
  COMMON, 9-18
  COMPLEX, 9-19
  DIMENSION, 9-24
  DOUBLE PRECISION, 9-28
  EQUIVALENCE, 9-37
  EXTERNAL, 9-39
  IMPLICIT, 9-48
  IMPLICIT NONE, 9-48
  INTEGER, 9-54
  INTRINSIC, 9-55
  NAMELIST, 9-58
  PARAMETER, 9-70
  REAL, 9-79
  SAVE, 9-82
  VIRTUAL, 9-86
  VOLATILE, 9-87
statements, 9-1
  ACCEPT, 9-6
  arithmetic, 9-8
  arithmetic IF, 9-45
  ASSIGN, 9-7
  assigned GOTO, 9-42
  BACKSPACE, 9-10
  BLOCK DATA, 9-12
  block IF, 9-46
  BYTE, 9-13
  CALL, 9-14
  CHARACTER, 9-15
  character, 9-9
  CLOSE, 9-16
  COMMON, 9-18
  COMPLEX, 9-19
  computed GOTO, 9-43
  CONTINUE, 9-20
  DATA, 9-21
  DECODE, 9-23
  DIMENSION, 9-24
  DO, 9-25
  DO WHILE, 9-27
  DOUBLE PRECISION, 9-28
  ELSE, 9-29
  ELSE IF, 9-30
  ENCODE, 9-31
  END, 9-32
  END DO, 9-33
  END IF, 9-34
  ENDFILE, 9-35
  ENTRY, 9-36
  EQUIVALENCE, 9-37
  EXTERNAL, 9-39
  FORMAT, 9-40
  FUNCTION, 9-41
  IMPLICIT, 9-48
  IMPLICIT NONE, 9-48
  INCLUDE, 9-49
  INQUIRE, 9-50
  INTEGER, 9-54
  INTRINSIC, 9-55
  LOGICAL, 9-57
  logical, 9-56
  logical IF, 9-47
  NAMELIST, 9-58
  nonexecutable, 2-1
  OPEN, 9-64
  OPTIONS, 9-68
  PARAMETER, 9-70
  PAUSE, 9-73
  PRINT, 9-74
  PROGRAM, 9-76
  READ, 9-77
  REAL, 9-79
  RETURN, 9-80
  REWIND, 9-81
  SAVE, 9-82
  STOP, 9-83
  SUBROUTINE, 9-84
  TYPE, 9-85
  unconditional GOTO, 9-44
  VIRTUAL, 9-86
  VOLATILE, 9-87
  WRITE, 9-88
structural statements, 9-5
  BLOCK DATA, 9-12
  ENTRY, 9-36

INDEX [continued]

structural statements *[continued]*
FUNCTION, 9-41
PROGRAM, 9-76
SUBROUTINE, 9-84
structure declaration, 11-1
subroutine, 6-5
substring, 4-11
symbolic name, 2-5
system subroutines
DATE, 10-2
ERRSNS, 10-2
EXIT, 10-2
GETARG, 10-3
IARGC, 10-3
IDATE, 10-3
MVBITS, 10-4
RAN, 10-4
SECNDS, 10-5
TIME, 10-5

union declaration, 11-3

variables, 4-5

Please comment on the publication's completeness, accuracy, and readability. We also appreciate any general suggestions you may have to improve this publication.

If you found any errors in this publication, please specify the page number or include a copy of the page with your remarks.

Your comments will be promptly investigated and appropriate action will be taken.

☐ If you require a written answer please check this box and include your address below.

Comments: ______________________________

Manual Title ______________________________

Manual Order Number ______________________ Issue Date ____________

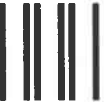
Name ______________________ Position ____________

Company ______________________________

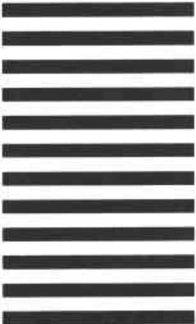
Address ______________________________

______________________ Telephone (   ) ____________

Please fold and tape.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 3624 FT. LAUDERDALE, FL 33309

POSTAGE WILL BE PAID BY ADDRESSEE

MODULAR COMPUTER SYSTEMS, INC.
1650 W. McNAB ROAD
P.O. BOX 6099
FT. LAUDERDALE, FL 33340–6099

Attention: TECHNICAL DOCUMENTATION, M.S. 880

MODCOMP, founded in 1970, is a worldwide supplier of high-performance, real-time computer systems, products, and services to the industrial automation, energy, transportation, scientific, and communications markets. MODCOMP is an AEG company.

Corporate Headquarters:
Modular Computer Systems, Inc.
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33340-6099
Tel: (305) 974-1380
Twx: 510-956-9414

International Headquarters:
Modular Computer Services, Inc.
The Business Centre
Molly Millars Lane
Wokingham, Berkshire
RG11 2JQ, UK
Tel: 0734-786808, TLX: 851849149

Latin American-Caribbean Headquarters:
Modular Computer Systems, Inc.,
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33340-6099
Tel: (305) 977-1795, TLX: 3727852

Canadian Headquarters:
MODCOMP Canada, Ltd.,
400 Matheson Blvd. East, Unit 24
Mississauga, Ontario
Canada L4Z 1N8
Tel: (416) 890-0666
Fax: (416) 890-0266

Sales & Service Locations
Throughout the World

Copyright © 1989, Modular Computer Systems, Inc.
MODCOMP is a registered trademark of Modular Computer Systems, Inc.

Printed in U.S.A. • "The technical contents of this document, while accurate as of the date of publication, are subject to change without notice."

AEG

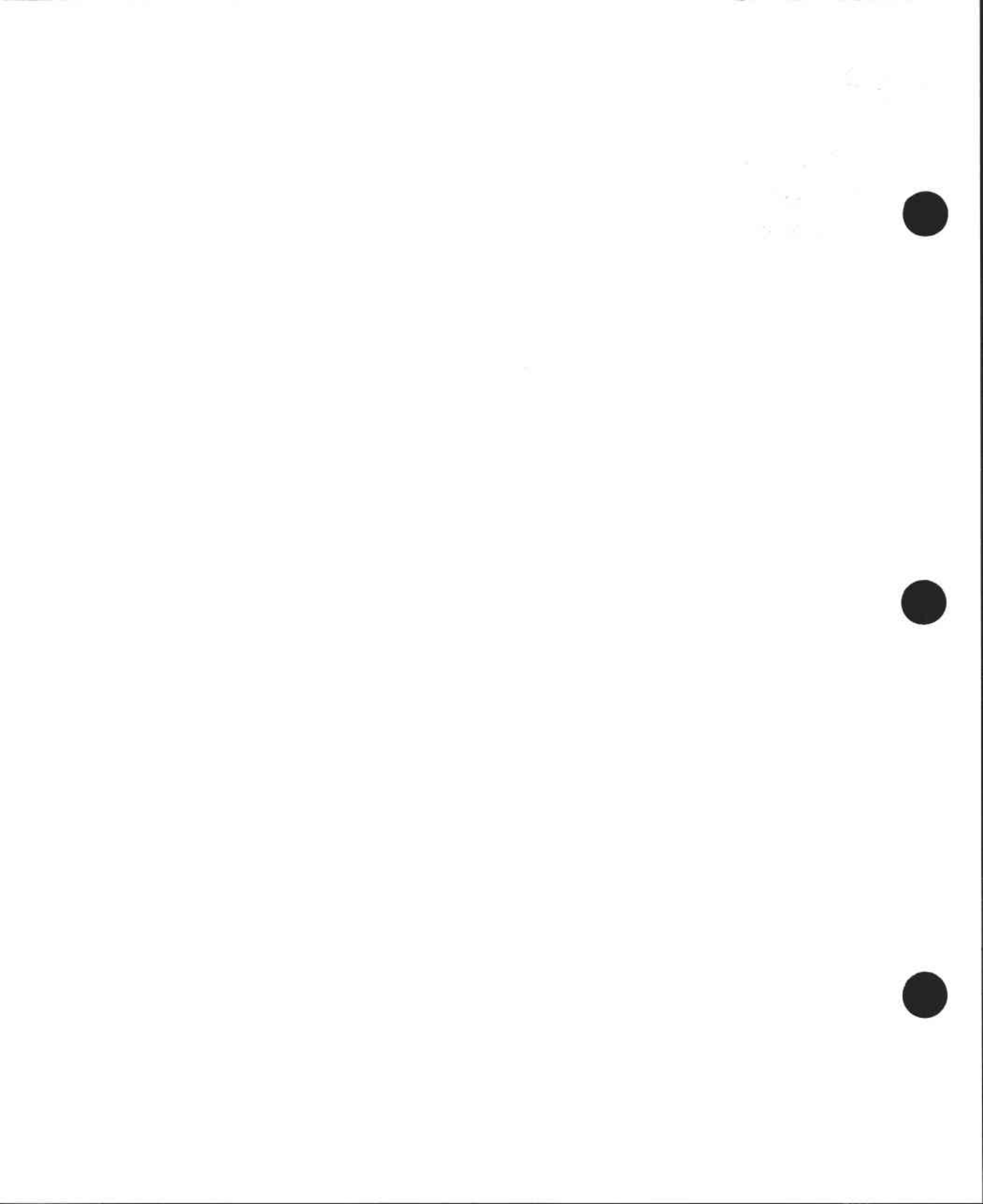
Library Reference Manual

GLS™ FORTRAN Interface to System Services

Open Architecture Systems

215–856001–002

MODCOMP



AEG

Library Reference Manual

GLS™ FORTRAN Interface to System Services

Open Architecture Systems

MODCOMP

Property
of
LOGICAL DATA CORPORATION

RECEIVED DEC 2 2 19[illegible]

MANUAL HISTORY

Manual Number: 215-856001-002

Title: GLS FORTRAN Interface to System Services Library Reference Manual

Revision Level	Date Issued	Description
000	11/89	Initial Issue.
001	05/90	Reissue. Corrected TIMES(2F) and pathnames of installed directories to reflect INSTALL script.
002	02/91	Reissue. Compatible with B.0 of GLS compilers for open architecture systems. Added new system calls.

Contents subject to change without notice.

MODCOMP, Tri-Dimensional, Tri-D, REAL/IX, and CLASSIC are registered trademarks of Modular Computer Systems, Inc.
GLS is a trademark of Modular Computer Systems, Inc.
UNIX is a registered trademark of UNIX System Laboratories, Inc.

Copyright © 1989, 1991 by Modular Computer Systems, Inc.
All Rights Reserved.
Printed in the United States of America.

PROPRIETARY NOTICE

THE INFORMATION AND DESIGNS DISCLOSED HEREIN WERE ORIGINATED BY AND ARE THE PROPERTY OF MODULAR COMPUTER SYSTEMS, INC. (MODCOMP). MODCOMP RESERVES ALL PATENT, PROPRIETARY DESIGN, MANUFACTURING, REPRODUCTION, USE, AND SALES RIGHTS THERETO, AND RIGHTS TO ANY ARTICLE DISCLOSED THEREIN, EXCEPT TO THE EXTENT RIGHTS ARE EXPRESSLY GRANTED TO OTHERS. THE FOREGOING DOES NOT APPLY TO VENDOR PROPRIETARY PARTS.

SPECIFICATIONS REMAIN SUBJECT TO CHANGE IN ORDER TO ALLOW THE INTRODUCTION OF DESIGN IMPROVEMENTS.

FOR GOVERNMENT USE THE FOLLOWING SHALL APPLY:

RESTRICTED RIGHTS LEGEND

USE, DUPLICATION, OR DISCLOSURE BY THE GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN RIGHTS IN DATA CLAUSES DOE 952.227-75, DOD 52.227-7013, AND NASA 18-52.227-74 (AS THEY APPLY TO APPROPRIATE AGENCIES).

MODULAR COMPUTER SYSTEMS, INC.
1650 WEST McNAB ROAD
P.O. BOX 6099
FORT LAUDERDALE, FL 33340-6099

THIS MANUAL IS SUPPLIED WITHOUT REPRESENTATION OR WARRANTY OF ANY KIND. MODULAR COMPUTER SYSTEMS, INC. THEREFORE ASSUMES NO RESPONSIBILITY AND SHALL HAVE NO LIABILITY OF ANY KIND ARISING FROM THE SUPPLY OR USE OF THIS PUBLICATION OR ANY MATERIAL CONTAINED HEREIN.

PREFACE

Audience

This manual is written for programmers using the General Language System (GLS) FORTRAN compiler. It assumes you have previous FORTRAN programming experience.

Subject

This manual describes how to install and use the GLS FORTRAN system calls library. Each system call includes a synopsis, description, and related error definitions.

Product Requirements

The FORTRAN interface system calls are used with the GLS FORTRAN compiler on an open architecture system under the REAL/IX Operating System.

Related Publications

Refer to the following manuals for additional information. When you order additional manuals, use the manual order number listed below. The most current revision level (REV) will be shipped.

GLS FORTRAN Language Reference Manual (210-856001-REV)
   Describes the form and interpretation of language elements specific to GLS FORTRAN.

GLS Programming Guide for Open Architecture Systems (216-856005-*REV*)
   Describes how to install and execute the GLS C, FORTRAN, and Pascal compilers.

GLS Symbolic Debugger User's Guide (216-856007-*REV*)
   Describes how to install and use the GLS Symbolic Debugger (mdb) and its rules and sytax.

REAL/IX Operating System, 97xx Systems Concepts and Characteristics (205-855001-*REV*)
   Gives an overview of the internals of the REAL/IX Operating System and introduces the available tools and facilities.

REAL/IX Reference Manual
Sections 1, 1M, and 1R
Sections 2, 3, and 5
   Contains manual pages for user commands (Section 1), administrative commands (Section 1M), realtime commands (Section 1R), system calls (Section 2), library routines (Sections 3C, 3M, 3N, 3S, and 3X), and miscellaneous information (Section 5).

MODCOMP Service and Assistance

MODCOMP® offers a variety of programs and services that demonstrate our commitment to customer satisfaction. Our Technical Education department provides comprehensive hands-on instruction either at our facilities or at customer-designated sites. Our worldwide field service organization is ready to provide installation assistance, free service during the warranty period, and flexible service programs tailored to your requirements.

Questions, Problems, and Suggestions

Your MODCOMP sales and service representatives can help you with any questions, problems, or suggestions you may have regarding our products and services. In addition, for your convenience MODCOMP maintains the following toll-free telephone numbers at which we can be reached for questions, problems, and suggestions. Please feel free to use the following numbers:

- **For questions, sales information, or suggestions**: in the U.S. and Canada, 1-800-255-2066 (In countries outside the U.S. and Canada, please call your regional sales support office or 1-305-974-1380 extension 1800 worldwide.)

- **For service**: in Florida, 1-800-432-1405; in the U.S., 1-800-327-8928; in Canada, 1-416-890-0666 (In countries outside the U.S. and Canada, please call your regional service/support office.)

- **For Technical Education information**: in the U.S., 1-305-977-1708 (In countries outside the U.S., please call your regional support office.)

For comments about documentation, please use the response form at the back of this manual.

REVISION SUMMARY

002

- New system calls were added

ACCEPT(2F)	Accepts a connection on a socket
BIND(2F)	Binds a name to a socket
CONNECT(2F)	Initiates a connection on a socket
GETMSG(2F)	Gets next message off a stream
GETPEERNAME(2F)	Gets name of connected peer
GETSOCKNAME(2F)	Gets socket name
GETSOCKOPT(2F)	Gets and sets options on sockets
LISTEN(2F)	Listens for connections on a socket
MEMCTL(2F)	Controls write/execute attributes of memory
PATHCONF(2F)	Gets configurable pathname variables
PUTMSG(2F)	Sends a message on a stream
READLINK(2F)	Reads value of a symbolic link
RECV(2F)	Receives a message from a socket
RENAME(2F)	Changes the name of a file
SEND(2F)	Sends a message from a socket
SETGROUPS(2F)	Sets group access list
SETPGID(2F)	Sets process group ID for job control
SETPSR(2F)	Sets/gets Processor Status Register (PSR)
SIGACTION(2F)	Examines or changes signal action
SIGPENDING(2F)	Examines pending signals
SIGPROMASK(2F)	Examines and changes blocked signals
SIGSETOPS(2F)	Manipulates signal sets
SLEEP(2F)	Suspends execution for interval
SOCKET(2F)	Creates an endpoint for communication
SYMLINK(2F)	Makes symbolic link to a file
UMOUNT(2F)	Unmounts a file system
WAITPID(2F)	Waits for child process to stop or terminate

001

- Corrected TIMES(2F) and pathnames of installed directories to reflect INSTALL script.

TABLE OF CONTENTS

SYSTEM CALLS (2F)

intro .... introduction to system calls and error numbers
absinterval .... set the expiration time of a process interval timer
acancel .... cancel one or more asynchronous I/O requests
accept .... accepts a connection on a socket
access .... determine accessibility of a file
acct .... enable or disable process accounting
alarm .... set a process alarm clock
aread .... read from file in an asynchronous manner
arinit .... initialize structures before requesting an asynchronous read
arwfree .... free internal resources for asynchronous I/O from the process
awinit .... initialize structures before requesting an asynchronous write
awrite .... write to file in an asynchronous manner
bind .... binds a name to a socket
brk .... change data segment space allocation
bsfree .... free a binary semaphore
bsget .... get a binary semaphore
chdir .... change working directory
chmod .... change mode of file
chown .... change owner and group of a file
chroot .... change root directory
cisema .... wait for a connected interrupt
close .... close a file descriptor
connect .... initiates a connection on a socket
creat .... create a new file or rewrite an existing one
dup .... duplicate an open file descriptor
estat .... get extended file status
evctl .... event control operations
evget .... get an event identifier
evpost .... post an event to a process
evrcv .... receive any queued event
evrcvl .... receive any queued event from a specified list
evrel .... release an event identifier
exec .... execute a file
exit .... terminate process
fcntl .... file control
fork .... create a new process
ftime .... get time
getdents .... read directory entries and put in a file system independent format
getinterval .... get the current value for a process interval timer
getmsg .... get next message off a stream
getpeername .... get name of connected peer
getpid .... get process, process group, and parent process IDs
getpri .... get scheduling priority
getsockname .... gets socket name
getsockopt .... get and set options on sockets
gettimer .... get the current value for a system-wide realtime timer
gettimerid .... get a unique identifier for a process interval timer
getuid .... get real user, effective user, real group, and effective group IDs
ioctl .... control device
kill .... send a signal to a process or a group of processes
link .... link to a file

listen .... listens for connections on a socket
lseek .... move read/write file pointer
memctl .... control write/execute attributes of memory
mkdir .... make a directory
mknod .... make a directory, or a special or ordinary file
msgctl .... message control operations
msgget .... get message queue
msgop .... message operations
nice .... change priority of a process
open .... open for reading or writing
pathconf .... get configurable pathname variables
pause .... suspend process until signal
pipe .... create an interprocess channel
plock .... lock process, text, or data in memory
prealloc .... preallocate contiguous file space
putmsg .... send a message on a stream
read .... read from file
readlink .... read value of a symbolic link
readv .... do multiple reads from a file
recv .... receive a message from a socket
relinquish .... voluntarily give up CPU
reltimerid .... release a process interval timer identifier
rename .... change the name of a file
resabs .... get resolution and maximum time value of process interval timer
resident .... make locked segments resident in memory
restimer .... get the resolution and maximum time values for a system-wide realtime timer
resume .... resume a suspended process
rmdir .... remove a directory
select .... synchronous I/O multiplexing
semctl .... semaphore control operations
semget .... get set of semaphores
semop .... semaphore operations
send .... sends a message from a socket
setgroups .... set group access list
setpgid .... set process group ID for job control
setpgrp .... set process group ID
setpri .... set scheduling priority
setpsr .... set/get Processor Status Register
setrt .... set realtime privileges
setrtusers .... set realtime privileged users list
setslice .... set CPU time slice size
settimer .... set the current value for a system-wide realtime timer
setuid .... set user and group IDs
shmctl .... shared memory control operations
shmget .... get shared memory segment identifier
shmop .... shared memory operations
sigaction .... examine or change signal action
signal .... specify what to do upon receipt of a signal
sigpending .... examine pending signals
sigprocmask .... examine and change blocked signals
sigset .... signal management
sigsetops .... manipulate signal sets
sleep .... suspend execution for interval
socket .... create an endpoint for communication
stat .... get file status
statfs .... get file system information
stime .... set time

stkexp .... expand the stack region of the data segment
suspend .... suspend the calling process
swtch .... switch into a process
symlink .... makes symbolic link to a file
sync .... update super block
sysfs .... get file system type information
sysm68k .... machine specific functions
times .... get process and child process times
trunc .... truncate file space
uadmin .... administrative control
ulimit .... get and set user limits
umask .... set and get file creation mask
umount .... unmount a file system
uname .... get name of current system
unlink .... remove directory entry
ustat .... get file system statistics
utime .... set file access and modification times
wait .... wait for child process to stop or terminate
waitpid .... wait for child process to stop or terminate
write .... write on a file
writev .... do multiple writes from a file

APPENDIX A (installation section)

INDEX

NAME

intro - introduction to system calls and error numbers

SYNOPSIS

 # include <sysf/errno.i>
integer*4 iretval
iretval = ftn_name (arg1, arg2,...)

DESCRIPTION

This section describes the FORTRAN interface to a subset of the C system calls library. The C system calls are described in section 2 of the *REAL/IX System Calls, Library Routines, and Files Reference Manual*.

To compile and link a program, enter a line similar to the following:

gf77 *source*.F -o *executable* **-lfs -I/usr/include/gls**

executable is the resultant executable file name. The **-lfs** option uses the FORTRAN System Calls Library (**libfs**). The **-I** option tells the preprocessor where to find the include files. The compiler invokes the preprocessor when it sees the **.F** source file extension.

Many calls use the C preprocessor directives. You must have the pound sign in column one. To execute the preprocessor, the FORTRAN source must be saved with a **.F** extension before compilation. The preprocessor is more powerful than the FORTRAN INCLUDE statement. For information concerning the preprocessor and its directives, see the **cpp(1)** manual page in the REAL/IX *Commands and Utilities Manual.*

The calls are listed in alphabetical order. Some pages describe more than one call. To find a call not otherwise found in the page headers, check the permuted index, center column, for the function name. The right-hand column provides the page header name.

The format for the FORTRAN Interface to System Calls section (2F), is similar to other REAL/IX manual pages. Each page uses the name of the call together with the section number in the page header. All entries are presented using the following format (not all of the following headers appear in every system call manual page):

- NAME - gives the primary name and a brief statement of purpose.
- SYNOPSIS - summarizes usage. Statements which begin with a pound sign (#), will be evaluated by the preprocessor. **Boldface** strings should be typed as they appear. *Italic* strings represent variable names. Square brackets [] indicate the argument is optional.
- DESCRIPTION - provides a detailed explanation of the system call and its parameters.
- EXAMPLE - shows the system call used in context. Consult the *GLS FORTRAN Language Reference Manual* for more information.
- ERROR CODES - describe possible error code return values. To use them, you must include the preprocessor statement **# include <sysf/errno.i>.**
- NOTES - provide additional helpful information.
- SEE ALSO - routes you to other sources of information.
- DIAGNOSTICS - describe return values and/or error codes.

Most of these calls have one or more error returns. A negative returned value indicates the error condition.

Functions that declare the string *character*SIZE,* can also pass that string explicitly, by defining it between quotes. For example, instead of:

```
string = 'a string'
iretval = func (string)
```

you can substitute the following code sequence:

```
iretval = func('a string')
```

The following list describes the error numbers and their names as they are defined in **<errno.i>**. Each system call description attempts to list all possible error numbers.

-1 EPERM Not owner
: Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.

-2 ENOENT No such file or directory
: This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

-3 ESRCH No such process
: No process can be found corresponding to that specified by *pid* in *kill*(2F) or *ptrace*(2F).

-4 EINTR Interrupted system call
: An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

-5 EIO I/O error
: Some physical I/O error has occurred. This error may in some cases occur on a call following the one to which it actually applies.

-6 ENXIO No such device or address
: I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

-7 E2BIG Arg list too long
: An argument list longer than 5,120 bytes is presented to a member of the *exec*(2F) family.

-8 ENOEXEC Exec format error
: A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number [see *a.out*(4)].

-9 EBADF Bad file number
: Either a file descriptor refers to no open file, or a *read*(2F) [respectively, *write*(2F)] request is made to a file which is open only for writing (respectively, reading).

-10 ECHILD No child processes
: A *wait* was executed by a process that had no existing or unwaited-for child processes.

-11 EAGAIN No more processes
: A *fork* failed because the system's process table is full or the user is not allowed to create any more processes. Or a system call failed because of insufficient memory or swap space.

-12 ENOMEM Not enough space
: During an *exec*(2F), *brk*(2F), or *sbrk*(2F), a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*(2F). If this error occurs on a resource associated with Remote File Sharing (RFS), it

indicates a memory depletion which may be temporary, dependent on system activity at the time the call was invoked.

-13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

-14 EFAULT Bad address
The system encountered a hardware fault in attempting to use an argument of a system call.

-15 ENOTBLK Block device required
A non-block file was mentioned where a block device was required, e.g., in *mount*(2F).

-16 EBUSY Device or resource busy
An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

-17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link*(2F).

-18 EXDEV Cross-device link
A link to a file on another device was attempted.

-19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

-20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path prefix or as an argument to *chdir*(2F).

-21 EISDIR Is a directory
An attempt was made to write on a directory.

-22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*(2F) or *kill*(2F); reading or writing a file for which *lseek*(2F) has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

-23 ENFILE File table overflow
The system file table is full, and temporarily no more *opens* can be accepted.

-24 EMFILE Too many open files
No process may have more than NOFILES (default 100) descriptors open at a time.

-25 ENOTTY Not a character device (or) Not a typewriter
An attempt was made to *ioctl*(2F) a file that is not a special character device.

-26 ETXTBSY Text file busy
An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed.

-27 EFBIG File too large
The size of a file exceeded the maximum file size or ULIMIT [see *ulimit*(2F)].

-28 ENOSPC No space left on device
During a *write*(2F) to an ordinary file, there is no free space left on the device. In *fcntl*(2F), the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

-29 ESPIPE Illegal seek
An *lseek*(2F) was issued to a pipe.

-30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

-31 EMLINK Too many links
An attempt to make more than the maximum number of links (1000) to a file.

-32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

-33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.

-34 ERANGE Result too large
The value of a function in the math package (3M) is not representable within machine precision.

This error number is also used for the EINPROG error code returned for asynchronous I/O operations.

-35 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue [see *msgop*(2F)].

-36 EIDRM Identifier removed
This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space [see *msgctl*(2F), *semctl*(2F), and *shmctl*(2F)].

This error number is also used for the ECANCELED error code returned for asynchronous I/O operations.

-37 through -44 are reserved numbers

-45 EDEADLK Deadlock
A deadlock situation was detected and avoided. This error pertains to file and record locking.

-46 ENOLCK No lock
In *fcntl*(2F) the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

-60 ENOSTR Not a stream (UNSUPPORTED)
A *putmsg*(2F) or *getmsg*(2F) system call was attempted on a file descriptor that is not a STREAMS device.

-62 ETIME Stream ioctl timeout (UNSUPPORTED)
: The timer set for a STREAMS *ioctl*(2F) call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl*(2F) operation is indeterminate.

-63 ENOSR No stream resources (UNSUPPORTED)
: During a STREAMS *open*(2F), either no STREAMS queues or no STREAMS head data structures were available.

-64 ENONET Machine is not on the network (UNSUPPORTED)
: This error is Remote File Sharing (RFS) specific. It occurs when users try to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network.

-65 ENOPKG No package
: This error occurs when users attempt to use a system call from a package which has not been installed.

-66 EREMOTE Resource is remote (UNSUPPORTED)
: This error is RFS specific. It occurs when users try to advertise a resource which is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.

-67 ENOLINK Virtual circuit is gone (UNSUPPORTED)
: This error is RFS specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.

-68 EADV Advertise error (UNSUPPORTED)
: This error is RFS specific. It occurs when users try to advertise a resource which has been advertised already, or try to stop the RFS while there are resources still advertised, or try to force unmount a resource when it is still advertised.

-69 ESRMNT Srmount error (UNSUPPORTED)
: This error is RFS specific. It occurs when users try to stop RFS while there are resources still mounted by remote machines.

-70 ECOMM Communication error (UNSUPPORTED)
: This error is RFS specific. It occurs when trying to send messages to remote machines but no virtual circuit can be found.

-71 EPROTO Protocol error
: Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

-74 EMULTIHOP Multihop attempted (UNSUPPORTED)
: This error is RFS specific. It occurs when users try to access remote resources which are not directly accessible.

-77 EBADMSG Bad message (UNSUPPORTED)
: During a *read*(2F), *getmsg*(2F), or *ioctl*(2F) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
  read(2F) - control information or a passed file descriptor
  getmsg(2F) - passed file descriptor
  ioctl(2F) - control or data information

-83 ELIBACC Cannot access a needed shared library
: Trying to *exec*(2F) an *a.out* that requires a shared library (to be linked in) and the shared library doesn't exist or the user doesn't have permission to use it.

-84 ELIBBAD Accessing a corrupted shared library
Trying to *exec*(2F) an *a.out* that requires a shared library (to be linked in) and *exec*(2F) could not load the shared library. The shared library is probably corrupted.

-85 ELIBSCN .lib section in *a.out* corrupted
Trying to *exec*(2F) an *a.out* that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the *a.out*. The .lib section tells *exec*(2F) what shared libraries are needed. The *a.out* is probably corrupted.

-86 ELIBMAX Attempting to link in more shared libraries than system limit
Trying to *exec*(2F) an *a.out* that requires more shared libraries (to be linked in) than is allowed on the current configuration of the system.

-87 ELIBEXEC Cannot exec a shared library directly
Trying to *exec*(2F) a shared library directly. This is not allowed.

The following error numbers and their names are supplied for network applications:

-90 EWOULDBLOCK Operation would block

-91 EINPROGRESS Operation now in progress

-92 EALREADY Operation already in progress

-93 ENOTSOCK Socket operation on non-socket

-94 EDESTADDRREQ Destination address required

-95 EMSGSIZE Message too long

-96 EPROTOTYPE Protocol wrong type for socket

-97 ENOPROTOOPT Protocol not available

-98 EPROTONOSUPPORT Protocol not supported

-100 EOPNOTSUPP Operation not supported on socket

-101 EPFNOSUPPORT Protocol family not supported

-102 EAFNOSUPPORT Proto fam doesn't support addr fam

-103 EADDRINUSE Address already in use

-104 EADDRNOTAVAIL Can't assign requested address

-105 ENETDOWN Network is down

-106 ENETUNREACH Network is unreachable

-107 ENETRESET Network dropped connection on reset

-108 ECONNABORTED Software caused connection abort

-109 ECONNRESET Connection reset by peer

-110 ENOBUFS No buffer space available

-114 ETOOMANYREFS Too many references: can't splice

-115 ETIMEDOUT Connection timed out

-116 ECONNREFUSED Connection refused

DEFINITIONS

Process ID Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30,000.

Parent Process ID A new process is created by a currently active process [see *fork*(2F)]. The parent process ID of a process is the process ID of its creator.

Process Group ID Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes [see *kill*(2F)].

tty Group ID Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group [see *exit*(2F) and *signal*(2F)].

Real User ID and Real Group ID Each user allowed on the system is identified by a positive integer (0 to 65535) called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set [see *exec*(2F)].

Superuser A process is recognized as a *superuser* process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Special Processes The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

proc0 is the scheduler. *proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Descriptor A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to (NOFILES - 1). A process may have no more than NOFILES file descriptors open simultaneously. A file descriptor is returned by system calls such as *open*(2F), or *pipe*(2F). The file descriptor is used as an argument by calls such as *read*(2F), *write*(2F), *ioctl*(2F), and *close*(2F).

File Name Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory. These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell [see *sh*(1)]. Although permitted, the use of unprintable characters in file names should be avoided.

Path Name and Path Prefix A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory. A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system.

File Access Permissions Read, write, and execute/search permissions on a file are granted to a process only if one or more of the following statements are true:

> The effective user ID of the process is superuser.
>
> The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion ('0700'O) of the file mode is set.
>
> The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion ('0070'O) of the file mode is set.
>
> The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion ('0007'O) of the file mode is set.

Message Queue Identifier A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2F) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
record      /ipc_perm/ msg_perm
integer*4   msg_first
integer*4   msg_last
integer*2   msg_cbytes
integer*2   msg_qnum
integer*2   msg_qbytes
integer*2   msg_lspid
integer*2   msg_lrpid
integer*4   msg_stime
integer*4   msg_rtime
integer*4   msg_ctime
```

msg_perm is an ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
integer*2   uid     !owner user id
integer*2   gid     !owner group id
integer*2   cuid    !creator user id
integer*2   cgid    !creator group id
integer*2   mode    !access permission
integer*2   seq     !slot usage sequence
integer*4   key     !key
```

msg_first is a pointer to the first message on the queue.

msg_last is a pointer to the last message on the queue.

msg_cbytes is the current number of bytes on the queue.

msg_qnum is the number of messages currently on the queue.

msg_qbytes is the maximum number of bytes allowed on the queue.

msg_lspid is the process id of the last process that performed a *msgsnd* operation.

msg_lrpid is the process id of the last process that performed a *msgrcv* operation.

msg_stime is the time of the last *msgsnd* operation.

msg_rtime is the time of the last *msgrcv* operation.

msg_ctime is the time of the last *msgctl*(2F) operation that changed a member of the above structure.

Message Operation Permissions In the *msgop*(2F) and *msgctl*(2F) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed, interpreted as follows:

'00400'O	Read by user
'00200'O	Write by user
'00040'O	Read by group
'00020'O	Write by group
'00004'O	Read by others
'00002'O	Write by others

Read and write permissions on a msqid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion ('0600'O) of **msg_perm.mode** is set.

The effective group ID of the process matches **msg_perm.cgid** or **msg_perm.gid** and the appropriate bit of the "group" portion ('060'O) of **msg_perm.mode** is set.

The appropriate bit of the "other" portion ('06'O) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier A semaphore identifier (semid) is a unique positive integer created by a *semget*(2F) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
record       /ipc_perm/ sem_perm     !operation permission struct
integer*4    sem_base                !ptr to first semaphore in set
integer*2    sem_nsems               !number of sems in set
integer*4    sem_otime               !last operation time
integer*4    sem_ctime               !last change time
                                     !Times measured in secs since
                                     !00:00:00 GMT, Jan. 1, 1970
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
integer*2    uid      !user id
integer*2    gid      !group id
integer*2    cuid     !creator user id
integer*2    cgid     !creator group id
integer*2    mode     !r/a permission
integer*2    seq      !slot usage sequence number
integer*4    key      !key
```

sem_nsems is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1.

sem_otime is the time of the last *semop*(2F) operation.

sem_ctime is the time of the last *semctl*(2F) operation that changed a member of the above structure.

A semaphore is a data structure called *sem* that contains the following members:

```
integer*2    semval     !semaphore value
integer*2    sempid     !pid of last operation
integer*2    semncnt    !# awaiting semval > cval
integer*2    semzcnt    !# awaiting semval = 0
```

semval is a non-negative integer which is the actual value of the semphore.

sempid is equal to the process ID of the last process that performed a semaphore operation on this semaphore.

semncnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

semzcnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions In the *semop*(2F) and *semctl*(2F) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

'00400'O	Read by user
'00200'O	Alter by user
'00040'O	Read by group
'00020'O	Alter by group
'00004'O	Read by others
'00002'O	Alter by others

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion ('0600'O) of **sem_perm.mode** is set.

The effective group ID of the process matches **sem_perm.cgid** or **sem_perm.gid** and the appropriate bit of the "group" portion ('060'O) of **sem_perm.mode** is set.

The appropriate bit of the "other" portion ('006'O) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2F) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as *shmid_ds* and contains the following members:

```
record      /ipc_perm/ shm_perm    !operation permission struct
integer*4   shm_segsz              !size of segment
integer*4   shm_reg                !ptr to region structure
integer*4   shm_paddr              !physical address
integer*2   shm_lpid               !pid of last operation
integer*2   shm_cpid               !creator pid
integer*2   shm_nattch             !number of current attaches
integer*2   shm_cnattch            !used only for shminfo
integer*4   shm_atime              !last attach time
integer*4   shm_dtime              !last detach time
integer*4   shm_ctime              !last change time
                                   !Times measured in secs since
                                   !00:00:00 GMT, Jan. 1, 1970
```

shm_perm is an ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
integer*2   cuid    !creator user id
integer*2   cgid    !creator group id
integer*2   uid     !user id
integer*2   gid     !group id
integer*2   mode    !r/w permission
integer*2   seq     !slot usage sequence #
integer*4   key     !key
```

shm_segsz specifies the size of the shared memory segment in bytes.

shm_cpid is the process id of the process that created the shared memory identifier.

shm_lpid is the process id of the last process that performed a *shmop*(2F) operation.

shm_nattch is the number of processes that currently have this segment attached.

shm_atime is the time of the last *shmat*(2F) operation.

shm_dtime is the time of the last *shmdt*(2F) operation.

shm_ctime is the time of the last *shmctl*(2F) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions In the *shmop*(2F) and *shmctl*(2F) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

'00400'O	Read by user
'00200'O	Write by user
'00040'O	Read by group
'00020'O	Write by group
'00004'O	Read by others
'00002'O	Write by others

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

The effective user ID of the process is superuser.

The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion ('0600'O) of **shm_perm.mode** is set.

The effective group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the "group" portion ('060'O) of **shm_perm.mode** is set.

The appropriate bit of the "other" portion ('06'O) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

STREAMS A set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream Head In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, such as a *multiplexer* or log *driver* [see *log*(7)], which is not associated with a hardware device.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream In a *stream*, the direction from *stream head* to *driver*.

Upstream In a *stream*, the direction from *driver* to *stream head*.

Message In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor A multiplexer is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process. STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

SEE ALSO

intro(3).

NAME

absinterval, incinterval - set the expiration time of a process interval timer

SYNOPSIS

```
#    include <sysf/time.i>
     integer*4 absinterval,timerid
     record /itimerstruc/ value, ovalue
     iretval = absinterval (timerid, value, ovalue)

     integer*4 incinterval, timerid
     record /itimerstruc/ value, ovalue
     iretval = incinterval (timerid, value, ovalue)
```

DESCRIPTION

absinterval sets the expiration time of the process interval timer, described by *timerid*, (see *gettimerid*(2F)) to an absolute time value.

incinterval sets the expiration time of the process interval timer, described by *timerid*, (see *gettimerid*(2F)) to a time value relative to the current timer value.

If the time value in the itimerstruc structure element it_value is pointed to by a value other than zero, then it_value indicates the time to the next expiration. In the case of an *incinterval* system call, it_value represents an offset from the current timer value. In the case of an *absinterval* system call, it_value represents an absolute time. If the specified expiration time value has passed, then the system call succeeds and the timer event is delivered.

If the time value in the itimerstruc element it_value is pointed to by value zero, then the process interval timer associated with *timerid* is removed from the timer expiration queue and no event is sent.

If the time value in the itimerstruc structure element it_interval is pointed to by a value other than zero, then it_interval indicates a time value to be used in reloading it_value when the process interval timer associated with *timerid* expires. For both *absinterval* and *incinterval*, it_interval represents an offset from the current expiration time.

If the time value in the itimerstruc structure element it_interval is pointed to by value zero, then the process interval timer associated with *timerid* is stopped after its next expiration (unless it_value is zero).

On return, the value in the it_value structure of *ovalue* is set to the amount of time left before the timer associated with *timerid* would have expired. The value in the it_interval structure of *ovalue* is set to the time value being used on timer expiration reloads.

EXAMPLE

```
        program absinter
#       include <sysf/time.i>
#       include <sysf/lock.i>
#       include <sysf/evt.i>

        integer*4 absinterval, incinterval, timerid, iretval
        record /itimerstruc/ value, ovalue

        integer*4 evget, eid, gettimerid, reltimerid
        integer*4 sec, nsec, isec, insec

c   Use setrt to set realtime mode
c   Use setpri to set new priority if necessary
c   Use plock to lock process in memory
c   Use resident to keep locked segements in memory
c   Use evget to get unique event id for this process

        eid = evget (EVT_QUEUE, 10, 0, %val(0))

c   Use gettimerid to get unique timer id for this process using event id

        timerid = gettimerid (TIMEOFDAY, MODCOMP_EVENTS, eid)
        if (timerid .lt. 0) write (*,*) 'gettimerid error:', timerid

c   Set up time for incinterval (offset if incinterval, otherwise absolute)
c   .5 sec

        sec = 0
        nsec = 500000000

c   Set up re-initialize interval time offset (0 = oneshot)

        isec = 0
        insec = 0

c   Set up timer structure and make the call

        value.it_value.tv_sec = sec
        value.it_value.tv_nsec = nsec
        value.it_interval.tv_sec = isec
        value.it_interval.tv_nsec = insec

        iretval = incinterval (timerid, value, ovalue)
        if (iretval .lt. 0) write (*,*) 'incinterval error:', iretval

c   Release timer

        iretval = reltimerid (timerid)
        if (iretval .lt. 0) write (*,*) 'reltimerid error:', iretval

        end
```

ERROR CODES

If *absinterval* or *incinterval* is not successful, a negative value is returned with one of the following error codes.

