Prime®

Subroutines Reference Guide

Volume I

Revision 21.0

DOC10080-2LA

Subroutines Reference Guide Volume I

Second Edition

by

John Breithaupt

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 21.0 (Rev. 21.0).

Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760

PRINTING HISTORY -- Subroutines Reference Guide, Volume 1

First Edition (DOC10080-1LA) August 1986 for Release 20.2
Second Edition (DOC10080-2LA) August 1987 for Release 21.0

CREDITS

Project Support	David Brooks, Len Bruns, Matthew Carr, Ellen Desmond, Camilla Haase, Sheryl Horowitz, Joan Karp, Alice Landy, Fran Litterio, Lee McGraw, Ewan Milne, Margaret Taft
Editorial Support	Thelma Henner
Graphic Support	Mingling Chang
Production Support	Judy Gordon
Document Support	Celeste Henry, Peg Theriault

HOW TO ORDER TECHNICAL DOCUMENTS

To order copies of documents, or to obtain a catalog and price list:

United States Customers	International
Call Prime Telemarketing, toll free, at 1-800-343-2533, Monday through Friday, 8:30 a.m. to 5:00 p.m. (EST).	Contact your local Prime subsidiary or distributor.

CUSTOMER SUPPORT

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838	(within Massachusetts)	1-800-541-8888	(within Alaska)
1-800-343-2320	(within other states)	1-800-651-1313	(within Hawaii)

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

About This Book

The Subroutines Reference Guide gives a systematic description of the standard Prime subroutine libraries. Each standard subroutine library is a binary file containing subroutines that perform a variety of related programming tasks. Whenever these tasks are to be performed, programmers can use the subroutines in the standard libraries instead of writing their own routines. Programmers must write subroutines only to perform specialized tasks for which no standard subroutines exist.

OVERVIEW OF THIS SERIES

The Subroutines Reference Guide comprises four volumes. The contents of each volume are as follows.

Volume I

Chapter 1 provides a general introduction to subroutines and subroutine libraries.

Chapter 2 describes how to call subroutines and functions.

Chapters 3 through 9 describe how to choose proper data types for parameters of subroutines called from programs written in the following languages: BASIC/VM, C, COBOL, FORTRAN, Pascal, PL/I, and PMA. Each of these chapters describes a language's data description formats and subroutine calling sequence. Each chapter also emphasizes the necessity of making data type descriptions in the calling language compatible with the data type descriptions used by the subroutines called; at the object code level, the calling language and the subroutines called must specify the same data types.

Most of the descriptions of subroutines in the Subroutines Reference Guide use a PL/I calling format. Chapters 3 through 8 in Volume I contain tables listing data types in the different Prime programming languages that are equivalent to those in PL/I and FORTRAN.

The remaining three volumes in the series describe in detail the different subroutine libraries.

Volume II

Volume II describes several functional groups of subroutines, dealing with the access to and management of file system entities, the manipulation of EPFs in the execution environment, and the use of a number of command environment functions. Three chapters are devoted to subroutines related to the file system, and one chapter each is devoted to those related to EPF management and to the command environment.

Volume III

Volume III describes system subroutines. The subroutines covered in this volume are the general system calls to the operating system and standard system library. This excludes file and EPF manipulation, which are described in Volume II.

Volume IV

Volume IV presents several mature libraries: the Input/Output Control System (IOCS) libraries and other I/O-related subroutines, the Application libraries, the Sort libraries, and MATHLB.

IOCS provides device-independent I/O. The chapters on IOCS provide descriptions of the device-independent subroutines as well as those device-dependent subroutines simplified by IOCS. Another section provides descriptions of the synchronous and asynchronous device-driver subroutines.

Sections on the Application Library, the Sort Libraries, and the FORTRAN Matrix Library provide descriptions of other program development subroutines especially useful for FORTRAN programs.

SUGGESTED REFERENCES

The Prime User's Guide (DOC4130-4LA) and its updates (UPD4130-41A, UPD4130-42A) contain information on system use, directory structure, the condition mechanism, CPL files, ACLs, and how to load and execute files with external subroutines. Language programmers will also need the reference guide for their particular languages.

Programmers who wish more advanced information on library management or I/O manipulation should consult the System Administrator's Guide, Volume 1: System Configuration (DOC10131-1LA) and System Administrator's Guide, Volume 2: Communication Lines and Controllers (DOC10132-1LA).

The following related Prime publications are also available:

Advanced Programmer's Guide, Volume 1: BIND and EPFs
(DOC10055-1LA)

Assembly Language Programmer's Guide
(DOC3059-2LA)

BASIC/VM Programmer's Guide
(FDR3058-101A, COR3058-001, COR3058-002, UPD3058-33A)

C User's Guide
(DOC7534-3LA)

COBOL 74 Reference Guide
(DOC5039-2LA, UPD5039-21A, UPD5039-22A)

CPL User's Guide
(DOC4302-3LA)

FORTRAN Reference Guide
(FDR3057-101A, COR3057-001, COR3057-002, UPD3057-33A, UPD3057-34A)

FORTRAN 77 Reference Guide
(DOC4029-4LA, UPD4029-41A, UPD4029-42A))

Pascal Reference Guide
(DOC4303-4LA, UPD4303-31A)

PL/I Reference Guide
(DOC5041-1LA, UPD5041-11A)

Programmer's Guide to BIND and EPFs
(DOC8691-1LA)

SEG and LOAD Reference Guide
(DOC3524-192L, UPD3524-21A)

System Architecture Reference Guide
(DOC9473-2LA)

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	LOGIN user-id
underlining in examples	In examples, user input is underlined but system prompts and output are not.	OK, SEG -LOAD
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items (0-n).	CALL xxx (key [,altrtn])
Braces { }	Braces enclose a vertical list of items. Choose one and only one of these items.	CALL $\left\{\begin{array}{l}\text{CLINEQ}\\ \text{LINEQ}\\ \text{DLINEQ}\end{array}\right\}$
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	CALL TIMDAT(array, n)
Hyphen -	Wherever a hyphen appears in a command line option, it is a required part of that option.	SPOOL -LIST

1 Subroutines and Libraries

INTRODUCTION

Subroutines are modules of object code that can be called by programs or other modules to perform commonly required tasks. Subroutines can also be called to perform tasks that the calling program or module cannot perform as efficiently, or cannot perform at all.

Prime supplies a number of standard subroutines, known as system subroutines. Some of these are part of the PRIMOS operating system. The others are contained in standard subroutine libraries. Subroutine libraries are files that contain subroutines which perform similar or related tasks.

Users can write their own subroutines to perform tasks not performed by the system subroutines. However, this guide deals only with system subroutines.

This chapter discusses the following topics:

- Major functions of system subroutines.
- Standard Libraries. Prime supplies a wide variety of standard libraries that support basic system operation.
- Shared and Unshared Libraries. More than one program can use a shared library. However, a program must use its own separate copy of an unshared library; the system loads this copy with the program that uses it.

- Static-mode libraries. Prime supplies these libraries. The SEG and BIND utilities link them to the programs that call them.
- Library EPFs (Executable Program Files). These libraries contain subroutines to which programs link themselves when they are executed. The BIND utility creates library EPFs. Prime supplies some library EPFs.
- DYNTs (Dynamic Entrypoints). Dynts are subroutine names that are linked with programs. When the programs are executed, the system uses the dynts to find the starting addresses of the subroutines.
- Search Rules Lists. These are lists of library EPFs and directives that contain the entrypoints to which dynts are converted at run time. The lists tell the system where to search for the entrypoints.
- Loading and Linking Libraries.
- Subroutines and addressing modes.

MAJOR FUNCTIONS OF SUBROUTINES

System subroutines perform a wide variety of functions. Among these functions are the following:

- File handling
- I/O processing
- Supporting synchronous and asynchronous controllers
- Semaphore handling
- Supporting the condition mechanism

File Handling

File-handling system subroutines support communication between the PRIMOS file structure and user programs. For example, they can verify the existence of a file before the program accesses it, delete a file, or verify that a filename entered by a user is valid. File-handling subroutines manage the ACL system, which controls file access.

Many of the file-handling subroutines allow a program to access files directly through file unit numbers, a method which is faster than access by filenames. For example, at the program level the filename TEXT and the file unit number 1 can be associated by the PRIMOS subroutine SRCH$$, as in the following call:

```
    CALL SRCH$$ (K$WRIT, 'TEXT', 4, 1, TYPE, code)
```

Afterwards, other subroutines can access the file by unit number.

Some file-handling subroutines are internal to PRIMOS, and others are available as application library routines. Volume II of this guide discusses file-handling subroutines.

I/O Processing

The I/O subroutines are those relating to data transfers and device operations. Subroutines managed by the Input/Output Control System (IOCS) perform input and output between the Prime computer and the disks, terminals, and peripheral devices within the system configuration.

The I/O subroutines include:

- Subroutines that function as device-independent drivers which route I/O requests to specific drivers, thus allowing the user to maintain device independence
- Disk subroutines that perform disk input/output operations
- Subroutines that transfer data between a user terminal or paper-tape device and memory
- Peripheral device routines that control line printers, a printer/plotter, serial and parallel card readers, and 7-track and 9-track tapes

Volume IV of this guide describes IOCS subroutines and other I/O-related subroutines.

Supporting Synchronous and Asynchronous Controllers

A number of subroutines move data for assigned synchronous or asynchronous lines. Volume IV of this guide describes these subroutines.

Semaphore Handling

Semaphores are hardware components which ensure that only a set number of users access certain system resources at a time, and that reallocation of the resource is orderly and controlled. PRIMOS includes a set of subroutines that provide access to Prime's semaphore primitives and to internal timing facilities. These subroutines support user applications that have realtime requirements or need to synchronize execution with other user programs. Volume III of this guide describes these subroutines.

Supporting the Condition Mechanism

A program can activate the condition mechanism when it encounters unexpected occurrences such as:

- End of file
- Illegal address
- An attempt to divide by 0
- Use of the BREAK key from a terminal

The condition mechanism either repairs the problem and restarts the program, or terminates the program in an orderly manner. To do this, the condition mechanism activates diagnostic or remedial code blocks called on-units.

Users writing in FORTRAN 66 (FTN), FORTRAN 77 (F77), PL/I, Pascal, or PMA can define their own on-units. However, all users are automatically protected by PRIMOS system on-units. When an error condition occurs, the condition mechanism looks for on-units within the executing procedure. If it finds none, or if the procedure's on-units continue to signal the condition, the condition mechanism searches first through the calling procedures' on-units and then through the system's on-units, activating the first appropriate on-unit it finds.

Volume III of this guide describes subroutines that support the condition mechanism.

STANDARD LIBRARIES

This section describes the functions of the following major libraries and subroutine groups:

- FORTRAN library
- General PRIMOS subroutines
- Matrix library
- Applications library
- Sort libraries
- Spool libraries

To load these libraries, use the LI command of the BIND and SEG utilities. For a fuller discussion of the use of this command, see the section in this chapter titled LINKING AND LOADING LIBRARIES.

FORTRAN Library

The FORTRAN library is indispensable for the functioning of most other libraries because references to system subroutines are resolved in the FORTRAN library. The FORTRAN library contains:

- Many PRIMOS subroutines, such as those in the IOCS library and all PRIMOS file-handling subroutines.
- FORTRAN function subroutines and mathematical subroutines described in the FORTRAN Reference Guide and the FORTRAN 77 Reference Guide.
- Arithmetic subroutines that the FORTRAN compiler uses. Some of these subroutines can also be called from PMA. These routines perform operations on single-precision integers, single-precision and double-precision floating point numbers, and complex numbers.

General PRIMOS Subroutines

General PRIMOS subroutines perform the following functions:

- Managing system information
- Managing global variables
- Handling phantoms
- File system management

Matrix Library

MATHLB (FORTRAN matrix subroutines) contains subroutines that

- Perform matrix operations
- Solve systems of simultaneous linear equations
- Generate permutations and combinations of elements

These subroutines are currently available only in R mode.

Applications Library

The Applications library contains easy-to-use service subroutines which range from simple subroutines which do little more than call lower-level subroutines, to subroutines that perform functions such as:

- String handling
- User query
- System information retrieval
- Mathematical operations
- Conversion
- File system management
- Parsing

Subroutines in this library often duplicate the work of subroutines in the File System library, or even call those routines. For example, to delete a file, you may use SRCH$$ or TSRC$$ in the File System library, or you may call DELE$A in the Applications library. DELE$A requires fewer arguments than the other subroutines, but it may be slightly slower because it makes calls to three subroutines.

Sort Libraries

Four libraries contain sort subroutines:

- VSRTLI, a V-mode library containing subroutines that perform most file sorting and merging operations
- SRTLIB, the R-mode version of VSRTLI
- VMSORT, a V-mode library containing several in-memory sort subroutines and a binary search subroutine
- MSORTS, the R-mode version of VMSORT

Spool Libraries

The spooler subsystem enables users of a Prime system or network to print their files in an efficient and organized manner. Options of the spooler subsystem enable the user to defer printing to some later time, specify the site at which the job is to print, and specify the number of copies to be printed. There are two libraries of subroutines that support spooler subsystems:

- SPOOL$.BIN, an R-mode library
- VSPOOL.BIN, a V-mode library

SHARED AND UNSHARED LIBRARIES

All libraries can exist in both shared and unshared versions. A shared library allows all programs to access its subroutines. For this reason, each system needs only one copy of a shared library.

A program that is to use subroutines in an unshared library must make its own copies of the subroutines. The system places the copies in the user space of the program's owner and links the program with the copies to form a single executable module. When you run a program linked to subroutines in an unshared library, the system allocates additional memory for it and for all the subroutines it calls. This memory remains allocated until the user logs out or frees segments. To save memory, use shared libraries whenever possible.

By convention, unshared libraries have names that begin with the letter N (for nonshared). For example, the unshared Pascal library is named NPASLIB.BIN, while the shared Pascal library is named PASLIB.BIN.

STATIC AND DYNAMIC LIBRARIES

All libraries belong to one of two categories: those that are loaded statically, and those that are loaded dynamically. Libraries that are loaded statically are referred to as static-mode libraries. Those that are loaded dynamically are referred to as library EPFs.

Static-mode Libraries

The system always loads a static-mode library into the same system segment. The library remains in this segment, where it is available for use whenever it is needed. All static-mode libraries are shared libraries.

Ordinarily, only the System Administrator needs to be aware that a given library is loaded in static mode. If you use the appropriate LI command in a BIND or SEG session, the standard system search rules ensure that the static-mode libraries are searched. Your system may have several static-mode shared libraries included in the system build; ask your System Administrator for details.

Library EPFs

A library EPF contains executable subroutines that link to a program when the program is executed. The program links to entrypoints within the library EPF by means of the dynamic linking mechanism. When a library EPF is called, PRIMOS maps it to any segment available. If the library is called again, the system may map the library into a totally different segment, provided that the library has not been mapped between calls.

Library EPFs are created by the BIND utility. They have the suffix .RUN (latest version) or .RPn (for versions in use when BIND creates a newer version).

Classes of Library EPFs

Library EPFs are divided into two classes, program class and process class.

Program-class libraries: Use these libraries when no data is to be passed from one program calling the subroutines to the next. For example, FORTRAN_IO_LIBRARY.RUN is a program-class library because it includes certain file control blocks and other variables that must be reset for each execution of a program.

Process-class libraries: Use these libraries when some data may be passed from one program calling the subroutines to the next. For example, SYSTEM_LIBRARY.RUN is a process-class library. It need not be reinitialized while the process continues, because it contains only linkages (Indirect Pointers and Entry Control Blocks) that do not change after the library is initialized.

For a thorough discussion of the difference between process-class libraries and program-class libraries, refer to the Advanced Programmer's Guide, Volume 1, BIND and EPFs.

DYNAMIC ENTRYPOINTS (DYNTS)

Programs can use dynamic entrypoints, or dynts, to access libraries of general PRIMOS subroutines, library EPFs, and static-mode libraries. To access a subroutine, the system converts the subroutine's dynt into an address for the start of that subroutine. The process of converting a dynt into an address is referred to as snapping the dynt. Dynts are snapped at runtime. BIND links the binary file containing the dynts with a program but does not link the subroutine code.

The code for any procedure referenced by a dynt can be in any of the following:

- A library EPF
- A segment used by a static-mode shared library
- PRIMOS

The location of the code depends on runtime conditions.

For example, suppose that a program requests a subroutine from the Pascal library PASLIB.BIN, and PASLIB.BIN holds a dynt to this subroutine, which is resident in the library EPF PASCAL_LIBRARY.RUN. If the dynt is not mapped, the system uses the search rules and system hashing tables to call the .RUN file to do the mapping. To access the subroutine, the program branches to the address currently given to the dynt; this address is established when the library EPF maps the dynt to the system. The program then branches to the subroutine it needs.

If PASLIB.BIN holds a dynt to a subroutine resident in a static-mode library, the program branches to its static address at the proper time. Individual user space holds a single copy of pure code for any subroutine. It is not burdened with reserving static system segments to hold possibly unused code.

Dynts for subroutines that are not in R mode may require several library EPFs. For example, giving the SEG or BIND command LI PASLIB may cause the system to use dynts that call subroutines from PASCAL_LIBRARY.RUN, which in turn contains dynts that require SYSTEM_LIBRARY.RUN and PRIMOS_LIBRARY.RUN. These library EPFs should be part of your system search rules list. (See the section titled Search Rules Lists below.)

For more information about how dynts are used to access subroutines, see the section titled Dynamic Linking Mechanism in the Advanced Programmer's Guide, Volume 1.

SEARCH RULES LISTS

PRIMOS uses search rules lists to determine pathnames for directories, files, and entrypoints. PRIMOS provides every user with five search lists: ATTACH$, INCLUDE$, BINARY$, COMMAND$, and ENTRY$. It uses ATTACH$ to search partitions for top-level directories. It uses INCLUDE$, BINARY$, and COMMAND$ to search directories for source, binary, and executable code files, respectively. It uses ENTRY$ to search library EPF files for entrypoints.

You can modify the contents of these search lists and create other search lists as required. These search lists and the search rules facility are described in greater detail in the Advanced Programmer's Guide, Volume II.