[EFAULT] *value* or *ovalue* points outside the allocated address space of the process.

[EINVAL] The *timerid* argument does not correspond to an ID returned by *gettimerid*(2F) or it has previously been released with a *reltimerid*(2F) call.

SEE ALSO

getinterval(2F), gettimerid(2F), reltimerid(2F), resabs(2F), resinc(2F), itimerstruc(4), timestruc(4).

NAME

acancel - cancel one or more asynchronous I/O requests

SYNOPSIS

```
# include <sysf/evt.i>
# include <sysf/aio.i>
# include <sysf/types.i>

  integer*4 acancel, fildes
  record /aiocb_t/ aiocb
  iretval = acancel (fildes, aiocb)
```

DESCRIPTION

acancel cancels one or more asynchronous I/O requests currently outstanding against the file descriptor *fildes*. The *aiocb* argument points to the *aiocb*(4) structure for a specific request that is to be cancelled. If the *aiocb* argument is NULL (i.e. %val(0)), then all outstanding asynchronous I/O requests against *fildes* are canceled.

EXAMPLE

```
      program acancel
#     include <sysf/types.i>
#     include <sysf/evt.i>
#     include <sysf/aio.i>
#     include <sysf/fcntl.i>
      integer*4 acancel, fildes
      record /aiocb_t/ aiocb (64)
      integer*4 iretval

c     See aread(2F) or awrite(2F)
c     Assume already have an open file with async I/O pending

c     Cancel I/O pending associated with aiocb (30)

      iretval = acancel (fildes, aiocb(30))

c     Cancel all I/O pending

      iretval = acancel (fildes, %val (0))
      end
```

ERROR CODES

[EBADF] *fildes* is not a valid file descriptor.

SEE ALSO

arwfree(2F), aread(2F), arinit(2F), awinit(2F), awrite(2F), creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), aiocb(4).

aio(D2X), comp_aio(D3X), comp_cancel_aio(D3X), areq(D4X) in the *Kernel Programming Reference Manual.*

DIAGNOSTICS

acancel returns one of the following:

0 The requested operation(s) were cancelled.

1 At least one of the requested operations are in progress and cannot be cancelled.

2 The operation is not queued and not in progress.

Otherwise, a negative value indicating the error is returned.

NAME

accept - accepts a connection on a socket

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*4 accept,s,addrlen
record/sockaddr/addr
ns = accept(s, addr, addrlen)

DESCRIPTION

The argument *s* is a socket which has been created with *socket*(2F), bound to an address with *bind*(2F), and is listening for connections after a *listen*(2F). *accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as nonblocking, *accept* blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket, *s*, remains open.

The argument *addr* is a result parameter which is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. Consult the manual entries in Sections 4 and 7 for detailed information. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return, it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

RETURN VALUE

The call returns -1 on error. If it succeeds, it returns a non-negative integer which is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The addr parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2F), connect(2F), listen(2F), socket(2F)

NAME

access - determine accessibility of a file

SYNOPSIS

integer*4 access, amode
character*SIZE path
iretval = access (path, amode)

DESCRIPTION

access checks the named file for accessibility according to the bit pattern contained in *amode. access* uses the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. *SIZE* is the maximum number of characters ever expected to be assigned to *path.* The maximum is currently 128 characters. *path* points to a path name naming a file. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*Path* points outside the allocated address space for the process.
[EINTR]	A signal was caught during the *access* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

EXAMPLE

```
        program access
#       include <sysf/errno.i>

        integer*4 access, amode
        character*40 path

        integer*4 i, j, iretval

c  check if source file is readable and writable

        path = '../example/access.F'
        amode = 4 .or. 2
        iretval = access (path, amode)
        if (iretval .ne. 0) write (*,*) 'access error:', iretval

c  check if executable file is readable and writable

        iretval = access ('../example/access', amode)
        if (iretval .eq. ETXTBSY) write (*,*) ' no access... executing'
        if (iretval .ne. ETXTBSY .and. iretval .lt. 0) then
          write (*,*) 'access error:', iretval
        endif
        end
```

SEE ALSO

chmod(2F), stat(2F).

DIAGNOSTICS

If the requested access is permitted, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

acct - enable or disable process accounting

SYNOPSIS

integer*4 acct
character*SIZE path

iretval = acct (path)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *acct* is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal [see *exit*(2F) and *signal*(2F)]. The effective user ID of the calling process must be super-user to use this call.

path points to a pathname naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is %val(0) or of null size and no errors occur during the system call.

acct will fail if one or more of the following are true:

[EPERM]	The effective user of the calling process is not super-user.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file path name do not exist.
[EACCES]	The file named by *path* is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* points to an illegal address.

EXAMPLE

```
      program acct
      integer*4 acct
      character*40 path
      integer*4 iretval
      path = './acct.fil'

c  Enable accounting

      iretval = acct (path)
      if (iretval .lt. 0) write (*,*) 'acct enable error:', iretval

c  Disable accounting

      iretval = acct ('')
      if (iretval .lt. 0) write (*,*) 'acct disable error:', iretval

      end
```

SEE ALSO

exit(2F), signal(2F), acct(4).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

alarm - set a process alarm clock

SYNOPSIS

integer*4 alarm, sec
iretval = alarm (sec)

DESCRIPTION

alarm instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed [see *signal*(2F)].

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

EXAMPLE

```
      program alarm
#     include <sysf/errno.i>
#     include <sysf/signal.i>

      integer*4 alarm, sec

      integer*4 prev_time
      integer*4 pause, iretval

c  Set alarm for 5 seconds, pause until alarm expires.
c  Expect the process to be killed, since the alarm signal
c  was not caught [see signal(2F)].

      sec = 5
      prev_time = alarm (sec)
      if (prev_time .lt. 0) write (*,*) 'alarm error:', prev_time
      iretval = pause ()
      write (*,*) 'Error, should not get here'
      end
```

SEE ALSO

pause(2F), signal(2F), sigpause(2F), sigset(2F).

DIAGNOSTICS

alarm returns the amount of time previously remaining in the alarm clock of the calling process.

NAME

aread - read from file in an asynchronous manner

SYNOPSIS

```
#   include <sysf/types.i>
#   include <sysf/evt.i>
#   include <sysf/aio.i>

    integer*4 aread, fildes, nbyte
    integer*1 buf(SIZE)
    record /aiocb_t/ aiocb
    iretval = aread (fildes, buf, nbyte, aiocb)
```

DESCRIPTION

aread attempts to read *nbyte* bytes from the file associated with *fildes,* into the buffer pointed to by *buf*. *fildes* is a file descriptor obtained from a *creat*(2F), *open*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call. The *aiocb* argument points to an *aiocb*(4) control block. *SIZE* is the number of bytes in the buffer.

aread returns when the read request has been queued to the file or device (even when the data cannot be delivered until later). If an error condition is encountered, *aread* returns without having queued the request. If *aiocb* is NULL (i.e. %val (0)), then no status is returned and the read operation begins from the current file pointer; no event notification is given upon completion of the read.

The *whence* and *offset* fields of the *aiocb* request an implicit *lseek*(2F) operation. The seek implied by these fields plus the *nbyte* parameter updates the file pointer when the function call returns without error.

INTERACTION WITH OTHER SERVICES

close(2F) Cancelable operations and queued operations for that file descriptor are deleted. Noncancelable operations are waited on.

exit(2F) and *exec*(2F)
Cancelable operations and queued operations for all file descriptors are deleted. Noncancelable operations are waited on.

open(2F) with O_TRUNC flag set and *creat*(2F)
Outstanding asynchronous I/O operations are not canceled.

fork(2F) No asynchronous I/O is inherited.

fcntl(2F) Used to set or check the bypass-buffer-cache flag before issuing the *aread* request. In this release, asynchronous I/O operations require this flag to be set.

arinit(2F) Can be called during process initialization, before the first *aread* call. This will improve the speed of all *aread* operations, especially the first one. The values of the *aread* arguments must match exactly with the values given to *arinit*.

EXAMPLE

```
        program aread
#       include <sysf/types.i>
#       include <sysf/evt.i>
#       include <sysf/aio.i>
#       include <sysf/fcntl.i>
        integer*4 arinit, aread, fildes, nbyte, iretval, i
        integer*4 error, buf (64), open, trunc
        record /aiocb_t/ aiocb (64)

c   Open a file

        fildes = open ('../example/aio.tst', O_RDWR)
        if (fildes .lt. 0) write (*,*) 'open error:', fildes

c   Initialize for asynchronous reading

        error = 0
        do 100 i = 1, 64
        aiocb (i).offset = (i - 1) * 4
        aiocb (i).whence = 0
        aiocb (i).rt_errno = -1
        aiocb (i).nobytes = 0
        aiocb (i).aioflag = 0
        aiocb (i).eid = 0
        nbyte = 4
        buf (i) = 64 - i
        iretval = arinit (fildes, buf (i), nbyte, aiocb (i))
        if (iretval .lt. 0 .and. error .eq. 0) error = iretval
100     continue
        if (error .ne. 0) write (*,*) 'arinit error:', error

c   Perform some asynchronous reading

        do 200 i = 1, 64
        iretval = aread (fildes, buf (i), nbyte, aiocb (i))
        if (iretval .lt. 0) write (*,*) 'aread error:', iretval
200     continue

c   Wait until i/o complete or error

        do 300 i = 1, 64
310     continue
        if (aiocb (i).rt_errno .lt. 0) goto 310
300     continue

c   Print buffers

        write (*,9000) (buf (i), i = 1, 64)
9000    format (' ', 8 (i8, x) )
        end
```

NOTES

When an *aread* operation is not emulated by a synchronous operation, *aread* does not attempt to enforce mandatory file/record locking (see *chmod*(2F)).

SEE ALSO

acancel(2F), arwfree(2F), arinit(2F), awrite(2F), creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), aiocb(4).

aio(D2X), comp_aio(D3X), comp_cancel_aio(D3X), areq(D4X) in the *Kernel Programming Reference Manual*.

DIAGNOSTICS

aread will fail if one or more of the following are true:

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated, system wide, for asynchronous I/O operations would be exceeded.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated for asynchronous I/O operations per process would be exceeded.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of processes using the asynchronous I/O facility, system wide, would be exceeded.

[EAGAIN] Insufficient system memory to lock down specified buffer areas, probably a transient condition.

[EAGAIN] Insufficient system memory to set up mapping.

[EBADF] *fildes* is not a valid file descriptor open for reading.

[EBUSY] Cannot read from a file when the bypass-buffer-cache flag is being changed or the file is being truncated.

[EFAULT] Data cannot be read into the specified area.

[EFAULT] *buf* points outside the allocated address space.

[EFAULT] *aiocb* points outside the allocated address space.

[EINVAL] The *whence* element of the *aiocb* is not a proper value, or the resulting file pointer would be invalid.

[EINVAL] The *flags* element of the *aiocb* is not set to 0 or EVT_POST.

[EINVAL] The *flags* element of the *aiocb* is set to EVT_POST and the *eid* element of the *aiocb* structure is not a valid event identifier.

[EINVAL] *nbyte* is less than 0.

[EPERM] Process has neither realtime nor superuser privileges.

Upon successful completion, the value zero is returned. Errors that are detected before the request is passed to the driver will result in a negative returned value in *iretval* indicating the error. Errors that occur after the request is passed to the driver are reported in the **rt_errno** member of the *aiocb* structure. This field is set to EINPROG before the operation is queued and to zero when the queued operation completes without error.

NAME

arinit - initialize structures before requesting an asynchronous read

SYNOPSIS

```
#    include <sysf/evt.i>
#    include <sysf/aio.i>
#    include <sysf/types.i>

     integer*4 arinit, fildes, nbyte
     integer*1 buf (SIZE)
     record /aiocb_t/aiocb
     iretval = arinit (fildes, buf, nbyte, aiocb)
```

DESCRIPTION

arinit can be called during process initialization to initialize structures used by asynchronous read operations. Using *arinit* significantly improves the speed of asynchronous read operations, especially the first one executed, although *aread*(2F) will execute correctly if *arinit* is not called first. It also guarantees that the necessary resources have been allocated to perform an *aread* with parameters that match those specified for *arinit*.

The arguments to *arinit* are:

fildes File descriptor obtained from a *creat*(2F), *open*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call.

buf Points to the buffer where data for subsequent *aread* calls is to be found. *SIZE* is the number of bytes in the buffer.

nbyte The byte count used in *aread* calls using the resources allocated and initialized by *arinit*.

aiocb Points to an *aiocb*(4) control block.

An *arinit* call gives a performance benefit to subsequent *aread* calls whose parameters meet the following conditions:

fildes As per *arinit*.

buf As per *arinit*.

nbytes As per *arinit*.

aiocb At the same address as the *aiocb* used in the *arinit*.

aiocb fields:

offset No restriction, to allow for positioning within the file.

whence No restriction, to allow for positioning within the file.

rt_errno Unused (return parameter).

nobytes Unused (return parameter).

rt_aio_pri Unused (reserved).

aioflag As per *arinit*.

eid As per *arinit* if *aioflag* is EVT_POST, otherwise ignored.

Note that an *arinit* allocates resources behind the scenes and these resources remain allocated until explicitly freed via *arwfree* or until asynchronous I/O operations on the file are curtailed by a *close*(2F), *exit*(2F), or *exec*(2F) system call.

arinit returns when the structures have been initialized.

If *aiocb* is NULL (i.e. %val(0)), then the *arinit* call initializes for *aread* calls that do not use an *aiocb* structure to return status information.

Errors that are reported via the **rt_errno** field are as follows:

[EAGAIN] Shortage of system resources prevented operation from being carried out.

[ENODEV] Asynchronous I/O is not supported for this *fildes* and emulation has not been requested.

[ENODEV] Asynchronous I/O is supported for this *fildes* but not for the transfer specified by the parameters to the *aread*. Reasons include:

The file pointer is not aligned on suitable boundary.
The number of bytes to transfer is too small or too large.
The number of bytes to transfer is not a multiple of the block size.
The buffer is not suitably aligned.

See the F_GETAIOREQ option in *fcntl*(2F) for further details. Note that ENODEV is returned only if synchronous emulation is not allowed. See the F_SETAIOEMUL option in *fcntl*(2F). If emulation is allowed the operation will be attempted in synchronous mode.

[EIO] Some physical I/O error occurred.

[ENXIO] I/O operation could not be started (e.g., tape not on-line).

If the asynchronous I/O operation is emulated by a synchronous operation, then, should the synchronous operation fail, the error code is reported in the **rt_errno**.

EXAMPLE

See aread(2F) for an example.

SEE ALSO

acancel(2F), arwfree(2F), aread(2F), awinit(2F), awrite(2F), creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), aiocb(4).

aio(D2X), comp_aio(D3X), comp_cancel_aio(D3X), areq(D4X) in the *Kernel Programming Reference Manual*.

DIAGNOSTICS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated, system wide, for asynchronous I/O operations would be exceeded.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated for asynchronous I/O operations per process would be exceeded.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of processes using the asynchronous I/O facility, system wide, would be exceeded.

[EAGAIN] Insufficient system memory to lock down specified buffer areas, probably a transient condition.

[EAGAIN] Insufficient system memory to set up mapping.

[EBADF] *fildes* is not a valid file descriptor open for reading.

[EFAULT] *buf* points outside the allocated address space.

[EFAULT] *aiocb* points outside the allocated address space.

[EINVAL] The *flags* element of the *aiocb* is not set to 0 or EVT_POST.

[EINVAL] The *flags* element of the *aiocb* is set to EVT_POST, and the *eid* element of the *aiocb* structure is not a valid event identifier.

[EINVAL] *nbyte* is less than 0.

[EINVAL] *nbyte* is equal to zero. Zero is a legal value for the *nbyte* parameter to *aread* but not for the *nbyte* parameter to *arinit*.

[ENODEV] Asynchronous I/O is not supported for this *fildes*.

[EPERM] Process has neither realtime nor superuser privileges.

NAME

arwfree - free internal resources for asynchronous I/O from the process

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/evt.i>
#    include <sysf/aio.i>

     integer*4 arwfree, fildes
     record /aiocb_t/ aiocb
     iretval = arwfree (fildes, aiocb)
```

DESCRIPTION

arwfree frees internal resources associated with the *aiocb*(4) structure from the process. If *arwfree* is not issued, the internal resources remain mapped into the process until the process exits or executes. The *aiocb* argument points to the *aiocb*(4) structure for a specific request that is to be freed. If the *aiocb* argument is NULL (i.e. %val(0)), then all resources for *aiocbs* associated with *fildes* are freed.

EXAMPLE

```
     program arwfree
#    include <sysf/types.i>
#    include <sysf/evt.i>
#    include <sysf/aio.i>
#    include <sysf/fcntl.i>
     integer*4 arwfree, fildes
     record /aiocb_t/ aiocb (64)
     integer*4 iretval

c  See aread(2F) or awrite(2F)
c  Assume already have an open file with async I/O complete

c  Free aiocb (30)

     iretval = arwfree (fildes, aiocb(30))

c  Free all aiocb's

     iretval = arwfree (fildes, %val (0))
     end
```

ERROR CODES

[EBADF] *fildes* is not a valid file descriptor.

SEE ALSO

acancel(2F), aread(2F), arinit(2F), awinit(2F), awrite(2F), creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), aiocb(4).

aio(D2X), comp_aio(D3X), comp_cancel_aio(D3X), areq(D4X) in the *Kernel Programming Reference Manual.*

DIAGNOSTICS

arwfree returns one of the following:

0 The requested resources(s) were freed.

1 An operation that uses the resources is in progress; no resources were freed.

2 No internal resources are associated with the specified *fildes* and *aiocb* parameters.

NAME

awinit - initialize structures before requesting an asynchronous write

SYNOPSIS

```
#    include <sysf/evt.i>
#    include <sysf/aio.i>
#    include <sysf/types.i>

     integer*4 awinit, fildes, nbyte
     integer*1 buf (SIZE)
     record /aiocb_t/ aiocb
     iretval = awinit (fildes, buf, nbyte, aiocb)
```

DESCRIPTION

awinit can be called during process initialization to initialize structures used by asynchronous write operations. Using *awinit* significantly improves the speed of asynchronous write operations, especially the first one executed, although *awrite*(2F) will execute correctly if *awinit* is not called first. It also guarantees that the necessary resources have been allocated to perform an *awrite* with parameters that match those specified for *awinit*.

The arguments to *awinit* are:

fildes A file descriptor obtained from a *creat*(2F), *open*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call.

buf Points to the buffer area where data for subsequent *awrite* operations is to be found. *SIZE* is the number of bytes in the buffer.

nbyte The byte count used in *awrite* operations using the resources allocated and initialized by *awinit*.

aiocb Points to an *aiocb*(4) control block.

An *awinit* call gives a performance benefit to subsequent *awrite* calls whose parameters meet the following conditions:

fildes As per *awinit*.

buf As per *awinit*.

nbytes As per *awinit*.

aiocb At the same address as the *aiocb* used in the *awinit*.

aiocb fields:

offset No restriction, to allow for positioning within the file.

whence No restriction, to allow for positioning within the file.

rt_errno Unused (return parameter).

nobytes Unused (return parameter).

rt_aio_pri Unused (reserved).

aioflag As per *awinit*.

eid As per *awinit* if *aioflag* is EVT_POST, otherwise ignored.

Note that an *awinit* allocates resources behind the scenes and these resources remain until explicitly freed via *arwfree* or until asynchronous I/O operations on the file are curtailed by a *close*(2F), *exit*(2F), or *exec*(2F) system call.

awinit returns when the structures have been initialized.

If *aiocb* is NULL (i.e. %val (0)), then the *awinit* call initializes for *awrite* calls that do not use an *aiocb*(4) structure to return status information.

EXAMPLE

See awrite(2F) for an example.

SEE ALSO

acancel(2F), arwfree(2F), aread(2F), arinit(2F), awrite(2F), creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), aiocb(4).

aio(D2X), comp_aio(D3X), comp_cancel_aio(D3X), areq(D4X) in the *Kernel Programming Reference Manual*.

DIAGNOSTICS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

[EAGAIN]	Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated, system wide, for asynchronous I/O operations would be exceeded.
[EAGAIN]	Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated for asynchronous I/O operations per process would be exceeded.
[EAGAIN]	Internal control blocks could not be allocated because a system-imposed limit on the maximum number of processes using the asynchronous I/O facility, system wide, would be exceeded.
[EAGAIN]	Insufficient system memory to lock down specified buffer areas, probably a transient condition.
[EAGAIN]	Insufficient system memory to set up mapping.
[EBADF]	*fildes* is not a valid file descriptor open for writing.
[EBUSY]	Cannot do an asynchronous write operation to a text file.
[EFAULT]	*buf* points outside the allocated address space.
[EFAULT]	*aiocb* points outside the allocated address space.
[EINVAL]	The *flags* element of the *aiocb* is not set to 0 or EVT_POST.
[EINVAL]	The *flags* element of the *aiocb* is set to EVT_POST, and the *eid* element of the *aiocb* structure is not a valid event identifier.
[EINVAL]	*nbyte* is less than 0.
[EINVAL]	*nbyte* is equal to zero. Zero is a legal value for the *nbyte* parameter to *awrite* but not for the *nbyte* parameter to *awinit*.
[ENODEV]	Asynchronous I/O is not supported for this *fildes*.
[EPERM]	Process has neither realtime nor superuser privileges.

NAME

awrite - write to file in an asynchronous manner

SYNOPSIS

```
#   include <sysf/types.i>
#   include <sysf/evt.i>
#   include <sysf/aio.i>

    integer*4 awrite, fildes, nbyte
    integer*1 buf(SIZE)
    record /aiocb_t/ aiocb
    iretval = awrite (fildes, buf, nbyte, aiocb)
```

DESCRIPTION

awrite attempts to write *nbyte* bytes to the file associated with *fildes,* from the buffer pointed to by *buf,* whose size is *SIZE* bytes. *fildes* is a file descriptor obtained from a *creat*(2F), *open*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call. The *aiocb* argument points to an *aiocb*(4) control block.

awrite returns when the write request has been queued to the file or device (even when the data cannot be delivered until later). If an error condition is encountered, *awrite* returns without having queued the request. If *aiocb* is NULL (i.e. %val(0)), then no status is returned and the write operation begins from the current file pointer; no event notification is given upon completion of the write.

The *whence* and *offset* fields of the *aiocb* request an implicit *lseek*(2F) operation. The seek implied by these fields plus the *nbyte* parameter updates the file pointer when the function call returns without error.

INTERACTION WITH OTHER SERVICES

close(2F) Cancelable operations and queued operations for that file descriptor are deleted. Noncancelable operations are waited on.

exit(2F) and *exec*(2F)
Cancelable operations and queued operations for all file descriptors are deleted. Noncancelable operations are waited on.

open(2F) with O_TRUNC flag set and *creat*(2F)
Outstanding asynchronous I/O operations are not canceled.

fork(2F) No asynchronous I/O is inherited.

fcntl(2F) Used to set or check the bypass-buffer-cache flag and the asynchronous-emulation flag before issuing the *awrite* request, as well as to obtain information about specifications for an asynchronous I/O operation (such as buffer alignment and maximum transfer size).

awinit(2F) Can be called during process initialization, before the first *awrite* call. This will improve the speed of all *awrite* operations, especially the first one.

EXAMPLE

```
      program awrite
#     include <sysf/types.i>
#     include <sysf/evt.i>
#     include <sysf/aio.i>
#     include <sysf/fcntl.i>
      integer*4 awinit, awrite, fildes, nbyte, iretval, i
      integer*4 error, buf (64), open, trunc
      record /aiocb_t/ aiocb (64)

c  Open a file and size it to 256 bytes

      fildes = open ('../example/aio.tst', O_RDWR .or.
&       O_CREAT .or. O_TRUNC, '777'o)
      if (fildes .lt. 0) write (*,*) 'open error:', fildes
      iretval = trunc (fildes, 256, 0)
      if (iretval .lt. 0) write (*,*) 'trunc error:', iretval

c  Initialize for asynchronous writing

      error = 0
      do 100 i = 1, 64
      aiocb (i).offset = (i - 1) * 4
      aiocb (i).whence = 0
      aiocb (i).rt_errno = -1
      aiocb (i).nobytes = 0
      aiocb (i).aioflag = 0
      aiocb (i).eid = 0
      nbyte = 4
      buf (i) = 64 - i
      iretval = awinit (fildes, buf (i), nbyte, aiocb (i))
      if (iretval .lt. 0 .and. error .eq. 0) error = iretval
100   continue
      if (error .ne. 0) write (*,*) 'awinit error:', error

c  Perform some asynchronous writing

      do 200 i = 1, 64
      iretval = awrite (fildes, buf (i), nbyte, aiocb (i))
      if (iretval .lt. 0) write (*,*) 'awrite error:', iretval
200   continue

c  Wait until i/o complete or error

      do 300 i = 1, 64
310   continue
      if (aiocb (i).rt_errno .lt. 0) goto 310
300   continue
      end
```

NOTES

When an *awrite* operation is not emulated by a synchronous operation, *awrite* does not attempt to enforce mandatory file/record locking (see *chmod*(2F)).

SEE ALSO

acancel(2F), arwfree(2F), arinit(2F), awrite(2F), creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), aiocb(4).

aio(D2X), comp_aio(D3X), comp_cancel_aio(D3X), areq(D4X) in the *Kernel Programming Reference Manual*.

DIAGNOSTICS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated, system wide, for asynchronous I/O operations would be exceeded.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of control blocks allocated for asynchronous I/O operations per process would be exceeded.

[EAGAIN] Internal control blocks could not be allocated because a system-imposed limit on the maximum number of processes using the asynchronous I/O facility, system wide, would be exceeded.

[EAGAIN] Insufficient system memory to lock down specified buffer areas, probably a transient condition.

[EAGAIN] Insufficient system memory to set up mapping.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EBUSY] Cannot write to a file when it is being truncated or when the bypass-buffer-cache flag is being changed.

[EFAULT] *buf* points outside the allocated address space.

[EFAULT] *aiocb* points outside the allocated address space.

[EINVAL] The *whence* element of the *aiocb* is not a proper value, or the resulting file pointer would be invalid.

[EINVAL] The *flags* element of the *aiocb* is not set to 0 or EVT_POST.

[EINVAL] The *flags* element of the *aiocb* is set to EVT_POST and the *eid* element of the *aiocb* structure is not a valid event identifier.

[EINVAL] *nbyte* is less than 0.

[EPERM] Process has neither realtime nor superuser privileges.

Upon successful completion, the value zero is returned. Errors that are detected before the request is passed to the driver will result in a negative returned value indicating the error. Errors that occur after the request is passed to the driver are reported in the **rt_errno** member of the *aiocb* structure. This field is set to EINPROG before the operation is queued and to zero when the queued operation completes without error.

Errors that are reported via the **rt_errno** field are as follows:

[EAGAIN] Shortage of system resources prevented operation from being carried out.

[ENODEV] Asynchronous I/O is not supported for this *fildes* and emulation has not been requested.

[ENODEV] Asynchronous I/O is supported for this *fildes* but not for the transfer specified by the parameters to the *aread*. Reasons include:

The file pointer is not aligned on suitable boundary.
The number of bytes to transfer is too small or too large.
The number of bytes to transfer is not a multiple of the block size.
The buffer is not suitably aligned.

See the F_GETAIOREQ option in *fcntl*(2F) for further details. Note that ENODEV is returned only if synchronous emulation is not allowed. See the F_SETAIOEMUL option in *fcntl*(2F). If emulation is allowed the operation will be attempted in synchronous mode.

[EFBIG] Attempt to write past the end-of-file.

[EIO] Some physical I/O error occurred.

[ENXIO] I/O operation could not be started (e.g., tape not on-line).

If the asynchronous I/O operation is emulated by a synchronous operation, then, should the synchronous operation fail, the error code is reported in the **rt_errno**.

NAME

bind - binds a name to a socket

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*4 bind,s,namelen
record/sockaddr/name
iretval= bind(s, name, namelen)

DESCRIPTION

bind assigns a name to an unnamed socket. When a socket is created with *socket*(2F) it exists in a name space (address family) but has no name assigned; *bind* requests the name be assigned to the socket.

NOTES

The rules used in name binding vary between communication domains. Consult the Reference manual entries in Sections 4 and 7 for detailed information.

RETURN VALUE

If the *bind* is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global errno.

ERRORS

The *bind* call will fail if:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The name parameter is not in a valid part of the user address space.

SEE ALSO

connect(2F), listen(2F), socket(2F), getsockname(2F)

NAME

brk, sbrk - change data segment space allocation

SYNOPSIS

integer*4 brk, endds
integer*4 sbrk, incr

iretval = brk (endds)
iretval = sbrk (incr)

DESCRIPTION

brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment [see *exec*(2F)]. The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

brk sets the break value to *endds* and changes the allocated space accordingly.

sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *incr* can be negative, in which case the amount of allocated space is decreased.

brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

[EAGAIN] Total amount of system memory available for a read during physical I/O is temporarily insufficient [see *shmop*(2F)]. This may occur even though the space requested was less than the system-imposed maximum process size [see *ulimit*(2F)].

[EAGAIN] The data segment is locked resulting in more resident pages being allocated than are currently available.

[EBUSY] Such a change would deallocate space that is still in use for asynchronous I/O or connected interrupts.

[ENOMEM] Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size [see *ulimit*(2F)].

EXAMPLE

```
        program brk
#       include <sysf/errno.i>
        integer*4 brk, endds
        integer*4 sbrk, incr
        integer*4 data_segment
        integer*4 iretval

c  increment current data segment by 512 bytes

        incr = 512
        data_segment = sbrk (incr)
        if (data_segment .lt. 0) then
              write (*,*) 'sbrk error:', data_segment
        else
              write (*,*) 'old data segment:', data_segment

c  increment new data segment by another 512 bytes

              endds = data_segment + 512
              iretval = brk (endds)
              if (iretval .lt. 0) write (*,*) 'brk error:', iretval
        endif
c       ...
        end
```

SEE ALSO

aread(2F), exec(2F), resident(2F), shmop(2F), ulimit(2F), end(3C), cintrio(7).

DIAGNOSTICS

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a negative value indicating the error is returned.

NAME

bsfree - free a binary semaphore

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/ipc.i>
#    include <sys/binsem.i>

     integer*4 bsfree, bid
     iretval = bsfree (bid)
```

DESCRIPTION

bid is a binary semaphore identifier obtained from a *bsget* system call. *bsfree* frees (releases) the binary semaphore identifier indicated by *bid*. *bsfree* does not unlock the binary semaphore.

[EINVAL] *bsfree* will fail if *bid* is not a valid binary semaphore identifier.

EXAMPLE

```
     program bsfree
#    include <sysf/types.i>
#    include <sysf/ipc.i>
#    include <sysf/binsem.i>
     integer*4 bsfree, bid, bsget, bsem, bslk, bsunlk, bslkc, iretval

c  Get a binary semaphore using agreed-upon key value

     bid = bsget (1234, bsem, '777'o .or. IPC_CREAT)
     if (bid .lt. 0) write (*,*) 'bsget error:', bid

c  Take the semaphore using bslk or bslkc (both shown as example)

     iretval = bslk (bsem, bid)

     iretval = bslkc (bsem)
     if (iretval .ne. 0) then
       write (*,*) 'bslkc locked the semaphore'
     else
       write (*,*) 'bslkc could not lock the semaphore'
     endif

c  Perform critical code here:
c  Now give up the semaphore

     iretval = bsunlk (bsem, bid)

c  Free up the semaphore

     iretval = bsfree (bid)
     if (iretval .lt. 0) write (*,*) 'bsfree error:', iretval
     end
```

SEE ALSO

bsget(2F), bslk(3C), bslkc(3C), bsunlk(3C).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

bsget - get a binary semaphore

SYNOPSIS

 # include <sysf/types.i>
 # include <sysf/ipc.i>
 # include <sysf/binsem.i>

integer*4 bsget, key, bsem, bsemflg
iretval = bsget (key, bsem, bsemflg)

DESCRIPTION

bsget creates and initializes the user-supplied binary semaphore associated with *key*. The caller must be superuser or have realtime privileges.

A binary semaphore identifier and associated data structure are created for *key* if one of the following are true:

> *key* is equal to **IPC_PRIVATE**.
>
> *key* does not already have a binary semaphore identifier associated with it and (*bsemflg* & **IPC_CREAT**) is "true."

bsem contains a pointer to the binary semaphore itself. Upon creation, the binary semaphore is initialized to an unlocked condition. *bsget* will fail if one or more of the following are true:

[EAGAIN] Insufficient system memory to lock down specified buffer areas.

[EAGAIN] Insufficient system memory to set up mapping.

[EEXIST] A binary semaphore identifier exists for *key* but ((*bsemflg* & **IPC_CREAT**) & (*bsemflg* & **IPC_EXCL**)) is "true."

[EFAULT] The locations containing the binary semaphore may not be written by the caller.

[EFAULT] The *bsem* parameter points outside the allocated address space.

[ENOENT] A binary semaphore identifier does not exist for *key* and (*bsemflg* & **IPC_CREAT**) is "false."

[ENOSPC] A binary semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed binary semaphore identifiers system wide would be exceeded.

[ENOSPC] A binary semaphore identifier is to be created but the system-imposed limit on the maximum number of processes using binary semaphore identifiers system wide would be exceeded.

[ENOSPC] A binary semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed binary semaphore identifiers for a process would be exceeded.

[EPERM] The sending process does not have realtime privileges and its user ID is not superuser.

EXAMPLE

See bsfree(2F) for an example.

SEE ALSO

bsfree(2F), bslk(3C), bslkc(3C), bsunlk(3C).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a binary semaphore identifier, is returned. Otherwise, a negative value indicating the error is returned.

NAME

chdir - change working directory

SYNOPSIS

integer*4 chdir
character*SIZE path
iretval = chdir (path)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to the path name of a directory. *chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

chdir will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chdir* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
        program chdir
#       include <sysf/fcntl.i>
        integer*4 chdir
        character*40 path

        integer*4 iretval
        integer*4 open, fildes

c   The current directory is 'example'.  Change to the
c   previous directory and then open 'example'.

        path = '..'
        iretval = chdir (path)
        fildes = open ('example', O_RDONLY)
        if (fildes .lt. 0) write (*,*) 'error:', fildes
        end
```

SEE ALSO

chroot(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

chmod - change mode of file

SYNOPSIS

integer*4 chmod, mode
character*SIZE path
iretval = chmod (path, mode)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to a path name naming a file. *chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

'04000'O	Set user ID on execution.
'020#0'O	Set group ID on execution if # is 7, 5, 3, or 1 Enable mandatory file/record locking if # is 6, 4, 2, or 0
'01000'O	Save text image after execution.
'00400'O	Read by owner.
'00200'O	Write by owner.
'00100'O	Execute (search if a directory) by owner.
'00070'O	Read, write, execute (search) by group.
'00007'O	Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file; If the effective user ID of the process is not super-user, mode bit '01000'O (save text image on execution) is cleared.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit '02000'O (set group ID on execution) is cleared.

If a '410'O executable file has the sticky bit (mode bit '01000'O) set, the operating system will not delete the program text from the swap area when the last user process terminates. If a '413'O executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

If the mode bit '02000'O (set group ID on execution) is set and the mode bit '00010'O (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may effect future calls to open(2F), creat(2F), read(2F), and write(2F) on this file.

chmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chmod* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
        program chmod
        integer*4 chmod, mode
        character*40 path

        integer*4 iretval

c  Change access permissions to read/write for owner, read only for
c  group and others

        path = '../example/chmod.F'
        mode = '400'o .or. '200'o .or. '40'o .or. '4'o
        iretval = chmod (path, mode)
        if (iretval .lt. 0) write (*,*) 'chmod error:', iretval
        end
```

SEE ALSO

chmod(1), chown(2F), creat(2F), fcntl(2F), mknod(2F), open(2F), read(2F), write(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

chown - change owner and group of a file

SYNOPSIS

integer*4 chown, owner, group
character*SIZE path
iretval = chown (path, owner, group)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, '04000'O and '02000'O respectively, will be cleared.

chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied on a component of the path prefix.

[EPERM] The effective user ID does not match the owner of the file and the effective user ID is not super-user.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *path* points outside the allocated address space of the process.

[EINTR] A signal was caught during the *chown* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program chown
      integer*4 chown, owner, group
      character*40 path

      integer*4 iretval
      integer*4 getuid, getgid

c Get current owner and group id's

      owner = getuid ()
      group = getgid ()

c Make new group owner for file

      group = group - 1
      path = '../example/chown.F'
      iretval = chown (path, owner, group)
      if (iretval .lt. 0) write (*,*) 'chown error:', iretval
      end
```

SEE ALSO

chown(1), chmod(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

chroot - change root directory

SYNOPSIS

integer*4 chroot
character*SIZE path
iretval = chroot (path)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to a path name naming a directory. *chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

chroot will fail and the root directory will remain unchanged if one or more of the following are true:

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EPERM]	The effective user ID is not super-user.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chroot* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program chroot
      integer*4 chroot
      character*40 path
      integer*4 iretval

c  Change root to itself

      path = '/'
      iretval = chroot (path)
      if (iretval .lt. 0) write (*,*) 'chroot error:', iretval
      end
```

SEE ALSO

chdir(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

cisema - wait for a connected interrupt

SYNOPSIS

integer*4 cisema, cid
iretval = cisema(cid)

DESCRIPTION

The *cisema* system call waits for an interrupt associated with the connected interrupt identifier specified by *cid*. *cid* is returned by a previous CI_CONNECT *ioctl*(2F) command (see *cintrio*(7)).

The *cisema* system call will fail if one or more of the following are true:

[EINVAL] *cid* is invalid.

[EINVAL] The connected interrupt associated with *cid* is not connected to the calling process.

[EINVAL] The delivery method is not CINTR_SEMA.

EXAMPLE

```
        program cisema
#       include <sysf/types.i>
#       include <sysf/cintrio.i>
#       include <sysf/evt.i>
        integer*4 cisema, cid
        integer*4 ioctl, fildes, iretval
        integer*4 eid
        record /cintrio/ arg

c   Initialize structure assuming posting events (see evpost)

        arg.ci_method = CINTR_EVENTS
        arg.ci_id = eid
        arg.ci_polloc = 0
        arg.ci_flags = 0

c   Connect the interrupt to an already opened fildes

        cid = ioctl (fildes, CI_CONNECT, arg)
        if (cid .lt. 0) write (*,*) 'ioctl error:', cid

c   Wait for connected interrupt to happen

        iretval = cisema (cid)
        if (iretval .lt. 0) write (*,*) 'cisema error:', iretval
        end
```

SEE ALSO

cintrio(7).

DIAGNOSTICS

A 0 value is returned if the *cisema* system call is successful. Otherwise, a negative value indicating the error is returned.

NAME

close - close a file descriptor

SYNOPSIS

integer*4 close, fildes
iretval = close (fildes)

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(2F)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2F) and *sigset*(2F)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

When *close* is issued against a *fildes* with which asynchronous I/O requests are associated, *close* waits for noncancelable I/O operations and cancels all cancelable and queued operations.

EXAMPLE

```
      program close
#     include <sysf/fcntl.i>
      integer*4 close, fildes
      integer*4 open, iretval

c   Open a file, then close it

      fildes = open ('../example/close.F', O_RDONLY)
      if (fildes .lt. 0) write (*,*) 'open error:', fildes
      iretval = close (fildes)
      if (iretval .lt. 0) write (*,*) 'close error:', iretval
      end
```

DIAGNOSTICS

The named file is closed unless one or more of the following are true:

[EBADF] *fildes* is not a valid open file descriptor.

[EINTR] A signal was caught during the *close* system call.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

acancel(2F), aread(2F), awrite(2F), creat(2F), dup(2F), exec(2F), fcntl(2F), intro(2F), open(2F), pipe(2F), signal(2F), sigset(2F), streamio(7).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

connect - initiates a connection on a socket

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*4 connect,s,namelen
record/sockaddr/name
iretval=connect(s, name, namelen)

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by a name which is an address in the communications space of the socket. Each communications space interprets the name parameter in its own way. Consult the Reference manual entries in Sections 4 and 7 for detailed information.

Generally, STREAM sockets may connect only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve their association by connecting to an invalid address, such as a null address.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a -1 is returned, and a more specific error code is stored in errno.

ERRORS

The call fails if:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The name parameter specifies an area outside the process address space.
[EWOULDBLOCK]	The socket is non-blocking and the connection cannot be completed immediately.

SEE ALSO

accept(2F), socket(2F), getsockname(2F)

NAME

creat - create a new file or rewrite an existing one

SYNOPSIS

integer*4 creat, mode
character*SIZE path
iretval = creat (path, mode)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *creat* creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*. If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process the group ID of the process is set to the effective group ID, of the process and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

> All bits set in the process's file mode creation mask are cleared [see *umask*(2F)].
> The "save text image after execution bit" of the mode is cleared [see *chmod*(2F)].

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls [see *fcntl*(2F)]. A new file may be created with a mode that forbids writing.

creat fails if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] A component of the path prefix does not exist.

[EACCES] Search permission is denied on a component of the path prefix.

[ENOENT] The path name is null.

[EACCES] The file does not exist and the directory in which the file is to be created does not permit writing.

[EROFS] The named file resides or would reside on a read-only file system.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

[EACCES] The file exists and write permission is denied.

[EISDIR] The named file is an existing directory.

[EMFILE] NOFILES file descriptors are currently open.

[EFAULT] *path* points outside the allocated address space of the process.

[ENFILE] The system file table is full.

[EAGAIN] The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2F)].

[EINTR] A signal was caught during the *creat* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

[ENOSPC] The file system is out of inodes.

EXAMPLE

```
      program creat
      integer*4 creat, mode
      character*40 path
      integer*4 fildes

c   Create a file and open for writing

      path = '../example/tst.x'
      mode = '600'o
      fildes = creat (path, mode)
      if (fildes .lt. 0) write (*,*) 'creat error:', fildes
      end
```

SEE ALSO

chmod(2F), close(2F), dup(2F), fcntl(2F), lseek(2F), open(2F), read(2F), umask(2F), write(2F).

DIAGNOSTICS

Upon successful completion, a non-negative file descriptor, is returned. Otherwise, a negative value indicating the error is returned.

NAME

dup - duplicate an open file descriptor

SYNOPSIS

integer*4 dup, fildes
iretval = dup (fildes)

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls [see *fcntl*(2F)].

The file descriptor returned is the lowest one available.

dup will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EINTR] A signal was caught during the *dup* system call.

[EMFILE] NOFILES file descriptors are currently open.

[ENOLINK] *Fildes* is on a remote machine and the link to that machine is no longer active.

EXAMPLE

```
      program dup
      integer*4 dup, fildes
      integer*4 iretval, new_fildes, creat, close

c   Redirect standard output to a file

      fildes = creat ('../example/std.out', '600'o)
      iretval = close (1)
      new_fildes = dup (fildes)
      if (new_fildes .lt. 0) write (*,*) 'error:', new_fildes
      write (*,*) 'this message should be in the file'
      end
```

SEE ALSO

close(2F), creat(2F), exec(2F), fcntl(2F), open(2F), pipe(2F), lockf(3C).

DIAGNOSTICS

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a negative value indicating the error is returned.

NAME

estat, efstat - get extended file status

SYNOPSIS

```
# include <sysf/types.i>
# include <sysf/stat.i>

integer*4 estat
character*SIZE path
record /estat/ buf
iretval = estat (path, buf)

integer*4 efstat, fildes
record /estat/buf
iretval = efstat (fildes, buf)
```

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to a path name where a file is located. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *estat* obtains information about the named file.

The information returned by *estat*, in a remote file sharing environment, depends upon the user/group mapping set up between the local and remote computers (see *idload*(1M)).

efstat gets information about an open file from the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

buf is a pointer to an *estat* structure where information is placed concerning the file. The following members are included in the contents of the structure:

```
integer*2  st_mode     !File mode (see mknod(2F))
integer*2  st_ino      !Inode number
integer*2  st_dev      ! ID of device containing
                       ! a directory entry for this file
integer*2  st_rdev     ! ID of device
                       ! This entry is defined only for
                       ! character special/block special files
integer*2  st_nlink    ! Number of links
integer*2  st_uid      ! User ID of the file's owner
integer*2  st_gid      ! Group ID of the file's group
integer*4  st_size     ! File size in bytes
integer*4  st_atime    ! Time of last access
integer*4  st_mtime    ! Time of last data modification
integer*4  st_ctime    ! Time of last file status change
                       ! Times measured in seconds since
                       ! 00:00:00 GMT, Jan. 1, 1970
record /extent_t/ st_extents(NF5EXT) ! Contiguous extent list
integer*4  st_lastw    ! Byte offset of last block written to
integer*4  st_flags    ! File characteristics (see fcntl(5))
```

st_mode The mode of the file as described in the *mknod*(2F) system call.

st_ino This field uniquely identifies the file in a given file system. The pair **st_ino** and **st_dev** uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. It's value can be used as input to the *ustat*(2F) system call to determine more information about the file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. For pipes or fifos, this is the count of the data currently in the file. For block special or character special, this is not defined.

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2F), *mknod*(2F), *pipe*(2F), *prealloc*(2F), *read*(2F), and *utime*(2F).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2F), *mknod*(2F), *pipe*(2F), *prealloc*(2F), *utime*(2F), and *write*(2F).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2F), *chown*(2F), *creat*(2F), *link*(2F), *mknod*(2F), *pipe*(2F), *prealloc*(2F), *unlink*(2F), *utime*(2F), and *write*(2F).

st_extents Array of contiguous file extent structures. Each structure contains the byte offset of the extent in the file system and the sum of that extent plus all previous extents.

st_lastw The byte offset in the file system of the last block written to this file.

st_flags Contains bit flags that describe certain characteristics of this file. For example, does this file have contiguous extents, will the physical space be removed on a truncate, will this file automatically grow, etc. See *fcntl*(5) for more information.

estat will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *buf* or *path* points to an invalid address.

[EINTR] A signal was caught during the *estat* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

efstat will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid open file descriptor.

[EFAULT] *buf* points to an invalid address.

[ENOLINK] *fildes* points to a remote machine and the link to that machine is no longer active.

EXAMPLE

```
        program estat
#       include <sysf/types.i>
#       include <sysf/stat.i>
#       include <sysf/fcntl.i>

        integer*4 estat, efstat, fildes
        character*40 path
        record /estat/ buf_path, buf_file
        integer*4 open, iretval

c    Get extended status of file using pathname

        path = '../example/estat.F'
        iretval = estat (path, buf_path)
        if (iretval .lt. 0) write (*,*) 'estat error:', iretval

c    Get extended status of file from file descriptor

        fildes = open (path, O_RDONLY)
        if (fildes .lt. 0) write (*,*) 'open error:', fildes
        iretval = efstat (fildes, buf_file)
        if (iretval .lt. 0) write (*,*) 'efstat error:', iretval

c    Print some of the information from both structures

        write (*,9000) buf_path.st_ino, buf_path.st_size, buf_path.st_flags
        write (*,9000) buf_file.st_ino, buf_file.st_size, buf_file.st_flags

9000    format (' Inode:', I5,', Size in bytes:',I9,', Flags:',z8)
        end
```

SEE ALSO

chmod(2F), chown(2F), creat(2F), link(2F), mknod(2F), pipe(2F), prealloc(2F), read(2F), stat(2F), time(2F), unlink(2F), utime(2F), write(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

evctl - event control operations

SYNOPSIS

 # include <sysf/evt.i>

integer*4 evctl, eid, cmd
iretval = evctl (eid, cmd)

DESCRIPTION

evctl provides a variety of event control operations as specified by *cmd*. The following *cmd*s are available:

EV_IGN Mark the event identifier *eid* as being ignored. Note that when an event identifier is first created (via the *evget* system call), the event identifier is not ignored.

EV_NOIGN Mark the event identifier *eid* as not being ignored.

evctl will fail if one or more of the following are true:

[EINVAL] *eid* is not a valid event identifier.

[EINVAL] *cmd* is not a valid command.

EXAMPLE

```
      program evctl
#     include <sysf/evt.i>
      integer*4 evctl, eid
      integer*4 evget, iretval

c  Use evget call to get an event identifier

      eid = evget (EVT_QUEUE, 10, 0, 0)

c  Now ignore events posted to this event id

      iretval = evctl (eid, EV_IGN)
      if (iretval .lt. 0) write (*,*) 'evctl error:', iretval
      end
```

SEE ALSO

evget(2F), evpost(2F), evrcv(2F), evrcvl(2F), evrel(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely an event identifier, is returned. Otherwise, a negative value indicating the error is returned.

NAME

evget - get an event identifier

SYNOPSIS

```
#    include <sysf/signal.i>
#    include <sysf/evt.i>

     integer*4 evget, method, quecnt, signo
     external func
     iretval = evget (method, quecnt, signo, func)
```

DESCRIPTION

evget creates an event identifier and defines the action that should be performed when the event is posted. The created event identifier is the lowest-numbered non-negative event identifier which was not in use by the process.

method specifies how to handle events posted to the event identifier.

If *method* is specified as EVT_SIGNAL, then posted events will result in a signal being delivered to the process. If *method* is specified as EVT_QUEUE, then posted events will be queued for the process. Posted events which are queued for the process may be read via the *evrcv* and *evrcvl* system calls.

quecnt specifies the number of internal system blocks to allocate the event identifier for queuing posted events or pending signals. When it is specified as zero, no internal blocks will be allocated for the event identifier (instead, internal system blocks will be allocated, from a per-process pool of internal system blocks, when the event is posted).

When *method* is specified as EVT_SIGNAL, and *quecnt* is specified as non-zero, then *quecnt* specifies the maximum number of signals which may be queued to the process for the event identifier at any time. When *method* is specified as EVT_QUEUE, and *quecnt* is specified as non-zero, then *quecnt* specifies the maximum number of posted events which may be queued to the process for the event identifier at any time.

If *quecnt* is specified as zero, then the only limit to the number of posted events or signals which may be queued to the process for the event identifier is the availability of internal system blocks (from a per-process pool of internal system blocks) when the event is posted.

If *method* is specified as EVT_SIGNAL, then *signo* is the signal number to deliver when the event is posted, and *func* is the signal-catching function to invoke. If *func* is specified as %val (0) then the default signal-catching function for *signo* (specified via the *signal* or *sigset* system service) will be invoked when the event is posted.

Signals delivered as a result of a posted event differ from signals sent for other reasons as follows:

> Signals delivered because of a posted event are queued. That is, if, for a given signal number, there are multiple signals to deliver to the process, they will all be delivered, and they will be delivered in the order that they were posted.

Signal-catching functions may determine the event information for the event which caused the signal to be delivered, since the third argument to a signal-catching function contains the event information. The first argument to a signal-catching function is an integer which is the signal number. The second argument is an integer whose value is defined by the signal number. The third argument is an event structure (of type event_t) defining the event information. If a signal-catching function is not invoked as a result of an event being delivered, then the event identifier field of the event structure will contain a -1.

If the same signal number is used by both the event mechanism and the standard Unix signal mechanism, there is a potential for the standard Unix signal sending mechanism to not deliver sent signals. Thus, it is suggested that the same signal number not be used for handling signals which result from events being posted and for handling signals which result from other sources.

It is suggested that the signal handling for signals resulting from posted events be defined via the *sigset* system service (and not via the *signal* system service).

evget will fail if one or more of the following are true:

[EINVAL] *method* is not a valid method of handling posted events.

[EINVAL] *method* is EVT_SIGNAL, and *signo* is an illegal signal number, including **SIGKILL**.

[ENOSPC] An event identifier is to be created but the system-imposed limit on the maximum number of processes using event identifiers system wide would be exceeded.

[ENOSPC] An event identifier is to be created but the system-imposed limit on the maximum number of allowed event identifiers for a process would be exceeded.

[ENOSPC] An event identifier is to be created and internal event blocks for it are to be preallocated, but the system-imposed limit on the maximum number of allowed internal event blocks for a process would be exceeded.

EXAMPLE

See evpost(2F) for an example.

SEE ALSO

evctl(2F), evpost(2F), evrcv(2F), evrcvl(2F), evrel(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely an event identifier, is returned. Otherwise, a negative value indicating the error is returned.

NAME

evpost - post an event to a process

SYNOPSIS

integer*4 evpost, pid, eid, dataitem
iretval = evpost (pid, eid, dataitem)

DESCRIPTION

evpost posts an event to the specified process and event identifier. The process to which the event is posted is specified by *pid*. The event identifier to post to is specified by *eid.* The data item to be posted is specified by *dataitem* (see *evrcv*(2F) and *evrcvl*(2F)).

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is superuser or the sending process has realtime privileges.

evpost will fail if one or more of the following are true:

[EPERM] The sending process does not have realtime privileges, its user ID is not superuser, and its real or effective user ID does not match the real or effective user ID of the receiving process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EINVAL] The process specified by *pid* does not have the event identifier specified by *eid*.

[EAGAIN] The event identifier specified by *eid*, for the process specified by *pid*, is currently being ignored.

[ENOSPC] An event is to be posted but the process-imposed limit (as specified to *evget*(2F)) on the maximum number of pending queued events or pending queued signals for the event identifier would be exceeded.

[ENOSPC] An event is to be posted but the system-imposed limit on the maximum number of pending queued events or pending queued signals for the process would be exceeded.

EXAMPLE

```
      program evpost
#     include <sysf/signal.i>
#     include <sysf/evt.i>

      integer*4 evget, func, eid, evpost, pid, dataitem, iretval
      integer*4 getpid, sigset
      external func

c  Setup default signal catching function

      iretval = sigset (SIGUSR1, func)
      if (iretval .lt. 0) write (*,*) 'sigset error:', iretval

c  Get an event identifier handling user1 signal

      eid = evget (EVT_SIGNAL, 2, SIGUSR1, func)
      if (eid .lt. 0) write (*,*) 'evget error: ', eid

c  Get process id and post user1 events

      pid = getpid ()
      if (pid .lt. 0) write (*,*) 'pid error:', pid
      do 100 dataitem = 1, 10
      iretval = evpost (pid, eid, dataitem)
      if (iretval .lt. 0) write (*,*) 'evpost error:', iretval
100   continue
      end

c  Subroutine to catch the event

c  The event structure contents are passed (not the address
c  of the event structure).

      subroutine func (signo, sigarg, arg1, arg2)
      integer*4 signo, sigarg, arg1, arg2
      integer*2 ev_eid, ev_type
      integer*4 ev_dataitem

c  Knowing the order that the event_t structure elements
c  are defined, get the values for the elements desired.

      ev_eid = %loc (arg1) / '10000'x        ! eid in upper half
      ev_type = %loc (arg1)        ! type in lower half
      ev_dataitem = %loc (arg2)
      write (*,9000) %loc (signo), %loc (sigarg), ev_dataitem,
&          ev_eid, ev_type
      return
9000  format (' signal: ', i6, ', uniq2sig: ', i6, ', dataitem: ', i6,
&          ', eid: ', i6, ', type: ', i6)
      end
```

SEE ALSO

evctl(2F), evget(2F), evrcv(2F), evrcvl(2F), evrel(2F), setrt(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

evrcv - receive any queued event

SYNOPSIS

```
#    include <sysf/evt.i>

     integer*4 evrcv, waitflg
     record /event_t/ event
     iretval = evrcv (waitflg, event)
```

DESCRIPTION

evrcv receives the first posted event which is queued to the process. The received event is dequeued.

If *event* is not %val(0), then the posted event information is stored in the location pointed to by *event*.

waitflg specifies the action to be taken if an event is not currently queued to the process. These are as follows:

> If *waitflg* is specified as zero, the calling process will return immediately with a return value of EAGAIN.
>
> Otherwise, the calling process will suspend execution until one of the following occurs:
>
> > Any event is queued to the process.
> >
> > The calling process receives a signal that is to be caught. In this case an event has not been queued to the process, and the calling process resumes execution in the manner prescribed in *signal*(2F)).

evrcv will fail if one or more of the following are true:

[EINVAL] Of the event identifiers for the process which are not currently being ignored, none are defined to have an event queued to the process when the event is posted (i.e., the *method* parameter to *evget*(2F) was not specified as EVT_QUEUE for any event identifier which is currently valid and not ignored for the process).

[EAGAIN] No events are queued for the process, and *waitflg* was specified as zero.

[EINTR] A signal was caught during the *evrcv* system call.

EXAMPLE

```
      program evrcv
#     include <sysf/evt.i>

      integer*4 evrcv, waitflg
      integer*4 evget, eid, eid2
      integer*4 evpost, pid, dataitem, iretval
      integer*4 getpid
      record /event_t/ event

c  Get two event identifiers

      eid = evget (EVT_QUEUE, 10, 0, 0)
      if (eid .lt. 0) write (*,*) 'evget error (call 1): ', eid
      eid2 = evget (EVT_QUEUE, 10, 0, 0)
      if (eid2 .lt. 0) write (*,*) 'evget error (call 2): ', eid2

c  Get process id and post events on the queues

      pid = getpid ()
      if (pid .lt. 0) write (*,*) 'pid error:', pid
      do 100 dataitem = 1, 10
      iretval = evpost (pid, eid, dataitem)
      if (iretval .lt. 0) write (*,*) 'evpost error (call 1):', iretval
      iretval = evpost (pid, eid2, dataitem+100)
      if (iretval .lt. 0) write (*,*) 'evpost error (call 2):', iretval
100   continue

c  Receive any queued event

      waitflg = .not. 0
      do 500 dataitem = 1, 20
      iretval = evrcv (waitflg, event)
      if (iretval .lt. 0) then
        write (*,*) 'evrcv error:', iretval
      else
        write (*,6000) event.ev_eid, event.ev_type, event.ev_dataitem
      endif
500   continue
6000  format (' eid:', i6, ' type:', i6, ' dataitem:', i6)
      end
```

SEE ALSO

evctl(2F), evget(2F), evpost(2F), evrcvl(2F), evrel(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

evrcvl - receive any queued event from a specified list

SYNOPSIS

 # include <sysf/evt.i>

integer*4 evrcvl, evl(64), evcnt, waitflg
record /event_t/ event
iretval = evrcvl (evl, evcnt, waitflg, event)

DESCRIPTION

evrcvl receives the first posted event, from the specified list, which is queued to the process. The received event is dequeued.

If *event* is not %val(0), then the posted event information is stored in the location pointed to by *event*.

evl points to a list of up to 64 specified event identifiers. *evcnt* is the number of event identifiers specified by *evl*. Only events posted to one of the specified event identifiers may be received by *evrcvl*.

waitflg specifies the action to be taken if an event from the specified list is not currently queued to the process. These are as follows:

> If *waitflg* is specified as zero, the calling process will return immediately with a return value of EAGAIN.
>
> Otherwise, the calling process will suspend execution until one of the following occurs:
>
> > Any event from the specified list is queued to the process.
> >
> > The calling process receives a signal that is to be caught. In this case an event from the specified list has not been queued to the process, and the calling process resumes execution in the manner prescribed in *signal*(2F).

evrcvl will fail if one or more of the following are true:

[EFAULT] *evl* points to an illegal address.

[EINVAL] *evcnt* is less than 1 or larger than 64.

[EINVAL] At least one of the specified event identifiers is not a valid event identifier.

[EINVAL] At least one of the specified event identifiers is currently being ignored, or is defined to not have an event queued to the process when the event is posted (i.e., the *method* parameter was not specified as EVT_QUEUE for the corresponding *evget* system call).

[EAGAIN] No events from the specified list are queued for the process, and *waitflg* was specified as zero.

[EINTR] A signal was caught during the *evrcvl* system call.

EXAMPLE

```
      program evrcvl
#     include <sysf/evt.i>

      integer*4 evrcvl, evl (2), evcnt, waitflg
      integer*4 evget, eid, eid2, eid3, eid4
      integer*4 evpost, pid, dataitem, iretval, getpid
      record /event_t/ event

c  Get four event identifiers

      eid = evget (EVT_QUEUE, 5, 0, 0)
      if (eid .lt. 0) write (*,*) 'evget error (call 1): ', eid
      eid2 = evget (EVT_QUEUE, 5, 0, 0)
      if (eid2 .lt. 0) write (*,*) 'evget error (call 2): ', eid2
      eid3 = evget (EVT_QUEUE, 5, 0, 0)
      if (eid3 .lt. 0) write (*,*) 'evget error (call 3): ', eid3
      eid4 = evget (EVT_QUEUE, 5, 0, 0)
      if (eid4 .lt. 0) write (*,*) 'evget error (call 4): ', eid4

c  Get process id and post events on the queues

      pid = getpid ()
      if (pid .lt. 0) write (*,*) 'pid error:', pid
      do 100 dataitem = 1, 5
      iretval = evpost (pid, eid, dataitem)
      if (iretval .lt. 0) write (*,*) 'evpost error (call 1):', iretval
      iretval = evpost (pid, eid2, dataitem+100)
      if (iretval .lt. 0) write (*,*) 'evpost error (call 2):', iretval
      iretval = evpost (pid, eid3, dataitem+200)
      if (iretval .lt. 0) write (*,*) 'evpost error (call 3):', iretval
      iretval = evpost (pid, eid4, dataitem+300)
      if (iretval .lt. 0) write (*,*) 'evpost error (call 4):', iretval
100   continue

c  Receive subset of queued events

      waitflg = .not. 0
      evl (1) = eid2
      evl (2) = eid4
      evcnt = 2
      do 500 dataitem = 1, 10
      iretval = evrcvl (evl, evcnt, waitflg, event)
      if (iretval .lt. 0) then
        write (*,*) 'evrcv error:', iretval
      else
        write (*,6000) event.ev_eid, event.ev_type, event.ev_dataitem
      endif
500   continue
6000  format (' eid:', i6, ' type:', i6, ' dataitem:', i6)
      end
```

SEE ALSO

evctl(2F), evget(2F), evpost(2F), evrcv(2F), evrel(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

evrel - release an event identifier

SYNOPSIS

integer*4 evrel, eid
iretval = evrel (eid)

DESCRIPTION

evrel releases (frees) the event identifier indicated by *eid*.

The argument *eid* is an event identifier which was obtained via an *evget* system call.

All queued posted events and queued pending signals for the event identifier are removed.

evrel will fail if the following is true:

[EINVAL] *eid* is not a valid event identifier.

[EBUSY] *eid* is currently in use by another subsystem on behalf of the process.

EXAMPLE

```
        program evrel
#       include <sysf/evt.i>
        integer*4 evrel, eid, iretval
        integer*4 evpost, pid, dataitem, getpid, evget

c   Get event identifier

        eid = evget (EVT_QUEUE, 10, 0, 0)
        if (eid .lt. 0) write (*,*) 'evget error:', eid

c   Post some events to it

        pid = getpid ()
        do 100 dataitem = 1, 10
        iretval = evpost (pid, eid, dataitem)
        if (iretval .lt. 0) write (*,*) 'evpost error:', iretval
100     continue

c   Release event identifier, ignoring posted events

        iretval = evrel (eid)
        if (iretval .lt. 0) write (*,*) 'evrel error:', iretval
        end
```

SEE ALSO

evctl(2F), evget(2F), evpost(2F), evrcv(2F), evrcvl(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

exec: execl, execv, execle, execve, execlp, execvp - execute a file

SYNOPSIS

integer*4 execl
character*SIZE path, arg0, arg1,..., argn
iretval = execl (path, arg0, arg1,..., argn, %val(0))

integer*4 execv, argv(SIZE)
character*SIZE path
iretval = execv (path, argv)

integer*4 execle, envp(SIZE)
character*SIZE path, arg0, arg1,..., argn
iretval = execle (path, arg0, arg1,..., argn, %val(0), envp)

integer*4 execve, argv(SIZE), envp(SIZE)
character*SIZE path
iretval= execve (path, argv, envp)

integer*4 execlp
character*SIZE file, arg0, arg1,..., argn
iretval = execlp (file, arg0, arg1,..., argn, %val(0))

integer*4 execvp, argv(SIZE)
character*SIZE file
iretval = execvp (file, argv)

DESCRIPTION

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header [see *a.out*(4)], a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

When a C program is executed, it is called as follows:

main (argc, argv, envp)
int argc;
char **argv, **envp;

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

SIZE can be any number between and including 1 through 128.

path points to a path name that identifies the new process file.

file points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" [see *environ*(5)]. The environment is supplied by the shell [see *sh*(1)].

arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component). The synopsis format is a guideline: arg0, arg1,..., argn do not have to be the length indicated by the synopsis.

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). To terminate *argv*, make the next member of *argv* zero.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. To terminate *envp*, make the next member of *envp* zero.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2F). For those file descriptors that remain open, the file pointer is unchanged.

exec waits on any outstanding, noncancelable asynchronous I/O requests; cancelable and queued asynchronous I/O requests are deleted.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(2F).

For signals set by *sigset*(2F), *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held.

If the set-user-ID mode bit of the new process file is set [see *chmod*(2F)], *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

If the calling process has a process, text or data lock, an *unlock* is performed [see *plock*(2F)].

The shared memory segments attached to the calling process will not be attached to the new process [see *shmop*(2F)].

The binary semaphores attached to the calling process will not be attached to the new process [see *bsget*(2F)].

The event identifiers attached to the calling process will not be attached to the new process [see *evget*(2F)].

The interrupts connected to the calling process will not be connected to the new process [see *cintrio*(7)].

Profiling is disabled for the new process; see *profil*(2F).

The new process also inherits the following attributes from the calling process:

nice value [see *nice*(2F)]
process ID
parent process ID
process group ID
semadj values [see *semop*(2F)]
tty group ID [see *exit*(2F) and *signal*(2F)]
trace flag [see *ptrace*(2F) request 0]
time left until an alarm clock signal [see *alarm*(2F)]
current working directory
root directory
file mode creation mask [see *umask*(2F)]
file size limit [see *ulimit*(2F)]
utime, *stime*, *cutime*, and *cstime* [see *times*(2F)]
file-locks [see *fcntl*(2F) and *lockf*(3C)]

exec will fail and return to the calling process if one or more of the following are true:

[ENOENT] One or more components of the new process path name of the file do not exist.

[ENOTDIR] A component of the new process path of the file prefix is not a directory.

[EACCES] Search permission is denied for a directory listed in the new process file's path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execution permission.

[ENOEXEC] The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.

[ETXTBSY] The new process file is a pure procedure (shared text) file that is currently open for writing by some process.

[ENOMEM] The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.

[E2BIG] The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes.

[EFAULT] Required hardware is not present.

[EFAULT] An a.out that was compiled with the MAU or 32B flag is running on a machine without a MAU or 32B.

[EFAULT] *path*, *argv*, or *envp* point to an illegal address.

[EAGAIN] Not enough memory.

[ELIBACC] Required shared library does not have execute permission.

[ELIBEXEC] Trying to *exec*(2F) a shared library directly.

[EINTR] A signal was caught during the *exec* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program exec
c  This program assumes that an executable program called hello
c  exists under '../example'.
c
c  A simple hello program might be:
c
c     program hello
c     integer*4 argc, i, iargc
c     character*20 argv
c     argc = iargc ()
c     if (argc .le. 0) goto 500
c     do 100 i = 1, argc
c     call getarg (i, argv)
c     write (*,*) argv
c100 continue
c     if (.true.) stop
c500 continue
c     write (*,*) 'executed without arguments'
c     end

      integer*4 execl, iretval
      character*20 path, arg1, arg2
      path = '../example/hello'
      arg1 = 'hello'
      arg2 = 'world!'
      iretval = execl (path, path, arg1, arg2, 'bye now', %val(0))
      write (*,*) 'execl error:', iretval
      end
```

NOTES

A FORTRAN program may obtain the call line arguments as follows:

```
      character*20 argv(10)
      num_args = iargc()
      if (num_args.gt.10) num_args = 10
      do 100 i = 1, num_args
      call getarg (i, argv(i))
100   continue
```

An environment variable may be obtained as follows:

```
      character*40 home_dir
      call getenv ("HOME", home_dir)
```

Entries in *argv* or *envp* may be set up using %loc() as follows:

```
      argv (i) = %loc (arg1)
```

SEE ALSO

sh(1), alarm(2F), exit(2F), fcntl(2F), fork(2F), nice(2F), plock(2F), ptrace(2F), semop(2F), signal(2F), sigset(2F), times(2F), ulimit(2F), umask(2F), lockf(3C), a.out(4), environ(5).

DIAGNOSTICS

If *exec* returns to the calling process an error has occurred; the return value will be a negative value indicating the error.

NAME

exit, _exit - terminate process

SYNOPSIS

integer*4 exit, status
iretval = exit (status)
integer*4 _exit, status
iretval = _exit (status)

DESCRIPTION

exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits '0377'O) of *status* are made available to it [see *wait*(2F)].

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sysf/proc.i>**) to be used by *times*.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2F)] inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value [see *semop*(2F)], that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock*(2F)].

exit waits on outstanding, noncancelable asynchronous I/O requests; all cancelable and queued asynchronous I/O requests are deleted.

An accounting record is written on the accounting file if the system's accounting routine is enabled [see *acct* (2F)].

If the process ID, tty group ID, and process group ID of the calling process are equal, the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process.

A death of child signal is sent to the parent.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

EXAMPLE

```
program exit
integer*4 exit, status, iretval
status = 5
write (*,*) 'Type "echo $?" to check exit status of ', status
iretval = exit (status)
end
```

SEE ALSO

acct(2F), intro(2F), plock(2F), semop(2F), signal(2F), sigset(2F), wait(2F).

WARNING

See *WARNING* in *signal*(2F).

DIAGNOSTICS

None. There can be no return from an *exit* system call.

NAME

fcntl - file control

SYNOPSIS

 # include <sysf/fcntl.i>

integer*4 fcntl, fildes, cmd, arg
iretval = fcntl (fildes, cmd, arg)

integer*4 fcntl, fildes, cmd
record /flock/ arg2
iretval = fcntl (fildes, cmd, arg2)

DESCRIPTION

fcntl provides for control over open files. *fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The commands available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2F) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (0 or 1 as above).

F_GETFL Get *file* status flags.

F_SETFL Set *file* status flags to *arg*. Only certain flags can be set [see *fcntl*(5)].

F_GETLK Get the first lock which blocks the lock description given by the structure *arg2* defined by *record/flock/arg2*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the structure *arg2* defined by *record/flock/arg2* [see *fcntl*(5)]. The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return immediately with an error value of -1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

F_CHKFL (check legality of file flag changes)

F_SETAIOEMUL Set or clear flag controlling the emulation of asynchronous I/O for a device that does not support asynchronous I/O. If *arg* 1 (default), an *aread*(2F) or *awrite*(2F) call emulates an asynchronous I/O operation and returns synchronously. If *arg* is 0, an

asynchronous I/O operation on a file or device that does not support asynchronous I/O causes an error condition. F_SETAIOEMUL can be executed only by processes with realtime or superuser permissions.

F_GETAIOEMUL Get the asynchronous emulation mode for *fildes*. If the return value is 0, emulation is turned on; if the return value is 1, emulation is turned off.

F_SETBYBCACHE Set the bypass-buffer-cache flag for a regular, directory, or pipe file. Note that, while this command is issued against a *fildes*, it is actually set on the inode for the file, to avoid problems that might result if one file receives both asynchronous and synchronous I/O. If the return value is 0 (default), I/O operations on this *fildes* will use the buffer cache; if *arg* 1, I/O operations on this *fildes* will bypass the buffer cache. F_SETBYBCACHE cannot be used with asynchronous I/O requests against a character special device file, only with asynchronous I/O requests to regular files and directories. F_SETBYBCACHE can be executed only by processes with realtime or superuser permissions.

If the *arg* to F_SETBYBCACHE is 0, an *aread*(2F) or *awrite*(2F) operation against the *fildes* runs in emulation mode; a *read*(2F) or *write*(2F) operation runs normally.

If the *arg* to F_SETBYCACHE is 1, an *aread*(2F) or *awrite*(2F) operation runs in non-emulation mode and a *read*(2F) or *write*(2F) request issued against *fildes* will cause an EBUSY error condition.

F_GETBYBCACHE Get the value of the bypass-buffer-cache flag. If the return value is 0, I/O requests to this *fildes* will not bypass the buffer cache; if the return value is not 0, I/O requests to this *fildes* will bypass the buffer cache; the return value indicates the number of *fildes* for that inode that have bypass permissions set.

F_GETAIOREQ Determine buffer alignment requirements, maximum transfer size, and other specifications for asynchronous I/O operations. See *fcntl*(5) for more information.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), process id (*l_pid*), and RFS system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2F) system call.

When mandatory file and record locking is active on a file, [see *chmod*(2F)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

fcntl will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid open file descriptor.

[EINVAL] *cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.

[EINVAL] *cmd* is F_GETLK, F_SETLK, or SETLKW and *arg2* or the data it points to is not valid.

[EACCES] *cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.

[ENOLCK] *cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.

[EDEADLK] *cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.

[EFAULT] *cmd* is F_SETLK, *arg2* points outside the program address space.

[EINTR] A signal was caught during the *fcntl* system call.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

[EPERM] The process that tried to set F_SETBYBCACHE or F_SETAIOEMUL did not have real-time or superuser permissions.

EXAMPLE

```
      program fcntl
#     include <sysf/fcntl.i>
      integer*4 fcntl, fildes, cmd
      integer*4 open, iretval
      record /flock/ arg2

c  Open a file

      fildes = open ('../example/fcntl.F', O_RDWR)
      if (fildes .lt. 0) write (*,*) 'open error:', fildes

c  Set read lock for entire file

      arg2.l_type = F_RDLCK
      arg2.l_whence = 0
      arg2.l_start = 0
      arg2.l_len = 0
      cmd = F_SETLK
      iretval = fcntl (fildes, cmd, arg2)
      if (iretval .lt. 0) write (*,*) '1 fcntl error:', iretval

c  Use the locked file
c  Remove the lock

      arg2.l_type = F_UNLCK
      cmd = F_SETLK
      iretval = fcntl (fildes, cmd, arg2)
      if (iretval .lt. 0) write (*,*) '2 fcntl error:', iretval
      end
```

SEE ALSO

close(2F), creat(2F), dup(2F), exec(2F), fork(2F), open(2F), pipe(2F), fcntl(5).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Positive value
F_GETFL	Value of file flags
F_SETFL	Positve value
F_GETLK	Positive value
F_SETLK	Positive value
F_SETLKW	Positive value
F_CHKFL	
F_SETAIOEMUL	Positive value
F_GETAIOEMUL	Value of 0 if emulation flag is turned on, value of 1 (or higher) if emulation flag is turned off.
F_SETBYBCACHE	Positive value
F_GETBYBCACHE	Value of 0 if bypass-buffer-cache flag is turned off; value 1 (or higher) if bypass-buffer-cache flag is turned on.
F_GETAIOREQ	

Otherwise, a negative value indicating the error is returned.

WARNINGS

Because in the future the variable *iretval* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

Only I/O operations to devices with drivers that include an *aio*(D3X) routine and an *ioctl*(D2X) routine that defines AIOGETREQ (as defined in *sysf/aio.i*) can use the F_GETAIOREQ command.

NAME

fork - create a new process

SYNOPSIS

integer*4 fork
iretval = fork ()

DESCRIPTION

fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag [see *exec*(2F)]
signal handling settings (i.e., **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value [see *nice*(2F)]
all attached shared memory segments [see *shmop*(2F)]
process group ID
tty group ID [see *exit*(2F)]
current working directory
root directory
file mode creation mask [see *umask*(2F)]
file size limit [see *ulimit*(2F)]

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared [see *semop*(2F)].

Binary semaphores are not inherited by the child [see *bsget*(2F)].

Event identifiers are not inherited by the child [see *evget*(2F)].

Connected interrupts are not inherited by the child [see *cintrio*(7)].

No asynchronous I/O is inherited [see *aread*(2F) and *awrite*(2F)].

Process locks, text locks and data locks are not inherited by the child [see *plock*(2F)].

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

[EAGAIN] Total amount of system memory available when reading via raw I/O is temporarily insufficient.

EXAMPLE

```
        program fork
        integer*4 fork, pid, cpid, getpid

c   Create a new process

        pid = fork ()
        if (pid .lt. 0) then
          write (*,*) 'fork error:', pid
          stop
        else if (pid .eq. 0) then

c   Here if child process

          cpid = getpid ()
          write (*,9000) 'Child here.', 'My process id:', cpid
          stop

        else

c   Here if parent (original) process

          write (*,9000) 'Parent here.', 'Child''s process id:', pid
          stop

        endif
9000    format (' ', a12, 2x, a20, i6)
        end
```

SEE ALSO

exec(2F), nice(2F), plock(2F), ptrace(2F), semop(2F), shmop(2F), signal(2F), sigset(2F), times(2F), ulimit(2F), umask(2F), wait(2F).

DIAGNOSTICS

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a negative value indicating the error is returned to the parent process and no child process is created.

NAME

ftime - get time

SYNOPSIS

integer*4 ftime, tloc
iretval = ftime (tloc)

DESCRIPTION

ftime returns the value of time in seconds since 00:00:00 GMT, January 1, 1970. This function is equivalent to the system call *time*(2).

If *tloc* is not %val(0), the return value is also stored in the location to which *tloc* points.

EXAMPLE

```
      program ftime

c  Print current local date and time

      integer*4 ftime, tloc, iretval
      integer*4 secs, mins, hours
      integer*4 month, day, year, dayofweek
      integer*4 daypyear, daypmonth (12)
      integer*4 diff
      character*3 weekday (7), monthname (12), zone
      character*2 minc, secc
      character*7 tz
      data daypmonth /31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31/
      data weekday /"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"/
      data monthname /"Jan", "Feb", "Mar", "Apr", "May", "Jun",
&       "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"/

c  Get time zone information

      call getenv ("TZ", tz)
      zone = tz (1:3)
      diff = ichar (tz (4:4)) - 48
      diff = diff * 3600

c  Get time in seconds since 00:00:00 GMT, January 1, 1970.

      iretval = ftime (tloc)
      if (iretval .lt. 0) write (*,*) 'ftime error:', iretval
      tloc = tloc - diff

c  Compute current local time

      secs = mod (tloc, 60)
      tloc = tloc / 60
      mins = mod (tloc, 60)
      tloc = tloc / 60
      hours = mod (tloc, 24)
      tloc = tloc / 24
      minc (1:1) = char ((mins / 10) + 48)
      minc (2:2) = char (mod (mins, 10) + 48)
      secc (1:1) = char ((secs / 10) + 48)
      secc (2:2) = char (mod (secs, 10) + 48)
```

```
c  Get day of week

      dayofweek = mod ((tloc + 7340036), 7) + 1

c  Get year

      year = 70
100   continue
      daypyear = 365
      if (mod (year, 4) .eq. 0) daypyear = 366
      if (tloc .lt. daypyear) goto 110
      tloc = tloc - daypyear
      year = year + 1
      goto 100

110   continue
      year = year + 1900

c  Get month

      month = 1
      daypmonth (2) = 28
      if (daypyear .eq. 366) daypmonth (2) = 29
200   continue
      if (tloc .le. daypmonth (month) ) goto 210
      tloc = tloc - daypmonth (month)
      month = month + 1
      goto 200

210   continue
      day = tloc + 1

c  Your mission:  Account for daylight savings time

c  Print date and time

      write (*,9000) weekday (dayofweek), monthname (month),
&       day, hours, minc, secc, zone, year
9000  format (' ', a3, x, a3, i3, x, i2, ':', a2, ':', a2, x, a3, x, i4)
      end
```

NOTES

The FORTRAN System Interface Call *time* was renamed to *ftime* to distinguish it from the VAX/VMS System Subroutine *time*.

SEE ALSO

stime(2F).

WARNING

ftime fails and its actions are undefined if *tloc* points to an illegal address.

DIAGNOSTICS

Upon successful completion, *ftime* returns the value of time. Otherwise, a negative value indicating the error is returned.

NAME

getdents - read directory entries and put in a file system independent format

SYNOPSIS

 # include <sysf/dirent.i>

integer*4 getdents, fildes, nbyte
integer*1 buf (SIZE)
iretval = getdents (fildes, buf, nbyte)

DESCRIPTION

fildes is a file descriptor obtained from an *open*(2F) or *dup*(2F) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf* of *SIZE* bytes. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents*, the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3X) routine [for a description see *directory*(3X)], and should not be used for other purposes.

getdents will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid file descriptor open for reading.

[EFAULT] *Buf* points outside the allocated address space.

[EINVAL] *nbyte* is not large enough for one directory entry.

[ENOENT] The current file pointer for the directory is not located at a valid entry.

[ENOLINK] *fildes* points to a remote machine and the link to that machine is no longer active.

[ENOTDIR] *fildes* is not a directory.

[EIO] An I/O error occurred while accessing the file system.

EXAMPLE

```
      program getdents
#     include <sysf/fcntl.i>
#     define MAXNAMSIZ 20
#     define MAXBUFSIZ 128
      integer*4 getdents, fildes, nbyte
      integer*1 buf(MAXBUFSIZ)
      integer*4 bc, open, iretval
      integer*4 i, index, i_off, inode, offset, entlen
      integer*4 skip, close, tmp
      character*MAXNAMSIZ name

c  Open directory

      fildes = open ('.', O_RDONLY)
      if (fildes .lt. 0) write (*,*) 'open error:', fildes

c  Loop on all directory entries

10    continue
      nbyte = MAXBUFSIZ
      bc = getdents (fildes, buf, nbyte)
      if (bc .eq. 0) then   ! all done
       iretval = close (fildes)
       stop
      else if (bc .lt. 0) then
       write (*,*) 'getdents error:', bc
       stop
      endif
      index = 1
      i_off = 0

c  Loop on directory entries read into buffer

      do while (bc .gt. 0)
       index = index + i_off
       i_off = 0
       inode = 0
       offset = 0
       entlen = 0
```

```
c  Format variables according to dirent(4)

         do 100 i = 1, 4
          tmp = buf (index + i_off)
          inode = inode * 256 + (tmp .and. '0ff'x)
          tmp = buf (index + i_off + 4)
          offset = offset * 256 + (tmp .and. '0ff'x)
          tmp = buf (index + i_off + 8)
          if (i .lt. 3) entlen = entlen * 256 + (tmp .and. '0ff'x)
          i_off = i_off + 1
100       continue

         i_off = i_off + 6
         name = ''
         skip = 1

c  Format name of null-terminated entry

         do 200 i = 1, entlen - i_off
          if (buf (index + i_off) .eq. 0) skip = -1
          if (skip .lt. 0) goto 210
          if (i .le. MAXNAMSIZ) name (i:i) = char (buf (index + i_off) )
210       continue
         i_off = i_off + 1
200      continue

         write (*,9000) inode, offset, entlen, name
         bc = bc - i_off
        end do

c  Do entire directory

        goto 10
9000 format (' inode:', i6, '  offset:', i6, '  entlen:', i6,
&         '  name: ', a MAXNAMSIZ)
        end
```

SEE ALSO

directory(3X), dirent(4).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a negative value indicating the error is returned.

NAME

getinterval - get the current value for a process interval timer

SYNOPSIS

```
#    include <sysf/time.i>

     integer*4 getinterval, timerid
     record /itimerstruc/ value
     iretval = getinterval (timerid, value)
```

DESCRIPTION

The *getinterval* system call is used to obtain a time value for the process interval timer described by *timerid* and returned by *gettimerid*(2F).

The *value* argument is a pointer to an itimerstruc where the it_value represents the amount of time remaining before the timer is to expire and it_interval is the current interval value if one exists (or 0 if one does not).

EXAMPLE

```
     program getinter
#    include <sysf/time.i>
     integer*4 getinterval, timerid, iretval
     record /itimerstruc/ value

c  Use evget to get an event id
c  Use gettimerid to get a timer id for the event id
c  Use incinterval or absinterval to start interval timer
c  Delay some time and use getinterval to check time remaining

     iretval = getinterval (timerid, value)
     if (iretval .lt. 0) write (*,*) 'getinterval error:', iretval

c  Print time remaining

     write (*,9000) value.it_value.tv_sec, value.it_value.tv_nsec
9000 format (' seconds remaining:', i8, '  nano-sec remaining:', i8)
     end
```

ERROR CODES

If *getinterval* is not successful, a negative value indicating the error is returned.

[EFAULT] The *value* argument points outside of the allocated address space.

[EINVAL] The *timerid* argument does not correspond to a valid timer identifier returned by *gettimerid*(2F) or it has been previously released with a *reltimerid*(2F) call.

SEE ALSO

absinterval(2F), gettimerid(2F), incinterval(2F), reltimerid(2F), resabs(2F), resinc(2F), itimerstruc(4), timestruc(4).

NAME

getmsg - get next message off a stream

SYNOPSIS

#include <stropts.i>

int getmsg(fd, ctlptr, dataptr, flags)
integer*2 fd
record/strbuf/ctlptr
record/strbuf/dataptr
integer*4 flags

DESCRIPTION

getmsg retrieves the contents of a message [see *intro*(2F)] located at the *stream head* read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* are each records of *strbuf* structure which contains the following members:

```
integer*4 maxlen          /* maximum buffer length */
integer*4 len      /* length of data       */
integer*4 buf      /* ptr to buffer    */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the *stream head* read queue and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTICS*. If information is retrieved from a *priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *stream head* read queue. If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, *getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

getmsg fails if one or more of the following are true:

[EAGAIN] The O_NDELAY flag is set, and no messages are available.