As stated above, PRIMOS uses the ENTRY$ search list to locate entrypoints when it resolves dynts. ENTRY$ contains a keyword, -STATIC_MODE_LIBRARIES, that causes PRIMOS to search static-mode libraries; if the entrypoint is found there, PRIMOS stops searching. ENTRY$ also contains a separate search rule for each library EPF. If the ENTRY$ search list does not contain a search rule for a library EPF, it cannot resolve dynts to entrypoints in that library at runtime. If a dynt is linked successfully, but is not replaced with the entrypoint link to the code at execution, an error occurs reporting that a subroutine cannot be found. When this error occurs, verify that the appropriate library EPF is listed in the ENTRY$ search list. If you create a private subroutines library, you must add the pathname of that library to your ENTRY$ search list.

There are three PRIMOS commands and several subroutines that you can use to check and modify the contents of your search lists. You can use the SET_SEARCH_RULES (SSR) command to set search rules in a search list. You can use the EXPAND_SEARCH_RULES (ESR) command to use a search list to determine the absolute pathname of a file or entrypoint. You can also use ESR to test whether a search list can locate a particular item. You can use the LIST_SEARCH_RULES (LSR) command to list the contents of your search lists. These commands are further described in the PRIMOS Commands Reference Guide; search rule subroutines are described in the Subroutines Reference Guide, Volume II.

Both the search rule and the file it refers to must be present for a runtime search to be successful. For example, if you are trying to run a CBL program, and within BIND you successfully linked LI CBLLIB, you still need the following:

- A copy of the library EPF CBL_LIBRARY.RUN on your system
- An ENTRY$ search list that includes the pathname to CBL_LIBRARY.RUN

If the ENTRY$ search list does not contain the pathname entry, you can add this entry to your own ENTRY$ search rules file and then use the SSR command to establish the new ENTRY$ search list; or, you can ask the System Administrator to add this entry to the default ENTRY$ search list for all users. If this pathname is present in ENTRY$ but the system cannot find the subroutines, then the library either was loaded with a different pathname or was not loaded at all. See your System Administrator for help.

Note

Be careful not to assign names of PRIMOS subroutines to your own library EPF entrypoints. If the name of one of your entrypoints is identical to one named in a public library (for example, PASCAL_LIBRARY.RUN), the library that is listed first in the ENTRY$ search list always provides the subroutine. This can result in a PRIMOS subroutine being executed instead of a user-written subroutine with the same name.

LINKING AND LOADING LIBRARIES

All PRIMOS subroutine libraries have been compiled before they are placed in the system. A source code library that has been compiled is known as a binary library. Binary libraries that are to be used by a program must be loaded into the program's runtime file (memory image). All object files loaded into one runtime file must be in the same addressing mode. (See the section titled Subroutines and Addressing Modes, below.)

Binary libraries are stored in the directory LIB. To get a list of all the libraries in the directory LIB, attach to that directory and give the LD command. Some libraries in LIB are not described in this guide. The subroutines in some of these libraries are discussed in the manuals for specific products, such as PRIMENET, FORTRAN, the Block Device Interface (BDVLIB), and MIDASPLUS (KIDALB and VKDALB). The calls to subroutines in other libraries, such as RPG, are produced automatically by compilers; the details do not concern the programmer.

When you run the SEG or BIND programs to link a binary library to a program, use the command

```
    LI library-name
```

where library-name designates the name of a library, such as VSRTLI or VAPPLB, to be loaded with the program. You do not need to include the .BIN suffix in the LI command. Use the command

```
    LI
```

at the end of a SEG or BIND session to link your program to the system default libraries, such as the FORTRAN library. You must use the LI command even if your program does not call any subroutines. While you do not need the default libraries command for this situation, you do not incur any system penalties by including it. The BIND program may display the message BIND COMPLETE before you give this command.

Separate versions of the libraries are required for use with R-mode and V-mode files. Table 1-1 describes the shared libraries and their corresponding pathnames for R mode and V mode. It is a good idea to be familiar with the names of these libraries, so that you do not inadvertently use any of these names for your own libraries.

Table 1-1
Shared Library Pathnames

Library	R-mode File	V-mode File
PRIMOS (including file system, condition mechanism, controllers, semaphore handlers, and IOCS)	LIB>FTNLIB.BIN	LIB>PFTNLB.BIN
Application	LIB>APPLIB.BIN	LIB>VAPPLB.BIN
In-memory sorts	LIB>MSORTS>BIN	LIB>VMSORT.BIN
Matrix	LIB>MATHLB.BIN	not available
Sort	LIB>SRTLIB.BIN	LIB>VSRTLI.BIN
Spool	LIB>SPOOL$.BIN	LIB>VSPOO$.BIN

If you get a runtime error message when you try to execute a program that calls subroutines, BIND the program again. Then, after the LI command, use MAP -UNDEFINED (or MAP 3 with SEG) to display the names of any missing subroutines. If necessary, refer to the subroutines' descriptions in the other volumes of this guide for information about the libraries required by the subroutines. MAP -UNDEFINED, along with other linking options, is explained in detail in the Programmer's Guide to BIND and EPFs and in Volume I of the Advanced Programmer's Guide. The MAP 3 option for SEG is explained in the SEG and LOAD Reference Guide.

The loading process is different for BASIC/VM, which performs the compiling, linking, loading, and execution within the special environment it creates. For information about BASIC/VM, see the BASIC/VM Programmer's Guide.

SUBROUTINES AND ADDRESSING MODES

Some subroutine libraries are available in only some of the addressing modes that Prime currently supports: R mode, V mode, and I mode. Most subroutines are available only in V mode and I mode. However, a number of older system subroutines exist only in R mode. R-mode subroutines can be called only from R-mode programs. To compile a program written in FTN in R mode, you must specify the compile option -32R or -64R.

All standard subroutines introduced with Revision 19.4 or later revisions are invoked through direct entry calls, by means of dynts. Direct entry calls execute subroutines within PRIMOS, and are faster than other calls. Direct entry calls are available only in V mode and I mode.

To find out which addressing modes a subroutine is available with, see the subroutine's description in one of the subsequent volumes of this guide. For subroutines available in I mode, and many subroutines in V mode, the usage descriptions are in PL/I notation. For certain subroutines meant for use by FORTRAN programmers, the Usage descriptions are written in FTN. For subroutines available only in R mode, the usage descriptions are in FTN. Some Input-Output Control System (IOCS) and applications library subroutines are meant to interface with FTN programs; accordingly, these Usage descriptions are written in FTN.

For more information about addressing modes, see the System Architecture Reference Guide.

2 Using Subroutines

INTRODUCTION

This chapter explains how to use the standard subroutines supplied by Prime. It covers the following topics:

- How to call subroutines and functions
- How to specify arguments of subroutines and functions
- Data types used by subroutines and functions written in FORTRAN and PL/I
- How to use and interpret key codes, argument codes, and error codes

CALLING SUBROUTINES

Each of the calling languages discussed in chapters 3 through 9 of this volume has its own statement for calling subroutines. For example, PL/I uses a statement of the following form to call subroutines.

```
CALL subroutine(argument1, argument2...);
```

<u>Subroutine</u> is the name of the subroutine called. The subroutine accepts input from and returns output to the arguments specified as <u>argument1</u>, <u>argument2</u>, and so on. For example, the subroutine GV$GET, as called by the PL/I statement shown below, retrieves the value of a variable named <u>gvname</u> and returns it to a variable named <u>gvalue</u>. The argument <u>size</u> specifies the length in characters of <u>gvalue</u>, and <u>code</u> receives the error code returned by GV$GET.

```
    CALL GV$GET(GVNAME, GVALUE, SIZE, CODE);
```

For information about the call statement for a particular calling language, see the chapter in this volume that describes how to call subroutines from that language.

CALLING FUNCTIONS

Some of the modules described in the other volumes of this guide are functions rather than subroutines. Functions differ from subroutines in the way that they must be invoked by the calling program and in the way that they return output to the calling program. A function accepts an argument or arguments and returns a value, which can then be assigned to a variable or used in expressions. To call functions, use formats such as the following:

```
    variable = function(argument or arguments);
```

For example, the function DELE$A, shown below, accepts two arguments and assigns a value to the variable <u>value1</u>:

```
    value1 = DELE$A(arg1, arg2);
```

You can also use a function with relational, arithmetic, or other operators.

SUBROUTINE ARGUMENTS

Most subroutines expect to receive from the calling program one or more arguments in a given order. If the subroutine receives fewer arguments than it expects, a message such as POINTER FAULT or ILLEGAL SEGNO is displayed when the program is executed. If too many arguments are passed, the subroutine ignores the extra arguments.

Subroutines and the programs that call them need not be written in the same language. However, the arguments that a program passes to a subroutine or function must be of data types that correspond to the data types expected by the subroutine or function. Chapters 3 through 9 of this volume describe how subroutine arguments must be declared in different calling languages in order to be acceptable to subroutines and functions. The following paragraphs describe the data types commonly expected and returned by system subroutines and functions written in PL/I and FTN.

PL/I Data Types

Subroutines and functions written in PL/I expect parameters and return values of the following data types:

CHAR(n)
: Also specified as CHARACTER(n), CHARACTER(n) NONVARYING. Specifies a character string or array of length _n_. A CHAR(n) string is stored as a byte-aligned string, one character per byte. A byte is 8 bits.

CHAR(*)
: Also CHARACTER(*), CHARACTER(*) NONVARYING. Specifies a character string or array whose length is unknown at the time of declaration. A CHAR(*) string is stored as a byte-aligned string, one character per byte.

CHAR(n) VAR
: Also CHARACTER(n) VARYING. Specifies a character string or array whose length can be a maximum of _n_ characters. The first 2 bytes (one halfword) of storage for a CHAR(n) VAR string contain an integer that specifies the current string length; these are followed by the string, one character per byte.

CHAR(*) VAR
: Also CHARACTER(*) VARYING. Specifies a character string or array whose maximum length is unknown at the time of declaration. The first 2 bytes (one halfword) of storage for a CHAR(*) VAR string contain an integer that specifies the current string length; these are followed by the string, one character per byte.

FIXED BIN
: Also FIXED BINARY, BIN, FIXED BIN(15). Specifies a 16-bit (halfword) signed integer.

FIXED BIN(31)
: Specifies a 32-bit signed integer.

(n) FIXED BIN
: Specifies an integer array of _n_ elements. See below for more information about arrays.

FLOAT BIN
: Also FLOAT BIN(23), FLOAT. Specifies a 32-bit (one-word) floating-point number.

FLOAT BIN(47)
: Specifies a 64-bit (double-word) floating-point number.

BIT(1)
: Specifies a logical (Boolean) value. A bit value of 1 means TRUE; a value of 0 means FALSE.

BIT(n)
: Specifies a bit string of length _n_. BIT(n) ALIGNED means that the bit string is to be aligned on a halfword boundary.

POINTER
: Also PTR. Specifies a POINTER data type. A pointer is stored in three halfwords (48 bits). If the pointer will point only to halfword-aligned data, it may occupy two halfwords (32 bits). The item to which the pointer points is declared with the BASED attribute (for example, BASED FIXED BIN).

POINTER OPTIONS (SHORT)
: Same as POINTER except that it always occupies only two halfwords and can point only to halfword-aligned data.

Note

When used as a parameter, POINTER can be used interchangeably with POINTER OPTIONS (SHORT).

When used as a returned function value, POINTER OPTIONS (SHORT) can be used in any high-level language except Pascal or 64V-mode C, which require returned pointers to be three halfwords; in these cases, POINTER must be used. C in 32I X-mode accepts only halfword-aligned, two-halfword pointers, and therefore requires the use of POINTER OPTIONS (SHORT).

Declaring Arrays and Structures in PL/I

Sometimes an argument is defined as an array or a structure. For example, the following DCL statement declares _item_ as an array of ten integers.

```
DCL ITEMS(10) FIXED BIN;
```

In the DCL statement above, you can replace the keywords FIXED BIN with any data type. By default, arrays are indexed starting with the subscript 1; the first integer in this array is ITEMS(1).

To declare an array with a starting subscript other than 1, use a range specification, as for example:

```
    DCL WORD(0:1023) BASED FIXED BIN;
```

WORD is an array indexed from 0 to 1023, and its elements are referenced by POINTER variables.

A structure is equivalent to a record in COBOL or Pascal. For example, the following DCL statement declares a structure named FS_DATE.

```
    DCL 1 FS_DATE,
            2 YEAR BIT(7),
            2 MONTH BIT(4),
            2 DAY BIT(5),
            2 QUADSECONDS FIXED BIN(15);
```

In the DCL statement above, the numbers 1 and 2 indicate the relative level numbers of the items in the structure. Always declare the name of the structure at level 1. After the level number, give the name of the data item and its data type. In this example, the structure occupies a total of 32 bits.

Since no names are given to data items in parameter lists, you can declare the array ITEMS simply as (10) FIXED BIN. Similarly, you can declare the structure FS_DATE as

```
    (..., 1, 2 BIT(7), 2 BIT(4), 2 BIT(5), 2 FIXED BIN(15), ...)
```

FTN Data Types

Subroutines and functions written in FTN expect parameters and return values of the following data types:

COMPLEX
: Specifies a 64-bit element to hold a complex number, defined as two 32-bit (REAL*4) entities, the first for its real and the second for its imaginary part.

INTEGER*2
: Also INTEGER. Specifies a 16-bit (halfword) signed integer. Bit 1 is the sign bit.

INTEGER*4
: Specifies a 32-bit signed integer. Bit 1 is the sign bit.

LOGICAL
: Specifies a logical (Boolean) value. Within a 16-bit halfword, the first 15 bits must be 0, and the 16th bit indicates .FALSE. with 0 and .TRUE. with 1.

REAL*4
: Also REAL. Specifies a 32-bit signed floating-point number. Bit 1 is the sign bit. Bits 2 to 24 are the mantissa. Bits 25 to 32 are the exponent.

REAL*8
: Also DOUBLE PRECISION. Specifies a 64-bit signed floating-point number. Bit 1 is the sign bit. Bits 2 to 48 are the mantissa. Bits 49 to 64 are the exponent.

Data Types Variants for FORTRAN

Other declarations in the _Usage_ section suggest the elements for which FTN has no data type:

BUFFER(1)
: Given the data type of INTEGER*2, this shorthand declaration for an array suggests a character string or array whose length is unknown at the time of declaration (an equivalent to CHAR(*) in PL/I). The user must DIMENSION the array with an adequate size. If the size is known to be (n), then the variable declaration is given as BUFFER(n).

LOC(variable)
: Specifies the equivalent of a POINTER data type. This built-in FORTRAN function automatically provides the prerequisite three halfwords (48 bits) for the pointer.

Key Codes and Argument Codes

In calls to many subroutines, key codes and argument codes can be used in place of numeric arguments. For example, in the subroutine call

```
CALL GPATH$ (K$INIA...other arguments...)
```

the key code K$INIA corresponds to the number 4 and tells GPATH$ to return the pathname of the user's origin directory.

Files in the SYSCOM directory define which numbers the codes represent. If the proper SYSCOM file is inserted in a program, the codes defined in that file can be used by the program as arguments in calls to many subroutines. For information about how to insert a SYSCOM file into a program, see the chapter in this volume that explains how to call subroutines from the program's language. It is good practice to use key codes and argument codes whenever possible.

Key codes are of the form K$*yyyy*, where *yyyy* is a string of up to four characters. For example, K$CURR is a key code. Key codes can be used with the subroutines described in Volumes II and III of this guide.

Argument codes are of the form A$*yyyy*, where *yyyy* is a string of up to four characters. For example, A$DEC is an argument code. Argument codes are used in calls to application library subroutines; these subroutines are described in Volume IV of this guide.

Key codes are associated with numeric values in the file SYSCOM>KEYS.INS.*language*; argument codes are associated with numeric values in the file SYSCOM>A$KEYS.INS.*language*. In these file names, *language* stands for an abbreviation designating the language of the calling program. Table 2-1 lists the abbreviations that can be substituted for *language* in the names of files in SYSCOM. Note that programs written in COBOL, CBL, and BASIC/VM cannot use the keys and codes defined in the SYSCOM files; in these languages, programs must specify the numeric equivalents of the keys and codes.

Table 2-1
Language Abbreviations in SYSCOM File Names

Language	Abbreviation
C	CC
FTN, F77	FTN
Pascal	PASCAL
PL/I	PL1
PMA	PMA

Some subroutines accept as a single argument a number of keys or key codes linked by plus signs (+). For example, the subroutine SRCH$$ is called by a statement of the following form.

```
    CALL SRCH$$ (action+ref+newfill,...other arguments...)
```

In this CALL statement, keys corresponding to the parameters *action*, *ref*, and *newfill* must be linked by plus signs. The subroutine SRCH$$ accepts the sum of these three keys as a single argument. For example, in the following call to SRCH$$

```
    CALL SRCH$$(K$RDWR+K$ISEG+K$NDAM,...other arguments...)
```

the key codes K$RDWR, K$ISEG, and K$NDAM are linked by plus signs (+) into a single argument.

Standard Error Codes

Many subroutines include an argument that the subroutine sets to a standard error code. The error code corresponds to a number reporting on the success or failure of the call or on some other condition worth noting.

Standard error codes are of the form E$xxxx, where xxxx is any combination of letters. For example, the error code

```
    E$DVIU
```

corresponds to the number 39, which means Device in Use.

Files named SYSCOM>ERRD.INS.language, where language is an abbreviation standing for the language of the calling program, associate standard error codes with numbers. Table 2-1 above lists the abbreviations that can be substituted for language in the names of files in SYSCOM.

Subroutines return the error code number whether or not you insert the SYSCOM>ERRD.INS file. However, if you wish to intercept errors and have your program write error messages, you should include the SYSCOM file and refer to the error by its code rather than its number.

For more information about standard error codes, see Volume 0 of the Advanced Programmer's Guide.

HOW TO READ SYSCOM FILES

You can learn the numeric equivalent of a key code, argument code, or error code by listing the SYSCOM file that defines the code for the calling language that you are using. A SYSCOM file defines a code on a line consisting of the code, the code's numeric equivalent, and a comment describing the significance of the code. A comment symbol (/*) in front of a code invalidates the code. Codes used with the same subroutine are grouped together in the SYSCOM file.