[EBADF] *fd* is not a valid file descriptor open for reading.

[EBADMSG]	Queued message to be read is not valid for *getmsg*.
[EFAULT]	*ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.
[EINTR]	A signal was caught during the *getmsg* system call.
[EINVAL]	An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexor.
[ENOSTR]	A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2F), read(2F), poll(2F), putmsg(2F), write(2F).

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL|MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the remainder of the message.

NAME

getpeername - get name of connected peer

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*4 getpeername,s,namelen
struct/sockaddr/name
iretval = getpeername(s, name, namelen)

DESCRIPTION

getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

ERRORS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

The call succeeds unless:

[EBADF] The argument *s* is not a valid descriptor.

[ENOTSOCK] The argument *s* is a file, not a socket.

[ENOTCONN] The socket is not connected.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[EFAULT] The *name* parameter points to memory not in a valid part of the process address space.

EXAMPLE

see socket(2F)

SEE ALSO

accept(2F), bind(2F), socket(2F), getsockname(2F)

NAME

getpid, getpgrp, getppid - get process, process group, and parent process IDs

SYNOPSIS

integer*4 getpid
iretval = getpid ()

integer*4 getpgrp
iretval = getpgrp ()

integer*4 getppid
iretval = getppid ()

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

EXAMPLE

```
      program getpid
      integer*4 getpid, getpgrp, getppid
      integer*4 pid, pgrp, ppid

c   Get process id, group, and parent process id

      pid = getpid ()
      pgrp = getpgrp ()
      ppid = getppid ()

      write (*,*) pid, pgrp, ppid
      end
```

SEE ALSO

exec(2F), fork(2F), intro(2F), setpgrp(2F), signal(2F).

NAME

getpri - get scheduling priority

SYNOPSIS

integer*4 getpri, pid
iretval = getpri (pid)

DESCRIPTION

pid is the process ID of the target process, or zero if the calling process is the target. *getpri* returns the scheduling priority of the target process.

If *pid* is nonzero, the requesting process must have superuser or realtime privileges which are granted via the *setrt*(2F) system call.

getpri will fail if one or more of the following are true:

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EPERM] The *pid* is nonzero and the requesting process does not have realtime or superuser permissions.

EXAMPLE

```
      program getpri
      integer*4 getpri, pid
      integer*4 pri

c   Get my priority

      pid = 0
      pri = getpri (pid)
      if (pri .lt. 0) then
        write (*,*) 'getpri error:', pri
      else
        write (*,*) 'my priority is', pri
      endif
      end
```

SEE ALSO

setpri(1R), getpid(2F), setpri(2F), setrt(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer is returned indicating the priority of the target process. Otherwise, a negative value indicating the error is returned.

NAME

getsockname - gets socket name

SYNOPSIS

integer*4 getsockname,s,namelen
record /sockaddr/ name
iretval=getsockname(s, name, namelen)

DESCRIPTION

getsockname returns the current name for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return, it contains the actual size of the name returned (in bytes).

ERRORS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

EXAMPLE

see socket(2F)

SEE ALSO

bind(2F), socket(2F)

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*4 getsockopt,s,level,optname,optlen
character*SIZE optval
iretval= getsockopt(s, level, optname, optval, optlen)

integer*4 setsockopt,s,level,optname,optlen
character*SIZE optval
iretval= setsockopt(s, level, optname, optval, optlen)

DESCRIPTION

getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to set option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) is to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. A valid option value pointer must always be passed.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<sysf/socket.i>* contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (7).

Socket-level options take an *int* pointer for *optval*. For *setsockopt*, the parameter must point to a non-zero value to enable a boolean option, or a zero value if the option is to be disabled. For *getsockopt*, the value returned in **optval == optname* if the option is enabled, or 0 if it is disabled (for boolean options).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keeping connections alive

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2F) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and calls on the socket will return -1 with *errno* set to ETIMEDOUT.

ERRORS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.

EXAMPLE

see socket(2F)

SEE ALSO

ioctl(2F), socket(2F), getprotoent(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

Options must be set/cleared via *setsockopt* prior to issuing a *connect* or *listen* on a SOCK_STREAM socket.

NAME

gettimer - get the current value for a system-wide realtime timer

SYNOPSIS

```
#    include <sysf/time.i>

     integer*4 gettimer, timer_type
     record /timestruc/ tp
     iretval = gettimer (timer_type, tp)
```

DESCRIPTION

gettimer returns the current value of the system-wide realtime timer specified by the *timer_type* argument.

The *timer_type* argument identifies the system-wide realtime timer used with this system call. TIMEOFDAY is a valid timer_type and corresponds to the system time-of-day clock representing the current time in seconds and nanoseconds since January 1, 1970. This timer is ascending in nature and is updated by the system at the frequency of the 64 Hz system clock.

tp is a pointer to a timestruc structure where the timer value is to be placed.

EXAMPLE

```
     program gettimer
#    include <sysf/time.i>
     integer*4 gettimer, timer_type
     integer*4 iretval, secs, mins, hours, tmp

     record /timestruc/ tp

     iretval = gettimer (TIMEOFDAY, tp)
     if (iretval .lt. 0) write (*,*) 'gettimer error:', iretval

     secs = mod (tp.tv_sec, 60)
     tmp = tp.tv_sec / 60
     mins = mod (tmp, 60)
     tmp = tmp / 60
     hours = mod (tmp, 24)
     write (*,9000) tp.tv_sec, tp.tv_nsec
     write (*,9001) hours, mins, secs
9000 format (' time since 1/1/1970: ',i10,' seconds,'i10,' nano-seconds')
9001 format (' time:  ',i2,':',i2,':',i2,' GMT')
     end
```

ERROR CODES

If successful, *gettimer* returns a value of 0. Otherwise a negative value indicating the error is returned.

[EFAULT] *tp* points outside the allocated address space of the process.

[EINVAL] The *timer_type* argument does not specify a valid system-wide realtime timer type.

SEE ALSO

restimer(2F), settimer(2F), timestruc(4).

NAME

gettimerid - get a unique identifier for a process interval timer

SYNOPSIS

```
#    include <sysf/time.i>
#    include <sysf/evt.i>

     integer*4 gettimerid, timer_type, event_type, eid
     iretval = gettimerid (timer_type, event_type, eid)
```

DESCRIPTION

The *gettimerid* system call allocates a process interval timer to the calling realtime process and assigns it a unique identifier (ID). This timer ID is required by all system calls that manipulate process interval timers.

The *timer_type* argument identifies the system-wide realtime timer associated with this timer identifier (ID). TIMEOFDAY is a valid *timer_type* and corresponds to the system time-of-day clock representing the current time in seconds and nanoseconds since January 1, 1970. This timer is ascending in nature and is updated by the system at the frequency of the 64 Hz system clock.

event_type identifies the type of event mechanism used to deliver an event when a process interval timer expiration occurs. The high-performance event notification facility is used by selecting MODCOMP_EVENTS as the *event_type*.

The *gettimerid* system call is used in conjunction with the event system calls. It must be preceeded by a call to *evget*(2F) which returns a unique event identifier (*eid*) for the process. When a timer previously activated by an *absinterval*(2F) or *incinterval*(2F) system call expires, the timer expiration event is sent to the process via the *eid* value. The process uses the *evrcv*(2F) or the *evrcvl*(2F) system calls to receive a timer signal or event.

EXAMPLE

See absinterval(2F) for an example.

NOTES

Interval timers are not inherited by a child process across a *fork*(2F) or an *exec*(2F).

ERROR CODES

If successful, *gettimerid* returns a value equal to that of the timer ID. Otherwise a negative value indicating the error is returned.

[EINVAL] The *timer_type* or *event_type* arguments do not specify a valid timer type or event type. The *eid* specified is not valid for the calling process.

[ENOSPC] The process interval timer cannot be allocated to this process. It exceeds the number of allowable timers for a single process.

[EPERM] The effective user ID does not have realtime privileges.

SEE ALSO

absinterval(2F), getinterval(2F), incinterval(2F), reltimerid(2F), resabs(2F), resinc(2F), restimer(2F).

NAME

getuid, geteuid, getgid, getegid - get real user, effective user, real group, and effective group IDs

SYNOPSIS

integer*4 getuid
iretval = getuid ()

integer*4 geteuid
iretval = geteuid ()

integer*4 getgid
iretval = getgid ()

integer*4 getegid
iretval = getegid ()

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

EXAMPLE

```
      program getuid
      integer*4 getuid, geteuid, getgid, getegid
      integer*4 uid, euid, gid, egid

c  Get and print user id, effective user id, group id, effective group id

      uid = getuid ()
      euid = geteuid ()
      gid = getgid ()
      egid = getegid ()

      write (*,*) uid, euid, gid, egid
      end
```

SEE ALSO

intro(2F), setuid(2F).

NAME

ioctl - control device

SYNOPSIS

integer*4 ioctl, fildes, request, arg
iretval = ioctl (fildes, request, arg)

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2F) and *write*(2F) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

fildes is an open file descriptor that refers to a device. *request* selects the control function to be performed and will depend on the device being addressed. *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl will fail for any type of file if one or more of the following are true:

[EBADF] *fildes* is not a valid open file descriptor.

[ENOTTY] *fildes* is not associated with a device driver that accepts control functions.

[EINTR] A signal was caught during the *ioctl* system call.

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true:

[EFAULT] *request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

[EINVAL] *request* or *arg* is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular subdevice.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(7).

EXAMPLE

See cisema(2F) for an example.

SEE ALSO

streamio(7), termio(7).

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a negative value indicating the error is returned.

NAME

kill - send a signal to a process or a group of processes

SYNOPSIS

integer*4 kill, pid, sig
iretval = kill (pid, sig)

DESCRIPTION

kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2F), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes [see *intro*(2F)] and will be referred to below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is -1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is -1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not -1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL] *sig* is not a valid signal number.

[EINVAL] *sig* is SIGKILL and *pid* is 1 (proc1).

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EPERM] The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process.

EXAMPLE

```
      program kill
#     include <sysf/signal.i>
      integer*4 kill, pid, sig
      integer*4 iretval

c  Kill all my processes (in other words, bye-bye)

      pid = 0
      sig = SIGKILL
      iretval = kill (pid, sig)
      if (iretval .lt. 0) then
        write (*,*) 'kill error:', iretval
      else
        write (*,*) 'should not have got this message'
      endif
      end
```

SEE ALSO

kill(1), getpid(2F), setpgrp(2F), signal(2F), sigset(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

link - link to a file

SYNOPSIS

integer*4 link
character*SIZE path1, path2
iretval = link (path1, path2)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path1* points to a path name naming an existing file. *path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

link will fail and no link will be created if one or more of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *path1* does not exist.
[EEXIST]	The link named by *path2* exists.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[ENOENT]	*path2* points to a null path name.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to a file would be exceeded.
[EINTR]	A signal was caught during the *link* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program link
      integer*4 link, unlink
      character*40 path1, path2
      integer*4 iretval

c   Rename a file by making a new directory entry for the file, then
c   deleting the old directory entry.

      path1 = '../example/link.F'
      path2 = '../example/newlink.F'

      iretval = link (path1, path2)
      if (iretval .lt. 0) write (*,*) 'link error:', iretval

      iretval = unlink (path1)
      if (iretval .lt. 0) write (*,*) 'unlink error:', iretval
      end
```

SEE ALSO

ln(1), unlink(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

listen - listens for connections on a socket

SYNOPSIS

integer*4 listen, s, backlog
iretval = listen(s,backlog)

DESCRIPTION

To accept connections, a socket is first created with *socket*(2F), a backlog for incoming connections is specified with *listen* and then the connections are accepted with *accept*(2F). The listen call applies only to sockets of type SOCK_STREAM.

The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, the client will receive an error with an indication of ECONNREFUSED.

ERRORS

Upon successful completion, the value zero is returned. Otherwise, a negative value indicating the error is returned.

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the *listen* operation.

EXAMPLE

see socket(2F)

SEE ALSO

accept(2F), connect(2F), socket(2F)

BUGS

The backlog is currently limited (silently) to 5.

NAME

lseek - move read/write file pointer

SYNOPSIS

integer*4 lseek, fildes, offset, whence
iretval = lseek (fildes, offset, whence)

DESCRIPTION

fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* will return the file pointer even if it is negative.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *fildes* is not an open file descriptor.

[ESPIPE] *fildes* is associated with a pipe or fifo.

[EINVAL and SIGSYS signal]
whence is not 0, 1, or 2.

[EINVAL] *fildes* is not a remote file descriptor, and the resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

EXAMPLE

```
      program lseek
#     include <sysf/fcntl.i>
      integer*4 lseek, fildes, offset, whence, fp
      integer*4 open

      fildes = open ('../example/lseek.F', O_RDONLY)
      if (fildes .lt. 0) write (*,*) 'open error:', fildes

c  Point to beginning of file

      whence = 0
      offset = 0
      fp = lseek (fildes, offset, whence)
      if (fp .lt. 0) write (*,*) 'lseek error:', fp
      write (*,9000) fp

c  Point to end of file

      whence = 2
      offset = 0
      fp = lseek (fildes, offset, whence)
      if (fp .lt. 0) write (*,*) 'lseek error:', fp
      write (*,9000) fp
9000  format (' current file pointer =', i10)
      end
```

SEE ALSO

creat(2F), dup(2F), fcntl(2F), open(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a negative value indicating the error is returned.

NAME

memctl - control write/execute attributes of memory.

SYNOPSIS

#include <sysf/m88kbcs.i> (on 88k machine only)

integer*4 memctl,start,length,mode
iretval = memctl(start,length,mode)

DESCRIPTION

memctl can change the access mode of a part of memory. It recognizes these modes:

MCT_TEXT	1	Readable and executable
MCT_TEXT	2	Readable and writable
MCT_TEXT	3	Readable only

The text section of a process is initially in mode 1. The data, bss, and stack sections are initially in mode 2. A process should only access its memory in the ways supported by the current mode, or unpredictable problems will result. The main purpose of this facility is to allow code to be written in the data section of a process and then be executed. This will not work correctly unless *memctl*() is called to make the relevant part of the data section executable after it has been modified and before it has been executed. If the memory is shared by several processes, all the processes must follow this procedure. *start* and *length* must be specified in bytes, and must be multiples of 4k.

RETURN VALUE

If the call fails, -1 will be returned, and the global errno set to reflect the error. Otherwise 0 will be returned.

ERRORS

[EINVAL] The *mode* is invalid, or the *start* or *length* are not multiples of 4k.

[EFAULT] The region of memory specified by the *start* and *length* parameters is not valid for the process.

NOTES

Currently on 88k machine only.

NAME

mkdir - make a directory

SYNOPSIS

integer*4 mkdir, mode
character*SIZE path
iretval = mkdir (path, mode)

DESCRIPTION

SIZE can be any number between and including 1 through 128. The routine *mkdir* creates a new directory with the name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument is modified by the process's mode mask [see *umask*(2F)].

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID. The newly created directory is empty with the possible exception of entries for "." and "..". *mkdir* will fail and no directory will be created if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[EACCES]	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to the parent directory would be exceeded.
[EIO]	An I/O error has occurred while accessing the file system.

EXAMPLE

```
      program mkdir
      integer*4 mkdir, mode
      character*40 path
      integer*4 iretval

c  Make directory "tstdir" in current directory

      path = 'tstdir'
      mode = '777'o
      iretval = mkdir (path, mode)
      if (iretval .lt. 0) write (*,*) 'mkdir error:', iretval
      end
```

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

mknod - make a directory, or a special or ordinary file

SYNOPSIS

integer*4 mknod, mode, dev
character*SIZE path
iretval = mknod (path, mode, dev)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *mknod* creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*. The value of *mode* is interpreted as follows:

'0170000'O file type; one of the following:

'0010000'O fifo special
'0020000'O character special
'0040000'O directory
'0060000'O block special
'0100000'O or '0000000'O ordinary file

'0004000'O set user ID on execution
'00020#0'O set group ID on execution if # is 7, 5, 3, or 1
enable mandatory file/record locking if # is 6, 4, 2, or 0
'0001000'O save text image after execution
'0000777'O access permissions; constructed from the following:

'0000400'O read by owner
'0000200'O write by owner
'0000100'O execute (search on directory) by owner
'0000070'O read, write, execute (search) by group
'0000007'O read, write, execute (search) by others

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared [see *umask*(2F)]. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

mknod may be invoked only by the superuser for file types other than FIFO special; *mknod* will fail and the new file will not be created if one or more of the following are true:

[EPERM] The effective user ID of the process is not superuser.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] A component of the path prefix does not exist.

[EROFS] The directory in which the file is to be created is located on a read-only file system.

[EEXIST] The named file exists.

[EFAULT] *path* points outside the allocated address space of the process.

[ENOSPC] No space is available.

[EINTR] A signal was caught during the *mknod* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program mknod
#     include <sysf/fcntl.i>
      integer*4 mknod, mode, dev
      character*20 path, buf
      integer*4 open, fd, write, read
      integer*4 iretval, i, fork
      data buf /'Message:'/

c  Create a named pipe

      path = './fifo'
      mode = '10777'o   ! fifo + full access permissions
      dev = 0
      iretval = mknod (path, mode, dev)
      if (iretval .lt. 0) write (*,*) 'mknod error:', iretval

c  Make a child process

      iretval = fork ()
      if (iretval .eq. 0) goto 3000

c  Write to the pipe

      fd = open ('fifo', O_WRONLY)
      if (fd .lt. 0) write (*,*) 'open error:', fd

      do 100 i = 1, 20
      buf (19:19) = char ((i/10) + 48)
      buf (20:20) = char (mod (i, 10) + 48)
      iretval = write (fd, buf, 20)
      if (iretval .lt. 0) write (*,*) 'write error:', iretval
100   continue
      if (.true.) stop

c  Here if child
c  Open pipe, read from it, and write to stdout

3000 continue
      fd = open ('fifo', O_RDONLY)
      if (fd .lt. 0) write (*,*) 'child open error:', fd

      do 3100 i = 1, 20
      buf = ' '
      iretval = read (fd, buf, 20)
      if (iretval .ge. 0) write (*,9000) buf
3100 continue
9000 format (' ', a20)

      end
```

SEE ALSO

mkdir(1), chmod(2F), exec(2F), umask(2F), fs(4).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

WARNING

If *mknod* is used to create a device in a remote directory (Remote File Sharing), the major and minor device numbers are interpreted by the server.

NAME

msgctl - message control operations

SYNOPSIS

 # include <sysf/types.i>
 # include <sysf/ipc.i>
 # include <sysf/msg.i>

integer*4 msgctl, msqid, cmd
record /msqid_ds/ buf
iretval = msgctl (msqid, cmd, buf)

DESCRIPTION

msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2F). {READ}

IPC_SET Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

msg_perm.uid
msg_perm.gid
msg_perm.mode !only low 9 bits
msg_qbytes

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

msgctl will fail if one or more of the following are true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EINVAL] *cmd* is not a valid command.

[EACCES] *cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process [see *intro*(2F)].

[EPERM] *cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is not equal to that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

[EPERM] *cmd* is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process is not equal to that of super user.

[EFAULT] *buf* points to an illegal address.

EXAMPLE

See msgop(2F) for an example.

SEE ALSO

intro(2F), msgget(2F), msgop(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

msgget - get message queue

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/ipc.i>
#    include <sysf/msg.i>

     integer*4 msgget, key, msgflg
     iretval = msgget (key, msgflg)
```

DESCRIPTION

msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure [see *intro*(2F)] are created for *key* if one of the following are true:

key is equal to **IPC_PRIVATE**.

key does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

Msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.

Msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.

Msg_ctime is set equal to the current time.

Msg_qbytes is set equal to the system limit.

msgget will fail if one or more of the following are true:

[EACCES] A message queue identifier exists for *key*, but operation permission [see *intro*(2F)] as specified by the low-order 9 bits of *msgflg* would not be granted.

[ENOENT] A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

[EEXIST] A message queue identifier exists for *key* but ((*msgflg* & **IPC_CREAT**) & (*msgflg* & **IPC_EXCL**)) is "true".

EXAMPLE

See msgop(2F) for an example.

SEE ALSO

intro(2F), msgctl(2F), msgop(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a negative value indicating the error is returned.

NAME

msgop: msgrcv, msgsnd - message operations

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/ipc.i>
#    include <sysf/msg.i>

     integer*4 msgsnd, msqid, msgsz, msgflg, msgtyp
     structure /msgbuff/
        integer*4 mtype
        character*SIZE mtext
     end structure

     record /msgbuff/ msgp
     iretval = msgsnd (msqid, msgp, msgsz, msgflg)
     iretval = msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
```

DESCRIPTION

msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *msgp* points to a structure containing the message. This structure is composed of the following members:

```
integer*4 mtype          !message type
character*SIZE  mtext  !message text
```

mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *mtext* is any text of length *msgsz* bytes. *msgsz* can range from 0 to a system-imposed maximum. *SIZE* must be greater or equal to *msgsz.*

msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes** [see *intro*(2F)].

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message will not be sent and the calling process will return immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

msqid is removed from the system [see *msgctl*(2F)]. When this occurs, *iretval* is set equal to EIDRM.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2F).

msgsnd will fail and no message will be sent if one or more of the following are true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process [see *intro*(2F)].

[EINVAL] *mtype* is less than 1.

[EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".

[EINVAL] *msgsz* is less than zero or greater than the system-imposed limit.

[EFAULT] *msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see intro (2F)].

msg_qnum is incremented by 1.

msg_lspid is set equal to the process ID of the calling process.

msg_stime is set equal to the current time.

msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
integer*4 mtype          !message type
character*SIZE mtext !message text
```

mtype is the received message's type as specified by the sending process. *mtext* is the text of the message. *msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process. *SIZE* may be greater or equal to *msgsz*.

msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

msqid is removed from the system. When this occurs, *iretval* is set equal to EIDRM.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2F).

msgrcv will fail and no message will be received if one or more of the following are true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *msgsz* is less than 0.

[E2BIG] mtext is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".

[ENOMSG] The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true".

[EFAULT] *msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see intro (2F)].

msg_qnum is decremented by 1.

msg_lrpid is set equal to the process ID of the calling process.

msg_rtime is set equal to the current time.

EXAMPLE

```
      program msgop
#     include <sysf/types.i>
#     include <sysf/ipc.i>
#     include <sysf/msg.i>
      integer*4 msgctl, msqid, cmd
      integer*4 msgget, iretval
      integer*4 msgsnd, msgrcv, msgflg, msgtyp, msgsz

      structure /msgbuff/
       integer*4 mtype
       character*40 mtext
      end structure

      record /msqid_ds/ buf
      record /msgbuff/ msgpsnd, msgprcv

c  Get msg queue id with full permissions

      msqid = msgget (101, ('777'o .or. IPC_CREAT))
      if (msqid .lt. 0) write (*,*) 'msgget error:', msqid

c  Send a message to myself

      msgpsnd.mtype = 1234
      msgpsnd.mtext = 'hello to myself'
      msgsz = 40
      msgflg = 0
      iretval = msgsnd (msqid, msgpsnd, msgsz, msgflg)
      if (iretval .lt. 0) write (*,*) 'msgsnd error:', iretval

c  Message now on queue, receive message

      msgprcv.mtype = 0
      msgprcv.mtext = 'testing 1, 2, 3 '
      msgtyp = 0
      iretval = msgrcv (msqid, msgprcv, msgsz, msgtyp, msgflg)
      if (iretval .lt. 0) write (*,*) 'msgrcv error:', iretval
```

```
c  Print message

      write (*,9000) msgprcv.mtype, msgprcv.mtext
9000 format (' Message received. Type:', i6, ' Text: ', a40)

c  Place contents of msg structure in buf and print some info

      cmd = IPC_STAT
      iretval = msgctl (msqid, cmd, buf)
      if (iretval .lt. 0) write (*,*) 'msgctl error:', iretval
      write (*,9001) buf.msg_lspid, buf.msg_lrpid, buf.msg_rtime
9001 format (' pid of last msgsnd:', i6,/
&          ' pid of last msgrcv:', i6,/
&          ' time stamp of msgrcv:', i10)

c  Remove message queue

      iretval = msgctl (msqid, IPC_RMID, buf)
      if (iretval .lt. 0) write (*,*) 'msgctl error on remove:', iretval
      end
```

SEE ALSO

intro(2F), msgctl(2F), msgget(2F), signal(2F).

DIAGNOSTICS

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of EINTR is returned to the calling process. If they return due to removal of *msqid* from the system, a value of EIDRM is returned.

Upon successful completion, the return value is as follows:

msgsnd returns a value of 0.

msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a negative value indicating the error is returned.

NAME

nice - change priority of a process

SYNOPSIS

integer*4 nice, incr
iretval = nice (incr)

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

[EPERM] *nice* will fail and not change the nice value if *incr* is negative or greater than 39 and the effective user ID of the calling process is not super-user.

EXAMPLE

```
        program nice
        integer*4 nice, incr, value

c   Determine current nice value

        incr = 0
        value = nice (incr)
        if (value .lt. 0) then
          write (*,*) '1st nice error:', value
        else
          write (*,*) 'nice value was:', value + 20
        endif

c   Decrease priority of this process

        value = nice (5)
        if (value .lt. 0) then
          write (*,*) '2nd nice error:', value
        else
          write (*,*) 'nice value now:', value + 20
        endif

        end
```

SEE ALSO

nice(1), exec(2F).

DIAGNOSTICS

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a negative value indicating the error is returned.

NAME

open - open for reading or writing

SYNOPSIS

```
#    include <sysf/fcntl.i>
     integer*4 open, oflag, mode
     character*SIZE path
     iretval = open (path, oflag, [,mode])
```

DESCRIPTION

path points to a path name naming a file. *SIZE* can be any number between and including 1 through 128. *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS [see *intro*(2F)] files, *oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes [see *read*(2F) and *write*(2F)].

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2F) will wait for both the file data and file status to be physically updated.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows [see *creat*(2F)]:

All bits set in the file mode creation mask of the process are cleared [see *umask*(2F)].

The "save text image after execution bit" of the mode is cleared [see *chmod*(2F)].

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY or-ed with either O_RDONLY, O_WRONLY or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of O_NDELAY affects the operation of STREAMS drivers and

certain system calls [see *read*(2F), *getmsg*(2F), *putmsg*(2F) and *write*(2F)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2F).

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls [see *fcntl*(2F)].

The named file is opened unless one or more of the following are true:

[EACCES] A component of the path prefix denies search permission.

[EACCES] *oflag* permission is denied for the named file.

[EAGAIN] The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod* (2F)].

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[EFAULT] *path* points outside the allocated address space of the process.

[EINTR] A signal was caught during the *open* system call.

[EIO] A hangup or error occurred during a STREAMS *open*.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EMFILE] NOFILES file descriptors are currently open.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

[ENFILE] The system file table is full.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOLINK] *path* points to a remote machine, and the link to that machine is no longer active.

[ENOMEM] The system is unable to allocate a send descriptor.

[ENOSPC] O_CREAT and O_EXCL are set, and the file system is out of inodes.

[ENOSR] Unable to allocate a *stream*.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[ENXIO] O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[ENXIO] A STREAMS module or driver open routine failed.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

EXAMPLE

```
      program open
#     include <sysf/errno.i>
#     include <sysf/fcntl.i>

      integer*4 open, oflag, mode
      character*80 path

      integer*4 fildes

c  open file for read/write, create if necessary, accessible to anyone

      path = 'tst.x'
      oflag = O_RDWR .or. O_CREAT
      mode = '0777'o
      fildes = open (path, oflag, mode)
      if (fildes .lt. 0) then
            write (*,*) 'open error ', fildes
            stop
      endif
c  "fildes" is the file descriptor of the opened file
c     ...
      end
```

SEE ALSO

chmod(2F), close(2F), creat(2F), dup(2F), fcntl(2F), intro(2F), lseek(2F), read(2F), getmsg(2F), putmsg(2F), umask(2F), write(2F).

DIAGNOSTICS

Upon successful completion, the file descriptor is returned. Otherwise, a negative value indicating the error is returned.

NAME

pathconf,fpathconf - get configurable pathname variables

SYNOPSIS

#include <unistd.i>

integer*4 pathconf,name
character*SIZE path
iretval = pathconf(path,name)

integer*4 fpathconf,filedes,name
iretval = fpathconf(filedes,name)

DESCRIPTION

SIZE can be any number 1 through 128. *pathconf* and *fpathconf* provide a method for an application to determine the current value of a configurable limit or option that is associated with a file or directory.

For *pathconf, path* points to a pathname of a file or directory. For *fpathconf, filedes* is an open file descriptor. *name* is the variable to be queried relative to the file or directory. The following variables can be queried.

_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_CHOWN_RESTRICTED
_PC_NO_TRUNC
_PC_BLKSIZE
_PC_VDISABLE

RETURN VALUE

If *name* is not a valid variable name, or if the variable cannot be associated with the specified file or directory, or if the process does not have permission to query the file specified by *path,* or *path* does not exist, *pathconf* will return -1, and *errno* will be set to indicate the error. If the named variable is not defined on the system, a value of -1 will be returned and *errno* will remain unchanged.

Otherwise, *pathconf* and *fpathconf* will return the current value associated with the variable for the file or directory.

ERRORS

pathconf and *fpathconf* will fail if one or more of the following are true:

[ENOTIDR]	A component of the path prefix is not a directory.
[EPERM]	A pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*path* points to an invalid address.

fpathconf will also fail if the following condition occurs:

[EBADF]	*filedes* is not a valid open file descriptor.

[EINVAL] Name is not equal to one of the allowable values above or name cannot be associated with the specified file or directory.

SEE ALSO

sysconf(3P)

NOTES

Currently on 88k machine only.

NAME

pause - suspend process until signal

SYNOPSIS

integer*4 pause
iretval = pause ()

DESCRIPTION

pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function [see *signal*(2F)], the calling process resumes execution from the point of suspension; with a return value of EINTR.

EXAMPLE

```
      program pause
      integer*4 pause, iretval

c  Suspend until some signal occurs, after which this process
c  is terminated.

      iretval = pause ()
      write (*,*) 'should not have gotten here', iretval
      end
```

SEE ALSO

alarm(2F), kill(2F), signal(2F), sigpause(2F), wait(2F).

NAME

pipe - create an interprocess channel

SYNOPSIS

integer*4 pipe, fildes(0:1)
iretval = pipe (fildes)

DESCRIPTION

pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*(0) and *fildes*(1). *fildes*(0) is opened for reading and *fildes*(1) is opened for writing.

Up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*(0) accesses the data written to *fildes*(1) on a first-in-first-out (FIFO) basis.

pipe will fail if:

[EMFILE]	NOFILES file descriptors are currently open.
[ENFILE]	The system file table is full.

EXAMPLE

```
        program pipe
        integer*4 pipe, fildes (0:1), dup, close, fork, iretval, i
        character*40 line

c   Create a pipe for interprocess communication

        iretval = pipe (fildes)

c   Make a new process

        iretval = fork ()
        if (iretval .gt. 0) goto 200

c   Child process
c
c   Read from pipe, write to standard output

100     continue
        iretval = close (0)
        iretval = dup (fildes (0) )
        iretval = close (fildes (0) )
        iretval = close (fildes (1) )

        do 150 i = 1, 15
        read (*,9000) line
        write (*,9001) line
150     continue
        stop

9000 format (a40)
9001 format (20x, a40)
```

```
c  Parent process
c
c  Write to pipe

200  continue
     iretval = close (1)
     iretval = dup (fildes (1) )
     iretval = close (fildes (0) )
     iretval = close (fildes (1) )

     do 250 i = 1, 15
     write (*,9002) 'This is line', i
250  continue
9002 format (' ', a15, i4)
     end
```

SEE ALSO

sh(1), read(2F), write(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

plock - lock process, text, or data in memory

SYNOPSIS

```
#    include <sysf/lock.i>

     integer*4 plock, op
     iretval = plock (op)
```

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be superuser or have realtime privileges to use this call. *op* specifies the following:

PROCLOCK lock text and data segments into memory (process lock)

TXTLOCK lock text segment into memory (text lock)

DATLOCK lock data segment into memory (data lock)

UNLOCK remove locks

plock will fail and not perform the requested operation if one or more of the following are true:

[EAGAIN] Not enough physical memory free at this time.

[EINVAL] *op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

[EINVAL] *op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.

[EINVAL] *op* is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.

[EINVAL] *op* is equal to **UNLOCK** and no type of lock exists on the calling process.

[EPERM] The effective user ID of the calling process is not superuser, or does not have realtime privileges.

EXAMPLE

```
      program plock
#     include <sysf/lock.i>
      integer*4 plock, op
      integer*4 iretval

c  Lock process in memory

      op = PROCLOCK
      iretval = plock (op)
      if (iretval .lt. 0) write (*,*) 'plock error:', iretval

c  ...

c  Remove locks

      iretval = plock (UNLOCK)
      if (iretval .lt. 0) write (*,*) '2nd plock error:', iretval
      end
```

SEE ALSO

exec(2F), exit(2F), fork(2F), resident(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a negative value indicating the error is returned.

NAME

prealloc - preallocate contiguous file space

SYNOPSIS

integer*4 prealloc, fd, hint, nbytes, flags
iretval = prealloc (fd, hint, nbytes, flags)

DESCRIPTION

The *prealloc* system call allocates contiguous file space to the file referenced by *fd*. *fd* is the file descriptor of a file open for writing. *nbytes* is the size in bytes of the contiguous file extent that is desired. The extent will logically start immediately after the last existing extent of the file. If *hint* is nonzero, it is used to determine the physical byte offset in the file system where the extent should be located. If the extent cannot be placed there, the system determines the physical location of the extent.

If the F5NOGROW bit is set in *flags*, then subsequent *write*(2F) system calls, that occur beyond the end of the file, fail and return an error of EFBIG to the user. If the F5NOGROW bit is reset, writes beyond the end of the file will automatically allocate space, but in a noncontiguous fashion.

If the F5SHRINK bit is set in *flags*, then subsequent *creat*(2F) and *open*(2F) with the O_TRUNC option or *trunc*(2F) system calls cause physical space to be freed from the file as well as logical space. If the F5SHRINK bit is not set, physical space will remain across such truncate operations.

If the F5NOZERO is set in *flags* and the user has realtime or superuser privileges, then the allocated space is not zeroed. Otherwise the space is zeroed. If not zeroed, the contents of the extent are undefined. If the space is zeroed, synchronous write operations are used so that the data and inode will both be on the disk when the *prealloc* call finishes.

If the F5NOCHANGE is set in *flags*, then the value of the F5NOGROW and F5SHRINK bits remains the same as they were before the call to *prealloc*.

prealloc will fail if one or more of the following are true:

[EBADF] *fd* is not a valid file descriptor open for writing.

[EEXIST] The request overlaps an area of the file that already had space allocated to it, or an attempt was made to preallocate space after noncontiguous growth had occurred.

[EFBIG] An attempt was made to *prealloc* a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2F)), or the number of extents in the file exceeds NF5EXT.

[EINVAL] *fd* does not describe a file on a F5 file system.

[ENOSPC] There is not enough contiguous space to satisfy the request.

[ENXIO] *fd* does not describe a regular file.

EXAMPLE

```
          program prealloc
#         include <sysf/fcntl.i>
          integer*4 prealloc, fd, hint, nbytes, flags
          integer*4 open, close, iretval

c  Open a file assuming have an F5 file system

          fd = open ('tmp', O_RDWR .or. O_CREAT, '777'o)
          if (fd .lt. 0) write (*,*) 'open error:', fd

c  Make room for some contiguous space

          hint = 0
          nbytes = 4096
          flags = F5NOCHANGE
          iretval = prealloc (fd, hint, nbytes, flags)
          if (iretval .lt. 0) write (*,*) 'prealloc error:', iretval
          iretval = close (fd)
          end
```

SEE ALSO

close(2F), creat(2F), dup(2F), exec(2F), fcntl(2F), fork(2F), open(2F), pipe(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

putmsg - send a message on a stream

SYNOPSIS

#include <sysf/stropts.i>

integer*2 fd
record /strbuf/ ctlptr, dataptr
integer*4 flags
irtval= putmsg (fd, ctlptr, dataptr, flags)

DESCRIPTION

putmsg creates a message [see *intro*(2F)] from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* are each records of a *strbuf* structure which contains the following members:

```
integer*4 maxlen        ! not used
integer*4 len           ! length of data
integer*4 buf           ! ptr to buffer
```

ctlptr is the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* [see *getmsg*(2F)]. In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

putmsg also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent.

putmsg fails if one or more of the following are true:

[EAGAIN] A non-priority message was specified, the O_NDELAY flag is set and the *stream* write queue is full due to internal flow control conditions.

[EAGAIN] Buffers could not be allocated for the message that was to be created.

[EBADF] *fd* is not a valid file descriptor open for writing.

[EFAULT] *ctlptr* or *dataptr* points outside the allocated address space.

[EINTR] A signal was caught during the *putmsg* system call.

[EINVAL] An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.

[EINVAL] The *stream* referenced by *fd* is linked below a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

[ENXIO] A hangup condition was generated downstream for the specified *stream*.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A *putmsg* also fails if a STREAMS error message had been processed by the *stream* head before the call to *putmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2F), read(2F), getmsg(2F), poll(2F), write(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

read - read from file

SYNOPSIS

integer*4 read, fildes, nbyte
integer*1 buf(SIZE)
iretval = read (fildes, buf, nbyte)

integer*4 read, fildes, nbyte
character*SIZE bufc
iretval = read (fildes, bufc, nbyte)

DESCRIPTION

SIZE is the maximum number of bytes in the buffer. *fildes* is a file descriptor obtained from a *creat*(2F), *open*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call.

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line [see *ioctl*(2F) and *termio*(7)], or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A *read* from a STREAMS [see *intro*(2F)] file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl* request [see *streamio*(7)], and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(2F) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded, and are not available for a subsequent *read* or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set [see *chmod*(2F)], and there is a blocking (i.e. owned by another process) write lock on the segment of the file to be read:

> If O_NDELAY is set, the read will return EAGAIN.
>
> If O_NDELAY is clear, the read will sleep until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data becomes available.

When attempting to read a file associated with a *stream* that has no data currently available:

If O_NDELAY is set, the read will return EAGAIN.

If O_NDELAY is clear, the read will block until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. *read* then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream*. When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head*.

read will fail if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw I/O is temporarily insufficient.

[EAGAIN] No message waiting to be read on a *stream* and O_NDELAY flag set.

[EBADF] *fildes* is not a valid file descriptor open for reading.

[EBADMSG] Message waiting to be read on a *stream* is not a data message.

[EDEADLK] The read was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

[EINVAL] Attempted to read from a *stream* linked to a multiplexor.

[ENOLCK] The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head*. In this case, *iretval* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

EXAMPLE

```
      program read
      integer*4 read, fildes, nbyte
      integer*1 buf (80)

      integer*4 bytcnt, open
      character*80 bufc

      equivalence (buf, bufc)

c  Open a file to read

      fildes = open ('../example/read.F', 0)
      if (fildes .lt. 0) write (*,*) 'open err: ', fildes

c  Read 80 bytes of the file

      nbyte = 80
      bytcnt = read (fildes, buf, nbyte)
      if (bytcnt .lt. 0) write (*,*) 'read err: ', bytcnt

c  bytcnt is number of bytes read in

      write (*, 9000) bufc
9000  format (' ', a80)
      end
```

NOTES

The *streams* features described in this manual page are not supported in this release.

SEE ALSO

creat(2F), dup(2F), fcntl(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), getmsg(2F), streamio(7), termio(7).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a negative value indicating the error is returned.

NAME

readlink - read value of a symbolic link

SYNOPSIS

integer*4 readlink,bufsiz
character*SIZE path,buf
iretval = readlink(path,buf,bufsiz)

DESCRIPTION

SIZE can be any number 1 through 128. *readlink* places the contents of the symbolic link *path* in the buffer *buf* which has size *bufsiz.* The contents of the link are not null terminated when returned.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a -1 if an error occurs, placing the error code in the global variable *errno.*

ERRORS

readlink will fail and the file mode will be unchanged if:

[EPERM]	The *path* argument contained a byte with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX.
[ELOOP]	Too many symbolic links were encountered in translating a pathname.
[ENOTDIR]	A component of the *path* prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the *path* prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not the superuser.
[EINVAL]	The named file is not a symbolic link.
[EFAULT]	*buf* extends outside the process's allocated address space.

SEE ALSO

stat(2F), lstat(2F), symlink(2F)

NOTES

Currently on 88k machine only.

NAME

readv - do multiple reads from a file

SYNOPSIS

```
#    include <sysf/uio.i>
     integer*4 readv, fildes, iovcnt
     record /iovec/ iov(16)
     iretval = readv (fildes, iov, iovcnt)
```

DESCRIPTION

readv attempts to read data from the object referenced by *fildes*. The input data is scattered into the *iovcnt* buffers specified by the members of the *iov* array: iov(1), iov(2), ..., iov(iovcnt). This allows you to do up to 16 read operations with one system call.

The *iovec* structure is defined as:

```
structure /iovec/
   integer*4   iov_base
   integer*4   iov_len
end structure
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *readv* fills one area completely before proceeding to the next.

Upon successful completion, *readv* returns the number of bytes actually read and placed in the buffer. The system will read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file; this is not guaranteed for other cases.

EXAMPLE

```
      program readv
#     include <sysf/uio.i>
#     include <sysf/fcntl.i>
      integer*4 readv, fildes, iovcnt, i, j, iretval, offset, open
      integer*1 buf(80,4)
      character*20 buf1, buf2, buf3, buf4
      equivalence (buf (1,1), buf1), (buf (1,2), buf2), (buf (1,3), buf3)
      equivalence (buf (1,4), buf4)
      record /iovec/ iov (16)

c   Open a file

      fildes = open ('../example/readv.F', O_RDONLY)
      if (fildes .lt. 0) write (*,*) 'open error:', fildes

c   Fill the iov structure for four buffers

      offset = 0
      do 100 i = 1, 4
      iov (i).iov_base = %loc (buf(1,i))
      iov (i).iov_len = 20
      offset = offset + 20
100   continue

c   Do the four read operations

      iovcnt = 4
      iretval = readv (fildes, iov, iovcnt)
      if (iretval .lt. 0) write (*,*) 'readv error:', iretval

c   Print the buffers

      write (*,9000) buf1, buf2, buf3, buf4
9000  format (' ',4a20)
      end
```

ERRORS

readv will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid file descriptor open for reading.

[EFAULT] *buf* or part of the *iov* points outside the allocated address space.

[EINTR] A read from a slow device was interrupted by a signal before any data arrived.

[EINVAL] The pointer associated with *d* was negative or greater than 16.

[EINVAL] *iovcnt* was negative or greater than 16.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] Part of the *iov* points outside the process's allocated address space.

[EIO] An I/O error occurred while reading from the file system.

NOTES

The following example can be used to place the address of a buffer into the *iov* array:

```
iov(2).iov_base = %loc (buf2)
```

SEE ALSO

creat(2F), dup(2F), fcntl(2F), getmsg(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), read(2F).

NAME

recv, recvfrom - receive a message from a socket

SYNOPSIS

```
#include <sysf/types.i>
#include <sysf/socket.i>

integer*2 recv,cc,s
character*SIZE buf
integer*4 len,flags
cc = recv(s, buf, len, flags)

integer*2 recvfrom,cc,s
character*SIZE buf
integer*4 len,flags
record/sockaddr/from
integer*4 fromlen
cc = recvfrom(s, buf, len, flags, from, fromlen)
```

DESCRIPTION

recv and *recvfrom* are used to receive messages from a socket.

The *recv* call may be used only on a connected socket (see *connect*(2F)), while *recvfrom* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is nonzero, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; see *socket*(2F).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *fcntl*(3)) in which case a *cc* of -1 is returned with the external variable errno set to EWOULDBLOCK. The *select*(2F) call may be used to determine when more data arrives.

The flags argument to a *recv* call is formed by or-ing one or more of the values,

```
#define MSG OOB     0x1   ! process out-of-band data
#define MSG PEEK    0x2   ! peek at incoming message
```

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

[EBADF] The argument *s* is an invalid descriptor.

[ENOTSOCK] The argument *s* is not a socket.

[EWOULDBLOCK] The socket is marked non-blocking and the receive operation would block.

[EINTR] The receive was interrupted by delivery of a signal before any data was available for the receive.

[EFAULT] The data was specified to be received into a nonexistent or protected part of the process address space.

SEE ALSO

read(2F), send(2F), socket(2F).

NAME

relinquish - voluntarily give up CPU

SYNOPSIS

integer*4 relinquish
iretval = relinquish ()

DESCRIPTION

relinquish causes a context switch to occur from the calling process to the next process that is runnable at the same priority of the calling process. If no other processes are runnable at that priority, the caller will immediately switch back to itself. Since processes are scheduled roundrobin within a priority level, the caller cannot run again until all other runnable processes at the callers priority level have run.

The caller must have realtime or superuser privileges.

relinquish will fail if the following is true:

[EPERM] The requesting process does not have the appropriate permission.

EXAMPLE

```
        program relinquish
#       include <sysf/errno.i>
        integer*4 relinquish, slicesiz
        slicesiz = relinquish ()
        if (slicesiz .eq. EPERM) then
          write (*,*) 'not superuser or no realtime privileges'
        else
          if (slicesiz .lt. 0) write (*,*) 'relinquish error:', slicesiz
        endif
        end
```

DIAGNOSTICS

Upon successful completion, a non-negative integer is returned indicating the previous slice size of the target process. Otherwise, a negative value indicating the error is returned.

NAME

reltimerid - release a process interval timer identifier

SYNOPSIS

```
#    include <sysf/time.i>

     integer*4 reltimerid, timerid
     iretval = reltimerid (timerid)
```

DESCRIPTION

The *reltimerid* system call deactivates and releases the process interval *timerid* previously allocated to the realtime process via a *gettimerid*(2F) system call. Any outstanding timer events associated with the specified *timerid* are cancelled and the timer freed when *reltimerid* returns.

EXAMPLE

See absinterval(2F) for an example.

ERROR CODES

If successful, **reltimerid** returns 0. Otherwise, a negative value indicating the error is returned.

[EINVAL] The *timerid* argument does not correspond to an ID returned by *gettimerid*(2F).

The *timerid* specified is not valid for the calling process.

SEE ALSO

absinterval(2F), getinterval(2F), gettimerid(2F), incinterval(2F), resabs(2F), resinc(2F).

NAME

rename - change the name of a file.

SYNOPSIS

integer*4 rename
character*SIZE old,new
iretval = rename(old,new)

DESCRIPTION

SIZE can be any number 1 through 128. *rename* changes the name of a file from *old* to *new*. If *old* is a file (not a directory), *new* cannot be a directory, and if *new* is an existing file, it will be removed and *old* renamed. If *old* is a directory, and *new* exists, *new* must be empty, in which case it will be removed and *old* renamed. If *old* and *new* refer to the same file, *rename* will return successfully without making any changes.

RETURN VALUE

0 value is returned if the operation succeeds, otherwise *rename* returns -1 and the global value *errno* indicates the reason for the failure.

ERRORS

rename will fail and neither of the files named as arguments will be affected if any of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory, or *old* names a directory, and *new* is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters while _POSIX_NO_TRUE is in effect, or an entire pathname exceeded PATH_MAX.
[ENOENT]	The link named by *old* does not exist or *old* or *new* points to an empty string.
[EACCES]	A component of either path prefix denies search permission, or one of the directories containing *old* or *new* denies write permission, or the requested link requires writing in a directory with a mode that denies write permission.
[EXDEV]	The link named by *new* and the file named by *old* are on different logical devices (file systems).
[EROFS]	The requested link requires writing in a directory on an read-only file system.
[EINVAL]	The *new* pathname contains a path prefix that names *old*.
[EBUSY]	The directory named by *old* or *new* cannot be remaned because it is being used by the system or another process.
[ENOEMPTY]	The directory named by *new* contains file other than "." or "..".
[EISDIR]	The *new* points to a directory, and *old* is not a directory.
[ENOSPC]	The directory that would enter *new* cannot be extended.

SEE ALSO

mv(1), link(2F), open(2F), symlink(2F), unlink(2F)

NOTES

Currently on 88k machine only.

NAME

resabs, resinc - get resolution and maximum time value of process interval timer

SYNOPSIS

```
#    include <sysf/time.i>

     integer*4 resabs, timerid
     record /timestruc/ res, max
     iretval = resabs (timerid, res, max)

     integer*4 resinc, timerid
     record /timestruc/ res, max
     iretval = resinc (timerid, res, max)
```

DESCRIPTION

resabs returns the resolution and maximum absolute time value for the process interval timer specified by *timerid* (returned by *gettimerid*(2F)). These are the values used by the operating system to validate expiration values specified by *absinterval*(2F).

resinc returns the resolution and maximum increment time value for the process interval timer specified by *timerid* (returned by *gettimerid*(2F)). These are the values used by the operating system to validate expiration values specified by *incinterval*(2F).

res is a pointer to a timestruc structure into which the resolution is placed.

max is a pointer to a timestruc structure into which the maximum time value is placed.

EXAMPLE

```
     program resabs
#    include <sysf/time.i>
     integer*4 resabs, timerid
     integer*4 resinc, iretval
     record /timestruc/ absres, absmax, incres, incmax

c  Use evget to get event id
c  Use gettimerid to get timerid for event
c  Get absolute resolution and max time value

     iretval = resabs (timerid, absres, absmax)
     if (iretval .lt. 0) write (*,*) 'resabs error:', iretval

c  Get incremental resolution and max value

     iretval = resinc (timerid, incres, incmax)
     if (iretval .lt. 0) write (*,*) 'resinc error:', iretval

c  Print results

     write (*,9000) absres.tv_sec, absmax.tv_sec
     write (*,9001) incres.tv_sec, incmax.tv_sec

9000 format (' absolute resolution in secs:', i8,/
&       ' absolute maximum in secs:', i8)
9001 format (' incremental resolution in secs:', i8,/
&       ' incremental maximum in secs:', i8)
     end
```

ERROR CODES

If *resabs* or *resinc* is not successful, a negative value is returned with one of the following error codes.

[EFAULT] *res* or *max* points outside the allocated address space of the process.

[EINVAL] The *timerid* argument does not correspond to an ID returned by *gettimerid*(2F) or it has previously been released with a *reltimerid*(2F) call.

SEE ALSO

absinterval(2F), getinterval(2F), gettimerid(2F), incinterval(2F), reltimerid(2F), itimerstruc(4), timestruc(4).

NAME

resident - make locked segments resident in memory

SYNOPSIS

 # include <sysf/lock.i>
 # include <sysf/evt.i>

integer*4 resident, flags, eid
iretval = resident (flags, eid)

DESCRIPTION

The *resident* system call causes all segments of the calling process that have been locked using *plock*(2F) or *shmctl*(2F) to be made resident in physical memory. The effective user ID of the calling process must be superuser or have realtime privileges to use this call.

If *resident* is not called after *plock*(2F) or *shmctl*(2F) with the **SHM_LOCK** option, then, although the pages of the locked segments will be prevented from being paged out, each page will still be loaded when it is initially accessed, causing possible degradation of realtime performance.

After a call of *resident*, any attempt to increase the size of a locked data segment (whether through natural stack growth, calls to *malloc*(3), or explicit calls to *brk*(2F) or *stkexp*(2F)) may cause degradation of realtime performance; for example, pages belonging to lower priority processes may have to be copied to disk, to free physical memory for the extra resident pages being created. Therefore, a realtime process should preallocate enough space for the data and stack regions of its data segment, using *brk*(2F) or *stkexp*(2F), before calling *resident*.

A mechanism is provided to inform a process if it has violated its realtime constraints by expanding its data segment. If (*flags* & **EVT_POST**) is "true", *eid* specifies an event identifier, initialized by the user, to which events may be posted with data items consisting of a combination of the following flags:

STKEXP Pages have been added to the stack region of the data segment as a result of a *stkexp*(2F) call, causing degradation of realtime performance.

STKGROW Pages have been added to the stack region of the data segment as a result of natural stack growth, causing degradation of realtime performance.

BRK Pages have been added to the data segment as a result of a *brk*(2F) or *sbrk*(2F) call, causing degradation of realtime performance.

DATUNLOCK The data segment (comprising the data and stack regions) has been unlocked, causing degradation of realtime performance. This occurs when pages have been added to the data segment, but there is no more physical memory available for resident segments.

This mechanism is reset (i.e. no further events will be posted) following a call to *fork*(2F), *exec*(2F), *plock*(2F) with the **UNLOCK** option, or a subsequent call to *resident* with (*flags* & **EVT_POST**) equal to "false".

The *resident* system call will fail if:

[EPERM] The effective user ID of the calling process is not superuser, or does not have realtime privileges.

[EINVAL] No segment of the calling process has been locked by *plock*(2F) or *shmctl*(2F).

[EINVAL] The calling process does not have the event identifier specified by *eid*.

[EINVAL] A valid *eid* was specified, but the data segment is not locked.

ERROR CODES

If successful, *restimer* returns a value of 0. Otherwise, a negative value indicating the error is returned with one of the following error codes:

[EFAULT] The *res* or *maxval* arguments point outside the allocated address space of the process.

[EINVAL] The *timer_type* argument does not specify a valid system-wide realtime timer type.

[EIO] A device error occurred while accessing the system-wide timer.

SEE ALSO

gettimer(2F), settimer(2F), timestruc(4).

NAME

resume - resume a suspended process

SYNOPSIS

integer* resume, pid
iretval = resume (pid)

DESCRIPTION

pid is the process ID of the target process to be resumed.

If the target process is suspended because of a *suspend*(2F), or *swtch*(2F) system call, it will resume execution after the *resume* is completed and when it becomes the highest runnable process in the system.

For a process to have permission to resume another process, the requesting process must have superuser or realtime privileges which are granted via the *setrt*(2F) system call. If the effective user ID of the requesting process is superuser, permission checks are bypassed.

resume will fail if one or more of the following are true:

[ESRCH] No process can be found corresponding to that specified by *pid*. Or *pid* refers to a process that is not suspended.

[EINVAL] pid refers to the requesting process or a process that was not suspended.

[EPERM] The requesting process does not have the appropriate permission to resume the target process.

EXAMPLE

```
      program resume
      integer*4 resume, pid
      integer*4 suspend, fork, iretval

c  Create a child process, print a line, suspend, print line, exit

      pid = fork ()
      if (pid .lt. 0) then
        write (*,9001) 'fork error:', pid
        stop
      else if (pid .eq. 0) then
        write (*,9000) 'child suspended'
        iretval = suspend ()
        if (iretval .lt. 0) write (*,9001) 'child suspend error:', iretval
        write (*,9000) 'child resumed, exiting'
      else

c  Here if parent, resume child

        write (*,9001) 'child id:', pid
        iretval = resume (pid)
        if (iretval .lt. 0) write (*,9001) 'parent resume error:', iretval
      endif
9000 format ('0', a40)
9001 format ('0', a40,3x,i6)
      end
```

SEE ALSO

setrtgroup(1), resume(1R), getpid(2F), setrt(2F), suspend(2F), swtch(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

rmdir - remove a directory

SYNOPSIS

integer*4 rmdir
character*SIZE path
iretval = rmdir (path)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *rmdir* removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "." and "..".

The named directory is removed unless one or more of the following are true:

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the directory to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while accessing the file system.
[ENOLINK]	*path* points to a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program rmdir
      integer*4 rmdir
      character*40 path
      integer*4 iretval

c  Remove directory "tstdir" in current directory

      path = 'tstdir'
      iretval = rmdir (path)
      if (iretval .lt. 0) write (*,*) 'rmdir error:', iretval
      end
```

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

SEE ALSO

rmdir(1), rm(1), mkdir(1), mkdir(2F).

NAME

select - synchronous I/O multiplexing

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/time.i>

     integer*4 select, nfds
     record /fd_set/readfds, writefds, exceptfds
     record /timeval/ timeout
     iretval = select (nfds, readfds, writefds, exceptfds, timeout)

     integer*4 fd
     record /fd_set/ fdset
     FD_SET (fd, fdset)
     FD_CLR (fd, fdset)
     FD_ZERO (fdset)
     iretval = FD_ISSET (fd, fdset)
```

DESCRIPTION

select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e. the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, fdset)* returns nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is not %val(0), it specifies a maximum interval to wait for the selection to complete. If *timeout* is %val(0), the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as %val(0) if no descriptors are of interest.

EXAMPLE

```
     program select

#    include <sysf/fcntl.i>
#    include <sysf/types.i>
#    include <sysf/time.i>

     integer*4 select, nfds
     integer*4 nrdy
     integer*4 open, oflag, fildes
     integer*4 read
     integer*4 i

     record /fd_set/ rwfds
     record /timeval/ timeout
```

```
c  Inititalize all descriptor values

      FD_ZERO (rwfds)
      nfds = 0

c  Open a file, set descriptor entry

      oflag = O_RDWR .or. O_NDELAY
      fildes = open ('../example/tst.x', oflag)
      if (fildes .ge. 0) FD_SET (fildes, rwfds)
      nfds = max (nfds, fildes)

c  Set timeout value to 5 milli-seconds

      timeout.tv_sec = 0
      timeout.tv_usec = 5000

c  Check 0, 1, and 2 also (stdin, stdout, stderr)

      do 100 i=1, 3
      FD_SET (i-1, rwfds)
100   continue

      write (*,9000) 'rwfds before = ', (rwfds.fds_bits (i), i=0, 7)
      nfds = nfds + 1

c  Check which files are ready for reading

      nrdy = select (nfds, rwfds, %val (0), %val (0), timeout)

c  Print results

      if (nrdy .lt. 0) then
        write (*,9001) 'select error ', nrdy
      else if (nrdy .gt. 0) then
        write (*,9001) 'number of files ready for I/O: ', nrdy
        do 200 i = 1, nfds
        if (FD_ISSET (i-1, rwfds) .ne. 0) goto 210
200     continue
        i = 0
210     continue
        fildes = i - 1
        write (*,9001) 'first file descriptor for I/O: ', fildes
      else
        write (*,9001) 'timeout occurred. files ready: ', nrdy
      endif
      write (*,9000) 'rwfds after =  ', (rwfds.fds_bits (i), i=0, 7)
9000  format (' ', a15, 8 (i6, 1x))
9001  format (' ', a30, i6)
      end
```

```
      subroutine memset (mem, value, bc)

c  This subroutine called by 'FD_ZERO'

      integer*1 mem (bc)
      integer*4 value, bc, i

      do 100 i=1, bc
      mem (i) = value
100   continue
      return
      end
```

RETURN VALUE

select returns the number of ready descriptors that are contained in the descriptor sets, or a negative number if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF] One of the descriptor sets specified an invalid descriptor.

[EINTR] A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL] The specified time limit is invalid. One of its components is negative or too large.

NOTES

The macros FD_SET, FD_CLR, and FD_ZERO expand to FORTRAN statements. FD_ISSET is a function. FD_ZERO requires the subroutine *memset* [see **memset**(3)].

SEE ALSO

accept(2F), connect(2F), read(2F), write(2F), recv(2F), send(2F), getdtablesize(2F)

BUGS

Although the provision of *getdtablesize* (2F) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of <sysf/types.i>.

select should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the *select* call.

NAME

semctl - semaphore control operations

SYNOPSIS

```
# include <sysf/types.i>
# include <sysf/ipc.i>
# include <sysf/sem.i>

integer*4 semctl, semid, cmd, semnum
integer*4 val, array (25)

record /semid_ds/ semid_buf
iretval = semctl (semid, semnum, cmd, val)
iretval = semctl (semid, semnum, cmd, semid_buf)
iretval = semctl (semid, semnum, cmd, array)
```

DESCRIPTION

semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL Return the value of semval [see *intro*(2F)]. {READ}

SETVAL Set the value of semval to *val*. {ALTER} When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of sempid. {READ}

GETNCNT Return the value of semncnt. {READ}

GETZCNT Return the value of semzcnt. {READ}

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL Place semvals into array pointed to by *array*. {READ}

SETALL Set semvals according to the array pointed to by *array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *semid_buf*. The contents of this structure are defined in *intro*(2F). {READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *semid_buf*:
sem_perm.uid
sem_perm.gid
sem_perm.mode !only low 9 bits

This cmd can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of superuser, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

semctl fails if one or more of the following are true:

[EINVAL] *semid* is not a valid semaphore identifier.

[EINVAL] *semnum* is less than zero or greater than **sem_nsems**.

[EINVAL] *cmd* is not a valid command.

[EACCES] Operation permission is denied to the calling process [see *intro*(2F)].

[ERANGE] *cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.

[EPERM] *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

[EFAULT] *semid_buf* points to an illegal address.

EXAMPLE

```
      program semctl
#     include <sysf/types.i>
#     include <sysf/ipc.i>
#     include <sysf/sem.i>
#     define MAXSEMNUM 8
      integer*4 semget, key, nsems, semflg
      integer*4 semctl, semid, cmd, semnum
      integer*4 val, array (0:MAXSEMNUM-1)
      integer*4 array_out (0:MAXSEMNUM-1)
      integer*4 iretval, i
      record /semid_ds/ semid_buf

c   Get an id for a number of semaphores

      key = 1234     ! some agreed upon value
      nsems = MAXSEMNUM
      semflg = '777'o .or. IPC_CREAT
      semid = semget (key, nsems, semflg)
      if (semid .lt. 0) write (*,*) 'semget error:', semid
      write (*,*) 'semaphore id:', semid

c   Set some values

      do 100 i = 0, MAXSEMNUM-1
      array (i) = i
      array_out (i) = -1
100   continue
      cmd = SETALL
      semnum = 0
      iretval = semctl (semid, semnum, cmd, array)
      if (iretval .lt. 0) write (*,*) 'semctl error:', iretval

c   Get some values

      iretval = semctl (semid, 0, GETALL, array_out)
      if (iretval .lt. 0) write (*,*) '2 semctl error:', iretval
      write (*,9000) 'semaphore values', array_out
9000  format (' ', a20, /, 8 (1x, i5))
```

```
c   Get and print value of a specific semaphore

       do 200 i = 0, MAXSEMNUM-1
       iretval = semctl (semid, i, GETVAL, 0)
       if (iretval .lt. 0) write (*,*) '3 semctl error:', iretval
       write (*,*) 'semaphore value:', i, iretval
200    continue

c   Get and print number of processes waiting on nonzero
c   semaphore value

       iretval = semctl (semid, 2, GETNCNT, 0)
       if (iretval .lt. 0) write (*,*) '4 semctl error:', iretval
       write (*,*) 'nonzero wait count, semaphore #2:', iretval

c   Get some info on these semaphores

       semid_buf.sem_nsems = 0
       iretval = semctl (semid, 0, IPC_STAT, semid_buf)
       if (iretval .lt. 0) write (*,*) '5 semctl error:', iretval
       write (*,9001) 'number of semaphores:', semid_buf.sem_nsems
9001   format (' ', a20, 3x, i10)

c   Remove semaphore

       iretval = semctl (semid, 0, IPC_RMID, 0)
       if (iretval .lt. 0) write (*,*) '6 semctl error:', iretval

       end
```

SEE ALSO

intro(2F), semget(2F), semop(2F).

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval
GETPID	The value of sempid
GETNCNT	The value of semncnt
GETZCNT	The value of semzcnt
All others	A value of 0

Otherwise, a negative value indicating the error is returned.

NAME

semget - get set of semaphores

SYNOPSIS

 # include <sysf/types.i>
 # include <sysf/ipc.i>
 # include <sysf/sem.i>

integer*4 semget, key, nsems, semflg
iretval = semget (key, nsems, semflg)

DESCRIPTION

semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores [see *intro*(2F)] are created for *key* if one of the following is true:

> *key* is equal to **IPC_PRIVATE**.
>
> *key* does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

> **Sem_perm.cuid, sem_perm.uid, sem_perm.cgid,** and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
>
> The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.
>
> **Sem_nsems** is set equal to the value of *nsems*.
>
> **Sem_otime** is set equal to 0 and **sem_ctime** is set equal to the current time.

semget fails if one or more of the following are true:

[EINVAL] *nsems* is either less than or equal to zero or greater than the system-imposed limit.

[EACCES] A semaphore identifier exists for *key*, but operation permission [see *intro*(2F)] as specified by the low-order 9 bits of *semflg* would not be granted.

[EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.

[ENOENT] A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is "false".

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.

[EEXIST] A semaphore identifier exists for *key* but ((*semflg* & **IPC_CREAT**) and (*semflg* & **IPC_EXCL**)) is "true".

EXAMPLE

```
        program semget
#       include <sysf/types.i>
#       include <sysf/ipc.i>
#       include <sysf/sem.i>
#       define MAXSEMNUM 8
        integer*4 semget, key, nsems, semflg
        integer*4 semid

c   Get an id for a number of semaphores

        key = 1234     ! some agreed upon value
        nsems = MAXSEMNUM
        semflg = '777'o .or. IPC_CREAT
        semid = semget (key, nsems, semflg)
        if (semid .lt. 0) write (*,*) 'semget error:', semid
        write (*,*) 'semaphore id:', semid

        end
```

SEE ALSO

intro(2F), semctl(2F), semop(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a negative value indicating the error is returned.

NAME

semop - semaphore operations

SYNOPSIS

 # include <sysf/types.i>
 # include <sysf/ipc.i>
 # include <sysf/sem.i>

integer*4 semop, semid, nsops
record /sembuf/ sops(NUM)
iretval = semop (semid, sops, nsops)

DESCRIPTION

semop is used to automatically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. *NUM* should be greater than or equal to *nsops.* The contents of each structure includes the following members:

```
integer*2  sem_num   !semaphore number
integer*2  sem_op    !semaphore operation
integer*2  sem_flg   !operation flags
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If semval [see *intro*(2F)] is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value [see *exit*(2F)] for the specified semaphore.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur.

Semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value for the specified semaphore.

The semid for which the calling process is awaiting action is removed from the system [see *semctl*(2F)]. When this occurs, *iretval* is set equal to EIDRM.

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2F).

If *sem_op* is a positive integer, the value of *sem_op* is added to semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If semval is zero, *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The semid for which the calling process is awaiting action is removed from the system. When this occurs, *iretval* is set equal to EIDRM.

The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2F).

semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL] *semid* is not a valid semaphore identifier.

[EFBIG] *sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG] *nsops* is greater than the system-imposed maximum.

[EACCES] Operation permission is denied to the calling process [see *intro*(2F)]

[EAGAIN] The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".

[ENOSPC] The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.

[EINVAL] The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit.

[ERANGE] An operation would cause a semadj value to overflow the system-imposed limit.

[EFAULT] *sops* points to an illegal address.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

EXAMPLE

```
      program semop
#     include <sysf/types.i>
#     include <sysf/ipc.i>
#     include <sysf/sem.i>
#     define MAXSEMNUM 1
      integer*4 semget, key, nsems, semflg
      integer*4 semop, nsops
      integer*4 semid, i, j, iretval, semctl
      integer*4 fork, pid, pause
      character*80 dummy
      record /sembuf/ sops (MAXSEMNUM)

c   Get an id for semaphores

      key = 1234     ! some agreed upon value
      nsems = MAXSEMNUM
      semflg = '777'o .or. IPC_CREAT
      semid = semget (key, nsems, semflg)
      if (semid .lt. 0) write (*,*) 'semget error:', semid
      write (*,*) 'semaphore id:', semid

c   Make a child

      pid = fork ()

c   Child and parent executing together.
c   Fill array of semaphore operation to take each semaphore

      do 100 i = 1, MAXSEMNUM
      sops (i).sem_num = i - 1
      sops (i).sem_op = 1   ! take a semaphore
      sops (i).sem_flg = 0   ! no flags
100   continue

c   Perform the semaphore operations

      nsops = MAXSEMNUM
      iretval = semop (semid, sops, nsops)
      if (iretval .lt. 0) write (*,*) 'semop error:', iretval

c   Inform of semaphore taken

      if (pid .gt. 0) write (*,*) 'semaphores taken by parent'
      if (pid .eq. 0) write (*,*) 'semaphores taken by child'
```

```
c  Release semaphores

      do 150 i = 1, MAXSEMNUM
      sops (i).sem_op = -1
      sops (i).sem_flg = SEM_UNDO
150   continue
      iretval = semop (semid, sops, nsops)
      if (iretval .lt. 0) write (*,*) '3 semop error:', iretval

c  Wait for semaphores to reach zero

      do 200 i = 1, MAXSEMNUM
      sops (i).sem_op = 0
200   continue
      iretval = semop (semid, sops, nsops)
      if (iretval .lt. 0) write (*,*) '2 semop error:', iretval

      if (pid .eq. 0) write (*,*) 'child exiting'
      if (pid .ne. 0) write (*,*) 'parent exiting'

      do 500 i = 1, 1000
      do 500 j = 1, 1000
      iretval = i
500   continue

c  If parent, remove semaphore

      if (pid .eq. 0) stop
      iretval = semctl (semid, 0, IPC_RMID, 0)
      if (iretval .lt. 0) write (*,*) 'could not remove semaphore', iretval
      end
```

SEE ALSO

exec(2F), exit(2F), fork(2F), intro(2F), semctl(2F), semget(2F).

DIAGNOSTICS

If *semop* returns due to the receipt of a signal, a value of EINTR is returned to the calling process. If it returns due to the removal of a *semid* from the system, a value of EIDRM is returned.

Upon successful completion, a value of zero is returned. Otherwise, a negative value indicating the error is returned.

NAME

send, sendto - sends a message from a socket

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*2 cc,s
character*SIZE msg
integer*4 send,len,flags
cc = send(s, msg, len, flags)

integer*2 cc,s
character*SIZE msg
integer*4 sendto,len,flags
record/sockaddr/to
integer*4 tolen
cc = sendto(s, msg, len, flags, to, tolen)

DESCRIPTION

SIZE can be any number 1 through 128. *send* and *sendto* are used to transmit a message to another socket. *send* may be used only when the socket is in a connected state, while *sendto* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of -1 indicate some locally detected errors.

If no message space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *select*(2F) call may be used to determine when it is possible to send more data.

The flags parameter may be set to MSG_OOB to send out-of-band data on sockets which support this notion (e.g., SOCK_STREAM).

See *recv*(2F) for a description of the msghdr structure.

ERRORS

The call returns the number of characters sent, or a negative value indicating an error occurred.

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.

EXAMPLE

see socket(2F)

SEE ALSO

recv(2F), socket(2F)

NAME

setgroups - set group access list

SYNOPSIS

#include <sysf/param.i>
#include <sysf/types.i>
integer*4 rename
character*SIZE old,new
iretval = rename(old,new)

DESCRIPTION

SIZE can be any number between 1 through 128. *rename* changes the name of a file from *old* to *new*. If *old* is a file (not a directory), *new* cannot be a directory, and if *new* is an existing file, it will be removed and *old* renamed. If *old* is a directory, and *new* exists, *new* must be empty, in which case it will be removed and *old* renamed. If *old* and *new* refer to the same file, *rename* will return successfully without making any changes.

RETURN VALUE

0 value is returned if the operation succeeds, otherwise *rename* returns -1 and the global value *errno* indicates the reason for the failure.

ERRORS

rename will fail and neither of the files named as arguments will be affected if any of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory, or *old* names a directory, and *new* is not a directory.
[ENAMETOOLONG]	A component of a pathname exceeded NAME_MAX characters while _POSIX_NO_TRUE is in effect, or an entire pathname exceeded PATH_MAX.
[ENOENT]	The link named by *old* does not exist or either *old* or *new* points to an empty string.
[EACCES]	A component of either path prefix denies search permission, or one of the directories containing *old* or *new* denies write permission, or the requested link requires writing in a directory with a mode that denies write permission.
[EXDEV]	The link named by *new* and the file named by *old* are on different logical devices (file systems).
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EINVAL]	The *new* pathname contains a path prefix that names *old*.
[EBUSY]	The directory named by *old* or *new* cannot be renamed because it is being used by the system or another process.
[ENOEMPTY]	The directory named by *new* contains file other than "." or "..".
[EISDIR]	The *new* points to a directory, and *old* is not a directory.
[ENOSPC]	The directory that would enter *new* cannot be extended.

SEE ALSO

mv(1), link(2), open(2), symlink(2), unlink(2)

NOTES

Currently on 88k machine only.

NAME

setpgid - Set process group ID for job control.

SYNOPSIS

integer*4 setpgid,pid,pgid
iretval = setpgid(pid,pgid)

DESCRIPTION

The *setpgid* function is used to either create a new process group or move the calling process or one of it's children into an already existing process group.

Upon successful completion, the process group ID of the process with a process ID which matches *pid* is set to *pgid*. If *pid* is zero, the *pid* of the calling process is used for *pid*. If *pgid* is set to 0, *pid* is used for *pgid*.

ERRORS

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

setpgid will not set the process group ID for process *pid* equal to *pgid* if one of the following are true:

[EACCESS] *pid* matches the process ID of a child process of the calling process which has successfully completed an *exec*(2) *call.*

[EINVAL] *pgid* does not fall within the range of valid process group ID numbers.

[EPERM] *pgid* matches the process ID of a session leader, or *pid* matches the process ID of a child of the calling process which does not belong to the calling process's session, or there is no process with a process ID which matches the *pgid* argument within the session of the calling process.

[ESRCH] *pid* does not match the process ID of the calling process or the process ID of a child of the calling process.

SEE ALSO

intro(2), exec(2), exit(2), fork(2), getpid(2), kill(2), sigaction(2), setsid(2), and terminos(7)

NOTES

Currently on 88k machine only.

NAME

setpgrp - set process group ID

SYNOPSIS

integer*4 setpgrp
iretval = setpgrp ()

DESCRIPTION

setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

EXAMPLE

```
      program setpgrp
      integer*4 setpgrp, gid

c   Set group id to process id

      gid = setpgrp ()
      write (*,*) 'group id:', gid
      end
```

SEE ALSO

exec(2F), fork(2F), getpid(2F), intro(2F), kill(2F), signal(2F).

DIAGNOSTICS

setpgrp returns the value of the new process group ID.

NAME

setpri - set scheduling priority

SYNOPSIS

integer*4 setpri, pid, pri
iretval = setpri (pid, pri)

DESCRIPTION

pid is the process ID of the target process, or zero if the calling process is the target. *pri* is the new scheduling priority that is to be assigned to the target process. The range of valid process priorities is 0 to 255, with lower values indicating higher priorities. Realtime processes have priorities in the range of 0 to 127. Time sharing processes have priorities in the range of 128 to 253. Priority 254 is to be used by user supplied idle tasks and priority 255 is reserved for system idle tasks. The priority of a realtime process or idle process is fixed unless explicitly changed via a *setpri* system call.

Time sharing processes are subject to fair-share priority migration similar to what occurs in standard UNIX. Using *setpri* to set the priority of a process within the fair-share range will subject that process to automatic priority adjustment by the system.

If *pid* is non-zero, the requesting process must have realtime privileges which are granted via the *setrt*(2F) system call. If *pid* is zero, the caller must have realtime or superuser permissions to make it's priority more favorable. If the effective user ID of the requesting process is superuser, permission checks are bypassed.

Care must be taken when using *setpri* because a process will be able to completely consume the bandwidth of a CPU by setting a high scheduling priority, therefore not relinquishing the CPU.

setpri will fail if one or more of the following are true:

[ESRCH] No process can be found corresponding to that specified by *pid*.

[EPERM] The requesting process does not have the appropriate permission to change the priority of the target process.

[EINVAL] *pri* is negative or greater than 255.

EXAMPLE