Figure 2-1 lists the portion of SYSCOM>KEYS.INS.FTN that defines the key codes that can be used by any FTN program which invokes the subroutine SRCH$$. For example, in Figure 2-1, K$RDWR is defined as the equivalent of the number 3. As the comment following the key definition indicates, you can use either the key K$RDWR or the number 3 as an argument of SRCH$$ to cause the subroutine to open a file for reading and writing.

```
X /******************** SRCH$$ ********************          */
X /*              ****** ACTION ******                       */
X /* K$READ = :1,      /* OPEN FOR READ                       */
X /* K$WRIT = :2,      /* OPEN FOR WRITE                      */
X    K$RDWR = :3,      /* OPEN FOR READING AND WRITING        */
X    K$CLOS = :4,      /* CLOSE FILE UNIT                     */
X    K$DELE = :5,      /* DELETE FILE                         */
X    K$EXST = :6,      /* CHECK FILE'S EXISTENCE              */
X    K$BKUP = :7,      /* OPEN FOR READ BY BACKUP UTILITY     */
X    K$VMR  = :20,     /* OPEN FOR VMFA READING               */
X    K$BKIO = :20000, /* OPEN FOR BLOCK MODE I/O              */
X    K$GETU = :40000, /* SYSTEM RETURNS UNIT NUMBER           */
X    K$RESV = :100000,/* reserved                             */
```

Excerpt from SYSCOM>KEYS.INS.FTN
Figure 2-1

3 Calling Subroutines From BASIC/VM

CALL FORMAT

Before a program written in BASIC/VM can call a subroutine, the program must declare the data types of the subroutine's parameters. To declare the data types of a subroutine's parameters in BASIC/VM, use a statement of the following form:

```
    SUB FORTRAN sub-name [(type, type...)]
```

In the SUB statement, only the FORTRAN-style data types INT, INT*4, REAL, or REAL*8 can be declared. BASIC/VM supports only two types of operand, strings and double-precision (64-bit) floating point. However, the BASIC/VM compiler performs all conversions of BASIC/VM operands to and from the subroutine argument types.

Note

> A BASIC/VM program can call only those subroutines that expect parameters of data types equivalent to INTEGER, INTEGER*2, LOGICAL, INT*4, REAL, or REAL*8. The SUB FORTRAN statement uses the type specifier INT to correspond to INTEGER, INTEGER*2, or LOGICAL.

To call a subroutine from a program written in BASIC/VM, use the following statement.

```
CALL sub-name [(argument1, argument2 ...)]
```

Literals can be used as arguments in BASIC/VM subroutine calls.

External functions cannot be called as functions from BASIC/VM. However, you can call most functions in this manual as subroutines, using the CALL statement described above.

For more information about BASIC/VM, see the BASIC/VM Programmer's Guide.

USING NUMERIC EQUIVALENTS OF SYSCOM KEYS

BASIC/VM does not recognize the key codes, argument codes, and error codes defined by files in the SYSCOM directory. In calls to subroutines, BASIC/VM programs must specify the numeric equivalents of these keys and codes. To learn the numeric equivalent of a SYSCOM key or code, list the SYSCOM file that defines the key or code. Chapter 2 of this volume explains which SYSCOM files define the key codes, argument codes, and error codes for each calling language.

SYSTEM SUBROUTINES NOT RECOGNIZED BY BASIC/VM

Some of the FORTRAN subroutines in VAPPLB are not recognized by the BASIC/VM compiler, and, therefore, cannot be called by BASIC/VM commands. If a program that calls a subroutine in VAPPLB compiles correctly but gives the runtime error message:

```
Entry name xxx not found
```

the subroutine is missing from the BASIC/VM compiler and must be installed. Your System Administrator may install more subroutines from VAPPLB (or user-written subroutines) in the BASIC/VM compiler, as explained in the System Administrator's Guide or the BASIC/VM Programmer's Guide.

DATA TYPES

Table 3-1 illustrates ways that FORTRAN and PL/I data types can be represented in a SUB FORTRAN declaration in BASIC/VM. The BASIC/VM numeric data type is REAL*8. When BASIC/VM interprets the CALL statement, it converts all scalars and arrays to INT or REAL.

For information about each data type, see the chapter titled "Overview of Subroutines" in Volume II, III, or IV of this guide. The sections that follow the table illustrate how arguments expected by subroutines coded in FORTRAN or PL/I can be declared in BASIC/VM programs.

Table 3-1
Data Type Equivalents: BASIC/VM

Generic Unit	Declared in SUB FORTRAN statement	PL/I	FTN	F77
16 bits (Halfword)	INT	FIXED BIN FIXED BIN(15)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2
32 bits (Word)	INT*4	FIXED BIN(31)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4
Varying character string	INT	CHAR(n) VARYING		
32 bits (Float single precision)	REAL	FLOAT BINARY FLOAT BIN(23)	REAL REAL*4	REAL REAL*4
64 bits (Float double precision)	REAL*8	FLOAT BIN(47)	REAL*8	REAL*8
Byte string (Max. 32767)	INT	CHAR(n)	Integer Array	CHARACTER *n

Note

String arrays in BASIC/VM cannot be passed as arguments to FORTRAN subroutines.

<u>BASIC/VM SUB Statement: INT</u>
<u>FTN and F77: INTEGER*2</u>
<u>PL/I: FIXED BIN(15) or FIXED BIN</u>

Use the data type INT in the BASIC/VM SUB FORTRAN statement to declare the FTN and F77 data type INTEGER*2 and the PL/I data type FIXED BIN(15) or FIXED BIN. In BASIC/VM, the variable or constant to be passed is the normal numeric operand, which is double-precision floating point, and is not declared.

For example, a BASIC/VM program that calls subroutine SRCH$$ must declare data types that correspond to the data types in the subroutine's DCL statement, as follows:

```
    DCL SRCH$$ ENTRY (FIXED BIN, CHAR (32) VAR, FIXED BIN, FIXED BIN,
                      FIXED BIN, FIXED BIN);
```

The following BASIC/VM statement declares each of the parameters of SRCH$$ as INT, which corresponds to the data types declared in the DCL statement above.

```
    40 SUB FORTRAN SRCH$$(INT, INT, INT, INT, INT, INT)
```

The following BASIC/VM statements assign values to the variables that are to be used as arguments of SRCH$$.

```
    50   N = 6
    60   F$ = 'CTRLFL'
    70   L = 6
    80   F = 1
    90   T = 0
```

The argument <u>N</u> illustrates how BASIC/VM calling programs use numeric arguments in place of SYSCOM keys. The value assigned to <u>N</u> is 6, the sum of the key K$EXST (=6), which instructs SRCH$$ to verify the existence of a file, and K$IUFD (=0), which instructs SRCH$$ to look for the file in the directory to which the user is currently attached. The argument F$ is assigned the name of the file for which SRCH$$ is to search ('CTRFL'). The values assigned to <u>L</u>, <u>F</u>, and <u>T</u> do not correspond to SYSCOM keys or codes.

The following BASIC/VM statement calls subroutine SRCH$$ with the arguments defined above.

```
    100 CALL SRCH$$(N,F$,L,F,T,C)
```

The variable C in the CALL statement above receives the standard error code reported by SRCH$$.

BASIC/VM SUB Statement: INT
FTN: LOGICAL
F77: LOGICAL*2

Use the data type INT in the BASIC/VM SUB FORTRAN statement to declare parameters of the data types LOGICAL or LOGICAL*2. In the BASIC/VM program, variables or constants to be passed to the subroutine should be used as normal numeric operands (not explicitly declared). They have a value of 0 (false) or 1 (true).

For example, subroutine TEXTO$ expects as parameters an integer array, two INTEGER*2 variables, and a LOGICAL variable. A BASIC/VM program that calls TEXTO$ must declare data types that correspond to the types expected by TEXTO$. Statement 50 below declares TEXTO$ as a BASIC/VM routine with four parameters of the data type INT.

```
    50  SUB FORTRAN TEXTO$(INT, INT, INT, INT)
```

The following statements cause values to be assigned to the arguments N$ and L1.

```
    60  N$ = '
    70  PRINT
    80  INPUT "ENTER NAME OF FILE TO BE CREATED: ", N$
    90  PRINT
    100 L1 = LEN(N$)
```

The following statement calls subroutine TEXTO$.

```
    110 CALL TEXTO$(N$, L1, L2, T)
```

The following statements specify conditional logic based on the value of T, the LOGICAL argument which TEXTO$ sets to 0 or to 1.

```
    120 IF T <> 0 GOTO 210
    130 REM
    140 REM              LOGICAL T IS FALSE
    150 REM
    160 PRINT "INVALID NAME - TRY AGAIN"
    170 GOTO 80
    180 REM
    190 REM              LOGICAL T IS TRUE
```

```
200 REM
210 IF T = 1 GOTO 240
220 PRINT "ERROR: TRY AGAIN"
230 GOTO 80
240 PRINT "LENGTH IS", L2
250 PRINT "TRUTH VALUE IS", T
260 PRINT "END OF RUN"
```

BASIC/VM SUB Statement: INT
PL/I: CHARACTER(n)NONVARYING

BASIC/VM can pass a character string to a subroutine or function expecting a parameter of the type CHARACTER(n)NONVARYING, usually declared CHARACTER(n). Declare the string INT in the BASIC/VM SUB FORTRAN statement. The BASIC/VM program must pass the number of characters expected by the subroutine.

For example, a BASIC/VM program that calls subroutine SPAS$$ must declare data types that correspond to the data types in the following DCL statement:

```
DCL SPAS$$ ENTRY (CHAR(6), CHAR(6), FIXED BIN);
```

The following BASIC/VM statement declares each of the parameters of SPAS$$ as INT, which corresponds to the data types declared in the DCL statement above.

```
20 SUB FORTRAN SPAS$$ (INT, INT, INT)
```

The following BASIC/VM statements assign values to the variables that are to be used as arguments of SPAS$$. Note that each of the strings to be passed to SPAS$$ consists of six characters, the number of characters expected by SPAS$$.

```
30 O = 'OWNSPW'
40 N = 'NOWNPN'
```

The following statement calls SPAS$$ with the arguments defined above.

```
50 CALL SPAS$$ (O,N,C)
```

In the CALL statement above, the variable C is the FIXED BIN parameter and receives the error code returned by the subroutine.

BASIC/VM SUB Statement: INT*4
FTN: INTEGER*4
F77: INTEGER*4 or LOGICAL*4
PL/I: FIXED BIN(31)

Use the data type INT*4 in the BASIC/VM SUB FORTRAN statement to declare parameters of the FTN data type INTEGER*4, the F77 data types INTEGER*4 and LOGICAL*4, and the PL/I data type FIXED BIN(31). In BASIC/VM, the variable or constant to be passed is the normal numeric operand, which is double-precision floating point, and is not declared.

For example, when invoked by a CALL statement, the subroutine RNUM$A expects a parameter of any data type, followed by two INTEGER*2 parameters and an INTEGER*4 parameter. The following BASIC/VM statement declares data types for the parameters of RNUM$A.

```
    50 SUB FORTRAN RNUM$A(INT, INT, INT, INT*4)
```

The following statements assign values to variables that are to be used as arguments of RNUM$A.

```
    20  F$ = 'ENTER A NUMBER'
    30  L = 14
    40  N = 1
```

The following statement calls function RNUM$A, using arguments defined in the statements given above.

```
    60 CALL RNUM$A(F$,L,N,V)
```

In the CALL statement above, V is the INTEGER*4 parameter and receives the returned value of the subroutine.

BASIC/VM SUB Statement: INT or INT*4
FTN: Integer Arrays

An FTN integer array should be declared in the BASIC/VM SUB FORTRAN statement as INT or INT*4, depending on the subroutine. Integer arrays in FTN can contain either numbers or characters. In the BASIC/VM CALL statement, the array should be called either as the array x(y), where _x_ is the variable name and _y_ is the dimension, or as the string _x$_ with the proper number of characters.

For example, the subroutine TIMDAT returns the date, the time, and other system information. A BASIC/VM program must call TIMDAT twice to collect all this information, because BASIC/VM cannot store both characters and integers in a single structure. Thus, BASIC/VM must call TIMDAT once to collect the integers in an array, and again to collect the characters in a string. The following data structure is the PL/I equivalent of the array that BASIC/VM must use to collect integers from TIMDAT:

```
1,
   2 CHAR(6),
   2,
     3 FIXED BIN,
     3 FIXED BIN,
     3 FIXED BIN,
   2,
     3 FIXED BIN,
     3 FIXED BIN,
   2,
     3 FIXED BIN,
     3 FIXED BIN,
   2 FIXED BIN,
   2 FIXED BIN,
   2 CHAR(32);
```

A BASIC/VM program which calls TIMDAT must declare data types that correspond to an array and to FIXED BIN. The final FIXED BIN parameter of TIMDAT must be declared INT in the BASIC/VM SUB FORTRAN statement; 28 is the usual value assigned this parameter.

The following statement declares subroutine TIMDAT as a FORTRAN subroutine with two parameters, each of the data type INT.

```
10   SUB FORTRAN TIMDAT (INT, INT)
```

Statement 20 below allocates an array, A, with 15 elements. Statement 30 calls subroutine TIMDAT to read information into array A.

```
15   REM COLLECT INTEGER DATA
20   DIM A(15)
30   CALL TIMDAT(A(), 28)
```

Statement 40 below writes 30 space characters into string A$. Statement 50 calls subroutine TIMDAT to read information into A$.

```
35   REM COLLECT CHARACTER DATA
40   A$ = SPA(30)
50   CALL TIMDAT(A$,28)
```

The following statements display the numeric and alphabetic information read into array A and string A$ by the two calls to TIMDAT.

```
    60   PRINT 'MONTH: ':LEFT(A$,2)
    70   PRINT 'DAY: ':MID(A$,3,2)
    80   PRINT 'YEAR: ':MID(A$,5,2)
    90   PRINT 'TIME IN MINUTES SINCE MIDNIGHT: ':A(3)
    100  PRINT 'TIME IN SECONDS: ':A(4)
    110  PRINT 'TIME IN TICKS: ' :A(5)
    120  PRINT 'LOGIN NAME: ':RIGHT(A$, 25)
```

Caution

Multidimensional arrays cannot be passed to FORTRAN from other languages, because FORTRAN is the only language to use a column-row format.

BASIC/VM SUB Statement: REAL
FTN and F77: REAL or REAL*4
PL/I: FLOAT BIN or FLOAT BIN(23)

Use the data type REAL in the BASIC/VM SUB FORTRAN statement to declare parameters of the FTN and F77 data type REAL or REAL*4, and of the PL/I data type FLOAT BIN(23), also known as FLOAT BIN. In BASIC/VM, the variable or constant to be passed should be used as the normal numeric operand, which is double-precision floating point, and is not declared.

BASIC/VM SUB Statement: REAL*8
FTN and F77: REAL*8
PL/I: FLOAT BIN(47)

Use the data type REAL*8 in the BASIC/VM SUB FORTRAN statement to declare parameters of the FTN and F77 data type REAL*8 and of the PL/I data type FLOAT BIN(47), also called FLOAT BIN. In BASIC/VM, the variable or constant to be passed should be the normal numeric operand, which is double-precision floating point, and is not declared.

BASIC/VM SUB Statement: INT
FTN and F77: INTEGER*2
PL/I: BIT(1) ALIGNED

The PL/I data type BIT(1) ALIGNED can be treated the same as the INTEGER*2 data type, whose value is -1 if false. Declare parameters of this type INT in the BASIC/VM SUB FORTRAN statement. Note that the PL/I data type BIT(1) cannot be passed from a BASIC/VM program.

4
Calling Subroutines From C

CALL FORMAT

To call a subroutine from a program written in C, use a statement of the following form:

```
    sub-name ([argument1, argument2,...,argumentn]);
```

In this statement, sub-name is the name of the subroutine, and the arguments in brackets are the arguments that the C program is to pass to the subroutine. The arguments must be separated by commas and the list of arguments must be delimited by parentheses. The data type of each argument must be declared in the C program that calls the subroutine.

To call a function from a program written in C, you can use a variety of statements. For example, you can use the same type of statement that you use to call a subroutine. You can also use a statement that assigns the value of the function to a variable, such as:

```
    value = function(argument);
```

In the statement above, the variable value receives the value of the subroutine.

The C program that calls a function must declare the data type of the function, as well as the data types of the function's arguments and of the variable that receives the function's value. The data type of the function is the data type of the value that it returns. For example, if function foo returns a double value, the function itself must be declared double, such as:

```
    double foo();
```

If variable value is to receive the value of foo, then value must also be declared double, such as:

```
    double value;
```

If the data type of the variable that receives the function's value is not declared, the C compiler assumes that the data type is int.

For more information about how to call subroutines and functions from C, see the C User's Guide.

The FORTRAN Storage Class

Any non-C subroutine called by a program written in C should be declared as the FORTRAN storage class. If the subroutine is not declared as the FORTRAN storage class, the C language by default converts the CHAR and SHORT INT data types to INT, and the FLOAT data type to DOUBLE. Declaring the subroutine as the FORTRAN storage class prevents C from performing this conversion. All the examples in this chapter use the FORTRAN storage class for PRIMOS subroutines.

See the C User's Guide for information about accessing common blocks, creating common blocks from C, transferring arguments in C, and passing arrays by reference.

USING THE -OLDFORTRAN AND -NEWFORTRAN OPTIONS

When you compile programs in 64V mode and do not declare subroutines as the FORTRAN storage class, you can use the compiler keywords -OLDFORTRAN and -NEWFORTRAN to tell the C compiler which language interface to use. The -OLDFORTRAN option selects the old interface, and the -NEWFORTRAN option selects the new interface. The 32IX-mode C compiler supports only the new interface.

If you specify neither the -OLDFORTRAN nor -NEWFORTRAN option on the command line, the new interface is selected by default. If the source code is not written for the new interface, you must either change the source code to adopt the new interface conventions, or specify -OLDFORTRAN on the command line when you invoke the compiler. The new interface conventions are documented in Prime's C User's Guide.

It is good practice to use the -NEWFORTRAN interface, which is faster and more flexible than the -OLDFORTRAN interface.

Note

> When the old interface is used, the ampersand character (&) must be placed in front of variables in calls to non-C subroutines to cause the variables to be passed by reference.

USING THE -NOCONVERT OPTION

If a C subroutine is being called from another Prime-supported language such as FORTRAN or PL/I, the conversion of CHAR, SHORT, and FLOAT data types does not occur. The C compiler, however, is not aware of this. Therefore, the -NOCONVERT compiler option must be used to inform the C compiler that data types of CHAR, SHORT, and FLOAT should not be converted. For more information about data type conversion and the -NOCONVERT option, see the C User's Guide.

USING SYSCOM FILES

To enable a program written in C to use standard error codes, insert the file SYSCOM>ERRD.INS.CC into the program by including the following statement in the program:

```
    #include <errd.ins.cc>
```

To enable a program written in C to use key codes, insert the file SYSCOM>KEYS.INS.CC into the program by including the following statement in the program:

```
    #include <keys.ins.cc>
```

Subroutines in VAPPLB use argument codes in the form A$yyyy. These codes are associated with numbers in the file SYSCOM>A$KEYS.INS.CC. To enable a program written in C to use argument codes, include the following statement in the program:

```
    #include <a$keys.ins.cc>
```

The BIND subcommand LI VAPPLB must be issued at load time.

DATA TYPES

Table 4-1 suggests ways that FORTRAN and PL/I data types can be represented in C. For information about each data type, see the chapter titled "Overview of Subroutines" in Volume II, III, or IV of this guide.

Table 4-1
Data Type Equivalents: C

Generic Unit	C	PL/I	FTN	F77
16 bits (Halfword)	short	FIXED BIN FIXED BIN(15)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2
32 bits (Word)	long int	FIXED BIN(31)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4
32 bits (Float single precision)	float	FLOAT BINARY FLOAT BIN(23)	REAL REAL*4	REAL REAL*4
64 bits (Float double precision)	double	FLOAT BIN(47)	REAL*8	REAL*8
1 bit	short	BIT BIT(1)		
1 left-aligned bit (Halfword)	short	BIT(1) ALIGNED		

Table 4-1 (continued)
Data Type Equivalents: C

Generic Unit	C	PL/I	FTN	F77
Bit string	unsigned int	BIT(n)		
16 bits (Halfword)	char		LOGICAL LOGICAL*1	LOGICAL LOGICAL*1
Byte string (Max. 32767)	char[n]	CHAR(n)	Integer Array	CHARACTER *n
Record	struct	CHAR(*) VARYING		
32 bits (Two halfwords)	pointer (32IX mode)	pointer OPTIONS (SHORT)		
48 bits (Three halfwords)	pointer (64V mode)	pointer		
Literal string or character array	literal string or character array	ASCII character string	ASCII character string	ASCII character string

Note

There are no equivalents in FTN or PL/I to the enumeration C data type or to the void C data type.

The following sections suggest how FORTRAN and PL/I data types can be represented in C.

C: short int
FTN: INTEGER*2 or LOGICAL
F77: INTEGER*2 or LOGICAL*2
PL/I: FIXED BIN(15) or FIXED BIN

The C data type short int, also known as short, can be used as an equivalent of the FTN data types INTEGER*2 and LOGICAL, of the F77 data types INTEGER*2 and LOGICAL*2, and of the PL/I data type FIXED BIN (15), also known as FIXED BIN.

For example, a C program that calls subroutine SRCH$$ must declare data types that correspond to the data types in the DCL statement:

```
DCL SRCH$$ ENTRY (FIXED BIN, CHAR (32) VAR, FIXED BIN, FIXED BIN,
                  FIXED BIN, FIXED BIN)
```

The following C statement declares five variables as the data type short.

```
short key, name_len, funit, type, code;
```

The following C statement declares that the subroutine SRCH$$ is of the FORTRAN storage class.

```
fortran srch$$();
```

The following C statements assign values to four variables which are to be used as arguments of SRCH$$.

```
key = k$exst + k$iufd;
name_len = 6;
funit = 0;
type = 0;
```

The following C statement calls subroutine SRCH$$, using arguments defined in the program code above.

```
srch$$ (key, "ctrlfl", name_len, funit, type, code);
```

C: long int
FTN: INTEGER*4
F77: INTEGER*4 or LOGICAL*4
PL/I: FIXED BIN(31)

The C data type long int, also known as long or int, can be used as an equivalent of the FTN and F77 data type INTEGER*4, of the F77 data type LOGICAL*4, and of the PL/I data type FIXED BIN(31).

For example, the function RNUM$A expects four parameters; the first parameter can be any data type, and the remaining three must be INTEGER*2, INTEGER*2, and INTEGER*4. C programs which call the function RNUM$A must declare data types that correspond to those expected by the function.

The following C statements declare data types for variables that are to be used as arguments of RNUM$A. The variable value is declared as int, which corresponds to INTEGER*4.

```
    static char msg[21] = "Please enter a number";
    short msglen, a$dec;
    int value;
```

The following C statement declares that the subroutine RNUM$A is of the FORTRAN storage class.

```
    fortran rnum$a();
```

The following C statements assign values to two of the variables that are to be used in the call to RNUM$A.

```
    msglen = 21;
    a$dec = 1;
```

The following C statement calls subroutine RNUM$A.

```
    rnum$a (msg, msglen, a$dec, value);
```

C: literal string or character array
FTN, F77, and PL/I: ASCII Character Strings

A C program should pass a literal string or character array to a FORTRAN or PL/I subroutine that expects an ASCII character string.

The example in the preceding section shows how the subroutine RNUM$A can be called with an argument msg, which is defined as a string of 21 characters. The argument msg is declared in the example by the following statement.

```
    static char msg[21] = "Please enter a number";
```

C: float
FTN and F77: REAL*4
PL/I: FLOAT BIN(23)

The C data type FLOAT can be used as an equivalent of the FTN and F77 data type REAL*4 and of the PL/I data type FLOAT BIN(23).

For example, a C program that calls function RAND$A must declare data types that correspond to INTEGER*4 and REAL*4, the data types of the parameters of RAND$A. The REAL*4 parameter of RAND$A can also be declared REAL*8.

The following C statements declare three variables, seed and number, that are to be used when RAND$A is called. The variable number is declared as FLOAT, which corresponds to REAL*4.

```
    int seed;
    float number;
    short k;
```

The following C statement declares function RAND$A as the FORTRAN storage class and its value as the data type FLOAT.

```
    fortran float rand$a();
```

The following C statements call function RAND$A to produce ten numbers at random and to print the numbers.

```
    seed = 1;
    for (k=1; k<=10; k++)
      {
        number = rand$a (seed);
        printf ("%e\n", number);
      }
```

C: double
FTN and F77: REAL*8
PL/I: FLOAT BIN(47)

The REAL*8 data type expected by FORTRAN subroutines is the FLOAT BIN(47) data type in PL/I. These two data types can be declared as double in C.

For example, the return value of function RAND$A (See the example in the preceding section.) can be received in a REAL*8 variable declared double in C. Such a variable, called number, can be declared as follows:

```
    double number;
```

If the return value of the function is received in a variable declared double, the function itself must be declared double, as follows:

```
    fortran double rand$a();
```

C: short
PL/I: BIT, BIT(1), or BIT(1) ALIGNED

The PL/I data type BIT or BIT(1) represents a logical (Boolean) value; a bit value of 1 means TRUE and a value of 0 means FALSE. This data type can be declared short in C.

The PL/I data type BIT(1) ALIGNED specifies a bit-aligned halfword (16 bits). This type can also be declared short in C.

The C programmer must know which of the 16 bits in the short data value is set by the function that returns the short value.

C: unsigned int
PL/I: BIT(n)

The PL/I data type BIT(n) specifies a bit string of length n. This data type can be declared in C as unsigned int, which specifies a bit string as represented by a machine word of 32 bits. There is no control of the length of a bit string in C except by use of existing data types.

C: char
FTN and F77: LOGICAL or LOGICAL*1

The FTN and F77 data types LOGICAL and LOGICAL*1 can be declared in C as char, with only the low order bit of the character being used.

For example, the function DELE$A returns a value to a variable which must be declared as LOGICAL or a corresponding data type. The following statement declares the variable log, which receives the return value of DELE$A, as type char, which corresponds to LOGICAL*1. Note that a variable declared char in C can be evaluated arithmetically. In this example, log, a char variable, is tested to determine whether it is set to zero or to a nonzero value.

```
    char log;
```

DELE$A expects two arguments of the data type INTEGER*2. The following statements declare data types for these arguments and assign values to them.

```
    static char filename[7] = "ctrlfl";
    short count = 6;
```

The first statement above declares an array, FILENAME, as type char, and assigns the characters "ctrlfl" to the array; the data type of this parameter of DELE$A does not matter. The second statement above declares the variable count as type short, and assigns the value 6 to the variable.

The following statement declares the function DELE$A as the FORTRAN storage class.

```
    fortran short dele$a();
```

The following statement calls DELE$A with the arguments declared above.

```
    log = dele$a (filename, count);
```

The following "if...else" statement performs one of two substatements, depending on whether the returned value of DELE$A (log) is zero or nonzero.

```
    if (log == 1)
       printf ("file deleted successfully\n");
    else
       printf ("no go\n");
```

C: Array of Integers and Characters
FTN: Integer and Character Arrays
F77: CHARACTER*n
PL/I: CHAR(n)

Arrays expected by FORTRAN and PL/I subroutines should be declared as an array of integers or as an array of characters in C, depending on the type of array being passed. A FORTRAN integer array containing both integer and character data can be declared in C as a structure of elements each of which is separately declared as an integer or character data type.

For example, a C program that calls subroutine TIMDAT, which returns system and user information, can declare an array that contains both integer and character data. The following C statement defines a data type named array1; this data type is a structure consisting of eleven fields, each of which is of the data type char or the data type short. The char fields are to contain characters and the short fields are to contain integers.

```
    static struct array1
        {
         char mmddyy[6];
         short time_min;
         short time_sec;
         short time_tck;
         short cpu_sec;
         short cpu_tck;
         short disk_sec;
         short disk_tck;
         short tck_sec;
         short user_num;
         char username[31];
        };
```

The following C statement declares the variable _intarray_ as of the data type _array1_, as defined above.

```
    static struct array1 intarray;
```

The following statement declares the variable _num_ as SHORT and assigns it a value of 28. This value must be specified as the second argument of TIMDAT.

```
    short num = 28;
```

The following C statement declares TIMDAT as a FORTRAN subroutine.

```
    fortran timdat();
```

The following C statement calls TIMDAT with _intarray_ and _num_ as arguments.

```
    timdat(intarray, num);
```

The following C statements print the information that TIMDAT has returned to array _intarray_.

```
    printf ("date is               %.6s\n", intarray.mmddyy);
    printf ("seconds elapsed       %d\n", intarray.time_sec);
    printf ("ticks elapsed         %d\n", intarray.time_tck);
    printf ("cpu seconds used      %d\n", intarray.cpu_sec);
    printf ("cpu ticks             %d\n", intarray.cpu_tck);
    printf ("disk seconds used     %d\n", intarray.disk_sec);
    printf ("user name             %.31s\n", intarray.username);
```

C: Two-element structure
PL/I: CHARACTER(*)VARYING

The PL/I data type CHARACTER(*)VARYING is implemented as a record structure, providing a count of the number of characters in the structure followed by the characters themselves. Figure 4-1 illustrates a CHAR(*)VAR record structure.

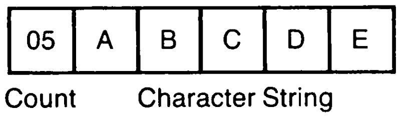

Figure 4-1
CHAR(*) VAR Record Structure

In C, the struct and typedef statements declare data types that are equivalent to the CHAR(*)VAR data type. For more information about these statements, see the C User's Guide.

Note

> The PL/I type CHAR(n) VARYING represents a character string whose length is given by the value n. This data type can be treated the same as CHAR(*) VARYING.

For example, a C program that calls subroutine GV$GET must declare data types that correspond to the data types in the subroutine's DCL statement, as follows:

```
    DCL GV$GET ENTRY (CHAR(*)VAR, CHAR(*)VAR, FIXED BIN, FIXED BIN);
```

The following C statements declare data types for variables that are to be used as arguments of GV$GET. The struct statement defines a data type, charvar, that corresponds to CHAR(*)VAR; charvar consists of a count-of-characters element, declared SHORT, and a five-character string. The static struct statements declare the variables varname and varval to be of the type charvar; varname is assigned the character count 4 and the string value ".MAX".

```
    short varsize, code;
    struct charvar
       {
       short nchars;
       char string1[5];
       };
    static struct charvar varname = {4, ".max"};
    static struct charvar varvalue;
```

The following C statement declares GV$GET as of the FORTRAN storage class.

```
    fortran gv$get();
```

The following statement assigns the value 5 to the variable varsize.

```
    varsize = 5;
```

The following statement calls subroutine GV$GET with the arguments defined above.

```
    gv$get (varname, varvalue, varsize, code);
```

C: pointer
PL/I: POINTER or POINTER OPTIONS (SHORT)

The C data type pointer can be used as an equivalent of the PL/I data type POINTER, also known as PTR. A POINTER item is usually stored in three halfwords (48 bits). If the POINTER item points only to halfword-aligned data, it may occupy two halfwords (32 bits). The item to which the POINTER item points is declared with the BASED attribute (for example, BASED FIXED BIN).

The POINTER OPTIONS (SHORT) data type is the same as POINTER except that it always occupies only two halfwords and can point only to halfword-aligned data.

Note

When used as a parameter, POINTER can be used interchangeably with POINTER OPTIONS (SHORT).

C in 32IX mode accepts only halfword-aligned, two-halfword pointers, and therefore requires the use of POINTER OPTIONS (SHORT). When used as a returned function value, POINTER OPTIONS (SHORT) cannot be used in C in 64V mode, which requires returned pointers to be three halfwords; in this case, POINTER must be used.

5 Calling Subroutines From COBOL or CBL

CALL FORMAT

To call a subroutine from a program written in COBOL or CBL, use a CALL statement of the following format:

```
  CALL 'sub-name' [USING data-name-1 [, data-name-2] ...]
```

In the CALL statement, 'sub-name' is the subroutine's name enclosed by single quotation marks. The data-names should be defined in the DATA division with level-number 01 or 77. In COBOL or CBL, arguments cannot be passed to or returned from a subroutine as literals.

External functions cannot be called from COBOL or CBL. However, most functions in this guide can be called as subroutines, using the CALL statement described above.

CBL incorporates features not supported by COBOL. For information about CBL, see the COBOL 74 Reference Guide.

USING NUMERIC EQUIVALENTS OF SYSCOM KEYS

COBOL and CBL do not recognize the key codes, argument codes, and error codes defined by standard files in the SYSCOM directory. In calls to subroutines, COBOL and CBL programs must specify the numeric equivalents of these keys and codes. To learn the numeric equivalent of a SYSCOM key or code, list the SYSCOM file that defines the key or code. Chapter 2 of this volume explains which SYSCOM files define the key codes, argument codes, and error codes for each calling language.

DATA TYPES

Table 5-1 suggests ways that FORTRAN and PL/I data types can be represented in COBOL or CBL.

Table 5-1
Data Type Equivalents: COBOL and CBL

Generic Unit	CBL	COBOL	PL/I	FTN	F77
16 bits (Halfword)	COMP PIC S9(1)- PIC S9(4)	COMP	FIXED BIN FIXED BIN(15)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2
32 bits (Word)	COMP PIC S9(5)- PIC S9(9)		FIXED BIN(31)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4
32 bits (Float single precision)	COMP-1		FLOAT BINARY FLOAT BIN(23)	REAL REAL*4	REAL REAL*4
64 bits (Float double precision)	COMP-2		FLOAT BIN(47)	REAL*8	REAL*8
Byte string (Max. 32767)	DISPLAY PIC A(n) PIC X(n) FILLER	DISPLAY PIC A(n) PIC X(n) FILLER	CHAR(n)	Integer Array	CHARACTER *n

Table 5-1 (continued)
Data Type Equivalents: COBOL and CBL

Generic Unit	CBL	COBOL	PL/I	FTN	F77
Byte string (2 digits per byte)	COMP-3	COMP-3	FIXED DECIMAL		
Record	two-element item	two-element item	CHAR(*) VARYING		

<u>Note</u>

COBOL has no data types that correspond to the INTEGER*4, FIXED BIN(31), REAL*4, REAL*8, or POINTER data types. CBL has no data type that corresponds to POINTER.

The following paragraphs explain how arguments of these types must be declared in COBOL programs that call subroutines. For more information about each data type, see the chapter titled "Overview of Subroutines" in Volume II, III, or IV of this guide.

<u>CBL: COMP with PIC S9(1) to S9(4)</u>
<u>COBOL: COMP</u>
<u>PL/I: FIXED BIN or FIXED BIN(15)</u>
<u>FTN or F77: INTEGER*2</u>

The COBOL data type COMP, signed or unsigned, can be used as an equivalent of the FTN and F77 data type INTEGER*2 and of the PL/I data type FIXED BIN, also called FIXED BIN(15).

The CBL data type COMP, with an item declared PIC S9(1) to PIC S9(4), can be used as an equivalent of the data types INTEGER*2 and FIXED BIN.

For example, a COBOL or CBL program that calls subroutine TNOUA must declare parameters of the proper COBOL or CBL data types for TNOUA. Subroutine TNOUA has two parameters, with the data types CHAR(*) and FIXED BIN, as indicated by the following DCL statement:

```
DCL TNOUA ENTRY (CHAR(*), FIXED BIN);
```

The following statements from the DATA DIVISION of a COBOL or CBL program declare two variables, ERRBUFF and COUNTER, with data types that correspond to CHAR(*) and FIXED BIN, respectively.

```
    WORKING-STORAGE SECTION.
    01 ERRBUFF                          PIC X(1) VALUE '^207'.
    01 COUNTER                          COMP PIC S9(4) VALUE 1.
```

Note

You can omit the PIC S9(4) clause from declarations of the data type COMP if you compile the program using the COBOL compiler. However, the CBL compiler returns an OBSERVATION error message if the PIC S9(4) clause is omitted from the COMP declarations.

The following statement from the PROCEDURE DIVISION of a COBOL or CBL program calls TNOUA, specifying ERRBUFF and COUNTER as arguments.

```
    CALL 'TNOUA' USING ERRBUFF, COUNTER.
```

COBOL and CBL: COMP
FTN: LOGICAL
F77: LOGICAL*2

The COBOL and CBL data type COMP can be used as an equivalent of the FTN data type LOGICAL and the F77 data type LOGICAL*2. Arguments of this data type must have a value of 0 (false) or 1 (true).

For example, subroutine TEXTO$ has four parameters, of the data types integer array, INTEGER*2, INTEGER*2, and LOGICAL. The following statements from the DATA DIVISION of a COBOL or CBL program declare four variables, FILENAME, NAMELENGTH, TRUELENGTH, and TEXTOK, with data types that correspond to the parameters of TEXTO$. The variable TEXTOK is declared as COMP PIC S9(4), which corresponds to the data type LOGICAL.

```
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01  FILENAME        PIC X(32).
    01  NAMELENGTH      COMP PIC S9(4) VALUE 32.
    01  TRUELENGTH      COMP PIC S9(4).
    01  TEXTOK          COMP PIC S9(4).
```

Note

> You can omit the PIC S9(4) clause from the declaration of the data type COMP if you compile the program using the COBOL compiler. However, the CBL compiler returns an OBSERVATION error message if the PIC S9(4) clause is omitted from the COMP declarations.