```
      program setpri
      integer*4 setpri, pid, pri
      integer*4 prevpri

c   Reduce my priority

      pid = 0
      pri = 253
      prevpri = setpri (pid, pri)
      if (prevpri .lt. 0) write (*,*) 'setpri error:', prevpri
      if (prevpri .ge. 0) write (*,*) 'previous priority:', prevpri
      end
```

SEE ALSO

setpri(1R), getpid(2F), getpri(2F), setrt(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer is returned indicating the previous priority of the target process. Otherwise, a negative value indicating the error is returned.

NAME

setpsr,getpsr - set/get Processor Status Register

SYNOPSIS

#include <sysf/m88kbcs.i> (on 88K machine only)
integer*4 setpsr,psr
iretval = setpsr(psr)

integer*4 getpsr
iretval = getpsr()

DESCRIPTION

setpsr sets certain bits in the Processor Status Register (PSR) of the calling process. The precise effects of these bits are defined in the Motorola *MC88100 User's Manual.*

Setting the C bit (PSR_C) sets the carry bit; clearing it will reset the carry bit.

Setting the MXM bit (PSR_MXM) will disable misaligned access exceptions. With the bit clear, a misaligned access will cause a SIGBUS signal to be delivered to the process.

Setting the BO bit (PSR_BO) will select Little-Endian byte order. Clearing this bit will select Big-Endian, which is the normal mode. All interfaces to the system must always be in Big-Endian byte order.

Setting the SER bit (PSR_SER) will select serial operation. Clearing this bit will select concurrent operation, which is the normal mode.

The *psr* value may be formed by the OR of the following:

PSR_C	0x20000000
PSR_MXM	0x10000000
PSR_BO	0x00000004
PSR_SER	0x40000000

RETURN VALUE

setpsr() returns the previous value of the PSR.
getpsr() returns the current value of the PSR.

NOTES

Currently on 88k machine only.

NAME

setrt - set realtime privileges

SYNOPSIS

integer*4 setrt
iretval = setrt()

integer*4 = clrrt
iretval = clrrt()

DESCRIPTION

setrt gives the calling process realtime privileges. Realtime privileges are required in order to perform such realtime operations as changing the scheduling priority of a process via *setpri*(2F) or suspending a process via *suspend*(2F).

The requesting processes effective user ID must appear in the kernel's realtime user privileged list which is set with the *setrtusers*(2F) system call. Realtime privileges are automatically obtained when a process acquires superuser privileges. Realtime privileges are also inherited by a child through a fork if the parent had realtime privileges.

clrrt removes realtime privileges from the calling process; although in certain circumstances the process may still be able to access realtime resources that were obtained while the process was privileged.

setrt will fail if the following is true:

[EPERM] The requesting process does not have permission to become a realtime process.

EXAMPLE

```
      program setrt
      integer*4 setrt, clrrt, iretval

c   Set realtime privileges if have permission

      iretval = setrt ()
      if (iretval .lt. 0) write (*,*) 'setrt error:', iretval

c   Do some realtime stuff
c   ...
c   Clear realtime privileges

      iretval = clrrt ()
      end
```

SEE ALSO

setrtusers(1M), setrtusers(2F).

DIAGNOSTICS

clrrt always returns a value of 0. Upon successful completion, *setrt* returns a value of 0. Otherwise, a negative value indicating the error is returned.

NAME

setrtusers - set realtime privileged users list

SYNOPSIS

integer*4 setrtusers, uidp (SIZE), count
iretval = setrtusers (uidp, count)

DESCRIPTION

setrtusers loads the system realtime privilege table with a list of user IDs that will be allowed to acquire realtime privileges, see *setrt*(2F). *SIZE* is the size of the array containing the user IDs.

setrtusers will fail and leave the system realtime privilege table unchanged if one or more of the following are true:

[EPERM] The requesting process does not have superuser permissions.

[EFAULT] *uidp* points to an invalid address.

[EINVAL] *count* specifies more user IDs than would fit in the system realtime privilege table.

EXAMPLE

```
      program setrtusers
      integer*4 setrtusers, uidp (10), count
      integer*4 getuid, iretval

c  Get my user id

      uidp (1) = getuid ()

c  Load realtime users list

      count = 1
      iretval = setrtusers (uidp, count)
      if (iretval .lt. 0) write (*,*) 'setrtusers error:', iretval
      end
```

SEE ALSO

setrtusers(1M), setrt(2F).

DIAGNOSTICS

Upon successful completion, *setrtusers* returns a value of 0. Otherwise, a negative value indicating the error is returned.

NAME

setslice - set CPU time slice size

SYNOPSIS

```
#    include <sysf/param.i>

     integer*4 setslice, pid, slice
     iretval = setslice (pid, slice)
```

DESCRIPTION

pid is the process ID of the target process, or zero if the calling process is the target. *slice* is the new CPU time slice, specified in ticks, that is to be assigned to the target process. A tick is 1/HZ of a second. The range of valid slice sizes is 1 to 2^{31}-1 (a little over 11 years with a 60 HZ system clock).

Processes will not share the CPU with other processes of the same priority except at the expiration of their time slice or when some other event in the system causes the CPU to be rescheduled (I/O completion, etc.).

For a process to have permission to set the time slice of the target process or increase the time slice of itself above the system default, the requesting process must have realtime privileges which are granted via the *setrt*(2F) system call. If the effective user ID of the requesting process is superuser, permission checks are bypassed.

setslice will fail if one or more of the following are true:

[EINVAL] *slice* is non-positive.

[EPERM] The requesting process does not have the appropriate permission to change the time slice of the target process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLE

```
     program setslice
#    include <sysf/param.i>
     integer*4 setslice, pid, slice
     integer*4 prevslice

c  Decrease time slice

     pid = 0  !  myself
     slice = HZ - 5
     prevslice = setslice (pid, slice)
     if (prevslice .lt. 0) write (*,*) 'setslice error:', prevslice
     write (*,*) 'previous slice value:', prevslice
     end
```

SEE ALSO

setpri(1R), getpid(2F), setrt(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer is returned indicating the previous slice size of the target process. Otherwise, a negative value indicating the error is returned.

NAME

settimer - set the current value for a system-wide realtime timer

SYNOPSIS

```
#     include <sysf/time.i>

      integer*4 settimer, timer_type
      record /timestruc/ tp
      iretval = settimer (timer_type, tp)
```

DESCRIPTION

The *settimer* system call sets the value of the system-wide realtime timer, specified by the *timer_type* argument, to the value pointed to by *tp*.

The *timer_type* argument identifies the system-wide realtime timer used with this system call. TIMEOFDAY is a valid *timer_type* and corresponds to the system time-of-day clock representing the current time in seconds and nanoseconds since January 1, 1970. This timer is ascending in nature and is updated by the system at the frequency of the 64 Hz system clock.

tp is a pointer to a timestruc structure where the timer value is taken from.

EXAMPLE

```
      program settimer
#     include <sysf/time.i>
      integer*4 settimer, timer_type
      record /timestruc/ tp
      integer*4 iretval, gettimer

c     Get current time

      iretval = gettimer (TIMEOFDAY, tp)
      if (iretval .lt. 0) write (*,*) 'gettimer error:', iretval

c     Modify for new time (bump by one hour)

      tp.tv_sec = tp.tv_sec + 3600
      iretval = settimer (TIMEOFDAY, tp)
      if (iretval .lt. 0) write (*,*) 'settimer error:', iretval
      end
```

ERROR CODES

If successful, **settimer** returns a value of 0. Otherwise, a negative value indicating the error is returned.

[EFAULT] *tp* points outside the allocated address space of the process.

[EINVAL] The *timer_type* argument does not specify a valid system-wide realtime timer type.

[EIO] A device error occurred while accessing the system-wide realtime timer.

[EPERM] The effective user ID does not have realtime privileges.

SEE ALSO

gettimer(2F), restimer(2F), timestruc(4).

NAME

setuid, setgid - set user and group IDs

SYNOPSIS

integer*4 setuid, uid
iretval = setuid (uid)

integer*4 setgid, gid
iretval = setgid (gid)

DESCRIPTION

setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2F) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

The *uid* is out of range. [EINVAL]

EXAMPLE

```
      program setuid
      integer*4 setuid, uid, setgid, gid
      integer*4 getuid, getgid
      integer*4 iretval

c     Get current user and group ids

      uid = getuid ()
      gid = getgid ()

c     Modify and update

      uid = uid + 1
      gid = gid + 1
      iretval = setuid (uid)
      if (iretval .lt. 0) write (*,*) 'setuid error:', iretval
      iretval = setgid (gid)
      if (iretval .lt. 0) write (*,*) 'setgid error:', iretval
      end
```

SEE ALSO

getuid(2F), intro(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

shmctl - shared memory control operations

SYNOPSIS

 # include <sysf/types.i>
 # include <sysf/ipc.i>
 # include <sysf/shm.i>

integer*4 shmctl, shmid, cmd
record /shmid_ds/ buf
iretval = shmctl (shmid, cmd, buf)

DESCRIPTION

shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2F). {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

shm_perm.uid
shm_perm.gid
shm_perm.mode !only low 9 bits

This *cmd* can only be executed by a process that has an effective user ID equal to that of superuser, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to that of superuser, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

SHM_LOCK Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that has realtime privileges, or has an effective user ID equal to superuser.

SHM_UNLOCK Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has realtime privileges, or has an effective user ID equal to superuser.

shmctl will fail if one or more of the following are true:

[EINVAL] *shmid* is not a valid shared memory identifier.

[EINVAL] *cmd* is not a valid command.

[EACCES] *cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process [see *intro*(2F)].

[EPERM] *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of superuser, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

[EPERM] *cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of superuser, or the calling process does not have realtime privileges.

[EFAULT] *buf* points to an illegal address.

[ENOMEM] *cmd* is equal to **SHM_LOCK** and there is not enough memory.

EXAMPLE

```
      program shmctl

c  See "shmop(2F)" to see how to link edit this program

#     include <sysf/types.i>
#     include <sysf/ipc.i>
#     include <sysf/shm.i>
      integer*4 shmctl, shmid, cmd
      record /shmid_ds/ buf
      integer*4 shmget, key, size, shmflg
      integer*4 iretval, varc1, varc2
      integer*4 cmnaddr, cmnsize

c  Define common

      external cmnsize   !  size of common
      common /c1/ varc1
      common /c2/ varc2

c  Get shared memory segment

      key = 1234   !  Some agreed upon value
      size = %loc (cmnsize)
      shmflg = '777'o .or. IPC_CREAT
      shmid = shmget (key, size, shmflg)
      if (shmid .lt. 0) write (*,*) 'shmget error:', shmid

c  Remove shared memory segment

      cmd = IPC_RMID
      iretval = shmctl (shmid, cmd, buf)
      if (iretval .lt. 0) write (*,*) 'shmctl error:', iretval
      end
```

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

SEE ALSO

resident(2F), shmget(2F), shmop(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

shmget - get shared memory segment identifier

SYNOPSIS

 # include <sysf/types.i>
 # include <sysf/ipc.i>
 # include <sysf/shm.i>

integer*4 shmget, key, size, shmflg ,physadr
iretval = shmget (key, size, shmflg [,physadr])

DESCRIPTION

shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes [see *intro*(2F)] are created for *key* if one of the following are true:

key is equal to **IPC_PRIVATE**.

key does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

shm_perm.cuid, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*.

shm_segsz is set equal to the value of *size*.

shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

shm_ctime is set equal to the current time.

If (*shmflg & IPC_PHYS*) is "true," then *shmget* retrieves the *physadr* argument and creates a shared memory segment starting at that physical memory address. This physical memory must not be within the kernel's free memory pool. When created, a physical shared memory segment does not remove the associated memory from the system free memory pool. Upon removal, the memory is not returned to the system free memory pool. *IPC_PHYS* may only be set by a process that has realtime privileges, or has an effective user ID equal to superuser.

For physical shared memory, if (*shmflg & IPC_NOCLEAR*) is "true," then the shared memory segment is not cleared on the first attach.

For physical shared memory, if (*shmflg & IPC_CI*) is "true," then the hardware cache, if any, is inhibited on this shared memory segment.

shmget will fail if one or more of the following are true:

[EINVAL] *size* is less than the system-imposed minimum or greater than the system-imposed maximum.

[EACCES] A shared memory identifier exists for *key* but operation permission [see *intro*(2F)] as specified by the low-order 9 bits of *shmflg* would not be granted.

[EINVAL] A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

[ENOENT] A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false".

[ENOSPC] A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.

[ENOMEM] A shared memory identifier and associated shared memory segment are to be created but the amount of available memory is not sufficient to fill the request.

[EEXIST] A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) and (*shmflg*& **IPC_EXCL**)) is "true".

[EPERM] A physical shared memory identifier is to be created but the calling process does not have realtime privileges, or the effective user ID of the calling process is not superuser.

EXAMPLE

See shmop(2F) for an example.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

SEE ALSO

intro(2F), shmctl(2F), shmop(2F).

DIAGNOSTICS

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a negative value indicating the error is returned.

NAME

shmop: shmat, shmdt - shared memory operations

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/ipc.i>
#    include <sysf/shm.i>

     integer*4 shmat, shmid, shmaddr, shmflg
     iretval = shmat (shmid, shmaddr, shmflg)

     integer*4 shmdt, shmaddr
     iretval = shmdt (shmaddr)
```

DESCRIPTION

shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

> If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.
>
> If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).
>
> If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "false", the segment is attached at the address given by *shmaddr*.

shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr.*

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

shmat will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL] *shmid* is not a valid shared memory identifier.

[EACCES] Operation permission is denied to the calling process [see *intro*(2F)].

[ENOMEM] The available data space is not large enough to accommodate the shared memory segment.

[EINVAL] *shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address.

[EINVAL] *shmaddr* is not equal to zero, (*shmflg* & **SHM_RND**) is "false", and the value of *shmaddr* is an illegal address.

[EMFILE] The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

shmdt will fail and not detach the shared memory segment if:

[EINVAL] *shmaddr* is not the data segment start address of a shared memory segment.

[EBUSY] The shared memory segment is still in use for asynchronous I/O or connected interrupts.

EXAMPLE

```
      program shmop

c  This program is an example of making a region of memory
c  GLOBAL to more than one process.  Each process must be
c  link edited with the common region and contain statements
c  to get and attach the region.  Following the end statement
c  of this program are steps to link edit and necessary files.

#     include <sysf/types.i>
#     include <sysf/ipc.i>
#     include <sysf/shm.i>
      integer*4 shmget, key, size, shmflg, physadr
      integer*4 shmat, shmid
      integer*4 iretval, varc1, varc2, suspend
      integer*4 cmnaddr, cmnsize

c  Define global common

      external cmnaddr  !  address of common
      external cmnsize  !  size of common
      common /c1/ varc1
      common /c2/ varc2

c  Get shared memory segment

      key = 1234  !  Some agreed upon value
      size = %loc (cmnsize)
      shmflg = '777'o .or. IPC_CREAT
      shmid = shmget (key, size, shmflg)
      if (shmid .lt. 0) write (*,*) 'shmget error:', shmid

c  Attach the common regions

      iretval = shmat (shmid, cmnaddr, shmflg)
      if (iretval .lt. 0) write (*,*) 'shmat error:', iretval

c  Print values of common variables and increment for next invocation

      write (*,9000) 'varc1:', varc1, 'varc2:', varc2
      varc1 = varc1 + 5
      varc2 = varc2 + 10

9000 format (' ', a6, 2x, i6, 2x, a6, 2x, i6)
      end

---------- common.f ----------

      block data common
      integer*4 varc1, varc2
      common /c1/ varc1
      common /c2/ varc2
      end
```

```
---------- ld.cmd ----------

/*  Save the following as "ld.cmd"   */

SECTIONS
{
	common 0x400000 (NOLOAD) :
	{
		cmnaddr_ = .;
		common.o [COMMON]
		cmnsize_ = . - cmnaddr_;
	}

	.text ALIGN (0x400000) :
	{
		*(.init)
		*(.text)
	}

	GROUP ALIGN (0x400000) :
	{
		.data : {}
		.bss  : {}
	}
}

/*  End of "ld" command file   */

---------- Steps to link edit ----------

gf77 -c common.f
gf77 -c shmop.F -lfs -I/usr/include/gls
ld ld.cmd /lib/crt0.o common.o shmop.o -o shmop   -L/usr/lib/gls -lf -lfs -lm -lc

---------- end ----------
```

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

SEE ALSO

aread(2F), exec(2F), exit(2F), fork(2F), intro(2F), shmctl(2F), shmget(2F), cintrio(7).

DIAGNOSTICS

Upon successful completion, the return value is as follows:

shmat returns the data segment start address of the attached shared memory segment.

shmdt returns a value of 0.

Otherwise, a negative value indicating the error is returned.

NAME

sigaction - examine or change signal action.

SYNOPSIS

#include <sysf/signal.i>

integer*4 sigaction,sig
record/sigaction/act,oact
iretval = sigaction(sig,act,oact)

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process.

All signals have the same priority. Signal routines invoked by *sigaction*(2) execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global "signal mask" defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a *sigprocmask*(2) call, a *sigsuspend*(2) call, or when a signal is delivered to the process.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* or *ignored* by the process then it is delivered to the process. When a signal is calculated (as described below), the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself (see *setjump*(*3C*) or *sigsetjump*(*3C*)).

sigaction allows the calling process to examine or specify the action to be taken on delivery of a signal. *sig* specifies the signal number. The sigaction structure is defined in <signal.i>:

```
STRUCTURE/sigaction/
 INTEGER*4 SA_HANDLER
 RECORD/sigset_t/SA_MASK
 INTEGER*4 SA_FLAGS
END STRUCTURE
```

If *act* is not NULL, it points to a structure specifying the action to be taken when the signal is delivered. If *oact* is not NULL, the action previously associated with the signal is stored in the location pointed to by *oact*. If *act* is NULL, signal handling is unchanged; thus if *act* is NULL, *sigaction* can be used to inquire about the current handling of a given signal.

The **sa_flags** field of *act* can be used to modify the delivery of a specific signal. If *sig* is SGHCHILP and the SA_CLDSTOP bit is set in **sa_flags,** SIGCHILP will be generated if a child process stops.

When a signal is caught by a signal-catching function, a new signal mask is calculated and installed for the duration of the signal-catching function or until *sigprocmask* or *sigsuspend* is called. This mask is formed by taking the union of the current signal mask and the set associated with the action for the signal being delivered (i.e., *sa_mask*), then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested by another call to *sigaction* or until one of the *exec* functions is called.

SIGKILL and **SIGSTOP** cannot be caught or ignored. **SIGCONT** cannot be ignored. The set of signals specified in **sa_mask** is not allowed to block these signals. This is silently enforced.

If *sigaction* fails, no new signal handler is installed.

The following signal names are listed in the include file <sysf/signal.i> :

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGABRT		
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmention violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user defined signal 1
SIGUSR2	17	user defined signal 2
SIGCHILD	18@	child status has changed
SIGPWR	19	power-fail restart
SIGWINCH	20@	window size change
SIGPOLL	22	pollable event occurred
SIGSTOP	23+	stop(cannot be caught, blocked, or ignored)
SIGTSTP	24+	stop signal generated from keyboard
SIGCONT	25@	continue after stop (cannot be blocked)
SIGTTIN	26+	background read attempted from control terminal
SIGTTOU	27+	background write attempted to control terminal
SIGURG	33@	urgent condition present on socket
SIGVTALRM	37	virtual time alarm (see *setitimer*(2))
SIGPROF	38	profiling timer alarm (see *setitimer*(2))

The starred signals (*) in the list above cause a core image if not caught or ignored.

The default action for a signal may be reinstated by setting **sv_handler** to **SIG_DFL;** this default is termination (with a core image for starred signals) except for signals marked with @ or +. Signals marked with a @ sign are discarded if the action is **SIG_DFL;** signals marked with a plus sign (+) cause the process to stop. If **sv_handler** is **SIG_IGN**, the signal is subsequently ignored, and pending instances of the signal are discarded.

After a *fork*(2), the child inherits all signals, the signal mask, and the signal stack.

exec(2) resets all caught signals to default action. Ignored signals remain ignored; the signal mask remains the same.

RETURN VALUE

Upon successful completion, a value of zero is returned. Otherwise a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, *sigaction* will return -1 and set *errno* to the corresponding value:

[EINVAL] The value of *sig* is not a valid signal number, or an attempt was made to supply an action for a signal that canot be caught or ignored.

[EFAULT] *act* and/or *oact* is an invalid address.

SEE ALSO

exec(2),kill(2),sigsetops(2),sigprocmask(2),sigsuspend(2),sigvec(2)

NOTES

Currently on 88k machine only.

NAME

signal - specify what to do upon receipt of a signal

SYNOPSIS

```
#    include <sysf/signal.i>
     integer*4 signal, sig, func
     external func
     iretval = signal (sig, func)
```

DESCRIPTION

signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *sig* specifies the signal and *func* specifies the choice.

sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1]	trace trap (not reset when caught)
SIGIOT	06[1]	IOT instruction
SIGEMT	07[1]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2F
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail
SIGPOLL	22[3]	selectable event pending

func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. **SIG_DFL**, and **SIG_IGN**, are defined in the include file *signal.i*. Each has a unique value that matches no declarable function.

The actions prescribed by the values of *func* are as follows:

SIG_DFL - terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2F). See NOTE [1] below.

SIG_IGN - ignore signal

The signal *sig* is to be ignored.

Note: the signal **SIGKILL** cannot be ignored.

function address - catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*(2F), a *write*(2F), an *open*(2F), or an *ioctl*(2F) system call on a slow device (like a terminal; but not a file), during a *pause*(2F) system call, or during a *wait*(2F) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return to the calling process with EINTR.

signal will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

Note: The signal **SIGKILL** cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending **SIGKILL** signal.

signal will fail if *sig* is an illegal signal number, including **SIGKILL**. [EINVAL]

EXAMPLE

```
      program signal
#     include <sysf/errno.i>
#     include <sysf/signal.i>

      integer*4 signal, sig, sighan, alarm, pause, iretval
      integer*4 prevwtd, remtime
      integer*4 sig_caught
      common /signal_c/ sig_caught
      external sighan

c  Set to terminate on quit signal, setup alarm signal handler

      prevwtd = signal (SIGALRM, sighan)
      if (prevwtd .lt. 0) stop 'sigalrm'
      prevwtd = signal (SIGQUIT, SIG_DFL)
      if (prevwtd .lt. 0) stop 'sigquit'

c  Cause alarm to occur in 5 seconds

      remtime = alarm (5)
      if (remtime .lt. 0) stop 'alarm error'

c  Pause until some signal occurs.
c  If the signal that occurs is due to an alarm, print
c  the signal caught (otherwise, do not expect to return
c  from pause).

      iretval = pause ()
      write (*,*) 'signal caught:', sig_caught
      end

      subroutine sighan (signo)
      integer*4 signo, sig_caught
      common /signal_c/ sig_caught

c  Place the signal which got us here into common

      sig_caught = %loc (signo)
      return
      end
```

NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of '0666'O modified by the file creation mask [see *umask*(2F)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

[2] For the signals **SIGCLD** and **SIGPWR**, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

SIG_DFL - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2F)].

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

In addition, **SIGCLD** affects the *wait*, and *exit* system calls as follows:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value *iretval* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2F)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

SEE ALSO

kill(1), intro(2F), kill(2F), pause(2F), ptrace(2F), wait(2F), setjmp(3C), sigset(2F).

DIAGNOSTICS

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of SIG_ERR indicating the error is returned. SIG_ERR is defined in the include file *signal.i*.

NAME

sigpending - examine pending signals

SYNOPSIS

#include <signal.h>

integer*4 sigpending
record/sigset_t/set
iretval = sigpending(set)

DESCRIPTION

sigpending stores the set of signals that are blocked from delivery and pending for the calling process at the location pointed to by *set.*

RETURN VALUE

Upon successful completion, zero is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

If the following condition occurs, *sigpending* will return -1 and set *errno* to the corresponding value:

[EFAULT] *set* points to an invalid address.

SEE ALSO

sigsetops(2),sigprocmask(2)

NOTES

Currently on 88k machine only.

NAME

sigprocmask - examine and change blocked signals

SYNOPSIS

#include <signal.h>

integer*4 sigprocmask,how
record/sigset_t/set,oset
iretval = sigprocmask(how,set,oset)

DESCRIPTION

sigprocmask allows the calling process to examine or change its signal mask. If the value of *set* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicated the manner in which the set is changed. The permitted values for *how* are:

SIG_BLOCK
The resulting set will be the union of the current set and the signal set pointed to by *set*

SIG_UNBLOCK
The resulting set will be the intersection of the current set and the complement of the signal set pointed to by *set.*

SIG_SETMASK
The resulting set will be the signal set pointed to by *set.*

If *oset* is not NULL, the previous mask is stored at the location pointed to by *set.* If the value of *set* is NULL, the value of *how* is ignored and the process's signal mask is unchanged. When *set* is NULL, *sigprocmask* can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to *sigprocmask,* at least one of those signals will be delivered before *sigprocmask* returns.

SIG_KILL and SIG_SIGSTOP cannot be caught or ignored. SIGCONT cannot be ignored. It is not possible to block these signals. This is silently enforced.

RETURN VALUE

Upon successful completion, zero is returned. Otherwise, -1 is returned and *errno* is set to indicate the error. If *oset* contains a valid address, its contents will contain the previous signal mask.

ERRORS

If the following condition occurs, *sigprocmask* will return -1 and set *errno* to the corresponding value.

[EINVAL] The value of *how* is invalid.

[EFAULT] *Set* or *oset* point to an invalid address.

SEE ALSO

sigaction(1), sigpending(2), sigsetops(3P), sigsuspend(2)

NOTES

Currently on 88k machine only.

NAME

sigset, sighold, sigrelse, sigignore, sigpause - signal management

SYNOPSIS

```
#    include <sysf/signal.i>
     integer*4 sigset, sig, func
     external func
     iretval = sigset (sig, func)

     integer*4  sighold, sig
     iretval = sighold (sig)

     integer*4 sigrelse, sig
     iretval = sigrelse (sig)

     integer*4 sigignore, sig
     iretval = sigignore (sig)

     integer*4 sigpause, sig
     iretval = sigpause (sig)
```

DESCRIPTION

These functions provide signal management for application processes. *sigset* specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

sig can be assigned any one of the following values except SIGKILL. Machine or implementation dependent signals are not included (see *NOTES* below).

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (can not be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2
SIGCLD	death of a child (see *WARNING* below)
SIGPWR	power fail (see *WARNING* below)
SIGPOLL	selectable event pending (see *NOTES* below)

See below under SIG_DFL regarding asterisks (*) in the above list.

The following values for the system-defined actions of *func* are also defined in <*signal.i*>. Each has a unique value that matches no declarable function.

SIG_DFL - default system action

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2F). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of '0666'O modified by the file creation mask [see *umask*(2F)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN - ignore signal
Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD - hold signal
The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process will call this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* will be passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action will be set to SIG_HOLD. During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process will resume execution at the point it was interrupted. However, when a signal is caught during a *read*(2F), a *write*(2F), an *open*(2F), or an *ioctl*(2F) system call during a *sigpause* system call, or during a *wait*(2F) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler will be executed and then the interrupted system call may return with *iretval* set to EINTR.

sighold and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to SIG_IGN (see above).

sigpause suspends the calling process until it receives a signal, the same as *pause*(2F). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal. *sigset* will fail if one or more of the following are true:

[EINVAL] *sig* is an illegal signal number (including SIGKILL) or the default handling of *sig* cannot be changed.

[EINTR] A signal was caught during the system call *sigpause*.

EXAMPLE

```
      program sigset
#     include <sysf/errno.i>
#     include <sysf/signal.i>

      integer*4 sigset, sig, sighan, alarm, iretval
      integer*4 sighold, sigpause, i, j
      integer*4 prevwtd, remtime
      integer*4 sig_caught
      common /signal_c/ sig_caught
      external sighan

c Set to terminate on quit signal, setup alarm signal handler

      prevwtd = sigset (SIGALRM, sighan)
      if (prevwtd .lt. 0) stop 'sigalrm'
      prevwtd = sigset (SIGQUIT, SIG_DFL)
      if (prevwtd .lt. 0) stop 'sigquit'

c Cause alarm to occur in 5 seconds

      sig_caught = -1
      remtime = alarm (5)
      if (remtime .lt. 0) stop 'alarm error'

c Prevent signal from occurring just yet, delay (hopefully > 5 secs)

      iretval = sighold (SIGALRM)
      if (iretval .lt. 0) write (*,*) 'sighold error:', iretval

      do 100 i = 1, 2000
      do 100 j = 1, 2000
      if (sig_caught .ge. 0) write (*,*) 'sighold not working'
100   continue

c Allow alarm signal to occur and pause until some signal occurs.
c If the signal that occurs is due to an alarm, print
c the signal caught (otherwise, do not expect to return).

      iretval = sigpause (SIGALRM)
      write (*,*) 'signal caught:', sig_caught
      end

      subroutine sighan (signo)
      integer*4 signo, sig_caught
      common /signal_c/ sig_caught

c Place the signal which got us here into common

      sig_caught = %loc (signo)
      return
      end
```

NOTES

SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(2F)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl*(2F) call [see *streamio*(7)]. Otherwise, the process will never receive **SIGPOLL**.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signal **SIGKILL** can not be changed from the default system action.

Specific implementations may have other implementation-defined signals. Also, additional implementation-defined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume execution at the point of interruption.

The signal type **SIGSEGV** is reserved for the condition that occurs on an invalid access to a data object. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(2F) and *pause*(2F), should not be used in conjunction with these routines for a particular signal type.

SEE ALSO

kill(2F), pause(2F), signal(2F), wait(2F), setjmp(3C).

DIAGNOSTICS

returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned. SIG_ERR is defined in <*signal.i*>.

For the other functions, upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

WARNING

Two signals that behave differently than the signals described above exist in this release of the system:

SIGCLD	death of a child (reset when caught)
SIGPWR	power fail (not reset when caught)

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL - ignore signal
: The signal is to be ignored.

SIG_IGN - ignore signal
: The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2F)].

function address - catch signal
: If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

The **SIGCLD** affects two other system calls [*wait*(2F), and *exit*(2F)] in the following ways:

wait If the *func* value of **SIGCLD** is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of *iretval* that is set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to SIG_IGN , the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NAME

sigaddset, sigdelset, sigismember, sigfillset - manipulate signal sets

SYNOPSIS

#include <signal.h>

integer*4 sigaddset,signo
record/sigset_t/set
iretval = sigaddset(set,signo)
record/sigset_t/set

integer*4 sigdelset,signo
record/sigset_t/set
iretval = sigdelset(signo)
record/sigset_t/set

integer*4 sigismember
record/sigset_t/set
iretval = sigismember(set)

integer*4 sigsfillset
iretval = sigfillset(set)

DESCRIPTION

sigaddset adds the signal specified by *signo* to the set pointed to by *set.*

sigdelset deletes the signal specified by *signo* from the set pointed to by *set.*

This system defines the following signals:

SIGABRT	SIGTERM	SIGIOT
SIGALRM	SIGUSR1	SIGEMT
SIGFPE	SIGUSR2	SIGBUS
SIGHUP	SIGCHILD	SIGSYS
SIGILL	SIGCONT	SIGPWR
SIGINT	SIGSTOP	SIGPOLL
SIGKILL	SIGTSTP	SIGURG
SIGPIPE	SIGTTIN	SIGWINCH
SIGQUIT	SIGTTOU	SIGVTALRM
SIGEGV	SIGTRAP	SIGPROF

sigfillset initializes the signal set pointed to by *set* such that all signals listed above are included.

sigfillset tests whether the signal specified by *signo* is a member of the set pointed to by *set.* Applications should call *sigemptyset(3P)* or *sigfillset(3P)* for each object of type **sigset_t(3P)** before any other use of the object.

RETURN VALUE

Upon successful completion *sigismember* returns 1 if the specified signal is a member of the specified set and zero if it is not. Upon successful completion, each of the other functions returns zero. For all the above functions, if an error is detected, the function will return -1 and set *errno* to indicate the error.

ERRORS

If the following condition occurs, the function shall return -1 and set *errno* to the corresponding value.

[EINVAL] The value of *signo* is not a valid signal number.

SEE ALSO

sigaction(1), sigpending(2), sigprocmask(3P), sigsuspend(2),sigvec(2)

NOTES

Currently on 88k machine only.

NAME

sleep - suspend execution for interval

SYNOPSIS

integer*4 sleep,seconds
iretval= sleep(seconds)

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Note that catching signals while within a *sleep* call is not recommended, see the warning below. The suspension time may also be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*. If the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the *sleep* routine returns. But if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening *sleep*.

SEE ALSO

alarm(2F), pause(2F), signal(2F), sighold(2F), sigrelse(2F), sigset(2F).

WARNING

Before calling the *sleep* any signals which might be caught during the *sleep* should be held via *sighold*(2). After the *sleep* the *sigrelse*(2) system call should be used to restore the previous signal handling. These actions are necessary because there is no guarantee that a signal handler which catches a signal during the *sleep* will complete before the *sleep* timeout expires. On expiration of the timeout, any signal handling still in progress is aborted and the system may be left in a state such that any further signals of that type will not be caught. The exception to this rule is SIGALRM, as *sleep* uses this internally and will automatically save and restore any SIGALRM handler.

NAME

socket - create an endpoint for communication

SYNOPSIS

#include <sysf/types.i>
#include <sysf/socket.i>

integer*4 s, socket, af, type, protocol
s = socket(af, type, protocol)

DESCRIPTION

socket creates an endpoint for communication and returns a descriptor.

The *af* parameter specifies an address family in which addresses specified in later operations using the socket should be interpreted. These families are defined in the include file **<sysf/socket.i>**. The currently understood family is

AF_INET (ARPA Internet protocols),

The socket has the indicated *type,* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network protocols and interfaces.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given address family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the *communication domain* in which communication is to take place; see *protocols*(4).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2F) call. Once connected, data may be transferred using *read*(2F) and *write*(2F) calls or some variant of the *send*(2F) and *recv*(2F) calls. When a session has been completed a *close*(2F) may be performed. Out-of-band data may also be transmitted as described in *send*(2F) and received as described in *recv*(2F). An *ioctl*(2F) call with the SIOCSPGRP command can be used to specify a process group to receive a SIGURG signal when out-of-band data arrives.

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets *warm* by forcing transmissions roughly every minute in the absence of other activity. An ETIMEDOUT error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (e.g. 3 minutes).

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *sendto*(2F) calls. Datagrams are generally received with *recvfrom*(2F), which returns the next datagram with its return address. Optionally, one may *connect*(2F) to a SOCK_DGRAM peer, in which case *send*(2F) and *write*(2F) may be used on the socket, to send packets to only that peer, until another *connect* is issued to change the association.

The operation of sockets is controlled by socket level *options*. These options are defined in the file <sysf/socket.i>. *setsockopt*(2F) and *getsockopt*(2F) are used to set and get options, respectively.

RETURN VALUE

A negative value is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EAFNOSUPPORT] The specified address family is not supported.

[ESOCKTNOSUPPORT] The specified socket type is not supported in the specified address family, or no address family can be found which supports it.

[EPROTONOSUPPORT] No protocol is supported or the specified protocol is not supported within this address family. Possibly because the Network Services Extension has not been installed, or the protocol has not been initialized yet (by invocation of *tpid*(1M)).

[EMFILE] The per-process descriptor table is full.

[ENFILE] The system file table is full.

[EACCESS] Permission to create a socket of the specified type and/or protocol is denied.

[ENOBUFS] Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

EXAMPLE

The following typifies a server/client relationship utilizing sockets.

```
          PROGRAM client

C   The include files need to be modified to reflect
C   fortran include files directory

#include "sysf/types.i"
#include "sysf/socket.i"
#include "sysf/in.i"

     INTEGER*4    datsock,s,cc,i,j,k,y,lenparam
     INTEGER*4    socket,bind,send,accept,gethostby,connect,read
     INTEGER*4    getsockopt,getsockname,getpeername
     CHARACTER*6  machname
     CHARACTER*40 buf
     RECORD /sockaddr_in/ server,client
     RECORD /hostent/  host
     RECORD /sockaddr/  temp


C      First create socket

       datsock = socket(AF_INET, SOCK_STREAM, 0)
       IF (datsock .lt. 0)  THEN
         write(*,*) 'ERROR opening data socket',datsock
       ENDIF
```

```
C       Let machname equal name of target machine
        machname='glsdev'
        lenparam=16

C       This goes into a c subroutine that calls gethostbyname
C       from inet library and also performs a memcpy from host
C       to server

        j = gethostby(host,machname,server)

        IF (j .lt. 0) THEN
        write(*,*) 'unknown host ',machname
        ENDIF
C       server.sin_port is port of your choice

        server.sin_port = 1225
        server.sin_family = AF_INET

        j=getsockopt(datsock,SOL_SOCKET,SO_REUSEADDR,buf,4)
        IF (j .lt. 0) THEN
          write(*,*) 'ERROR is getsockopt ',j
        ENDIF

        j=getsockname(datsock,temp,lenparam)
        IF (j .lt. 0) THEN
          write(*,*) 'ERROR is getsockname ',j
        ENDIF

        j=connect(datsock, server, lenparam )
        IF (j .lt. 0) THEN
          write(*,*) 'ERROR connect data socket',j
        ENDIF

        write(*,*) 'Client - got a new socket ', datsock

        j=getpeername(datsock,server,lenparam)
        IF (j .lt. 0) THEN
          write(*,*) 'ERROR is getpeername ',j
        ENDIF
        cc=0
        DO WHILE (cc .GE. 0)
          write(*,*)'buf is ',buf
          cc=read(datsock,buf,40)
          write(*,*)'Read bytes :',cc
        END DO
        write(*,*) 'Client - read bytes : ',cc

C       Now close socket

        close(datsock)
        return
        END
```

```
          PROGRAM server

C  The include files need to be modified to reflect fortran
C  include files directory

#include "sysf/types.i"
#include "sysf/socket.i"
#include "sysf/in.i"

          INTEGER*4    datsock,s,cc,i,j,k,a
          INTEGER*4    socket,bind,send,accept,gethostnm
          INTEGER*4    setsockopt
          CHARACTER*7  machname
          CHARACTER*40 buf
          RECORD /sockaddr_in/ server,client
          RECORD /hostent/  host

C       First create  socket

        datsock = socket(AF_INET, SOCK_STREAM, 0)
        IF (datsock .lt. 0)   THEN
          write(*,*) 'error opening data socket',datsock
        ENDIF

C       Next is to fill what is to be sent through socket
        buf='characters'

C       Let machname equal name of target machine
        machname='glsdev'

C       This goes into a c subroutine that calls gethostbynm
C       from inet library and also performs a memcpy from host
C       to server

        i = gethostnm(host,machname)
        IF (i .lt. 0) THEN
         write(*,*) 'unknown host ',machname
        ENDIF

C       server.sin_port is port of your choice
        server.sin_port = 1225

        server.sin_family = AF_INET

        j=setsockopt(datsock,SOL_SOCKET,SO_KEEPALIVE,1,4)
        IF (j .lt. 0) THEN
         write(*,*) 'ERROR setsockopt ',j
        ENDIF

        j=bind(datsock, server, 16 )
        IF (j .lt. 0) THEN
          write(*,*) 'error binding data socket',j
        ENDIF
```

```
        j=listen(datsock, 1)
        IF ( j .lt. 0) THEN
          write(*,*) 'error listening ',j
        ENDIF
        s = accept(datsock, client, k)
        IF ( s .lt. 0) THEN
          write(*,*) 'error listening ',s
        ENDIF
        write(*,*) 'Server - got a new socket ', s

        a=0
        DO WHILE (a .LT.25)
        write(*,*) 'Sending character and 40 bytes'
        cc=send(s,buf,40,0)
        i=sleep(2)
        a=a+1
        END DO
C       Close socket
        close(s)
        return
        END
```

The following are the C interface calls that will work interactively with the above Fortran programs.

```
            gethostby_.c

#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <errno.h>
#include <malloc.h>

extern int          errno;

int   gethostby_(result,xname,serv,xlen)
      struct hostent  *result;
      char *xname;
      struct sockaddr_in  *serv;
      int xlen;
{
       struct hostent  *temp;
       char            *pbuf;
       int i,j,k;

       if ((pbuf = malloc(xlen+1))==NULL)
          return(-ENOMEM);
       g_char(xname, xlen, pbuf);
       temp = gethostbyname(xname);
       if (temp==NULL)
         return(-errno);
```

```
        memcpy(result,temp,sizeof (struct hostent));
        free(pbuf);
        memcpy(&serv->sin_addr,result->h_addr,result->h_length);
    }

                    gethostnm-FORTRAN interface to the system
                gethostbyname call

#include <stdio.h>
#include <netdb.h>
#include <errno.h>
#include <malloc.h>

extern int          errno;

int                 gethostnm_(result,xname,xlen)
        struct hostent  *result;
        char *xname;
        int xlen;
{
        struct hostent  *temp;
                    char      *pbuf;
        int i,j,k;

        if ((pbuf = malloc(xlen+1))==NULL)
          return(-ENOMEM);
                    g_char(xname, xlen, pbuf);
                    temp = gethostbyname(xname);
        if (temp==NULL)
          return(-errno);
        memcpy(result,temp,sizeof (struct hostent));
                    free(pbuf);
                    return((int)result);
}
```

SEE ALSO

accept(2F), bind(2F), connect(2F), getsockname(2F), getsockopt(2F), ioctl(2F), listen(2F), read(2F), recv(2F), select(2F), send(2F), shutdown(2F), write(2F), tcp(7), udp(7)

NAME

stat, fstat - get file status

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/stat.i>

     integer*4 stat
     character*SIZE path
     record /stat/ buf
     iretval = stat (path, buf)

     integer*4 fstat, fildes
     record /stat/ buf
     iretval = fstat (fildes, buf)
```

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *stat* obtains information about the named file.

Note that in a Remote File Sharing environment, the information returned by *stat* depends upon the user/group mapping set up between the local and remote computers. [See *idload*(1M)].

fstat obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
integer*2  st_mode    !File mode [see mknod(2F)]
integer*2  st_ino     !Inode number
integer*2  st_dev     !ID of device containing
                      !a directory entry for this file
integer*2  st_rdev    !ID of device
                      !This entry is defined only for
                      !character special or block special files
integer*2  st_nlink   !Number of links
integer*2  st_uid     !User ID of the file's owner
integer*2  st_gid     !Group ID of the file's group
integer*4  st_size    !File size in bytes
integer*4  st_atime   !Time of last access
integer*4  st_mtime   !Time of last data modification
integer*4  st_ctime   !Time of last file status change
                      !Times measured in seconds since
                      !00:00:00 GMT, Jan. 1, 1970
```

st_mode The mode of the file as described in the *mknod*(2F) system call.

st_ino This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. Its value may be used as input to the *ustat*(2F) system call to determine more information about this file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. For pipes or fifos, this is the count of the data currently in the file. For block special or character special, this is not defined.