The following statement from the COBOL program's PROCEDURE DIVISION calls subroutine TEXTO$.

```
    CALL 'TEXTO$' USING FILENAME,NAMELENGTH,TRUELENGTH,TEXTOK.
```

CBL: COMP with PIC S9(5) through PIC S9(9)
FTN: INTEGER*4
F77: INTEGER*4 or LOGICAL*4
PL/I: FIXED BIN(31)

The CBL data type COMP, with PIC S9(5) through PIC S9(9), can be used as an equivalent of the FTN data type INTEGER*4, of the F77 data types INTEGER*4 and LOGICAL*4, and of the PL/I data type FIXED BIN(31).

For example, the function DATE$ returns the current date and time in binary format as a 32-bit value. In the function, this value is declared as data type FIXED BIN(31), as indicated by the following DCL statement:

```
    DCL DATE$ ENTRY RETURNS (FIXED BIN(31));
```

The following statements from the DATA DIVISION of a CBL program declare the variable FSDATE as COMP PIC S9(5), which corresponds to the data type FIXED BIN(31):

```
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 FSDATE COMP PIC S9(5).
```

The following statement from the PROCEDURE DIVISION of a CBL program calls DATE$, assigning its value to the variable FSDATE.

```
    CALL 'DATE$' USING FSDATE.
```

<u>CBL: COMP-1</u>
<u>PL/I: FLOAT BINARY(23) or FLOAT BINARY</u>
<u>FTN and F77: REAL or REAL*4</u>

The CBL data type COMP-1 can be used as an equivalent of the PL/I data type FLOAT BINARY(23) and of the FTN and F77 data type REAL*4, which can be specified simply as REAL.

For example, the function DTIM$A outputs the disk time since login, in centiseconds, to a variable that must be INTEGER*4; the function value is disk time in seconds, and is returned to a variable that must be REAL*4 or REAL*8. The following statements from the DATA DIVISION of a COBOL or CBL program declare the output variable <u>DSKTIM</u> as COMP PIC S9(5), corresponding to INTEGER*4, and the variable <u>RTVAL</u> as COMP-1, corresponding to REAL*4.

```
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 DSKTIM COMP PIC S9(5).
    01 RTVAL COMP-1.
```

The following statement from the PROCEDURE DIVISION of a COBOL or CBL program calls function DTIM$A; the function returns its value to <u>RTVAL</u>.

```
    CALL 'DTIM$A' USING RTVAL.
```

<u>CBL: COMP-2</u>
<u>PL/I: FLOAT BIN(47)</u>
<u>FTN and F77: REAL*8</u>

The CBL data type COMP-2 can be used as an equivalent of the PL/I data type FLOAT BIN(47) and of the FTN and F77 data type REAL*8.

For example, the function FDAT$A accepts information from RDEN$$ about the date in the format YYYYYYYMMMMDDDD and converts it into the format <u>DAY</u>, <u>MON</u> <u>DD</u> <u>YEAR</u> (for example, FRI, JAN 16 1987). The returned value of the function is the date in the format <u>MM/DD/YY</u> and must be received in a REAL*8 variable. The following lines from a CBL program declares the variable <u>rtval</u> as COMP-2, which corresponds to REAL*8.

```
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 RTVAL COMP-2.
```

The following statement from the PROCEDURE DIVISION of a CBL program calls function FDAT$A.

```
    CALL 'FDAT$A' USING RTVAL.
```

COBOL and CBL: PIC 9(n), PIC X(n), or PIC A(n)
FTN: ASCII Character String

A COBOL or CBL program must declare an ASCII string as PIC 9(n), PIC X(n), or PIC A(n) if it is to pass the string to a FORTRAN subroutine or function.

For example, a COBOL or CBL program that calls subroutine SRCH$$ must pass to SRCH$$ six arguments, including a character string representing a file name. The following DCL statement declares the data types expected by SRCH$$.

```
    DCL SRCH$$ ENTRY (FIXED BIN, CHAR(32) VAR, FIXED BIN,
                      FIXED BIN, FIXED BIN, FIXED BIN);
```

The following statements from such a program declare data types and values for all of the arguments that are to be passed to SRCH$$. These arguments include a variable, <u>NAME</u>, that is declared PIC X(6) and is assigned the value 'CTRLFL', a file name.

```
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 K-EXST       COMP PIC S9(4) VALUE 6.
    01 NAME         PIC X(6) VALUE 'CTRLFL'.
    01 NAMELENGTH   COMP PIC S9(4) VALUE 6.
    01 FUNIT        COMP PIC S9(4) VALUE 0.
    01 TYPE         COMP PIC S9(4) VALUE 0.
    01 CODE         COMP PIC S9(4).
```

The following statement from the PROCEDURE DIVISION of a COBOL or CBL program calls subroutine SRCH$$ with the parameters defined by the DATA DIVISION statements above.

```
    CALL 'SRCH$$' USING K-EXST, NAME, NAMELENGTH, FUNIT, TYPE, CODE.
```

Note

> You can omit the PIC S9(4) clause from declarations of the data type COMP if you compile the program using the COBOL compiler. However, the CBL compiler returns an OBSERVATION error message if the PIC S9(4) clause is omitted from the COMP declarations.

CBL: PIC A(n) or PIC X(n)
COBOL: PIC A(n), PIC X(n), or PIC 9(n)
PL/I: CHARACTER(n)NONVARYING

A COBOL or CBL program can declare as PIC A or PIC X items of _n_ characters any data strings that are to be passed to subroutines or functions expecting data of the PL/I data type CHARACTER(n)NONVARYING, also called CHAR(n).

For example, subroutine uses three parameters, as declared in the following DCL statement:

```
DCL SPAS$$ ENTRY (CHAR(6), CHAR(6), FIXED BIN);
```

The following statements declare data types for three variables, OWNER, NONOWN, and CODE, that are to be used as arguments in the call to SPAS$$ from COBOL or CBL.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 OWNER     PIC X(6).
01 NONOWN    PIC X(6).
01 CODE COMP PIC S9(1).
```

The following statement calls SPAS$$ with the arguments defined above.

```
CALL 'SPAS$$' USING OWNER, NONOWN, CODE.
```

COBOL or CBL: COMP-3
PL/I: FIXED DECIMAL

The COBOL and CBL data type COMP-3 can be used as an equivalent of the PL/I data type FIXED DECIMAL. COMP-3 represents a packed decimal item. The PICTURE clause of a COMP-3 data type declaration can contain only 9, S, V, or P. FIXED DECIMAL represents a data item consisting of one or more decimal digits, and which may include a decimal point or sign.

For more information about these data types, see the COBOL 74 Reference Guide and the PL/I Reference Guide.

COBOL or CBL: Array
FTN: Integer Array

In COBOL or CBL, a table of the correct data type can be passed to a FORTRAN subroutine expecting an integer array. An integer array in FTN can contain either alphabetic or numeric information.

Multidimensional arrays cannot be passed to a FORTRAN subroutine.

For example, the subroutine TIMDAT returns an integer array containing alphabetic and numeric information. A COBOL or CBL program that calls TIMDAT must declare a separate array for each type of information. The following statements from a COBOL or CBL program define separate arrays for alphabetic and numeric characters.

```
DATA DIVISION.
WORKING-STORAGE SECTION
01 ARRAY.
  05 TABLE  PIC X(30).
  05 CHAR-ARRAY REDEFINES TABLE OCCURS 15, PIC X(2).
  05 NUM-ARRAY REDEFINES TABLE OCCURS 15, COMP PIC S9(4).
01  NUMBER  COMP PIC S9(4) VALUE 15.
```

Note

You can omit the PIC S9(4) clause from declarations of the data type COMP if you compile the program using the COBOL compiler. However, the CBL compiler returns an OBSERVATION error message if the PIC S9(4) clause is omitted from the COMP declarations.

The following statement from the PROCEDURE DIVISION of a COBOL or CBL program calls subroutine TIMDAT.

```
CALL 'TIMDAT' USING ARRAY, NUMBER.
```

In this CALL statement, ARRAY is the array of system and user information returned by the subroutine, and NUMBER specifies the number of elements in the array; this value must be less than or equal to 28.

The following DISPLAY statements from the PROCEDURE DIVISION display on the terminal the information returned by TIMDAT. Each DISPLAY statement refers to a field either of the alphabetic array or of the numeric array. The final DISPLAY statement displays the first 6 characters of the 32 character user name returned by TIMDAT.

```
    DISPLAY 'MONTH IS:  ', CHAR-ARRAY(1).
    DISPLAY 'DAY IS:  ', CHAR-ARRAY(2).
    DISPLAY 'YEAR IS:  ', CHAR-ARRAY(3).
    DISPLAY 'MINUTES SINCE MIDNIGHT:  ', NUM-ARRAY(4).
    DISPLAY 'TIME IN SECONDS:  ', NUM-ARRAY(5).
    DISPLAY 'TIME IN TICKS:  ', NUM-ARRAY(6).
    DISPLAY 'CPU TIME IN SECONDS:  ', NUM-ARRAY(7).
    DISPLAY 'CPU TIME IN TICKS:  ', NUM-ARRAY(8).
    DISPLAY 'DISK I/O TIME IN SECONDS:  ', NUM-ARRAY(9).
    DISPLAY 'DISK I/O TIME IN TICKS:  ', NUM-ARRAY(10).
    DISPLAY 'TICKS PER SECOND:  ', NUM-ARRAY(11).
    DISPLAY 'USER-NUMBER:  ', NUM-ARRAY(12).
    DISPLAY 'LOGIN NAME:  ', CHAR-ARRAY(13), CHAR-ARRAY(14),
                             CHAR-ARRAY(15).
```

COBOL and CBL: Two-element group item
PL/I: CHARACTER(*)VARYING

The PL/I data type CHARACTER(*)VARYING is a record structure consisting of a count-of-characters field and a field containing characters. In COBOL or CBL, a comparable record structure must be declared as a two-element group item. The first element should be the count-of-characters field and should be defined as a COMP data type. The second element should be the character field and should be defined as PIC X(n), where n is equal to the length of the character field. The following group item illustrates these requirements.

```
    01 CHAR-VAR.
      05 CHAR-COUNT  PIC S9(4) VALUE 5 COMP.
      05 CHAR-STRING PIC X(5) VALUE 'ABCDE'.
```

In the CHAR-COUNT field above, VALUE 5 specifies the number of characters to be passed.

In the CHAR-STRING field above, PIC X(5) specifies that there are five characters in the character string; the characters themselves, A, B, C, D, and E, are specified after VALUE'.

For example, the subroutine COM$AB, which expands a line of text using the PRIMOS abbreviation preprocessor, uses one CHAR(*) VAR and two FIXED BIN parameters, as indicated by the following DCL statement:

```
    DCL COM$AB ENTRY (CHAR(*) VAR, FIXED BIN, FIXED BIN);
```

The following lines from the DATA DIVISION of a COBOL or CBL program define data types for variables that can be used for each of these parameters.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COMMAND.
   05 CLENGTH PIC S9(4) VALUE 6 COMP.
   05 CFIELD PIC X(6).
01 COMSIZE PIC S9(4) VALUE 6 COMP.
01 CODE PIC S9(4) COMP.
```

The following statement moves the characters "sl" to <u>cfield</u>.

```
MOVE 'SL' TO CFIELD.
```

The following statement from the PROCEDURE DIVISION of a COBOL or CBL program illustrates a call to subroutine COM$AB.

```
CALL 'COM$AB' USING COMMAND, COMSIZE, CODE.
```

6 Calling Subroutines From FORTRAN

CALL FORMAT

A subroutine can be called from a program written in FORTRAN 66 (FTN) or FORTRAN 77 (F77) by a statement of the following form:

```
    CALL sub-name[(argument [, argument]...)]
```

In the CALL statement, sub-name is the name of the subroutine and argument can be either a literal or a data-name.

FTN and F77 can invoke functions in a variety of ways. For information about how FORTRAN calls subroutines and functions, see the FORTRAN Reference Guide or the FORTRAN 77 Reference Guide.

USING SYSCOM FILES

You can insert the SYSCOM file that defines error codes into a FORTRAN program by including the following statement in the program:

```
    $INSERT SYSCOM>ERRD.INS.FTN;
```

You can insert the SYSCOM file that defines key codes into a FORTRAN program by including the following statement in the program:

```
    $INSERT SYSCOM>KEYS.INS.FTN;
```

You can insert the SYSCOM file that defines argument codes into a FORTRAN program by the following statement in the program:

```
    $INSERT SYSCOM>A$KEYS.INS.FTN;
```

In F77, use the INCLUDE statement to insert SYSCOM files into a program. The file name specified as the argument of the INCLUDE statement must be enclosed by single quotation marks. For example, the following statement inserts into a program the SYSCOM file that defines argument codes.

```
    INCLUDE 'SYSCOM>A$KEYS.INS.FTN'
```

For more information about the INCLUDE statement, see the Rev. 21 update of the FORTRAN 77 Reference Guide.

DATA TYPES

Table 6-1 suggests ways that FORTRAN and PL/I data types can be represented in FTN and F77.

Table 6-1
Data Type Equivalents: FORTRAN

Generic Unit	FTN	F77	PL/I
16 bits (Halfword)	INTEGER INTEGER*2 LOGICAL	 INTEGER*2 LOGICAL*2	FIXED BIN FIXED BIN(15) BIT(1) ALIGNED
32 bits (Word)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	FIXED BIN(31)
32 bits (Float single precision)	REAL REAL*4	REAL REAL*4	FLOAT BINARY FLOAT BIN(23)
64 bits (Float double precision)	REAL*8 double precision	REAL*8 double precision	FLOAT BIN(47)
Byte string (Max. 32767)	Integer Array	CHAR*n	CHAR(n)
Record Structure	Use EQUIVALENCE Statement	Use EQUIVALENCE Statement	CHAR(*) VARYING

<u>Note</u>

Neither FTN nor F77 has data types that correspond to the PL/I data types BIT, BIT(1), BIT(n), CHAR(n) VARYING, POINTER OPTIONS (SHORT), or POINTERs.

The following sections suggest how PL/I data types can be declared in FTN or F77. For more information about each data type, see the chapter titled "Overview of Subroutines" in Volume II, III, or IV of this guide.

FTN or F77: INTEGER*2 or INTEGER*4
PL/I: FIXED BIN(15) or FIXED BIN(31)

The FTN and F77 data types INTEGER*2 and INTEGER*4 can be used as equivalents of the PL/I data types FIXED BIN(15) and FIXED BIN(31), respectively.

By default, FTN treats as INTEGER*2 any data type declared simply as INTEGER. To make FTN treat the INTEGER data type as INTEGER*4, use the -INTL (integer long) option every time you compile an FTN program. For example, to compile program LOGICAL.FTN so that INTEGER is interpreted as INTEGER*4, execute the following command.

```
    OK, FTN LOGICAL.FTN -INTL
```

By default, F77 treats as INTEGER*4 any data type declared simply as INTEGER. To make F77 treat the INTEGER data type as INTEGER*2, use the -INTS (short integer) option every time you compile an F77 program. For example, to compile program LOGICAL.F77 so that INTEGER is interpreted as INTEGER*2, execute the following command.

```
    OK, F77 LOGICAL.F77 -INTS
```

Although the data type INTEGER is valid in FTN and F77, it is good practice to declare all integer arguments as either INTEGER*2 or INTEGER*4.

For example, a program written in FTN or F77 that calls the subroutine TI$MSG must declare data types that correspond to the data types declared for the subroutine in the following DCL statement:

```
    DCL TI$MSG ENTRY (FIXED BIN(15), FIXED BIN(31), FIXED BIN(31),
                      FIXED BIN(31));
```

The following statements from an FTN or F77 program declare corresponding data types for the four parameters of TI$MSG:

```
    INTEGER*2 RESERV
    INTEGER*4 CONNECT, CPU, IO
```

The following statement from an FTN or F77 program calls TI$MSG with the four arguments whose data types are declared above.

```
    CALL TI$MSG (RESERV, CONNECT, CPU, IO);
```

<u>FTN: LOGICAL</u>
<u>F77: LOGICAL*2</u>
<u>PL/I: FIXED BIN or FIXED BIN(15)</u>

The FTN data type LOGICAL and the F77 data type LOGICAL*2 can be used as equivalents of the PL/I data type FIXED BIN and FIXED BIN(15).

For example, the subroutine BREAK$ expects a parameter of the data type FIXED BIN, as indicated by the following DCL statement:

```
    DCL BREAK$ ENTRY (FIXED BIN);
```

Any variable that is to be used in the call to BREAK$ must be declared LOGICAL*2, the F77 equivalent of FIXED BIN. For example, the following statement from a program written in F77 declares the variable <u>logic</u> as LOGICAL*2.

```
    LOGICAL*2 LOGIC
```

The program then calls subroutine BREAK$:

```
    CALL BREAK$ (LOGIC)
```

<u>FTN and F77: INTEGER*2</u>
<u>PL/I: BIT(1) ALIGNED</u>

The FTN and F77 data type INTEGER*2 can be used as an equivalent of the PL/I data type BIT(1) ALIGNED. If the argument is declared in the PL/I program as BIT(1) ALIGNED, it can be treated as a 16-bit integer, with a value of 0 for false and -32768 for TRUE.

For example, an FTN or F77 program that calls function IDCHK$ must declare data types for the parameters of IDCHK$ that correspond to the following:

```
    DCL IDCHK$ ENTRY (FIXED BIN, CHAR(*)VAR) RETURNS (BIT (1));
```

Note that BIT(1) values returned by functions are always ALIGNED.

The following statement from an FTN or F77 program declares variable return as INTEGER*2; return could thus be used to receive the returned value of IDCHK$.

```
    INTEGER*2 return
```

FTN and F77: REAL or REAL*4
PL/I: FLOAT BIN or FLOAT BIN(23)

The FTN and F77 data type REAL, or REAL*4, can be used as an equivalent of the PL/I data type FLOAT BIN, or FLOAT BIN(23).

FTN and F77: REAL*8
PL/I: FLOAT BIN(47)

The FTN and F77 data type REAL*8 can be used as an equivalent of the PL/I data type FLOAT BIN(47).

FTN and F77: Two-element record, defined by EQUIVALENCE statement
PL/I: CHARACTER(*)VARYING

The PL/I data type CHARACTER(*)VARYING is implemented as a record structure, consisting of a count of characters followed by the characters themselves. Figure 6-1 illustrates the record's structure.

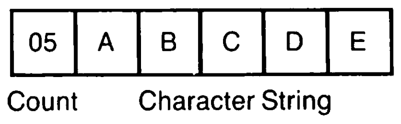

Figure 6-1
CHAR(*) VAR Record Structure

In FTN and F77, the corresponding structure is a two-element record. The record consists of an INTEGER*2 element containing a count of the characters in the record and a field containing a character string. This field can be CHARACTER*n in F77, or INTEGER*2 in FTN, and should contain the characters to be passed.

The EQUIVALENCE statement can be used to create such a record by assigning values to the two elements of the array. For example, the following FTN code sets up a two-element array that corresponds in structure to the CHARACTER(*)VARYING data type of PL/I.

```
INTEGER*2 STRING(10), LENGTH
INTEGER*2 VARSTRING(11)
EQUIVALENCE (LENGTH, VARSTRING(1))
EQUIVALENCE (STRING(1), VARSTRING(2))
STRING(1) = 'MY'
STRING(2) = 'FI'
STRING(3) = 'LE'
LENGTH = 6
```

Figure 6-2 illustrates this record's structure.

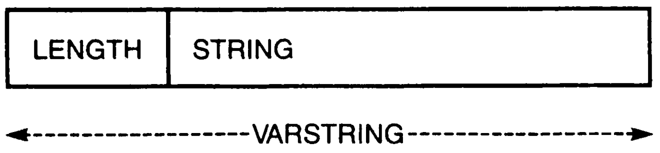

Figure 6-2
The Record VARSTRING

In the code given above, VARSTRING is declared as an array of eleven characters. The first EQUIVALENCE statement declares the first element of VARSTRING equal to the variable LENGTH. The second EQUIVALENCE statement declares elements 2 through 11 of VARSTRING equal to elements 1 through 10, respectively, of an array named STRING. The characters 'MYFILE' are assigned to array STRING, two characters to an element. The value 6 is assigned to LENGTH because six characters are assigned to STRING. The effect of this code is to make VARSTRING a two-element record that corresponds in structure to the CHAR*VARYING data type of PL/I.

In F77 all of STRING can be assigned at once, as follows.

```
INTEGER*2 LENGTH, VARSTRING(11)
CHARACTER*20 STRING
EQUIVALENCE(LENGTH, VARSTRING(1))
EQUIVALENCE(VARSTRING(2), STRING)
STRING(1:6) = 'MYFILE'
LENGTH = 6
```

For more information about the FORTRAN EQUIVALENCE statement, see the FORTRAN Reference Guide or the FORTRAN 77 Reference Guide.

For example, suppose a program written in FTN or F77 is to call subroutine GV$GET. The program must declare data types for the parameters of GV$GET that correspond to the data types declared for the subroutine in the following DCL statement:

```
    DCL GV$GET ENTRY (CHAR(*)VAR, CHAR(*)VAR, FIXED BIN, FIXED BIN);
```

The following FTN statements declare that two CHAR(*)VAR arrays, VARNAM and VARVAL, consist of eleven elements each, and that each element is of the data type INTEGER*2.

```
    INTEGER*2 VARNAM(11)
    INTEGER*2 VARVAL(11)
```

VARNAM and VARVAL are each to consist of an element containing a count of characters and a ten-element field containing the characters. The following statement defines these elements of VARNAM and VARVAL:

```
    INTEGER*2 LEN1, STR1(10), LEN2, STR2(10)
```

The preceding statement declares as INTEGER*2:

- The character count (LEN1) of VARNAM
- Each of the ten elements of (STR1(10)), the character field of VARNAM
- The character counter (LEN2) of VARVAL
- Each of the ten elements of (STR2(10)), the character field of VARVAL

The following statement declares as INTEGER*2 the second FIXED BIN parameter of GV$GET:

```
    INTEGER*2 CODE
```

The first EQUIVALENCE statement below declares that LEN1 corresponds to the first element of VARNAM. The second EQUIVALENCE statement pairs elements of VARNAM with elements of STR1: the second element of VARNAM corresponds to the first element of STR1, the third element of VARNAM corresponds to the second element of STR1, and so on.

```
    EQUIVALENCE(LEN1, VARNAM(1))
    EQUIVALENCE(VARNAM(2), STR1(1))
```

Figure 6-3 illustrates the effect of the EQUIVALENCE statements above.

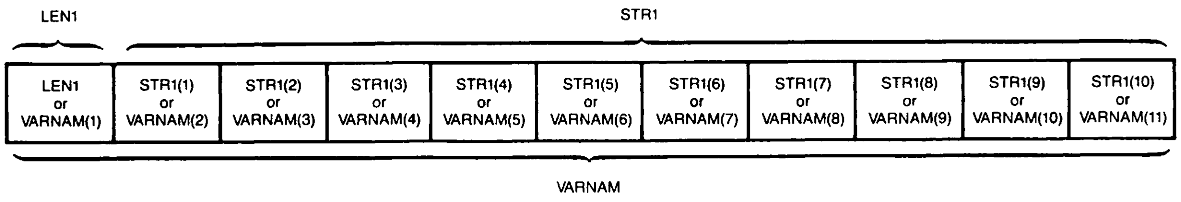

Figure 6-3
The Array VARNAM

The first EQUIVALENCE statement below declares that LEN2 corresponds to the first element of VARVAL. The second EQUIVALENCE statement pairs elements of VARVAL with elements of STR2: the second element of VARVAL corresponds to the first element of STR2, the third element of VARVAL corresponds to the second element of STR2, and so on.

```
    EQUIVALENCE(LEN2, VARVAL(1))
    EQUIVALENCE(VARVAL(2), STR2(1))
```

Figure 6-4 illustrates the effect of the EQUIVALENCE statements above. If an array is specified in an EQUIVALENCE statement without a subscript, the subscript is assumed by default to be (1).

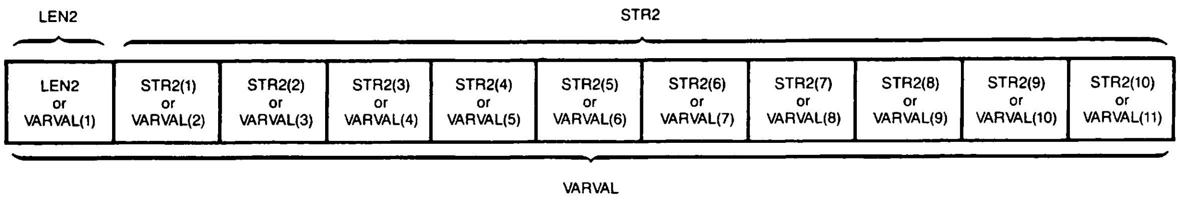

Figure 6-4
The Array VARVAL

The following statement calls subroutine GV$GET with the arguments defined above; note that the third argument, declared as a FIXED BIN parameter, can be expressed as the numeric literal 20.

```
    CALL GV$GET(VARNAM,VARVAL, 20, CODE)
```

In F77, the parameters of GV$GET can be declared as follows:

```
    INTEGER*2 CODE, LEN1, LEN2, VARLEN
    CHARACTER*20 STR1, STR2
    INTEGER*2 VARNAM(11)
    INTEGER*2 VARVAL(11)
    EQUIVALENCE(LEN1, VARNAM(1))
    EQUIVALENCE(LEN2, VARVAL(1))
    EQUIVALENCE(VARNAM(2), STR1(1))
    EQUIVALENCE(VARVAL(2), STR2(1))
```

The following statement calls GV$GET with the arguments defined in the code above.

```
    CALL GV$GET(VARNAM, VARVAL, VARLEN, CODE)
```

<u>F77: CHARACTER*n</u>
<u>FTN: Integer Array</u>
<u>PL/I: CHARACTER(n)NONVARYING</u>

The F77 data type CHARACTER*n can be used as an equivalent of the PL/I data type CHARACTER(n)NONVARYING, usually declared as CHARACTER(n).

FTN can pass short integer arrays to subroutines expecting parameters of the type CHARACTER(n). Use one array element for each two characters to be passed. Thus, the dimension of the integer array should be one-half the value of the (n) in CHARACTER(n), rounded up.

For example, an FTN program that calls subroutine TIMDAT must declare data types that correspond to the data types declared for the subroutine in the following DCL statement:

```
    DCL TIMDAT (1...., FIXED BIN);
```

The following statements declare data types for the parameters of TIMDAT that correspond to the data types declared in the preceding DCL statement:

```
    INTEGER*2 STRING(28)
    INTEGER*2 NUM, DATE(3)
    INTEGER*2 TIME, TIME1, TIME2, NAME(3)
```

The preceding statements declare the following as INTEGER*2.

- Each of the 28 elements of array STRING,
- The variable NUM,
- Each of the three elements of array DATE,
- The variables TIME, TIME1, and TIME2, and
- Each of the three elements of the array NAME.

The following EQUIVALENCE statements subdivide array STRING into five fields, each of which holds separate items of system information as returned by TIMDAT:

```
    EQUIVALENCE (STRING(1), DATE)
    EQUIVALENCE (STRING(4), TIME)
    EQUIVALENCE (STRING(5), TIME1)
    EQUIVALENCE (STRING(6), TIME2)
    EQUIVALENCE (STRING(13), NAME)
```

The five equivalence statements equate the values of DATE, TIME, TIME1, TIME2, and NAME with specified elements in array STRING. The three elements of array DATE are equated with the first three elements of array STRING, TIME is equated with the fourth element of array STRING, and so on. Thus the ASCII and numeric characters of system and user information returned by subroutine TIMDAT are all assigned to different elements of array STRING.

The following statement calls subroutine TIMDAT.

```
    CALL TIMDAT(STRING, NUM)
```

Note

Variables declared as CHARACTER*n are not necessarily aligned on word boundaries. Thus, passing CHARACTER*n parameters to FTN subroutines may cause serious errors.

PL/I: POINTER

Neither FTN nor F77 supports a pointer data type. PL/I subroutines that expect this data type should not be called from FORTRAN. Only experienced programmers should attempt to pass the expression LOC(*name*) to a non-PL/I subroutine that expects a pointer.

There is no convenient FORTRAN data type for storing a 48-bit pointer. Currently, most Prime system subroutines use 32 bits of the pointer available, ignoring the extra 16 bits if they are present. FORTRAN can create only two-word pointers using LOC(*name*).

FORTRAN cannot directly handle a pointer returned to it. If you want to use a pointer that has been returned to a program written in FTN or F77, receive the pointer in a variable declared INT*4, and use the resulting value as an argument of MOVEW$ to gain access to the data pointed at.

7
Calling Subroutines From Pascal

CALL FORMAT

Before a Prime subroutine or function can be called by a Pascal program, it must be declared as a procedure or a function. To declare a subroutine or a function as a procedure, use a statement of the following format:

```
    PROCEDURE sub-name[([VAR]  arg:type[;  [VAR] arg:type]...)];EXTERN;
```

The keyword EXTERN must be added to the end of the PROCEDURE or FUNCTION declaration for any procedure or function that is compiled separately from the Pascal calling program.

To call a subroutine as a procedure from a program written in Pascal, use a statement of the following format:

```
    sub-name[(argument [,argument]...)];
```

In the Pascal procedure statement, the element sub-name must be the name of a subroutine, and the arguments can be data names or constants.

To declare a subroutine as a function in a program written in Pascal, use a FUNCTION statement of the following format:

```
  FUNCTION function-name[([VAR] arg: type [;[VAR] arg:type]...)]: type;
    EXTERN;
```

To call a function, use statements similar to the following:

```
    X := function(data...);

    IF function(data...) = X THEN ...;
```

Note

Any arguments that are supplied or changed by the subroutine must be declared as variable parameters, preceded by the reserved word VAR. These arguments are described as OUTPUT parameters or INPUT/OUTPUT parameters in the subroutine descriptions in the other volumes of this guide.

USING SYSCOM FILES

You can insert the SYSCOM file that defines error codes into a Pascal program by including the following statement in the CONST section of the program:

```
    %INCLUDE 'SYSCOM>ERRD.INS.PASCAL';
```

You can insert the SYSCOM file that defines key codes into a Pascal program by including the following statement in the CONST section of the program:

```
    %INCLUDE 'SYSCOM>KEYS.INS.PASCAL';
```

You can insert the SYSCOM file that defines argument codes into a Pascal program by including the following statement in the CONST section of the program:

```
    %INCLUDE 'SYSCOM>A$KEYS.INS.PASCAL';
```

DATA TYPES

Table 7-1 suggests how PL/I and FORTRAN data types can be represented in Pascal.

Table 7-1
Data Type Equivalents: Pascal

Generic Unit	Pascal	PL/I	FTN	F77
16 bits (Halfword)	INTEGER Enumerated	FIXED BIN FIXED BIN(15)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2
32 bits (Word)	LONGINTEGER	FIXED BIN(31)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4
32 bits (Float single precision)	REAL	FLOAT BINARY FLOAT BIN(23)	REAL REAL*4	REAL REAL*4
64 bits (Float double precision)	LONGREAL	FLOAT BIN(47)	REAL*8	REAL*8
1 left-aligned bit (Halfword)	BOOLEAN	BIT(1) ALIGNED		
Bit string	SET	BIT(n)		
Byte string (Max. 32767)	CHAR PACKED ARRAY[1..n] OF CHAR	CHAR(n)	Integer Array	CHARACTER *n

Table 7-1 (continued)
Data Type Equivalents: Pascal

Generic Unit	Pascal	PL/I	FTN	F77
Varying character string	STRING[n] STRING	CHAR(n) VARYING CHAR(*) VARYING		
48 bits (Three halfwords)	pointer	POINTER		
Record Structure	RECORD	Structure		

The following sections suggest how FORTRAN and PL/I data types can be declared in Pascal. For more information about each data type, see the chapter titled OVERVIEW OF SUBROUTINES in Volume II, III, or IV of this guide.

Boolean Values

Some functions return Boolean (true/false) values through FIXED BIN parameters. In Pascal, these parameters should be declared as INTEGER.

Subroutines that are written in FORTRAN and return Boolean values return the Boolean values through the LOGICAL data type. In Pascal, declare these parameters as INTEGER.

<u>Pascal: INTEGER</u>
<u>PL/I: FIXED BIN(15)</u>
<u>FTN: INTEGER, INTEGER*2, or LOGICAL</u>
<u>F77: INTEGER*2 or LOGICAL*2</u>

The Pascal data type INTEGER can be used as an equivalent of the PL/I data type FIXED BIN(15) (also called FIXED BIN); of the FTN data types INTEGER, INTEGER*2, and LOGICAL; and of the F77 data types INTEGER*2 and LOGICAL*2.

For example, a Pascal program that calls the subroutine SRCH$$, which is written in PL/I, must declare for the parameters of SRCH$$ the data types that correspond to the data types expected by the subroutine, as declared in the following DCL statement:

```
    DCL SRCH$$ ENTRY (FIXED BIN, CHAR(32) VAR, FIXED BIN,
                      FIXED BIN, FIXED BIN, FIXED BIN);
```

The following TYPE and PROCEDURE statements declare Pascal data types that correspond to the PL/I data types declared for the parameters of SRCH$$ in the DCL statement above.

```
    TYPE
      NAMETYPE = PACKED ARRAY[1..8] OF CHAR;
    PROCEDURE SRCH$$(KEY        : INTEGER;
                     FILENAME   : NAMETYPE;
                     NAMELENGTH : INTEGER;
                     FILEUNIT   : INTEGER;
                     FILETYPE   : INTEGER;
                     VAR CODE   : INTEGER); EXTERN;
```

The PROCEDURE statement declares five parameters of SRCH$$ as INTEGER, the Pascal data type that corresponds to the PL/I data type FIXED BIN. The PROCEDURE statement also declares the second parameter as <u>NAMETYPE</u>; the preceding TYPE statement declares NAMETYPE as equivalent to PACKED ARRAY[1..8] OF CHAR, which corresponds to CHAR(32) VAR. The programmer selects for <u>NAMETYPE</u> the length that is likely to be needed (here, 8).

<u>Pascal: LONGINTEGER</u>
<u>PL/I: FIXED BIN(31)</u>
<u>FTN: INTEGER*4</u>
<u>F77: INTEGER, INTEGER*4, LOGICAL, or LOGICAL*4</u>

The Prime Pascal data type LONGINTEGER can be used as an equivalent of the PL/I data type FIXED BIN(31), of the FTN data type INTEGER*4, and of the F77 data types INTEGER, INTEGER*4, LOGICAL, and LOGICAL*4. The LONGINTEGER data type is a Prime extension to ANSI and ISO standard Pascal.

For example, a Pascal program that calls the subroutine RNUM$A, which is written in FORTRAN, must declare the INTEGER*4 parameter of RNUM$A as LONGINTEGER. The parameter declarations in subroutine RNUM$A are as follows.

```
    INTEGER*2 msg(1), msglen, numkey
    INTEGER*4 value
```

The following TYPE and PROCEDURE statements declare Pascal data types that correspond to the FORTRAN data types declared for the parameters of RNUM$A in the subroutine description above.

```
    TYPE
      MSGTYPE = PACKED ARRAY[1..14] OF CHAR;
    PROCEDURE RNUM$A(MSG  : MSGTYPE;
                     MSGLEN  : INTEGER;
                     NUMKEY  : INTEGER;
                     VAR VALUE : LONGINTEGER);EXTERN;
```

MESSAGE, the first parameter of RNUM$A, is declared as MSGTYPE, which corresponds to PACKED ARRAY[1..14] OF CHAR; the array can be of any length required to accommodate the message. The second and third parameters are declared as INTEGER, which correspond to the FORTRAN data type INTEGER*2. The fourth parameter is declared as LONGINTEGER, which corresponds to the FORTRAN data type INTEGER*4.

Pascal: REAL
PL/I: FLOAT BIN, FLOAT BIN(23)
FTN and F77: REAL or REAL*4

The Pascal data type REAL can be used as an equivalent of the PL/I data types FLOAT BIN (also called FLOAT BIN(23)), and of the FTN and F77 data types REAL and REAL*4. When a Pascal program calls a subroutine that uses FLOAT BIN, REAL, or REAL*4 parameters, it must declare these parameters as REAL. Constants passed as real arguments to FORTRAN functions should be in scientific format (x.xEyy).

For example, a Pascal program that calls the function RAND$A must declare for each parameter of RAND$A the data type that corresponds to the data type declared for the parameter in the function. Function RAND$A has two parameters, a seed value which is used to generate random numbers, and a return value (rt_val) which is assigned the value of each random number generated. The seed value and return value must be declared as INTEGER*4 and REAL*4, respectively. The return value can also be declared REAL*8.