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2F), *mknod*(2F), *pipe*(2F), *utime*(2F), and *read*(2F).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2F), *mknod*(2F), *pipe*(2F), *utime*(2F), and *write*(2F).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2F), *chown*(2F), *creat*(2F), *link*(2F), *mknod*(2F), *pipe*(2F), *unlink*(2F), *utime*(2F), and *write*(2F).

stat will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *buf* or *path* points to an invalid address.

[EINTR] A signal was caught during the *stat* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

fstat will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid open file descriptor.

[EFAULT] *buf* points to an invalid address.

[ENOLINK] *fildes* points to a remote machine and the link to that machine is no longer active.

EXAMPLE

```
        program stat
#       include <sysf/types.i>
#       include <sysf/stat.i>
#       include <sysf/fcntl.i>

        integer*4 stat, fstat, fildes
        character*40 path
        record /stat/ buf_path, buf_file
        integer*4 open, iretval

c   Get extended status of file using pathname

        path = '../example/stat.F'
        iretval = stat (path, buf_path)
        if (iretval .lt. 0) write (*,*) 'stat error:', iretval

c   Get extended status of file from file descriptor

        fildes = open (path, O_RDONLY)
        if (fildes .lt. 0) write (*,*) 'open error:', fildes
        iretval = fstat (fildes, buf_file)
        if (iretval .lt. 0) write (*,*) 'fstat error:', iretval

c   Print some of the information from both structures

        write (*,9000) buf_path.st_ino, buf_path.st_size, buf_path.st_mode
        write (*,9000) buf_file.st_ino, buf_file.st_size, buf_file.st_mode

9000 format (' Inode:', I5,', Size in bytes:',I9,', Mode:',o8)
        end
```

SEE ALSO

chmod(2F), chown(2F), creat(2F), link(2F), mknod(2F), pipe(2F), read(2F), time(2F), unlink(2F), utime(2F), write(2F).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

statfs, fstatfs - get file system information

SYNOPSIS

```
# include <sysf/types.i>
# include <sysf/statfs.i>

integer*4 statfs, len, fstyp
character*SIZE path
record /statfs/ buf
iretval = statfs (path, buf, len, fstyp)

integer*4 fstatfs, len, fstyp, fildes
record /statfs/ buf
iretval = fstatfs (fildes, buf, len, fstyp)
```

DESCRIPTION

statfs returns a "generic superblock" describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which will be filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *Len* must be no greater than the number of bytes in the *statfs* structure and ordinarily it will contain exactly that value; if it holds a smaller value the system will fill the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

If the file system of interest is currently mounted, *path* should name a file which resides on that file system. *SIZE* can be any number between and including 1 through 128. In this case the file system type is known to the operating system and the *fstyp* argument must be zero. For an unmounted file system *path* must name the block special file containing it and *fstyp* must contain the (non-zero) file system type. In both cases read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The *statfs* structure pointed to by *buf* includes the following members:

```
integer*2    f_fstyp    !File system type
integer*4    f_bsize    !Block size
integer*4    f_frsize   !Fragment size
integer*4    f_blocks   !Total number of blocks
integer*4    f_bfree    !Count of free blocks
integer*4    f_files    !Total number of file nodes
integer*4    f_ffree    !Count of free file nodes
character*6  f_fname    !Volume name
character*6  f_fpack    !Pack name
```

fstatfs is similar, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *filedes* obtained from a successful *open*(2F), *creat*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call.

statfs obsoletes *ustat*(2F) and should be used in preference to it in new programs.

statfs and *fstatfs* will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *buf* or *path* points to an invalid address.

[EBADF] *fildes* is not a valid open file descriptor.

[EINVAL] *fstyp* is an invalid file system type; *path* is not a block special file and *fstyp* is nonzero; *len* is negative or is greater than the number of bytes in the *statfs* structure.

[ENOLINK] *path* points to a remote machine, and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
        program statfs
#       include <sysf/types.i>
#       include <sysf/statfs.i>
        integer*4 statfs, len, fstyp
        character*40 path
        record /statfs/ buf
        integer*4 iretval

c  Get information about a file system

        path = '/usr'
        len = 38
        fstyp = 0
        iretval = statfs (path, buf, len, fstyp)
        if (iretval .lt. 0) write (*,*) 'statfs error:', iretval

c  Print the information

        write (*,9000) buf.f_fstyp, buf.f_bsize, buf.f_frsize,
&         buf.f_blocks, buf.f_bfree, buf.f_files, buf.f_ffree,
&         buf.f_fname, buf.f_fpack
9000 format (' Type: ', i5,/
&         ' Block size: ', i8,/
&         ' Fragment size: ', i8,/
&         ' Total blocks: ', i8,/
&         ' Total free blocks: ', i8,/
&         ' Total nodes: ', i8,/
&         ' Total free nodes: ', i8,/
&         ' Volume name: ', a6,/
&         ' Pack name: ', a6)
        end
```

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

SEE ALSO

chmod(2F), chown(2F), creat(2F), link(2F), mknod(2F), pipe(2F), read(2F), time(2F), unlink(2F), utime(2F), write(2F), fs(4).

NAME

stime - set time

SYNOPSIS

integer*4 stime, tp
iretval = stime (tp)

DESCRIPTION

stime sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM] *stime* will fail if the effective user ID of the calling process is not super-user.

EXAMPLE

```
        program stime
        integer*4 stime, tp
        integer*4 ftime, iretval

c   Get current time and date

        tp = ftime (%val(0))

c   Bump it by 1 hour

        tp = tp + 3600
        iretval = stime (tp)
        if (iretval .lt. 0) write (*,*) 'stime error:', iretval
        end
```

SEE ALSO

time(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

stkexp - expand the stack region of the data segment

SYNOPSIS

 #   include <sysf/lock.i>

integer*4 stkexp, incr, flags
iretval = stkexp (incr, flags)

DESCRIPTION

stkexp is used to expand the amount of space allocated for the stack region of the data segment. It is recommended that a realtime process should use this call, in conjunction with *plock*(2F) and *resident*(2F), to preallocate sufficient space to contain the maximum size to which its stack can grow. If this call is not used, pages will be added to the stack region of the data segment as needed by natural stack growth, with possible violation of realtime constraints.

incr is the number of bytes to be added to the stack region. It will be rounded up to the nearest page boundary. If (*flags* & STKSZ) is true, then *incr* is the new total size, in bytes, of the stack region. It will be rounded up to the nearest page boundary. (The number of bytes per page, NBPC, is defined in **<sysf/param.i>**.)

The *stkexp* system call will fail without making any change in the allocated space if:

[EAGAIN] The data segment is locked resulting in more resident pages being allocated than are currently available.

[EINVAL] The requested new size is less than the current size.

[ENOMEM] Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2F)).

EXAMPLE

```
      program stkexp
      integer*4 stkexp, incr, flags, oldsize

c  Expand stack by 2048 bytes

      incr = 2048
      flags = 0
      oldsize = stkexp (incr, flags)
      if (oldsize .lt. 0) write (*,*) 'stkexp error:', oldsize
      oldsize = (oldsize + 1023) / 1024
      write (*,9000) 'old stack size:', oldsize

c  Get current stack size

      incr = 0
      oldsize = stkexp (0, 0)
      if (oldsize .lt. 0) write (*,*) '2 stkexp error:', oldsize
      oldsize = (oldsize + 1023) / 1024
      write (*,9000) 'new stack size:', oldsize
9000 format (' ', a20, i4, 'Kb')
      end
```

SEE ALSO

brk(2F), plock(2F), resident(2F).

DIAGNOSTICS

Upon successful completion, *stkexp* returns the old size of the stack region. Otherwise, a negative value indicating the error is returned.

NAME

suspend - suspend the calling process

SYNOPSIS

integer*4 suspend
iretval = suspend()

DESCRIPTION

The calling process is suspended until a *resume*(2F), or *swtch*(2F) is performed by another process.

suspend will fail if the following is true:

[EINTR] A signal was caught during the *suspend* system call.

EXAMPLE

See resume(2F) for an example.

SEE ALSO

setrtgroup(1), resume(1R), getpid(2F), resume(2F), setrt(2F), swtch(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

swtch - switch into a process

SYNOPSIS

integer*4 swtch, pid
iretval = swtch (pid)

DESCRIPTION

pid is the process ID of the target process.

swtch is functionally equivalent to the following code fragment:

```
...
...
ires_retval = resume (target_pid)
isus_retval = suspend ()
...
...
```

The difference being that the overhead of one system call is saved. The target process will resume when it becomes the highest priority process in the system.

For a process to have permission to *swtch* into another process the requesting process must have superuser or realtime privileges which are granted via the *setrt*(2F) system call.

swtch will fail if one or more of the following are true:

[EINVAL] The target process was not suspended via a *suspend*(2F) system call or a *swtch*(2F) system call.

[EPERM] The requesting process does not have the appropriate permission to suspend the target process.

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLE

```
      program swtch
      integer*4 swtch, pid
      integer*4 fork, ppid, getppid
      integer*4 suspend, resume
      integer*4 iretval, i

c  Make a child

      pid = fork ()
      if (pid .eq. 0) goto 2000

c  Print messages back and forth

      do 100 i = 1, 5
      write (*,9000) 'parent', i
      iretval = swtch (pid)
      if (iretval .lt. 0) write (*,9001) 'parent', iretval
100   continue
      iretval = resume (pid)
      if (.true.) stop

c  Here if child
c  Get parent process id

2000 continue

      ppid = getppid ()
      iretval = suspend ()

c  Print message back and forth

      do 2100 i = 1, 5
      write (*,9000) 'child', i
      iretval = swtch (ppid)
      if (iretval .lt. 0) write (*,9001) 'child', iretval
2100 continue
      iretval = resume (ppid)
9000 format ('0', a10, x, i4)
9001 format (' ', a6, ' swtch error:', x, i6)
      end
```

SEE ALSO

setrtusers(1M), resume(1R), getpid(2F), resume(2F), setrt(2F), suspend(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

symlink - makes symbolic link to a file

SYNOPSIS

integer*4 symlink
character*SIZE name1,name2
iretval = symlink(name1,name2)

DESCRIPTION

SIZE can be any number 1 through 128. A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name can be an arbitrary path name; the files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in *errno* and a -1 value is returned.

ERRORS

The symbolic link is made unless one or more of the following are true:

[EPERM] Either value of *name1* or *name2* contains a character with the high-order bit set.

[EPERM] A pathname contains a character with the high-order bit set.

[ENAMETOOLONG] A component of a pathname exceeded NAME_MAX characters, or an entire path-name exceeded PATH_MAX.

[ELOOP] Too many symbolic links were encountered in translating a pathname.

[ENOENT] One of the pathnames specified was too long.

[ENOTDIR] A component of the *name2* prefix is not a directory.

[EEXIST] *name2* already exists.

[EACCES] A component of the *name2* path prefix denies search permission.

[EROFS] The file *name2* would reside on a read-only file system.

[EFAULT] *name1* or *name2* points outside the process' allocated address space.

SEE ALSO

ln(1), link(2), readlink(2), unlink(2)

NOTES

Currently on 88k machine only.

NAME

sync - update super block

SYNOPSIS

integer*4 sync
iretval = sync ()

DESCRIPTION

sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a re-boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

EXAMPLE

```
        program sync
        integer*4 sync, iretval

c   Flush disk data

        iretval = sync ()
        if (iretval .lt. 0) write (*,*) 'sync error:', iretval
        end
```

NAME

sysfs - get file system type information

SYNOPSIS

```
#   include <sysf/fstyp.i>
#   include <sysf/fsid.i>

    integer*4 sysfs, opcode
    character*SIZE fsname
    iretval = sysfs (opcode, fsname)

    integer*4 sysfs, opcode, fs_index
    character*SIZE buf
    iretval = sysfs (opcode, fs_index, buf)

    integer*4 sysfs, opcode
    iretval = sysfs (opcode)
```

DESCRIPTION

sysfs returns information about the file system types configured in the system. The number of arguments accepted by *sysfs* varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are described below:

GETFSIND translates *fsname*, a null-terminated file-system identifier, into a file-system type index. *SIZE* can be any number between and including 1 through 128.

GETFSTYP translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf* of *SIZE* bytes; this buffer must be at least of size **FSTYPSZ** as defined in *<sysf/fstyp.i>*.

GETNFSTYP returns the total number of file system types configured in the system.

sysfs will fail if one or more of the following are true:

[EINVAL] *fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.

[EFAULT] *buf* or *fsname* point to an invalid user address.

EXAMPLE

```
        program sysfs
#       include <sysf/fstyp.i>
#       include <sysf/fsid.i>

#       if FSTYPSZ > 20
#       define BUFSIZ 20
#       else
#       define BUFSIZ FSTYPSZ
#       endif

        integer*4 sysfs, opcode, fs_index
        character*FSTYPSZ buf
        integer*4 iretval, i, j

c   Get file system names

        do 100 i = 0, 20
        opcode = GETFSTYP
        fs_index = i
        iretval = sysfs (opcode, fs_index, buf)
        if (iretval .ge. 0) write (*,9000) fs_index, buf (1:BUFSIZ)
100     continue
9000 format (' Index:', i3, ' Name: ', a BUFSIZ)
        end
```

DIAGNOSTICS

Upon successful completion, *sysfs* returns the file-system type index if the *opcode* is **GETFSIND**, a value of 0 if the *opcode* is **GETFSTYP**, or the number of file system types configured if the *opcode* is **GETNFSTYP**. Otherwise, a negative value indicating the error is returned.

NAME

sysm68k - machine specific functions

SYNOPSIS

 # include <sysf/sysm68k.i>

integer*4 sysm68k, cmd, arg1, arg2
iretval = sysm68k (cmd, arg1, arg2)

DESCRIPTION

sysm68k implements machine specific functions. The *cmd* argument determines the function performed. The number of arguments expected is dependent on the function.

Command S68ADDMEM

When *cmd* is S68ADDMEM, the argument is used as the number of pages to add to the free list. Note that this command is available to the superuser only. If more pages are added with this command than were deleted with S68DELMEM, only the amount previously deleted will be added back.

Command S68BCACHEOFF

When *cmd* is S68BCACHEOFF, the cache is disabled. Note this command is available to the superuser only.

Please note that board caching is not available on the MVME147. This command will fail if attempted when executing on the MVME147.

Command S68BCACHEON

When *cmd* is S68BCACHEON, argument is used as the value to be written to the cache mask register and the cache is then enabled. Note this command is available to the superuser only.

Please note that board caching is not available on the MVME147. This command will fail if attempted when executing on the MVME147.

Command S68BRDSTAT

When *cmd* is S68BRDSTAT, a structure containing processor board specific status is returned. Two arguments are required. The first argument is a pointer to a *boardid* structure into which information is placed concerning the processor board. Refer to **sysf/mvmecpu.i** for a declaration of this structure. The second argument is the number of bytes to be transferred. This should be the size of the structure for the first argument or smaller. This command may be executed by any user. This command is only available on some of the newer processor boards. Refer to the specific BUG manual for the processor board being used and to the ROM calls available, which may return similar detailed information about the processor board.

Command S68CACHERD

When *cmd* is executed the current mask used for setting the cacr (cache control register) is returned. No other arguments are necessary. This command may be executed by any user. Refer to the next command, S68CACHESET, and to the *Motorola MC68030 User's Manual* for a description of the control bit values returned.

Command S68CACHESET

When *cmd* is S68CACHESET, the processor instruction and data caches may be enabled or disabled with various options. A value for setting the cacr (cache control register) is passed as the only argument. If the command is successful, this value is returned as the return value. This command may only be executed as superuser. The following control bits are available in the cacr with the MC68030 microprocessor:

CACR_EI	'0001'x	Enable Instruction Cache
CACR_FI	'0002'x	Freeze Instruction Cache
CACR_CEI	'0004'x	Clear Entry in Instruction Cache
CACR_CI	'0008'x	Clear Instruction Cache
CACR_IBE	'0010'x	Instruction Burst Enable
CACR_ED	'0100'x	Enable Data Cache
CACR_FD	'0200'x	Freeze Data Cache
CACR_CED	'0400'x	Clear Entry in Data Cache
CACR_CD	'0800'x	Clear Data Cache
CACR_DBE	'1000'x	Data Burst Enable
CACR_WA	'2000'x	Write Allocate

Refer to **sysf/sysm68k.i** for a declaration of these defines and other cacr related information. Any flags relating to the data cache are only available on the MC68030. The meaning and use of these flags is described in the *Motorola MC68030 User's Manual*. However, when running under REAL/IX only certain limited combinations of the above will be legal. Only the following flags will be allowed to be turned on (or set to a 1):

CACR_EI	Enable Instruction Cache
CACR_CI	Clear Instruction Cache
CACR_IBE	Instruction Burst Enable
CACR_ED	Enable Data Cache
CACR_CD	Clear Data Cache
CACR_DBE	Data Burst Enable
CACR_WA	Write Allocate

Command S68CONT

When *cmd* is S68CONT, the kernel will continue with the instruction that was interrupted by a bus error signal to the calling routine.

Command S68CPUBRD

When *cmd* is S68CPUBRD, no arguments are expected. A value corresponding to the processor board on which the operating system is running is returned. Refer to **sysf/mvmecpu.i** for the mnemonic names used for the CPU board values.

Command S68DELMEM

When *cmd* is S68DELMEM, the argument is used as the number of pages to delete from the free list. Note that this command is available to the superuser only. This command is intended to allow stress tests to verify system behavior with low free memory.

Command S68FPEX

When *cmd* is S68FPEX, the floating-point operand that caused the floating-point exception is returned to the user at the address specified by *arg1*. This command should be executed only after a floating-point exception has been indicated to the caller, otherwise an undetermined operand will be returned to the user.

Command S68FPHW

When *cmd* is S68FPHW, a flag is set at the address specified by the argument that indicates whether or not the floating-point hardware chip is present on the system. A flag of NOFPHW will be stored if there is not a floating-point chip, a flag of MC68881 will be stored if there is.

Command S68ICACHEOFF

When *cmd* is S68ICACHEOFF, the internal cache of the MC68030 chip is disabled. Note that this command is available to the superuser only.

Command S68ICACHEON

When *cmd* is S68ICACHEON, the internal cache of the MC68030 chip is enabled. Note that this command is available to the superuser only.

Command S68MEMSIZE

When *cmd* is S68MEMSIZE, no arguments are expected. The size of the virtual memory space and the amount of physical memory (in bytes) are returned.

Command S68RTODC

When *cmd* is S68RTODC, the value of the realtime clock (rtc) is returned to the address specified by the argument. If there is no realtime clock on the system, the current time is returned. Note that this command is available to the superuser only.

Command S68SETNAME

When *cmd* is S68SETNAME, the argument is expected to be a pointer to a character string. The system name and node name are set to the character string specified by the argument. Note that this command is available to the superuser only.

Command S68STACK

>>> This system call is obsolete and is included only for compatibility with previous releases.

When *cmd* is S68STACK, the available stack space is increased by the number of bytes (rounded to the nearest page boundary). If this system call succeeds, the new value of the stack pointer is returned.

Command S68STIME

When *cmd* is S68STIME, the argument is used as the new value for the system time and date. The argument contains the time as measured in seconds from **00:00:00 GMT January 1,1970**. Note that this command is only available to the superuser. This command is redundant in that *stime*(2F) may also be used to set the system time but this command is included for compatibility with previous releases.

Command S68SWAP

When *cmd* is S68SWAP, individual swapping areas may be added or deleted, or the current areas determined. The address of an appropriately primed swap buffer is passed as the only argument. (Refer to **sysf/swap.i** header file for details of loading the buffer.)

The format of the swap buffer is:

```
structure /swpi_t/
      integer*1  si_cmd     !command: list, add, delete
      integer*4  si_buf     !swap file path pointer
      integer*4  sw_swplo   !start block
      integer*4  si_nblks   !swap size
end structure
```

Note that the add and delete options of the command may only be exercised by the superuser.

Typically, a swap area is added by a single call to *sysm68k*. First, the swap buffer is primed with appropriate entries for the structure members. Then *sysm68k* is invoked.

```
# include <sysf/sysm68k.i>
# include <sysf/swap.i
record /swpi_t/ swapbuf
sysm68k(S68SWAP, swapbuf)
```

Command S68TODCSTAT

When *cmd* is S68TODCSTAT, no arguments are expected. The integer return value reflects the status of the time-of-day clock. A useful time-of-day clock is indicated by a return value equal to GOOD_TODC.

Command S68WPOSTOFF

When *cmd* is S68WPOSTOFF, write-posting is disabled. No other arguments are necessary. This command may only be executed by superuser. This command is not available on the MVME147 processor board, however, it may well become available in future processor boards.

Command S68WPOSTON

When *cmd* is S68WPOSTON, write-posting is enabled. No other arguments are necessary. This command may only be executed by superuser. This command is not available on the MVME147 processor board, however, it may well become available in future processor boards.

Command R68VMEADDR

When *cmd* is R68VMEADDR, important VME bus memory map information is returned to the user. This information identifies the starting physical address and size, in bytes, of each range in question. The information is returned via a typedef'ed structure, supplied by the user, which is pointed to by *arg1*. Below is an code fragment illustrating proper use:

```
# include <sysf/sysm68k.i>
# include <sysf/vme.i>/
record /vme_t/ vme
sysm68k(R68VMEADDR, vme, 0)
```

The second argument identifies which VME chassis is of interest to the user. For machines with only one VME chassis, this argument must be zero.

vme.i contains the following:

```
#define VMESIZE     20 !#array elements (with reserve space)

#define VMEA16D16    0 !16bit address, 8/16 bit data
#define VMEA24D16    1 !24bit address, 8/16 bit data
#define VMEA24D32    2 !24bit address, 32 bit data
#define VMEA32D16    3 !32bit address, 8/16 bit data
#define VMEA32D32    4 !32bit address, 32 bit data
```

```
!This structure is returned by the R68VMEADDR system service
!and holds the starting address and the size of the various memory
!mapped i/o regions of the vme bus.  Space for unanticipated future
!extensions is available.

structure /vme_t/
  integer*4 addr(VMESIZE) !starting physical address of range
  integer*4 size(VMESIZE) !size of range, in bytes
  integer*4 res1(VMESIZE) !reserved
  integer*4 res2(VMESIZE) !reserved
end structure
```

EXAMPLE

```
        program sysm68k
#       include <sysf/sysm68k.i>
        integer*4 sysm68k, cmd, arg1, arg2
        integer*4 iretval

c   Get size of virtual memory space

        iretval = sysm68k (S68MEMSIZE, arg1, arg2)
        if (iretval .lt. 0) write (*,*) 'sysm68k error:', iretval
        write (*,9000) (iretval + 1023) / 1024
9000    format (' Memory size: ', i10, 'Kb')
        end
```

SEE ALSO

swap(1M).

NAME

times - get process and child process times

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/times.i>

     integer*4 times
     record /tms/ buffer
     iretval = times (buffer)
```

DESCRIPTION

times fills the structure pointed to by *buffer* with time-accounting information. The following are the contents of this structure:

```
structure /tms/
        integer*4        tms_utime
        integer*4        tms_stime
        integer*4        tms_cutime
        integer*4        tms_cstime
end structure
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are reported in clock ticks per second. Clock ticks are a system-dependent parameter. The specific value for an implementation is defined by the variable HZ, found in the include file *param.i*.

tms_utime is the CPU time used while executing instructions in the user space of the calling process.

tms_stime is the CPU time used by the system on behalf of the calling process.

tms_cutime is the sum of the *tms_utime*s and *tms_cutime*s of the child processes.

tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

[EFAULT] *times* will fail if *buffer* points to an illegal address.

EXAMPLE

```
      program times
#     include <sysf/times.i>
      integer*4 times
      integer*4 init_tim, dummy, elap_time, utime, stime
      integer*4 cutime, cstime, i, j

      record /tms/ buffer

c   get initial elapsed time and delay a while

      init_tim = times (buffer)
      do 110 i=1, 1000
      do 110 j=1, 2000
      dummy = dummy + i
110   continue

c   get current elapsed time and print results

      elap_time = times (buffer)
      utime = buffer.tms_utime
      stime = buffer.tms_stime
      cutime = buffer.tms_cutime
      cstime = buffer.tms_cstime
      write (*,9000) utime, stime, cutime, cstime
9000  format (' utime: ', i8, ' stime: ', i8,
&     ' cutime: ', i8, ' cstime: ', i8)
      write (*,9001) utime + stime, elap_time - init_tim
9001  format (' Total time used via:'/
&     '  Utime + Stime: ', i4/
&     '  Elapsed Time: ', i4)
      end
```

SEE ALSO

exec(2F), fork(2F), time(2F), wait(2F).

DIAGNOSTICS

Upon successful completion, *times* returns the elapsed real time, in clock ticks per second, from an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a negative value indicating the error is returned.

NAME

trunc - truncate file space

SYNOPSIS

integer*4 trunc, fd, length, flags
iretval = trunc (fd, length, flags)

DESCRIPTION

trunc truncates the file referenced by *fd* to *length* bytes. The file referenced by *fd* must be open for writing. If the F5NOZERO bit is set in *flags*, any potential physical space that exists after the new logical end-of-file will not be zeroed; the default is to zero such space. If F5NOZERO is used, it is possible to creat holes in the file (using *lseek*) that will not read as zeroes. If the F5SHRINK bit is set and there is contiguous file extents for this file (see *prealloc*(2F)), physical and logical space will be truncated. If *prealloc*(2F) has never been used on this particular file, both logical and physical space for the file will be truncated.

trunc will fail if one or more of the following are true:

[EBADF] *fd* is not a valid file descriptor open for writing.

[ENXIO] *fd* is not a regular file.

EXAMPLE

```
      program trunc
#     include <sysf/fcntl.i>
      integer*4 trunc, fd, length, flags
      integer*4 open, close, iretval

c  Open and optionally create a file

      fd = open ('tmp', O_RDWR .or. O_CREAT, '777'o)
      if (fd .lt. 0) write (*,*) 'open error:', fd

c  Truncate to 2048 bytes zero filled

      length = 2048
      flags = 0
      iretval = trunc (fd, length, flags)
      if (iretval .lt. 0) write (*,*) 'trunc error:', iretval

c  Close the file

      iretval = close (fd)
      end
```

SEE ALSO

close(2F), creat(2F), dup(2F), exec(2F), fcntl(2F), fork(2F), open(2F), pipe(2F), prealloc(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

uadmin - administrative control

SYNOPSIS

```
#    include <sysf/uadmin.i>

     integer*4 uadmin, cmd, fcn, mdep
     iretval = uadmin (cmd, fcn, mdep)
```

DESCRIPTION

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN
: The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

  AD_HALT — Halt the processor and turn off the power.

  AD_BOOT — Reboot the system, using /realix.

  AD_IBOOT — Interactive reboot; user is prompted for system name.

A_REBOOT
: The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.

A_REMOUNT
: The root file system is mounted again after having been fixed. This should be used only during the startup process.

A_KILLALL
: Sends a signal to all active processes not directly related to the shutdown procedure (see *killall*(1M). The *fcn* argument should contain the signal number to be sent (as defined on *signal*(2F)); the *mdep* argument should contain the process group ID of processes to be signaled.

uadmin fails if the following is true:

[EPERM] — The effective user ID is not superuser.

EXAMPLE

```
     program uadmin
#    include <sysf/uadmin.i>
     integer*4 uadmin, cmd, fcn, mdep
     integer*4 iretval

c  Reboot the system

     cmd = A_SHUTDOWN
     fcn = AD_BOOT
     mdep = 0
     iretval = uadmin (cmd, fcn, mdep)
     if (iretval .lt. 0) write (*,*) 'uadmin error:', iretval
     end
```

DIAGNOSTICS

Upon successful completion, the value returned depends on *cmd* as follows:

A_SHUTDOWN	Never returns
A_REBOOT	Never returns
A_REMOUNT	0

Otherwise, a negative value indicating the error is returned.

NAME

ulimit - get and set user limits

SYNOPSIS

integer*4 ulimit, cmd, newlimit
iretval = ulimit (cmd, newlimit)

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the regular file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *ulimit* fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit. [EPERM]

3 Get the maximum possible break value [see *brk*(2F)].

4 Get the maximum possible number of file descriptors that may be used by a process at a time. This is normally configured to NOFILES. (See *system*(1M) for reconfiguring the value of NOFILES.)

EXAMPLE

```
      program ulimit
      integer*4 ulimit, cmd, newlimit
      integer*4 maxfilesize, maxbrk, maxnumfd

c  Get current file size limit

      cmd = 1
      maxfilesize = ulimit (cmd, newlimit)
      if (maxfilesize .lt. 0) write (*,*) '1 ulimit error:', maxfilesize

c  Get maximum break value

      cmd = 3
      maxbrk = ulimit (cmd, newlimit)
      if (maxbrk .lt. 0) write (*,*) '2 ulimit error:', maxbrk

c  Get maximum number of file descriptors

      cmd = 4
      maxnumfd = ulimit (cmd, newlimit)
      if (maxnumfd .lt. 0) write (*,*) '3 ulimit error:', maxnumfd

c  Print results

      write (*,9000) maxfilesize, maxbrk, maxnumfd
9000 format (' Max file size in blocks:', i6,/
&       ' Max break value:', z8,/
&       ' Max number of file descriptors:', i6)
      end
```

SEE ALSO

brk(2F), close(2F), creat(2F), dup(2F), open(2F), sysgen(1M), write(2F).

WARNING

ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 5,120 bytes.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. Otherwise, a negative value indicating the error is returned.

NAME

umask - set and get file creation mask

SYNOPSIS

integer*4 umask, cmask
iretval = umask (cmask)

DESCRIPTION

umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

EXAMPLE

```
      program umask
      integer*4 umask, cmask
      integer*4 old_cmask

c  Set and get file creation mask

      cmask = '777'o  ! Full permissions
      old_cmask = umask (cmask)
      write (*,9000) 'previous creation mask:', old_cmask
9000 format (' ', a30, o5)
      end
```

SEE ALSO

mkdir(1), sh(1), chmod(2F), creat(2F), mknod(2F), open(2F).

DIAGNOSTICS

The previous value of the file mode creation mask is returned.

NAME

umount - unmount a file system

SYNOPSIS

integer*4 umount
character*SIZE path

iretval=umount(path)

DESCRIPTION

SIZE can be any number 1 through 128. *umount* requests that a previously mounted file system contained on the block special device or directory identified by *path* be unmounted. *Path* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

umount may be invoked only by the super-user.

umount will fail if one or more of the following are true:

[EPERM] The process's effective user ID is not super-user.

[EINVAL] *File* does not exist.

[EINVAL] *File* is not a block special device.

[EINVAL] *File* is not mounted.

[EBUSY] A file on *path* is busy.

[EFAULT] *File* points to an illegal address.

[EREMOTE] *File* is remote.

[ENOLINK] *File* is on a remote machine, and the link to that machine is no longer active.

[EMULTIHOP] Components of the path pointed to by *path* require hopping to multiple remote machines.

SEE ALSO

mount(2F).

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

uname - get name of current system

SYNOPSIS

```
#    include <sysf/utsname.i>

     integer*4 uname
     record /utsname/ name
     iretval = uname (name)
```

DESCRIPTION

uname stores information identifying the current system in the structure pointed to by *name*.

uname uses the structure defined in **<sysf/utsname.i>** whose members are:

```
character*9 sysname
character*9 nodename
character*9 release
character*9 version
character*9 machine
```

uname returns a null-terminated character string naming the current system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the system is running on.

[EFAULT] *uname* will fail if *name* points to an invalid address.

EXAMPLE

```
     program uname
#    include <sysf/utsname.i>
     integer*4 uname, iretval
     record /utsname/ name

c    Get name of current system and print

     iretval = uname (name)
     if (iretval .lt. 0) write (*,*) 'uname error:', iretval
     write (*,9000) name.sysname, name.nodename, name.release,
&       name.version, name.machine
9000 format (' Sysname:  ', a9,/
&       ' Nodename: ', a9,/
&       ' Release:  ', a9,/
&       ' Version:  ', a9,/
&       ' Machine:  ', a9)
     end
```

SEE ALSO

uname(1).

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. Otherwise, a negative value indicating the error is returned.

NAME

unlink - remove directory entry

SYNOPSIS

integer*4 unlink
character*SIZE path
iretval = unlink (path)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *unlink* removes the directory entry named by the path name pointed to by *path*.

The named file is unlinked unless one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EINTR]	A signal was caught during the *unlink* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

EXAMPLE

See link(2F) for an example.

SEE ALSO

rm(1), close(2F), link(2F), open(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

ustat - get file system statistics

SYNOPSIS

```
#    include <sysf/types.i>
#    include <sysf/ustat.i>

     integer*4 ustat
     integer*2 dev
     record /ustat/ buf
     iretval = ustat (dev, buf)
```

DESCRIPTION

ustat returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a *ustat* structure that includes the following elements:

```
integer*4   f_tfree        !Total free blocks
integer*2   f_tinode       !Number of free inodes
character*6  f_fname       !Filsys name
character*6  f_fpack       !Filsys pack name
```

ustat will fail if one or more of the following are true:

[EINVAL] *dev* is not the device number of a device containing a mounted file system.

[EFAULT] *buf* points outside the process's allocated address space.

[EINTR] A signal was caught during a *ustat* system call.

[ENOLINK] *dev* is on a remote machine and the link to that machine is no longer active.

[ECOMM] *dev* is on a remote machine and the link to that machine is no longer active.

EXAMPLE

```
      program ustat
#     include <sysf/types.i>
#     include <sysf/ustat.i>
#     include <sysf/stat.i>
      integer*4 ustat, stat
      integer*2 dev
      record /ustat/ buf
      record /stat/ buf2
      integer*4 iretval, i

c  Get device number for a file

      iretval = stat ('../example/ustat.F', buf2)
      if (iretval .lt. 0) write (*,*) 'stat error:', iretval
      dev = buf2.st_dev

c  Get file system statistics

      iretval = ustat (dev, buf)
      if (iretval .lt. 0) write (*,*) 'ustat error:', iretval, i
      write (*,9000) buf.f_tfree, buf.f_tinode,
&       buf.f_fname, buf.f_fpack
9000  format (' Total free blocks: ', i8,/
&       ' Total free inodes: ', i6,/
&       ' File system name:  ', a6,/
&       ' File pack name:    ', a6)
      end
```

SEE ALSO

stat(2F), fs(4).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

utime - set file access and modification times

SYNOPSIS

```
#    include <sysf/types.i>
     integer*4 utime
     character*SIZE path
     structure /utimbuf/
        integer*4 actime
        integer*4 modtime
     end structure
     record /utimbuf/ times
     iretval = utime (path, times)
```

DESCRIPTION

SIZE can be any number between and including 1 through 128. *path* points to a path name naming a file. *utime* sets the access and modification times of the named file.

If *times* is NULL (i.e. %val(0)), the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
structure /utimbuf/
        integer*4 actime      !access time
        integer*4 modtime     !modification time
end structure
```

utime will fail if one or more of the following are true:

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EACCES] Search permission is denied by a component of the path prefix.

[EPERM] The effective user ID is not super-user and not the owner of the file and *times* is not NULL.

[EACCES] The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *times* is not NULL and points outside the process's allocated address space.

[EFAULT] *path* points outside the process's allocated address space.

[EINTR] A signal was caught during the *utime* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

EXAMPLE

```
      program utime
      integer*4 utime, iretval
      character*40 path
      structure /utimbuf/
        integer*4 actime
        integer*4 modtime
      end structure
      record /utimbuf/ times
      integer*4 hours, mins, secs, month, day, year
      integer*4 gmt, getgmt, iretval

c  Modify a file with new times.
c  Pick November 3, 1989 at 13:45:27 as the new time

      hours = 13
      mins = 45
      secs = 27
      month = 11
      day = 3
      year = 1989

c  Your mission:  write function getgmt (returns -1 if error)

      gmt = getgmt (hours, mins, secs, month, day, year)
      if (gmt .lt. 0) write (*,*) 'time computation error'

c  Modify access and modification time of a file

      times.actime = gmt
      times.modtime = gmt
      iretval = utime ('../example/utime.F', times)
      if (iretval .lt. 0) write (*,*) 'utime error:', iretval
      end
```

SEE ALSO

stat(2F).

DIAGNOSTICS

Upon successful completion, a value of 0 is returned. Otherwise, a negative value indicating the error is returned.

NAME

wait - wait for child process to stop or terminate

SYNOPSIS

integer*4 wait, stat_loc
iretval = wait (stat_loc)

DESCRIPTION

wait suspends the calling process until one of the immediate children terminates or until a child that is being traced stops, because it has hit a break point. The *wait* system call will return prematurely if a signal is received and if a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. The status can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

> If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to '0177'O.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit* [see *exit*(2F)].
>
> If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit '200'O) is set, a "core image" will have been produced [see *signal*(2F)].

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes [see *intro*(2F)].

wait will fail and return immediately if the following is true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINTR] The calling process returned due to the receipt of a signal.

EXAMPLE

```
      program wait
      integer*4 wait, stat_loc
      integer*4 pid, fork
      integer*4 i, j

c  Make a child

      pid = fork ()
      if (pid .eq. 0) goto 2000

c  Wait until child terminates

      pid = 0
      pid = wait (stat_loc)
      if (pid .lt. 0) write (*,*) 'wait error:', pid

c  Print information
      write (*,9000) pid, stat_loc
9000 format (' Terminating child id:', i7, ' Status:', o8)
      if (.true.) stop

c  Here if child
c  Delay a while

2000 continue
      do 2100 i = 1, 2000
      do 2100 j = 1, 2000
      pid = i
2100 continue
      end
```

SEE ALSO

exec(2F), exit(2F), fork(2F), intro(2F), pause(2F), ptrace(2F), signal(2F).

WARNING

wait fails and its actions are undefined if *stat_loc* points to an invalid address.

DIAGNOSTICS

If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a negative value indicating the error is returned.

NAME

waitpid - wait for child process to stop or terminate

SYNOPSIS

#include <sysf/wait.i>

integer*4 waitpid,pid,statloc,options
iretval = waitpid(pid,statloc,options)

DESCRIPTION

waitpid provides both non-blocking status collection and collection of status of children that are stopped.

If *statloc* (taken as an integer) is nonzero, 16 bits of information called *status* are stored in the low order 16 bits of the location pointed to by *statloc*. *status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, *status* identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

> If the child process stopped, the high order 8 bits of *status* will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.
>
> If the child process terminated due to an *exit* call, the low order 8 bits of *status* will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit;* see **exit** (2).
>
> If the child process terminated due to a signal, the high order 8 bits of *status* will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit(i.e,. bit 200) is set, a "core image" will have been produced; see **signal** (3).

If *options* is zero, the behavior of *waitpid* is identical to *wait* (2). Otherwise, *options* consists of the logical OR of one or both of the following flags:

WNOHANG

> Return immediately, even if there are no children to wait for. In this case, a return value of zero indicates that no children have terminated (or stopped, if WUNTRACED is also set).

WUNTRACED

> Return the status of stoppped children. If the child process has stopped due to the delivery of a **SIGTTIN, SIGTTOU, SIGSTP,** or **SIGSTOP** signal, its status may be collected by setting this flag.

If **WUNTRACED** is set and the *status* of a stopped child process is reported, the high order 8 bits of *status* shall contain the number of the signal that caused the process to stop and the low order eight bits shall be set to the octal value 0177.

RETURN VALUE

waitpid returns -1 if there are no children not previously waited for; zero is returned if **WNOHANG** is specified and there are no stopped or terminated children.

ERRORS

If any of the following conditions occur, *waitpid* will return -1 and set *errno* to the corresponding value:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EINVAL] *waitpid* was called with an invalid *options* value.

SEE ALSO

exit(2), wait(2), wait3(2N)

NOTES

Currently on 88k machine only.