In Pascal, the following VAR and FUNCTION statements declare Pascal data types that correspond to the data types expected by the function. The parameter _seed_ must be declared REAL in Pascal.

```
VAR
  SEED1, THISONE : REAL;
  INDEX : INTEGER;
FUNCTION RAND$A(VAR SEED : REAL) : REAL; EXTERN;
BEGIN
  SEED1 := 1.2;
  FOR INDEX := 1 TO 10 DO
    BEGIN
      THISONE := RAND$A(SEED1);
      WRITELN(INDEX, ':', THISONE);
    END
```

The FUNCTION statement declares the function's return value and the seed value _seed_ as real numbers. This is necessary because the return value of function RAND$A is a real number.

Pascal: LONGREAL
PL/I: FLOAT BIN(47)
FTN and F77: REAL*8

The Prime Pascal data type LONGREAL can be used as an equivalent of the PL/I data type FLOAT BIN(47) and of the FORTRAN data type REAL*8. The LONGREAL data type is a Prime extension to ANSI and ISO standard Pascal.

For example, the return value of function RAND$A can be received by a REAL*8 variable, declared LONGREAL in Pascal. The function can be called as in the example in the preceding section, with the variable that is to receive the return value, _thisone_, declared LONGREAL, as follows:

```
THISONE :  LONGREAL;
```

The return value must also be declared LONGREAL in the function declaration, as follows:

```
FUNCTION RAND$A(VAR SEED :  REAL) :  LONGREAL;  EXTERN;
```

Pascal: BOOLEAN or SET OF 0..x
PL/I: BIT(1) ALIGNED or BIT(n)

The Pascal data type BOOLEAN can be used as an equivalent of the PL/I data type BIT(1) ALIGNED. In Pascal, the PL/I types '0'B and '1'B can be read as FALSE and TRUE, respectively.

If the _n_ of a BIT(_n_) data type is greater than 1, this data type corresponds to the Pascal data type SET OF 0..x. The base type of SET OF 0..x must be an INTEGER subrange starting at 0, with _x_ equal to _n - 1_. For example, a PL/I data type BIT(11) can be declared in Pascal as SET OF 0..10; a BIT(48) data type can be declared as SET OF 0..47; and so on.

Whatever the _n_ of a BIT(_n_) data type, if the BIT data items are elements of a structure, all the adjacent bits can be summed into a single SET.

If the _n_ of a BIT(_n_) data type is 16 or if the number of bits in a structure totals 16, you can declare the parameter as an INTEGER.

For example, a Pascal program that calls subroutine UID$BT must declare for the parameter of UID$BT the data type that corresponds to the data type declared for the parameter in the following DCL statement:

```
DCL UID$BT ENTRY (BIT(48) ALIGNED);
```

As the value of (_n_) in this BIT(_n_) declaration is greater than 1, the Pascal program can declare the parameter as SET OF 0..47. The following TYPE statement defines a non-standard data type, BITSET, as equivalent to SET OF 0..47; the following PROCEDURE statement declares UID$BT as a Pascal procedure whose parameter is data type BITSET:

```
TYPE
  BITSET = SET OF 0..47;
PROCEDURE UID$BT (VAR BITS : BITSET); EXTERN;
```

Note

The data type BOOLEAN can also be declared for parameters of the type BIT(16) ALIGNED or for parameters whose description states that only the most significant bit is used.

Pascal: PACKED ARRAY[1..n] OF CHAR
PL/I: CHARACTER(n)
FTN: Integer Array
F77: CHARACTER*n

The Pascal data type PACKED ARRAY[1..n] OF CHAR can be used as an equivalent of the PL/I data type CHARACTER(n) NONVARYING, also called CHARACTER(n) or CHAR(n). The CHAR(n) parameter must be passed an array that contains exactly n characters.

The Pascal data type PACKED ARRAY[1..n] OF CHAR also corresponds to an FTN integer array, and to the F77 data type CHARACTER*n.

A function that returns any of these data types cannot be called from Pascal, because Pascal functions cannot return arrays. Multidimensional arrays should not be passed between FORTRAN and Pascal, because columns and rows will be reversed.

For example, a Pascal program that calls the subroutine DELE$A, which is written in FORTRAN, must declare the subroutine's INTEGER*2 parameter as PACKED ARRAY[1..n] OF CHAR. The parameter declarations in the subroutine include the following statements:

```
    INTEGER*2  name(1), namlen
    LOGICAL  log
```

When declared as INTEGER*2, the variable name(1) represents an array or character string whose length is unknown at the time of declaration. The following TYPE and PROCEDURE statements declare Pascal data types that correspond to the PL/I data types in the subroutine description of DELE$A:

```
    TYPE
      NAMETYPE = PACKED ARRAY[1..8] OF CHAR;
    VAR
```

The following FUNCTION statement declares DELE$A as a Pascal function with parameters of the data types NAMETYPE and INTEGER; the FUNCTION statement also declares that the value of the function, when executed, is an INTEGER:

```
    FUNCTION DELE$A(FILNAM : NAMETYPE;
                    LEN    : INTEGER) : INTEGER; EXTERN;
```

Pascal: STRING[n]
PL/I: CHARACTER(n) VARYING

The Pascal data type STRING[n] can be used as an equivalent of the PL/I data type CHARACTER(n) VARYING. The STRING[n] data type is a Prime extension of ANSI and ISO standard Pascal.

Note

Some system subroutines (such as SRCH$$) that expect a parameter of the type CHAR(n) VAR may require you to declare the parameter as PACKED ARRAY[1..n] OF CHAR. If an error occurs when you declare the parameter STRING[n], declare the parameter as PACKED ARRAY.

In the Pascal data type STRING[n], the value of _n_, which specifies the data type's maximum length, can be any integer up to 32767. The Pascal data type must have the same maximum length as the PL/I data type to which it corresponds.

For example, a Pascal program that calls subroutine AC$SET must declare for each parameter of AC$SET the data type that corresponds to the data type declared for the parameter in the following DCL statement. This DCL statement declares the second parameter of AC$SET as CHAR(128) VAR.

```
    DCL AC$SET ENTRY (FIXED BIN, CHAR(128) VAR, PTR, FIXED BIN);
```

The following TYPE statement defines the data type STRING7 as equivalent to the standard Pascal data type STRING[7] (Note that STRING7 could be declared any length up to 128.). The PROCEDURE statement declares NAME, the second parameter of AC$SET, as equivalent to STRING7.

```
    TYPE
      STRING7 = STRING[7];
      ...
    PROCEDURE AC$SET(KEY : INTEGER;
                     NAME : STRING7;
                     PTR  : ACL_PTR;
                     VAR CODE : INTEGER); EXTERN;
```

Pascal: STRING or STRING[n]
PL/I: CHARACTER(*) VARYING

The Prime Pascal data types STRING or STRING[n] can be used as an equivalent of the PL/I data type CHARACTER(*) VARYING. The data type STRING is a Prime extension to ANSI and ISO standard Pascal. An argument declared as a STRING, with no declared maximum length, can be up to 80 characters long.

A STRING is implemented as a structure that contains a count of the characters in the structure followed by the characters themselves, as shown in the diagram below.

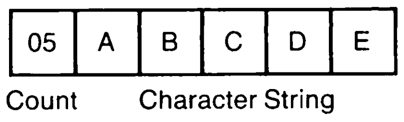

Figure 7-1
CHAR(*) VAR Record Structure

CHARACTER(*) VARYING is identical to the CHARACTER(n) VARYING data type, except that it has no specified maximum length, whereas CHARACTER(n) VARYING has a maximum length specified by n.

A Pascal program that calls GV$GET must declare for each parameter of GV$GET the data type that corresponds to the data type declared for the parameter in the following DCL statement:

```
    DCL GV$GET ENTRY (CHAR(*)VAR, CHAR(*)VAR, FIXED BIN, FIXED BIN);
```

The following TYPE and PROCEDURE statements declare Pascal data types that correspond to the PL/I data types declared for the parameters of GV$GET in the DCL statement above:

```
    TYPE
      CHARVAR = STRING[4];
    PROCEDURE GV$GET(NAME       : CHARVAR;
                     VAR VALUE  : CHARVAR;
                     LENGTH     : INTEGER;
                     VAR CODE   : INTEGER); EXTERN;
```

The TYPE statement defines data type CHARVAR as equivalent to the standard Pascal data type STRING[4]. The PROCEDURE statement declares the first two parameters of GV$GET as CHARVAR, and the second two parameters as INTEGER, the Pascal data type that corresponds to the PL/I data type FIXED BIN.

Pascal: pointer
PL/I: POINTER

The Pascal data type pointer can be used as an equivalent of the PL/I data type POINTER, also known as PTR. A pointer is stored in three halfwords (48 bits). The item to which the pointer points is declared in PL/I with the BASED attribute (for instance, BASED FIXED BIN).

Although all Pascal pointers are three-halfword pointers, a subroutine that uses the two-halfword PTR OPTIONS(SHORT) type can be called from Pascal, provided that the pointer points to halfword-aligned data. However, a Pascal pointer cannot be used within a structure if the pointer is declared in a PL/I routine as OPTIONS(SHORT).

You can call functions that return PTR OPTIONS(SHORT) from Pascal only by some method such as the following: declare a variable as a record with two variants, LONGINTEGER and pointer. Declare the function with LONGINTEGER as the data type of the returned value. Your Pascal program can use the returned value through the pointer variant. For example, the following code shows how function STR$AL can be called from Pascal in this manner:

```
TYPE
   PTR = ^INTEGER;
   FAKE = RECORD
            CASE BOOLEAN OF
             FALSE: (LI : LONGINTEGER);
             TRUE:  (PT : PTR)
          END;
VAR
   F : FAKE;
   CODE : INTEGER;

FUNCTION STR$AL(RESERVED1 : INTEGER;
                BLOCK_SIZE : LONGINTEGER;
                RESERVED2 : INTEGER;
                VAR CODE : INTEGER) : LONGINTEGER; EXTERN;

BEGIN
  F.LI := STR$AL(0, 2, 0, CODE);
  WRITELN('LONGINT VALUE: ', F.LI);
  F.PT^ := 0;
  WRITELN('F.PT^ : ', F.PT^);
  IF (CODE = E$ALSZ) OR (CODE = E$ROOM) OR (CODE = E$HPER) THEN
    WRITELN('STANDARD ERROR CODE RETURNED');
  WRITELN('CODE: ', CODE);
END.
```

When a Pascal program calls a subroutine that uses a POINTER argument, it must declare this parameter as pointer. For example, the subroutine AC$SET uses a POINTER argument, as indicated by the following DCL statement:

```
DCL AC$SET ENTRY (FIXED BIN, CHAR(128) VAR, PTR, FIXED BIN);
```

The following TYPE statement defines special data types that can be used to declare data types for the parameters of AC$SET. Among these special data types is the type ACL_PTR, which corresponds to the pointer data type ^ACLTYPE:

```
TYPE
  STRING7 = STRING[7];
  RANGETYPE = 1..2;
  ACL_PTR = ^ACLTYPE;
  ACLTYPE = RECORD
              VERSION : INTEGER;
              ENTRY_COUNT : RANGETYPE;
              ENTRIES : ARRAY[RANGETYPE] OF STRING;
            END;
```

The following VAR statement declares data types for the parameters of AC$SET. Among these parameters is THISPTR, which is declared as ACL_PTR, a pointer data type:

```
VAR
  KEY     : INTEGER;
  ACLNAME : STRING7;
  THISPTR : ACL_PTR;
  ERRCODE : INTEGER;
  ACL     : ACLTYPE;
```

The following PROCEDURE statement declares AC$SET as a Pascal procedure, with parameters that correspond in data type to the data types declared in the DCL statement above:

```
PROCEDURE AC$SET(KEY : INTEGER;
                 NAME : STRING7;
                 PTR  : ACL_PTR;
                 VAR CODE : INTEGER); EXTERN;
```

The following statement calls AC$SET with arguments that correspond in data type to the data types declared in the DCL statement above. The third argument, THISPTR, is of the data type pointer:

```
    AC$SET (KEY, ACLNAME, THISPTR, ERRCODE);
```

Pascal: RECORD
PL/I: Structure

The Prime Pascal data type RECORD can be used as an equivalent of a PL/I structure. The data types of the Pascal RECORD fields must correspond to the data types of the members of the PL/I structure.

When a Pascal program calls a subroutine that uses a structure, it must declare this structure as a RECORD parameter. For example, a Pascal program can call the subroutine TIMDAT, which is written in PL/I, to read system and user information into a Pascal record.

TIMDAT expects two parameters, a PL/I structure and a FIXED BIN data item. The structure can be declared in Pascal as a record consisting of 11 different fields. The FIXED BIN parameter must be declared INTEGER in Pascal; 28 is the usual value assigned this parameter.

The following statements define the data type tabletype, corresponding to the PL/I structure expected by TIMDAT:

```
    TYPE
      TABLETYPE = RECORD
                    MMDDYY    : PACKED ARRAY[1..6] OF CHAR;
                    TIME_MIN  : INTEGER;
                    TIME_SEC  : INTEGER;
                    TIME_TCK  : INTEGER;
                    CPU_SEC   : INTEGER;
                    CPU_TCK   : INTEGER;
                    DISK_SEC  : INTEGER;
                    DISK_TCK  : INTEGER;
                    TCK_SEC   : INTEGER;
                    USER_NUM  : INTEGER;
                    USERNAME  : PACKED ARRAY[1..32] OF CHAR
                  END;
```

The following statement declares the variable table as the type tabletype:

```
    VAR
      TABLE    : TABLETYPE;
```

The following PROCEDURE statement declares subroutine TIMDAT as a Pascal procedure with two parameters, one as data type <u>tabletype</u> and one as INTEGER:

```
PROCEDURE TIMDAT(VAR ARR : TABLETYPE;
                    SIZE : INTEGER); EXTERN;
```

In the following Pascal code, the call:

```
TIMDAT(TABLE, 28);
```

reads the contents of the system and user information into a record of 11 fields. The following statements display this information.

```
BEGIN
  TIMDAT(TABLE, 28);
  WITH TABLE DO
    BEGIN
      WRITELN('DATE IS                ', MMDDYY);
      WRITELN('MINUTES USED       ',TIME_MIN);
      WRITELN('SECONDS ELAPSED    ',TIME_SEC);
      WRITELN('TICKS ELAPSED      ',TIME_TCK);
      WRITELN('CPU SECONDS USED   ',CPU_SEC);
      WRITELN('CPU TICKS          ',CPU_TCK);
      WRITELN('DISK SECONDS USED  ',DISK_SEC);
      WRITELN('DISK TICKS USED    ',DISK_TCK);
      WRITELN('TICKS PER SECOND   ',TCK_SEC);
      WRITELN('USER NUMBER        ',USER_NUM);
      WRITELN('USER NAME               ',USERNAME);
    END
END.
```

8 Calling Subroutines From PL/I

CALL FORMAT

Programs written in PL/I must declare as external procedures any subroutines that they call. In PL/I, subroutine declarations are of the following form:

```
    DECLARE sub-name EXTERNAL ENTRY[(type [,type]...)];
```

In the DECLARE statement, sub-name is the name of the subroutine to be called, and type is the data type of an argument to be passed to the subroutine.

Subroutines are called by statements of the following form:

```
    CALL sub-name[(argument, [,argument]...)];
```

In the CALL statement, argument may be either a constant or a data name.

PL/I programs must also declare any functions that they are to call. Function declarations are of the following form:

```
    DECLARE function-name EXTERNAL ENTRY[(type ...)] RETURNS (type);
```

PL/I can call a function using a statement that evaluates the function and assigns its value to a variable; such statements are of the following form:

```
    X = function-name[(identifier ...)];
```

PL/I can also call a function using a control statement, such as an IF/THEN statement, which performs a specified action if the function is of a specified value. Such statements are of the following form:

```
    IF function[(identifier...)] = 0 THEN ...action;
```

Note

In this chapter, the term PL/I stands for both full PL/I and PL/I Subset G (PL/I-G).

THE OPTIONS (SHORTCALL) DECLARATION

The OPTIONS(SHORTCALL) declaration calls PMA procedures with the PMA instruction JSXB instead of the more common PCL instruction. A procedure call of this type is faster than one using PCL. However, the called procedure must be written to expect this kind of call. As of Rev. 20.2, the only system subroutines that can and must be declared in this way are MKONU$ and ALOC$S.

The OPTIONS(SHORTCALL) declaration is of the following form:

```
    DECLARE procedure-name EXTERNAL ENTRY [(arg1 [,arg2]...)]
                           OPTIONS(SHORTCALL [(stack-size)] );
```

In the DECLARE statement, stack-size specifies the extra space needed for the calling procedure's stack. The default size is 8, but the descriptions of MKONU$ and ALOC$S explain which stack size to specify. This call does not create a new stack for storage, as does PCL. The calling procedure's stack space is used. Thus it may be necessary to specify stack size in the declaration in order to enlarge the calling stack. For example, MKONU$ requires a 28-word stack, so the user's stack must be large enough to accommodate this requirement. If the stack is not large enough, the return from the subroutine will cause unpredictable error messages.

Arguments can be used with the SHORTCALL option. The computer will set up the L register to point to an array containing the addresses of the arguments, or, if there is only one argument, to the address of the argument itself. No type checking is done. Both MKONU$ and ALOC$S take more than one argument.

USING SYSCOM FILES

The SYSCOM file that defines error codes can be inserted in a PL/I program by including the following statement in the program before the subroutine declaration:

```
    %INCLUDE 'SYSCOM>ERRD.INS.PL1';
```

You can insert the SYSCOM file that defines key codes in a PL/I program by including the following statement before the subroutine declaration:

```
    %INCLUDE 'SYSCOM>KEYS.INS.PL1';
```

You can insert the SYSCOM file that defines argument codes in a PL/I program by including the following statement in the program before the subroutine declaration:

```
    %INCLUDE 'SYSCOM>A$KEYS.INS.PL1'.
```

DATA TYPES

Many PRIMOS subroutines are written in a version of PL/I. Moreover, most of the usage descriptions in the Subroutines Reference Guide use PL/I terminology. To declare and call these subroutines from PL/I, use the same terminology.

Some subroutines, however, are written in FORTRAN, and some of these have usage sections that use FORTRAN terminology. Most subroutines that use FORTRAN terminology are described in Volume IV.

Table 8-1 summarizes the argument types of FORTRAN subroutines and functions that can be called from PL/I.