NAME

write - write on a file

SYNOPSIS

integer*4 write, fildes, nbyte
integer*1 buf (SIZE)
iretval = write (fildes, buf, nbyte)

integer*4 write, fildes, nbyte
character*SIZE bufc
iretval = write (fildes, bufc, nbyte)

DESCRIPTION

SIZE can be any number between and including 1 through 128. *fildes* is a file descriptor obtained from a *creat*(2F), *open*(2F), *dup*(2F), *fcntl*(2F), or *pipe*(2F) system call.

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

For regular files, if the O_SYNC flag of the file status flags is set, the write will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(2F)], and there is a record lock owned by another process on the segment of the file to be written. If O_NDELAY is not set, the write will sleep until the blocking record lock is removed.

For STREAMS [see *intro*(2F)] files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the *stream*. These values are contained in the topmost *stream* module. Unless the user pushes [see I_PUSH in *streamio*(7)] the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write* will fail with *iretval* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY is not set and the *stream* can not accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY will prevent a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* can not accept data, *write* will fail. If O_NDELAY is set and part of the buffer has been written when a condition in which the *stream* can not accept additional data occurs, *write* will terminate and return the number of bytes written.

write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw I/O is temporarily insufficient.

[EAGAIN] Attempt to write to a *stream* that can not accept data with the O_NDELAY flag set.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EDEADLK] The write was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *buf* points outside the process's allocated address space.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size [see *ulimit*(2F)].

[EINTR] A signal was caught during the *write* system call.

[EINVAL] Attempt to write to a *stream* linked below a multiplexor.

[ENOLCK] The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

[ENOSPC] During a *write* to an ordinary file, there is no free space left on the device.

[ENXIO] A hangup occurred on the *stream* being written to.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[ERANGE] Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* [see *ulimit*(2F)] or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *iretval* is set to the value included in the error message.

EXAMPLE

```
        program write
        integer*4 write, fildes, nbyte
        integer*1 buf (80)

        integer*4 open, bytwrit
        character*80 bufc

        equivalence (buf, bufc)

        bufc = 'Buffer test Data'
        nbyte = 80

c   Open file for read/write, create it if necessary

        fildes = open ('tst.x', '0402'o, '0666'o)
        if (fildes .lt. 0) write (*,*) 'create err: ', fildes
        bytwrit = write (fildes, buf, nbyte)
        if (bytwrit .lt. 0) write (*,*) 'write err: ', bytwrit
        end
```

NOTES

The *streams* features described in this manual page are not supported in this release.

SEE ALSO

creat(2F), dup(2F), fcntl(2F), intro(2F), lseek(2F), open(2F), pipe(2F), ulimit(2F).

DIAGNOSTICS

Upon successful completion the number of bytes actually written is returned. Otherwise, a negative value indicating the error is returned.

NAME

writev - do multiple writes from a file

SYNOPSIS

```
#   include <sysf/uio.i>
    integer*4 writev, fildes, iovcnt
    record /iovec/ iov(16)
    iretval = writev (fildes, iov, iovcnt)
```

DESCRIPTION

writev attempts to write data from the object referenced by *fildes*. The input data is scattered into the *iovcnt* buffers specified by the members of the *iov* array: iov(1), iov(2), ..., iov(iovcnt). This allows you to do up to 16 write operations with one system call.

The *iovec* structure is defined as:

```
structure /iovec/
   integer*4 iov_base
   integer*4 iov_len
end structure
```

Each *iovec* entry specifies the base address and length of an area in memory where data exists. *writev* writes to one area completely before proceeding to the next.

Upon successful completion, *writev* returns the number of bytes actually written and placed in the buffer. The system will write the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file; this is not guaranteed for other cases.

ERRORS

writev will fail if one or more of the following are true:

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EFAULT] *buf* or part of the *iov* points outside the allocated address space.

[EINTR] A write from a slow device was interrupted by a signal before any data arrived.

[EINVAL] The pointer associated with *d* was negative or greater than 16.

[EINVAL] *iovcnt* was negative or greater than 16.

[EINVAL] One of the *iov_len* values in the *iov* array was negative.

[EINVAL] Part of the *iov* points outside the process's allocated address space.

[EIO] An I/O error occurred while writing from the file system.

EXAMPLE

```
        program writev
#       include <sysf/uio.i>
#       include <sysf/fcntl.i>
        integer*4 writev, fildes, iovcnt
        integer*4 i, j, iretval, offset, open
        integer*1 buf(80,4)
        character*20 buf1, buf2, buf3, buf4
        equivalence (buf (1,1), buf1)
        equivalence (buf (1,2), buf2)
        equivalence (buf (1,3), buf3)
        equivalence (buf (1,4), buf4)
        record /iovec/ iov (16)

c   Open a file

        fildes = open ('../example/writev.tst', O_WRONLY + O_CREAT, '777'o)
        if (fildes .lt. 0) write (*,*) 'open error:', fildes

c   Fill the iov structure for four buffers

        offset = 0
        do 100 i = 1, 4
        iov (i).iov_base = %loc (buf(1,i))
        iov (i).iov_len = 20
        offset = offset + 20
100     continue

c   Fill the output buffers

        buf1 = 'Line 1'
        buf2 = 'Line two'
        buf3 = 'This is line three'
        buf4 = 'This should be 4'
        do 110 i = 1, 4
        buf (20, i) = 10
110     continue

c   Do the four write operations

        iovcnt = 4
        iretval = writev (fildes, iov, iovcnt)
        if (iretval .lt. 0) write (*,*) 'writev error:', iretval

        write (*,*) 'check writev.tst for the data'
        end
```

NOTES

The buffer address can be placed in the structure as follows:

```
iov (index).iov_base = %loc (buffer)
```

SEE ALSO

creat(2F), dup(2F), fcntl(2F), getmsg(2F), ioctl(2F), intro(2F), open(2F), pipe(2F), read(2F).

Appendix A

Installing the FORTRAN Library

The following sections tell how to install the General Language System (GLS™) FORTRAN System Calls Library (**libfs**) on open architecture systems. Before installing the library package you should be familiar with the **sysadm** installation tool. Refer to the **sysadm**(1M) manual page in the *REAL/IX Reference Manual* for a description of this tool.

Product Requirements

The following list includes the minimum requirements for installing and using **libfs**:

- An open architecture system
- One 150 Mbyte cartridge tape drive
- One auxiliary input device for batch processing (any MODCOMP-supported terminal)
- Approximately 480 Kbytes of disc space for **libfs** files

Installation Tape Contents

The GLS FORTRAN System Calls Library installation tape contains four IGF images (refer to **igf**(1M) in the *REAL/IX Reference Manual Sections*. IGF image 3 contains the scripts required to install **libfs**. These files are removed when installation is complete. IGF image 4 contains all the product files required to install and support this product. Table A–1 provides a breakdown of the IGF images.

IGF FILES/IMAGE CONTENTS	TEMPORARY DIRECTORY†	INSTALLED DIRECTORY	DESCRIPTION
IGF IMAGE 1: Volume ID Header			
IGF IMAGE 2: Tape Directory			lists all directories
IGF IMAGE 3:			
README_libfs	*/usr/tmp/igfdir*	*/libfs/README_libfs*	*README* file
INSTALL	*/usr/tmp/igfdir*		installation script
UNINSTALL	*/usr/tmp/igfdir*		remove script
Rlist	*/usr/tmp/igfdir*		list of files removed when uninstalling
postinstall	*/usr/tmp/igfdir*		links similar calls to the same man page
IGF IMAGE 4:. *./libfs/usr/lib/gls/libfs.a*		*/usr/lib/gls/libfs.a*	FORTRAN system call interface library
*./libfs/usr/include/gls/sysf/*i*		*/usr/include/gls/sysf/*i*	FORTRAN system call include files
*./libfs/usr/catman/p_man/manf/*z*		*/usr/catman/p_man/manf/*z*	FORTRAN (2F) man pages

† */usr/tmp/igfdir* is removed after installation is complete

Preinstallation

You must remove any prior versions of this product before installing the new release. Follow the instructions in the section "Removing libfs From Your System" at the end of this chapter.

Installation Procedure

If you are updating your version of **libfs**, remove the older version before installing the new release. To install **libfs**:

1. Log on to the operating system as superuser, also known as *root*.
2. Invoke the **sysadm**(1M) tool. You will be presented with the **sysadm** menu.
3. Select item **2** from the menu to display the *Software Management Menu*.
4. Choose item **1** from the menu to select *installpkg*.
5. Insert the installation tape into the cartridge tape drive and press carriage return (<CR>).
6. Choose item 2 to install a MODCOMP® tape. The *INSTALL* script completes the installation procedure. If any errors occur during the installation, a message describing the corrective action is displayed at the terminal.

After installation, the *README* file resides in */libfs/README_libfs*. You can remove this file from the system if desired.

Removing libfs From Your System

If you are updating your version of **libfs**, remove the older version before installing the new release. Use the cartridge tape that contains the *prior* release and follow these instructions before installing the new version. To remove **libfs** from your system:

1. Log on to the operating system as superuser, also known as *root*.
2. Invoke the **sysadm**(1M) tool. You will be presented with the **sysadm** menu.
3. Select item **2** from the menu to display the *Software Management Menu*.
4. Choose item **3** from the menu to select *removepkg*.
5. Insert the *original* installation tape into the cartridge tape drive and press carriage return (<CR>). The *Rlist* file, which was originally removed after the installation procedure, is reinstalled and referenced by **sysadm**. The *UNINSTALL* script removes all **libfs** product files from your system. If an error occurs during this process, a message describing the corrective action is displayed at the terminal.

PERMUTED INDEX

set the expiration time of a/	absinterval	absinterval(2F)
cancel one or more asynchronous/	acancel	acancel(2F)
accepts a connection on a socket	accept	accept(2F)
	accepts a connection on a socket	accept(2F)
determine accessibility of a/	access	access(2F)
set file	access and modification times	utime(2F)
set group	access list	setgroups(2F)
determine	accessibility of a file	access(2F)
enable or disable process	accounting	acct(2F)
enable or disable process/	acct	acct(2F)
examine or change signal	action.	sigaction(2F)
	administrative control	uadmin(2F)
set a process alarm clock	alarm	alarm(2F)
set a process	alarm clock	alarm(2F)
change data segment space	allocation	brk(2F)
read from file in an/	aread	aread(2F)
initialize structures before/	arinit	arinit(2F)
free internal resources for/	arwfree	arwfree(2F)
free internal resources for	asynchronous I/O from the process	arwfree(2F)
read from file in an	asynchronous manner	aread(2F)
write to file in an	asynchronous manner	awrite(2F)
initialize structures before requesting an	asynchronous read	arinit(2F)
initialize structures before requesting an	asynchronous write	awinit(2F)
cancel one or more	asynchronous I/O requests	acancel(2F)
control write/execute	attributes of memory.	memctl(2F)
initialize structures before/	awinit	awinit(2F)
write to file in an/	awrite	awrite(2F)
free a	binary semaphore	bsfree(2F)
get a	binary semaphore	bsget(2F)
binds a name to a socket	bind	bind(2F)
	binds a name to a socket	bind(2F)
update super	block	sync(2F)
examine and change	blocked signals	sigprocmask(2F)
change data segment space/	brk	brk(2F)
free a binary semaphore	bsfree	bsfree(2F)
get a binary semaphore	bsget	bsget(2F)
suspend the	calling process	suspend(2F)
introduction to system	calls and error numbers	intro(2F)
requests	cancel one or more asynchronous I/O	acancel(2F)
examine and	change blocked signals	sigprocmask(2F)
	change data segment space allocation	brk(2F)
	change mode of file	chmod(2F)
	change owner and group of a file	chown(2F)
	change priority of a process	nice(2F)
	change root directory	chroot(2F)
examine or	change signal action.	sigaction(2F)
	change the name of a file.	rename(2F)
	change working directory	chdir(2F)
create an interprocess	channel	pipe(2F)
change working directory	chdir	chdir(2F)
get process and	child process times	times(2F)
wait for	child process to stop or terminate	wait(2F)
wait for	child process to stop or terminate	waitpid(2F)

change mode of file	chmod	chmod(2F)
change owner and group of a file	chown	chown(2F)
change root directory	chroot	chroot(2F)
wait for a connected interrupt	cisema	cisema(2F)
set a process alarm	clock	alarm(2F)
close a file descriptor	close	close(2F)
	close a file descriptor	close(2F)
create an endpoint for	communication	socket(2F)
get	configurable pathname variables	pathconf(2F)
initiates a connection on a/	connect	connect(2F)
wait for a	connected interrupt	cisema(2F)
get name of	connected peer	getpeername(2F)
accepts a	connection on a socket	accept(2F)
initiates a	connection on a socket	connect(2F)
listens for	connections on a socket	listen(2F)
preallocate	contiguous file space	prealloc(2F)
administrative	control	uadmin(2F)
file	control	fcntl(2F)
	control device	ioctl(2F)
event	control operations	evctl(2F)
message	control operations	msgctl(2F)
semaphore	control operations	semctl(2F)
shared memory	control operations	shmctl(2F)
	control write/execute attributes of memory.	memctl(2F)
Set process group ID for job	control.	setpgid(2F)
create a new file or rewrite an/	creat	creat(2F)
	create a new file or rewrite an existing one	creat(2F)
	create a new process	fork(2F)
	create an endpoint for communication	socket(2F)
	create an interprocess channel	pipe(2F)
set and get file	creation mask	umask(2F)
get name of	current system	uname(2F)
get the	current value for a process interval timer	getinterval(2F)
get the	current value for a system-wide realtime timer	gettimer(2F)
set the	current value for a system-wide realtime timer	settimer(2F)
lock process, text, or	data in memory	plock(2F)
expand the stack region of the	data segment	stkexp(2F)
change	data segment space allocation	brk(2F)
close a file	descriptor	close(2F)
duplicate an open file	descriptor	dup(2F)
	determine accessibility of a file	access(2F)
control	device	ioctl(2F)
change root	directory	chroot(2F)
change working	directory	chdir(2F)
make a	directory	mkdir(2F)
remove a	directory	rmdir(2F)
independent format read	directory entries and put in a file system	getdents(2F)
remove	directory entry	unlink(2F)
make a	directory, or a special or ordinary file	mknod(2F)
enable or	disable process accounting	acct(2F)
duplicate an open file/	dup	dup(2F)
	duplicate an open file descriptor	dup(2F)
get real user, effective user, real group, and	effective group IDs	getuid(2F)
IDs get real user,	effective user, real group, and effective group	getuid(2F)
get extended file status	efstat	estat(2F)
	enable or disable process accounting	acct(2F)
create an	endpoint for communication	socket(2F)

format read directory	entries and put in a file system independent	getdents(2F)
remove directory	entry	unlink(2F)
introduction to system calls and	error numbers	intro(2F)
get extended file status	estat	estat(2F)
event control operations	evctl	evctl(2F)
receive any queued	event	evrcv(2F)
	event control operations	evctl(2F)
receive any queued	event from a specified list	evrcvl(2F)
get an	event identifier	evget(2F)
release an	event identifier	evrel(2F)
post an	event to a process	evpost(2F)
get an event identifier	evget	evget(2F)
post an event to a process	evpost	evpost(2F)
receive any queued event	evrcv	evrcv(2F)
receive any queued event from a/	evrcvl	evrcvl(2F)
release an event identifier	evrel	evrel(2F)
	examine and change blocked signals	sigprocmask(2F)
	examine or change signal action.	sigaction(2F)
	examine pending signals	sigpending(2F)
execute a file	exec:	exec(2F)
execute a file	execl	exec(2F)
execute a file	execle	exec(2F)
execute a file	execlp	exec(2F)
	execute a file	exec(2F)
suspend	execution for interval	sleep(2F)
execute a file	execv	exec(2F)
execute a file	execve	exec(2F)
execute a file	execvp	exec(2F)
create a new file or rewrite an	existing one	creat(2F)
terminate process	exit	exit(2F)
	expand the stack region of the data segment	stkexp(2F)
set the	expiration time of a process interval timer	absinterval(2F)
get	extended file status	estat(2F)
file control	fcntl	fcntl(2F)
change mode of	file	chmod(2F)
change owner and group of a	file	chown(2F)
determine accessibility of a	file	access(2F)
do multiple reads from a	file	readv(2F)
do multiple writes from a	file	writev(2F)
execute a	file	exec(2F)
link to a	file	link(2F)
make a directory, or a special or ordinary	file	mknod(2F)
makes symbolic link to a	file	symlink(2F)
read from	file	read(2F)
write on a	file	write(2F)
read directory entries and put in a	file system independent format	getdents(2F)
set	file access and modification times	utime(2F)
	file control	fcntl(2F)
set and get	file creation mask	umask(2F)
close a	file descriptor	close(2F)
duplicate an open	file descriptor	dup(2F)
read from	file in an asynchronous manner	aread(2F)
write to	file in an asynchronous manner	awrite(2F)
create a new	file or rewrite an existing one	creat(2F)
move read/write	file pointer	lseek(2F)
preallocate contiguous	file space	prealloc(2F)
truncate	file space	trunc(2F)

get extended	file status	estat(2F)
get	file status	stat(2F)
unmount a	file system	umount(2F)
get	file system information	statfs(2F)
get	file system statistics	ustat(2F)
get	file system type information	sysfs(2F)
change the name of a	file.	rename(2F)
create a new process	fork	fork(2F)
entries and put in a file system independent	format read directory	getdents(2F)
get configurable pathname/	fpathconf	pathconf(2F)
	free a binary semaphore	bsfree(2F)
I/O from the process	free internal resources for asynchronous	arwfree(2F)
get file status	fstat	stat(2F)
get file system information	fstatfs	statfs(2F)
get time	ftime	ftime(2F)
machine specific	functions	sysm68k(2F)
read directory entries and put/	getdents	getdents(2F)
get real user, effective user,/	getegid	getuid(2F)
get real user, effective user,/	geteuid	getuid(2F)
get real user, effective user,/	getgid	getuid(2F)
get the current value for a/	getinterval	getinterval(2F)
get next message off a stream	getmsg	getmsg(2F)
get name of connected peer	getpeername	getpeername(2F)
get process, process group, and/	getpgrp	getpid(2F)
get process, process group, and/	getpid	getpid(2F)
get process, process group, and/	getppid	getpid(2F)
get scheduling priority	getpri	getpri(2F)
set/get Processor Status/	getpsr	setpsr(2F)
	gets socket name	getsockname(2F)
gets socket name	getsockname	getsockname(2F)
get and set options on sockets	getsockopt	getsockopt(2F)
get the current value for a/	gettimer	gettimer(2F)
get a unique identifier for a/	gettimerid	gettimerid(2F)
get real user, effective user,/	getuid	getuid(2F)
voluntarily	give up CPU	relinquish(2F)
set	group access list	setgroups(2F)
Set process	group ID for job control.	setpgid(2F)
change owner and	group of a file	chown(2F)
send a signal to a process or a	group of processes	kill(2F)
set process	group ID	setpgrp(2F)
set user and	group IDs	setuid(2F)
user, effective user, real group, and effective	group IDs get real	getuid(2F)
get real user, effective user, real	group, and effective group IDs	getuid(2F)
get process, process	group, and parent process IDs	getpid(2F)
synchronous	I/O multiplexing	select(2F)
Set process group	ID for job control.	setpgid(2F)
get an event	identifier	evget(2F)
get shared memory segment	identifier	shmget(2F)
release a process interval timer	identifier	reltimerid(2F)
release an event	identifier	evrel(2F)
get a unique	identifier for a process interval timer	gettimerid(2F)
set the expiration time of a/	incinterval	absinterval(2F)
directory entries and put in a file system	independent format read	getdents(2F)
get file system type	information	sysfs(2F)
get file system	information	statfs(2F)
asynchronous read	initialize structures before requesting an	arinit(2F)
asynchronous write	initialize structures before requesting an	awinit(2F)

	initiates a connection on a socket	connect(2F)
I/O from the process free	internal resources for asynchronous	arwfree(2F)
create an	interprocess channel	pipe(2F)
wait for a connected	interrupt	cisema(2F)
suspend execution for	interval	sleep(2F)
get a unique identifier for a process	interval timer	gettimerid(2F)
get the current value for a process	interval timer	getinterval(2F)
resolution and maximum time value of process	interval timer get	resabs(2F)
set the expiration time of a process	interval timer	absinterval(2F)
release a process	interval timer identifier	reltimerid(2F)
introduction to system calls/	intro	intro(2F)
	introduction to system calls and error numbers	intro(2F)
control device	ioctl	ioctl(2F)
Set process group ID for	job control.	setpgid(2F)
send a signal to a process or a/	kill	kill(2F)
get and set user	limits	ulimit(2F)
link to a file	link	link(2F)
read value of a symbolic	link	readlink(2F)
	link to a file	link(2F)
makes symbolic	link to a file	symlink(2F)
receive any queued event from a specified	list	evrcvl(2F)
set group access	list	setgroups(2F)
set realtime privileged users	list	setrtusers(2F)
listens for connections on a/	listen	listen(2F)
	listens for connections on a socket	listen(2F)
	lock process, text, or data in memory	plock(2F)
make	locked segments resident in memory	resident(2F)
move read/write file pointer	lseek	lseek(2F)
	machine specific functions	sysm68k(2F)
	makes symbolic link to a file	symlink(2F)
signal	management	sigset(2F)
	manipulate signal sets	sigsetops(2F)
read from file in an asynchronous	manner	aread(2F)
write to file in an asynchronous	manner	awrite(2F)
set and get file creation	mask	umask(2F)
get resolution and	maximum time value of process interval timer	resabs(2F)
timer get the resolution and	maximum time values for a system-wide realtime	restimer(2F)
control write/execute/	memctl	memctl(2F)
lock process, text, or data in	memory	plock(2F)
make locked segments resident in	memory	resident(2F)
shared	memory control operations	shmctl(2F)
shared	memory operations	shmop(2F)
get shared	memory segment identifier	shmget(2F)
control write/execute attributes of	memory.	memctl(2F)
	message control operations	msgctl(2F)
receive a	message from a socket	recv(2F)
sends a	message from a socket	send(2F)
get next	message off a stream	getmsg(2F)
send a	message on a stream	putmsg(2F)
	message operations	msgop(2F)
get	message queue	msgget(2F)
make a directory	mkdir	mkdir(2F)
make a directory, or a special/	mknod	mknod(2F)
change	mode of file	chmod(2F)
set file access and	modification times	utime(2F)
	move read/write file pointer	lseek(2F)
message control operations	msgctl	msgctl(2F)

get message queue	msgget	msgget(2F)
message operations	msgop:	msgop(2F)
message operations	msgrcv	msgop(2F)
message operations	msgsnd	msgop(2F)
do	multiple reads from a file	readv(2F)
do	multiple writes from a file	writev(2F)
synchronous I/O	multiplexing	select(2F)
gets socket	name	getsockname(2F)
change the	name of a file.	rename(2F)
get	name of connected peer	getpeername(2F)
get	name of current system	uname(2F)
binds a	name to a socket	bind(2F)
get	next message off a stream	getmsg(2F)
change priority of a process	nice	nice(2F)
introduction to system calls and error	numbers	intro(2F)
open for reading or writing	open	open(2F)
duplicate an	open file descriptor	dup(2F)
	open for reading or writing	open(2F)
event control	operations	evctl(2F)
message control	operations	msgctl(2F)
message	operations	msgop(2F)
semaphore control	operations	semctl(2F)
semaphore	operations	semop(2F)
shared memory control	operations	shmctl(2F)
shared memory	operations	shmop(2F)
get and set	options on sockets	getsockopt(2F)
make a directory, or a special or	ordinary file	mknod(2F)
change	owner and group of a file	chown(2F)
get process, process group, and	parent process IDs	getpid(2F)
get configurable pathname/	pathconf	pathconf(2F)
get configurable	pathname variables	pathconf(2F)
suspend process until signal	pause	pause(2F)
get name of connected	peer	getpeername(2F)
examine	pending signals	sigpending(2F)
create an interprocess channel	pipe	pipe(2F)
lock process, text, or data in/	plock	plock(2F)
move read/write file	pointer	lseek(2F)
	post an event to a process	evpost(2F)
preallocate contiguous file/	prealloc	prealloc(2F)
	preallocate contiguous file space	prealloc(2F)
get scheduling	priority	getpri(2F)
set scheduling	priority	setpri(2F)
change	priority of a process	nice(2F)
set realtime	privileged users list	setrtusers(2F)
set realtime	privileges	setrt(2F)
change priority of a	process	nice(2F)
create a new	process	fork(2F)
for asynchronous I/O from the	process free internal resources	arwfree(2F)
post an event to a	process	evpost(2F)
resume a suspended	process	resume(2F)
suspend the calling	process	suspend(2F)
switch into a	process	swtch(2F)
terminate	process	exit(2F)
enable or disable	process accounting	acct(2F)
set a	process alarm clock	alarm(2F)
get	process and child process times	times(2F)
Set	process group ID for job control.	setpgid(2F)

set	process group ID	setpgrp(2F)
get process,	process group, and parent process IDs	getpid(2F)
get a unique identifier for a	process interval timer	gettimerid(2F)
get resolution and maximum time value of	process interval timer	resabs(2F)
get the current value for a	process interval timer	getinterval(2F)
set the expiration time of a	process interval timer	absinterval(2F)
release a	process interval timer identifier	reltimerid(2F)
send a signal to a	process or a group of processes	kill(2F)
get process and child	process times	times(2F)
wait for child	process to stop or terminate	wait(2F)
wait for child	process to stop or terminate	waitpid(2F)
suspend	process until signal	pause(2F)
get process, process group, and parent	process IDs	getpid(2F)
IDs get	process, process group, and parent process	getpid(2F)
lock	process, text, or data in memory	plock(2F)
send a signal to a process or a group of	processes	kill(2F)
set/get	Processor Status Register	setpsr(2F)
read directory entries and	put in a file system independent format	getdents(2F)
send a message on a stream	putmsg	putmsg(2F)
get message	queue	msgget(2F)
receive any	queued event	evrcv(2F)
receive any	queued event from a specified list	evrcvl(2F)
read from file	read	read(2F)
structures before requesting an asynchronous	read initialize	arinit(2F)
system independent format	read directory entries and put in a file	getdents(2F)
	read from file	read(2F)
	read from file in an asynchronous manner	aread(2F)
	read value of a symbolic link	readlink(2F)
move	read/write file pointer	lseek(2F)
open for	reading or writing	open(2F)
read value of a symbolic link	readlink	readlink(2F)
do multiple	reads from a file	readv(2F)
do multiple reads from a file	readv	readv(2F)
get real user, effective user,	real group, and effective group IDs	getuid(2F)
effective group IDs get	real user, effective user, real group, and	getuid(2F)
set	realtime privileged users list	setrtusers(2F)
set	realtime privileges	setrt(2F)
and maximum time values for a system-wide	realtime timer get the resolution	restimer(2F)
get the current value for a system-wide	realtime timer	gettimer(2F)
set the current value for a system-wide	realtime timer	settimer(2F)
specify what to do upon	receipt of a signal	signal(2F)
	receive a message from a socket	recv(2F)
	receive any queued event	evrcv(2F)
	receive any queued event from a specified list	evrcvl(2F)
receive a message from a socket	recv	recv(2F)
receive a message from a socket	recvfrom	recv(2F)
expand the stack	region of the data segment	stkexp(2F)
set/get Processor Status	Register	setpsr(2F)
	release a process interval timer identifier	reltimerid(2F)
	release an event identifier	evrel(2F)
voluntarily give up CPU	relinquish	relinquish(2F)
release a process interval/	reltimerid	reltimerid(2F)
	remove a directory	rmdir(2F)
	remove directory entry	unlink(2F)
change the name of a file.	rename	rename(2F)
initialize structures before	requesting an asynchronous read	arinit(2F)
initialize structures before	requesting an asynchronous write	awinit(2F)

cancel one or more asynchronous I/O	requests	acancel(2F)
get resolution and maximum time/	resabs	resabs(2F)
make locked segments resident/	resident	resident(2F)
make locked segments	resident in memory	resident(2F)
get resolution and maximum time/	resinc	resabs(2F)
interval timer get	resolution and maximum time value of process	resabs(2F)
system-wide realtime timer get the	resolution and maximum time values for a	restimer(2F)
the process free internal	resources for asynchronous I/O from	arwfree(2F)
get the resolution and maximum/	restimer	restimer(2F)
resume a suspended process	resume	resume(2F)
	resume a suspended process	resume(2F)
create a new file or	rewrite an existing one	creat(2F)
remove a directory	rmdir	rmdir(2F)
change	root directory	chroot(2F)
change data segment space/	sbrk	brk(2F)
get	scheduling priority	getpri(2F)
set	scheduling priority	setpri(2F)
expand the stack region of the data	segment	stkexp(2F)
get shared memory	segment identifier	shmget(2F)
change data	segment space allocation	brk(2F)
make locked	segments resident in memory	resident(2F)
synchronous I/O multiplexing	select	select(2F)
free a binary	semaphore	bsfree(2F)
get a binary	semaphore	bsget(2F)
	semaphore control operations	semctl(2F)
	semaphore operations	semop(2F)
get set of	semaphores	semget(2F)
semaphore control operations	semctl	semctl(2F)
get set of semaphores	semget	semget(2F)
semaphore operations	semop	semop(2F)
sends a message from a socket	send	send(2F)
	send a message on a stream	putmsg(2F)
processes	send a signal to a process or a group of	kill(2F)
	sends a message from a socket	send(2F)
sends a message from a socket	sendto	send(2F)
	set a process alarm clock	alarm(2F)
	set and get file creation mask	umask(2F)
	set file access and modification times	utime(2F)
	set group access list	setgroups(2F)
get	set of semaphores	semget(2F)
get and	set options on sockets	getsockopt(2F)
	Set process group ID for job control.	setpgid(2F)
	set process group ID	setpgrp(2F)
	set realtime privileged users list	setrtusers(2F)
	set realtime privileges	setrt(2F)
	set scheduling priority	setpri(2F)
realtime timer	set the current value for a system-wide	settimer(2F)
timer	set the expiration time of a process interval	absinterval(2F)
	set time	stime(2F)
	set user and group IDs	setuid(2F)
get and	set user limits	ulimit(2F)
	set CPU time slice size	setslice(2F)
	set/get Processor Status Register	setpsr(2F)
set user and group IDs	setgid	setuid(2F)
set group access list	setgroups	setgroups(2F)
Set process group ID for job/	setpgi	setpgid(2F)
set process group ID	setpgrp	setpgrp(2F)

set scheduling priority	setpri	setpri(2F)
set/get Processor Status/	setpsr	setpsr(2F)
set realtime privileges	setrt	setrt(2F)
set realtime privileged users/	setrtusers	setrtusers(2F)
manipulate signal	sets	sigsetops(2F)
set CPU time slice size	setslice	setslice(2F)
get and set options on sockets	setsockopt	getsockopt(2F)
set the current value for a/	settimer	settimer(2F)
set user and group IDs	setuid	setuid(2F)
	shared memory control operations	shmctl(2F)
	shared memory operations	shmop(2F)
get	shared memory segment identifier	shmget(2F)
shared memory operations	shmat	shmop(2F)
shared memory control operations	shmctl	shmctl(2F)
shared memory operations	shmdt	shmop(2F)
get shared memory segment/	shmget	shmget(2F)
shared memory operations	shmop:	shmop(2F)
examine or change signal action.	sigactio	sigaction(2F)
manipulate signal sets	sigaddset	sigsetops(2F)
manipulate signal sets	sigdelset	sigsetops(2F)
manipulate signal sets	sigfillset	sigsetops(2F)
signal management	sighold	sigset(2F)
signal management	sigignore	sigset(2F)
manipulate signal sets	sigismember	sigsetops(2F)
specify what to do upon receipt of a	signal	signal(2F)
specify what to do upon receipt/	signal	signal(2F)
suspend process until	signal	pause(2F)
examine or change	signal action.	sigaction(2F)
	signal management	sigset(2F)
manipulate	signal sets	sigsetops(2F)
send a	signal to a process or a group of processes	kill(2F)
examine and change blocked	signals	sigprocmask(2F)
examine pending	signals	sigpending(2F)
signal management	sigpause	sigset(2F)
examine pending signals	sigpending	sigpending(2F)
examine and change blocked/	sigprocmask	sigprocmask(2F)
signal management	sigrelse	sigset(2F)
signal management	sigset	sigset(2F)
set CPU time slice	size	setslice(2F)
suspend execution for interval	sleep	sleep(2F)
set CPU time	slice size	setslice(2F)
accepts a connection on a	socket	accept(2F)
binds a name to a	socket	bind(2F)
create an endpoint for/	socket	socket(2F)
initiates a connection on a	socket	connect(2F)
listens for connections on a	socket	listen(2F)
receive a message from a	socket	recv(2F)
sends a message from a	socket	send(2F)
gets	socket name	getsockname(2F)
get and set options on	sockets	getsockopt(2F)
preallocate contiguous file	space	prealloc(2F)
truncate file	space	trunc(2F)
change data segment	space allocation	brk(2F)
make a directory, or a	special or ordinary file	mknod(2F)
machine	specific functions	sysm68k(2F)
receive any queued event from a	specified list	evrcvl(2F)
	specify what to do upon receipt of a signal	signal(2F)

expand the	stack region of the data segment	stkexp(2F)
get file status	stat	stat(2F)
get file system information	statfs	statfs(2F)
get file system	statistics	ustat(2F)
get extended file	status	estat(2F)
get file	status	stat(2F)
set/get Processor	Status Register	setpsr(2F)
set time	stime	stime(2F)
expand the stack region of the/	stkexp	stkexp(2F)
wait for child process to	stop or terminate	wait(2F)
wait for child process to	stop or terminate	waitpid(2F)
get next message off a	stream	getmsg(2F)
send a message on a	stream	putmsg(2F)
read initialize	structures before requesting an asynchronous	arinit(2F)
write initialize	structures before requesting an asynchronous	awinit(2F)
update	super block	sync(2F)
suspend the calling process	suspend	suspend(2F)
	suspend execution for interval	sleep(2F)
	suspend process until signal	pause(2F)
	suspend the calling process	suspend(2F)
resume a	suspended process	resume(2F)
	switch into a process	swtch(2F)
switch into a process	swtch	swtch(2F)
read value of a	symbolic link	readlink(2F)
makes	symbolic link to a file	symlink(2F)
makes symbolic link to a file	symlink	symlink(2F)
update super block	sync	sync(2F)
	synchronous I/O multiplexing	select(2F)
get file system type information	sysfs	sysfs(2F)
machine specific functions	sysm68k	sysm68k(2F)
get name of current	system	uname(2F)
unmount a file	system	umount(2F)
introduction to	system calls and error numbers	intro(2F)
read directory entries and put in a file	system independent format	getdents(2F)
get file	system information	statfs(2F)
get file	system statistics	ustat(2F)
get file	system type information	sysfs(2F)
get the current value for a	system-wide realtime timer	gettimer(2F)
set the current value for a	system-wide realtime timer	settimer(2F)
the resolution and maximum time values for a	system-wide realtime timer get	restimer(2F)
wait for child process to stop or	terminate	wait(2F)
wait for child process to stop or	terminate	waitpid(2F)
	terminate process	exit(2F)
lock process,	text, or data in memory	plock(2F)
and maximum time value of process interval	timer get resolution	resabs(2F)
get a unique identifier for a process interval	timer	gettimerid(2F)
get the current value for a process interval	timer	getinterval(2F)
maximum time values for a system-wide realtime	timer get the resolution and	restimer(2F)
set the expiration time of a process interval	timer	absinterval(2F)
the current value for a system-wide realtime	timer get	gettimer(2F)
the current value for a system-wide realtime	timer set	settimer(2F)
release a process interval	timer identifier	reltimerid(2F)
get process and child process	times	times(2F)
get process and child process/	times	times(2F)
set file access and modification	times	utime(2F)
truncate file space	trunc	trunc(2F)
	truncate file space	trunc(2F)

get file system	type information	sysfs(2F)
administrative control	uadmin	uadmin(2F)
get and set user limits	ulimit	ulimit(2F)
set and get file creation mask	umask	umask(2F)
unmount a file system	umount	umount(2F)
get name of current system	uname	uname(2F)
get a	unique identifier for a process interval timer	gettimerid(2F)
remove directory entry	unlink	unlink(2F)
	unmount a file system	umount(2F)
suspend process	until signal	pause(2F)
	update super block	sync(2F)
specify what to do	upon receipt of a signal	signal(2F)
set	user and group IDs	setuid(2F)
get and set	user limits	ulimit(2F)
group IDs get real	user, effective user, real group, and effective	getuid(2F)
IDs get real user, effective	user, real group, and effective group	getuid(2F)
set realtime privileged	users list	setrtusers(2F)
get file system statistics	ustat	ustat(2F)
set file access and/	utime	utime(2F)
get the current	value for a process interval timer	getinterval(2F)
get the current	value for a system-wide realtime timer	gettimer(2F)
set the current	value for a system-wide realtime timer	settimer(2F)
read	value of a symbolic link	readlink(2F)
get resolution and maximum time	value of process interval timer	resabs(2F)
get the resolution and maximum time	values for a system-wide realtime timer	restimer(2F)
get configurable pathname	variables	pathconf(2F)
	voluntarily give up CPU	relinquish(2F)
wait for child process to stop/	wait	wait(2F)
	wait for a connected interrupt	cisema(2F)
	wait for child process to stop or terminate	wait(2F)
	wait for child process to stop or terminate	waitpid(2F)
wait for child process to stop/	waitpid	waitpid(2F)
change	working directory	chdir(2F)
structures before requesting an asynchronous	write initialize	awinit(2F)
write on a file	write	write(2F)
	write on a file	write(2F)
	write to file in an asynchronous manner	awrite(2F)
control	write/execute attributes of memory.	memctl(2F)
do multiple	writes from a file	writev(2F)
do multiple writes from a file	writev	writev(2F)
open for reading or	writing	open(2F)
voluntarily give up	CPU	relinquish(2F)
set	CPU time slice size	setslice(2F)
free internal resources for asynchronous	I/O from the process	arwfree(2F)
cancel one or more asynchronous	I/O requests	acancel(2F)
set process group	ID	setpgrp(2F)
effective user, real group, and effective group	IDs get real user,	getuid(2F)
get process, process group, and parent process	IDs	getpid(2F)
set user and group	IDs	setuid(2F)
terminate process	_exit	exit(2F)

Please comment on the publication's completeness, accuracy, and readability. We also appreciate any general suggestions you may have to improve this publication.

If you found any errors in this publication, please specify the page number or include a copy of the page with your remarks.

Your comments will be promptly investigated and appropriate action will be taken.

☐ If you require a written answer please check this box and include your address below.

Comments: ____

____

____

____

____

____

____

____

____

____

____

____

____

Manual Title ____

Manual Order Number ______________________ Issue Date ____________

Name ______________________ Position ____________

Company ____

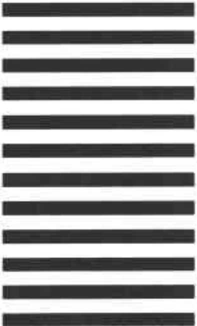
Address ____

______________________ Telephone (   ) ____________

MODCOMP
an AEG company

Please fold and tape.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 3624   FT. LAUDERDALE, FL 33309

POSTAGE WILL BE PAID BY ADDRESSEE

MODULAR COMPUTER SYSTEMS, INC.
1650 W. McNAB ROAD
P.O. BOX 6099
FT. LAUDERDALE, FL 33340–6099

Attention: TECHNICAL DOCUMENTATION, M.S. 880

MODCOMP, founded in 1970, is a worldwide supplier of high-performance, real-time computer systems, products, and services to the industrial automation, energy, transportation, scientific, and communications markets. MODCOMP is an AEG company.

Corporate Headquarters:
Modular Computer Systems, Inc.
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33340-6099
Tel: (305) 974-1380
Twx: 510-956-9414

International Headquarters:
Modular Computer Services, Inc.
The Business Centre
Molly Millars Lane
Wokingham, Berkshire
RG11 2JQ, UK
Tel: 0734-786808, TLX: 851849149

Latin American-Caribbean Headquarters:
Modular Computer Systems, Inc.,
1650 West McNab Road
P.O. Box 6099
Ft. Lauderdale, FL 33340-6099
Tel: (305) 977-1795, TLX: 3727852

Canadian Headquarters:
MODCOMP Canada, Ltd.,
400 Matheson Blvd. East, Unit 24
Mississauga, Ontario
Canada L4Z 1N8
Tel: (416) 890-0666
Fax: (416) 890-0266

Sales & Service Locations
Throughout the World

Copyright © 1989, Modular Computer Systems, Inc.
MODCOMP is a registered trademark of Modular Computer Systems, Inc.

Printed in U.S.A. • "The technical contents of this document, while accurate as of the date of publication, are subject to change without notice."