Table 8-1
Data Type Equivalents: PL/I

Generic Unit	PL/I	FTN	F77
16 bits (Halfword)	FIXED BIN FIXED BIN(15)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2
32 bits (Word)	FIXED BIN(31)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4
32 bits (Float single precision)	FLOAT BINARY FLOAT BIN(23)	REAL REAL*4	REAL REAL*4
64 bits (Float double precision)	FLOAT BIN(47)	REAL*8	REAL*8
Byte string (Max. 32767)	CHAR(n)	Integer Array	CHARACTER *n

The following sections explain how these argument types relate to PL/I. For more information, see the chapter titled "Overview of Subroutines" in Volume II, III, or IV of this guide.

PL/I: FIXED BIN or FIXED BIN(15)
FTN: INTEGER, INTEGER*2
F77: INTEGER*2

The PL/I data type FIXED BIN(15), also called FIXED BIN, can be used as an equivalent of the FTN and F77 data type INTEGER*2.

For example, a PL/I program that calls subroutine RNUM$A must declare data types that correspond to the types in the subroutine's description, as follows:

```
    INTEGER*2  msg(1), msglen, numkey
    INTEGER*4  value
```

The following statement in the PL/I program declares corresponding PL/I data types:

```
DCL RNUM$A EXTERNAL ENTRY (CHAR(14), FIXED BIN, FIXED BIN,
                           FIXED BIN(31));
```

The following PL/I statement inserts the file SYSCOM>A$KEYS.PL1 into the PL/I program; this file makes it possible to use argument keys as arguments in the call to RNUM$A:

```
%INCLUDE 'SYSCOM>A$KEYS.PL1';
```

The following PL/I statement declares the variable numvalue as FIXED BIN(31); this variable is to be used as the INTEGER*4 parameter of RNUM$A:

```
DCL NUMVALUE FIXED BIN(31);
```

The following PL/I statement calls subroutine RNUM$A with the arguments defined above:

```
CALL RNUM$A ('ENTER A NUMBER', 14, A$DEC, NUMVALUE);
```

PL/I: FIXED BIN(31)
FTN: INTEGER*4
F77: INTEGER, INTEGER*4, LOGICAL, LOGICAL*4

The PL/I data type FIXED BIN(31) can be used as an equivalent of the FTN data type INTEGER*4 and of the F77 data types INTEGER, INTEGER*4, LOGICAL, and LOGICAL*4.

For example, a PL/I program that calls subroutine RAND$A must declare data types that correspond to the types in the function's description, as follows:

```
      INTEGER*4  seed
      REAL*8     rt_val
C   rt_val can also be declared REAL*4
```

The following PL/I statement declares the variable seed as FIXED BIN(31), the PL/I data type that corresponds to INTEGER*4. The statement also initializes seed with the value 1. This variable is to be used as the argument of RAND$A.

```
    DCL SEED STATIC FIXED BIN(31) INITIAL (1);
```

The following PL/I statement declares the variable return as FLOAT, the PL/I data type that corresponds to REAL*4. This variable is to receive the return value of function RAND$A.

```
    DCL RETURN FLOAT;
```

The following PL/I statement declares function RAND$A with parameters of data types that correspond to the data types declared in the function description above; note how the seed parameter is declared FIXED BIN(31), the PL/I data type that corresponds to INTEGER*4.

```
    DCL RAND$A EXTERNAL ENTRY (FIXED BIN(31)) RETURNS (FLOAT);
```

The following statements call RAND$A with the arguments declared above:

```
    DCL INDEX FIXED BIN;
    DO INDEX = 1 TO 10;
      RETURN = RAND$A(SEED);
      PUT SKIP LIST(REAL4);
     END;
```

PL/I: FLOAT BIN or FLOAT BIN(23)
FTN and F77: REAL or REAL*4

The PL/I data type FLOAT BIN, also known as FLOAT or FLOAT BIN(23), can be used as an equivalent of the FTN and F77 data types REAL or REAL*4. Constants passed to a FORTRAN function that expects REAL arguments should be in scientific format (x.xE+yy).

For an illustration of how to call a function that returns a REAL*4 value, see the preceding section.

PL/I: FLOAT BIN(47)
FTN and F77: REAL*8

The PL/I data type FLOAT BIN(47) can be used as an equivalent of the

FTN and F77 data type REAL*8. Data of this type should be in scientific format (x.xE+yy).

For example, the section about the FIXED BIN(31) data type above contains an example of a call to function RAND$A. The variable that receives the return value of this function can be declared REAL*8, or in PL/I, FLOAT BIN(47). Thus, in the example, the variable return could also be declared as follows:

```
DCL RETURN FLOAT BIN(47);
```

PL/I: Integer or Character Array
FTN and F77: Integer Arrays

An integer array expected by a FORTRAN subroutine should be declared in PL/I either as an array of FIXED BINARY(15) elements, or as a character array, depending on the kind of information to be passed.

If the subroutine parameter is a character array, you can declare it either as CHAR(n) NONVARYING or as CHAR(*) NONVARYING. Multidimensional arrays cannot be passed between FORTRAN (FTN or F77) and PL/I, because columns and rows would be reversed.

For example, a PL/I program that calls function DELE$A must declare data types that correspond to the data types in the function's description, as follows:

```
INTEGER*2  name(1), namlen
LOGICAL    log
```

The following PL/I statement declares DELE$A as an external function with arguments that correspond in type to the types declared in the function's description above. The CHAR(8) argument is a string of eight characters.

```
DCL DELE$A EXTERNAL ENTRY(CHAR(8),FIXED BIN)
                           RETURNS(FIXED BIN);
```

The following PL/I statements call DELE$A and specify the action to be taken when DELE$A returns 1 and the action to be taken when it returns a value other than 1:

```
IF DELE$A ('OBSOLETE', 8) = 1 THEN
  PUT SKIP LIST ('FILE DELETED');
ELSE PUT SKIP LIST ('NO GO');
```

PL/I: FIXED BIN(15)
FTN: LOGICAL
F77: LOGICAL*2

The PL/I data type FIXED BIN(15) can be used as an equivalent of the FTN data type LOGICAL and to the F77 data type LOGICAL*2. Arguments declared as LOGICAL or LOGICAL*2 must have a value of 0 (false) or 1 (true).

The example in the section above concerning PL/I integer and character arrays illustrates a call to a function, DELE$A, that returns a LOGICAL value.

PL/I: CHAR(n) NONVARYING or literal
FTN: ASCII Character (String or Array)

An ASCII string expected by a subroutine should be declared in PL/I as CHAR(n) NONVARYING or passed as a literal.

9 Calling Subroutines From PMA

CALL FORMAT

To call a subroutine from a program written in PMA, use a statement of the following form:

```
    CALL sub-name
```

The CALL statement is followed on succeeding lines by statements that list the arguments to be passed. The succeeding statements begin with AP (address-pointer), followed by S or SL, as discussed in the following section.

You can also use the PCL machine instruction to call subroutines. However, the PCL instruction requires that you declare the subroutine as an external subroutine by means of the EXT statement, and that you code the pointers to the subroutine. Thus, the CALL statement provides the more convenient way to call subroutines from PMA.

Functions should be called from PMA as if they were subroutines. However, not all functions can be called in this way. If the function has an OPTIONAL RETURNED ARGUMENT, you can call the function as a subroutine. If the function has a RETURNED ARGUMENT, you cannot call the function as a subroutine.

For more information about how to call subroutines from PMA, see the Advanced Programmer's Guide and Chapter 12 of the Assembly Language Programmer's Guide.

CALLING SUBROUTINES FROM V-MODE AND I-MODE PMA

When PMA calls an external subroutine in V mode or I mode, arguments are passed by reference using the AP instruction. Each AP instruction except the last one in a call uses S as its second operand; the last AP instruction uses SL. All examples in this chapter can be used either in V mode or in I mode.

CALLING SUBROUTINES FROM R-MODE PMA

When PMA calls an external subroutine in R mode, arguments are passed by reference using the DAC pseudo-operation. If there is more than one argument, the last DAC pseudo-operation is followed by DATA 0. This is a convention of the operating system, not an architectural feature. If there is only one argument, DATA 0 must not be used.

USING SYSCOM FILES

You can insert the SYSCOM file that defines error codes into a PMA program by including the following statement in the program.

```
    $INSERT 'SYSCOM>ERRD.INS.PMA';
```

You can insert the SYSCOM file that defines key codes into a PMA program by including the following statement in the program.

```
    $INSERT 'SYSCOM>KEYS.INS.PMA';
```

Programs written in PMA cannot use the argument keys defined in the SYSCOM files; the numeric equivalents of these keys must be used instead. To learn the numeric equivalents of the argument keys, list the file SYSCOM>A$KEYS.INS.FTN, or any of the other SYSCOM files that define argument keys. For more information about argument keys, see Chapter 2 of this volume.

DATA TYPES

The following sections describe the argument types of FORTRAN and PL/I subroutines that can be called from PMA. For more information about each data type, refer to the Assembly Language Programmer's Guide, or to the chapter titled "Overview of Subroutines" in Volume II, III, or IV of this guide.

PMA: BSS pseudo-operation
FTN or F77: INTEGER*2
PL/I: FIXED BIN(15)

The FTN and F77 data type INTEGER*2 and the PL/I data type FIXED BIN(15), also called FIXED BIN, can be declared in PMA with the BSS pseudo-operation.

For example, the subroutine TEXTO$ expects four arguments, of the data types integer array, INTEGER*2, INTEGER*2, and LOGICAL. A PMA program that calls this subroutine must pass it arguments of the corresponding data types. The following EXT instruction declares TEXTO$ as an external subroutine; this instruction is required only when the subroutine is called by a PCL instruction, as in this example:

```
    EXT TEXTO$
```

The following PMA statements declare the data types of the last two arguments of TEXTO$:

```
    LEN      DATA  6         INTEGER*2 ARGUMENT
    OK       BSS   1         LOGICAL ARGUMENT
```

The following statement calls TEXTO$, with the first two arguments specified as literals:

```
             EXT   TEXTO$
    MAIN     PCL   TEXT_IP,*
             AP    =C'CTRLFL',S
             AP    =6,S
             AP    LEN,S
             AP    OK,SL
    TEXT_IP  IP    TEXTO$
```

Note

> Although BSS is ordinarily used in PMA to declare an INTEGER*2 parameter, the DYNM, BSZ, OCT, DEC, HEX, or DATA instructions could also be used. For information about these instructions, see the Assembly Language Programmer's Guide and the Instruction Sets Guide.

PMA: BSS 2 pseudo-operation
FTN or F77: INTEGER*4
PL/I: FIXED BIN(31)

The FTN and F77 data type INTEGER*4 and the PL/I data type FIXED BIN(31) are defined in PMA with the BSS 2 pseudo-operation.

For example, the subroutine RNUM$A expects four arguments of the types INTEGER*2, INTEGER*2, INTEGER*2, and INTEGER*4. A PMA program that calls RNUM$A must declare data types that correspond to those expected by the subroutine.

The following PMA statements declare the data types of the last two arguments of RNUM$A:

```
A$BIN    DATA 9            INTEGER*2 ARGUMENT
ITEM     BSS  2            INTEGER*4 ARGUMENT
```

The first two arguments can be specified as literals in the call to RNUM$A, as follows:

```
STRT     CALL RNUM$A       CALL SUBROUTINE TO ACCEPT NUMBER
         AP   =C'ENTER A NUMBER',S
         AP   =14,S        MESSAGE LENGTH
         AP   A$BIN,S      SYSCOM>A$KEY FOR BINARY
         AP   ITEM,SL      RETURNED VALUE
```

Note

> Although BSS 2 is ordinarily used in PMA to declare an INTEGER*4 parameter, it is also possible to use the DYNM x (2) or DATA xxxxL instructions. For information about these instructions, see the Assembly Language Programmer's Guide.

PMA: Declared using DEC statement
FTN: LOGICAL
F77: LOGICAL*2

The PMA DEC statement can be used to declare the FTN data type LOGICAL and the F77 data type LOGICAL*2. Both LOGICAL and LOGICAL*2 specify a 16-bit integer, with a value of 1 for true or 0 for false.

For example, the following PMA statement declares the variable L as a 16-bit halfword and initializes the variable to 0:

```
    L DEC 0
```

For another example of how to declare a LOGICAL parameter in PMA, see the section above that describes how to declare INTEGER*2 parameters.

PMA: BSS 2 pseudo-operation
FTN or F77: REAL*4 or REAL
PL/I: FLOAT BIN(23) or FLOAT BIN

The FTN and F77 data types REAL and REAL*4, and the PL/I data type FLOAT BIN(23) or FLOAT BIN, can be declared in PMA with the BSS 2 pseudo-operation.

For example, the function DTIM$A expects an INTEGER*4 argument and the function's value is received by a variable that must be REAL*4 or REAL*8. The following statements declare these data types for the two variables:

```
    DSKTIM  BSS 2       INTEGER*4 ARGUMENT
    RTVAL   BSS 2       REAL*4 ARGUMENT
```

Note

Although BSS 2 is ordinarily used in PMA to declare a REAL*4 parameter, it is also possible to use the DATA pseudo-operation to define the parameter as a data item with a decimal point or scientific notation (*nn*E*nn*). For information about these instructions, see the *Assembly Language Programmer's Guide*.

PMA: BSS 4 pseudo-operation
FTN and F77: REAL*8
PL/I: FLOAT BIN(47)

The FTN and F77 data type REAL*8 and the PL/I data type FLOAT BIN(47) can be declared in PMA with the BSS 4 pseudo-operation.

For example, the function CTIM$A expects an argument of the data type INTEGER*4 and its value is received by a variable of the data type REAL*8:

```
CPUTIM  BSS 2    INTEGER*4 ARGUMENT
RTVAL   BSS 4    REAL*8 ARGUMENT
```

Note

> Although BSS 4 is ordinarily used in PMA to declare a REAL*8 parameter, it is also possible to use the pseudo-operator DATA to declare the parameter as a data item with a decimal point or scientific notation and with (nnDnn) appended to it. For information about these instructions, see the Assembly Language Programmer's Guide.

PMA: Quad precision nnQnn format
F77: REAL*16

The PL/I data type REAL*16 is a quad precision floating-point number, implemented as a 128-bit value. It corresponds to the PMA format nnQnn, but can be passed to and from F77 only as a REAL*16 number. For details, see the Assembly Language Programmer's Guide.

PMA: Alphabetic or Integer Array
FTN or F77: Integer Array

An integer array in FTN or F77 can contain either alphabetic or numeric data. This may be passed as any data type.

For example, the subroutine TIMDAT expects two arguments, as indicated by the following DCL statement:

```
DCL TIMDAT (1..., FIXED BIN)
```

The first argument is an array to which TIMDAT is to return system and user information, in alphabetic and numeric form. The second argument is an integer value which must be set to 28. The first statement below declares the data type of variable string. TIMDAT writes system and user information into string. The second statement assigns num the value 28.

```
    STRING BSS 28
    NUM EQU 28
```

The following statements call TIMDAT with the arguments defined above:

```
    CALL TIMDAT
    AP STRING,S
    AP NUM,SL
```

PMA: C-string or BCI-string
FTN or F77: ASCII character string

ASCII characters can be passed to a FORTRAN subroutine as a constant string after the DATA statement. The string can be preceded by =C and enclosed in single quotation marks; for example, DATA =C'STEP 1'. The string can also be used in a BCI statement and enclosed by any delimiter. The maximum number of characters after C is 32. After BCI, you can use as many characters as fit on the same statement line.

For example, the subroutine SRCH$$ expects to receive arguments of the data types indicated by the following DCL statement:

```
    DCL SRCH$$ ENTRY (FIXED BIN, CHAR(32) VAR, FIXED BIN,
                      FIXED BIN, FIXED BIN, FIXED BIN);
```

In the DCL statement, the CHAR(32) VAR parameter is an ASCII character string to be passed to SRCH$$.

The following statements call subroutine TNOUA to display the message CODE; the first AP statement defines the text of the string, and the second AP statement specifies the number of characters in the string:

```
    MAIN      CALL TNOUA
              AP  =C'CODE ',S
              AP  =5,SL
```

The following statement inserts the SYSCOM>KEYS.INS.PMA file into program SRCH:

```
    $INSERT SYSCOM>KEYS.INS.PMA
```

The following statements call subroutine SRCH$$ to verify that CTRLFL exists in the UFD to which the user is attached:

```
    CALL  SRCH$$
    AP     =K$EXST+K$IUFD,S
    AP    =C'CTRLFL',S
    AP    =6,S
    AP    =0,S
    AP    =0,S
    AP    CODE,SL
```

The code that calls subroutine SRCH$$ uses the following statement to define the ASCII character string that gives the filename.

```
    AP    =C'CTRLFL',S
```

PMA: Record structure
PL/I: CHARACTER(*)VARYING

The PL/I data type CHARACTER(*)VARYING is implemented as a record structure containing a count of characters followed by the characters themselves. The record structure can be pictured as follows:

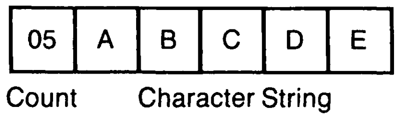

Figure 9-1
CHAR(*) VAR Record Structure

For example, a PMA program that calls subroutine GV$GET must declare data types that correspond to the types declared in the subroutine's DCL statement, as follows:

```
    DCL GV$GET ENTRY (CHAR(*)VAR, CHAR(*) VAR,
                      FIXED BIN, FIXED BIN);
```

The following PMA statement calls GV$GET with four arguments, each specified by an AP statement.

```
MAIN     CALL  GV$GET
         AP    NAME,S    CHAR*VAR ARGUMENT
         AP    VAL,S     CHAR*VAR RETURN ARGUMENT
         AP    SIZE,S    ONE-WORD ARGUMENT
         AP    CODE,SL   ONE-WORD RETURN ARGUMENT
```

The following PMA statements declare data types for the arguments specified in the CALL statement above. Note how the CHARACTER(*)VARYING arguments are defined as structures consisting of one-word integers followed by character strings.

```
NAME     DATA  4        ONE-WORD INTEGER +
         BCI   '.MAX'       FOUR-CHAR NAME
VAL      DATA  4        ONE-WORD INTEGER(SUPPLIED) +
         BSS   2            FOUR-CHARACTERS RETURNED
SIZE     DATA  4            16-BIT INTEGER
CODE     BSS   1        16-BIT INTEGER
```

PMA: DATA C'xxx...' or literal
F77: CHARACTER*n
PL/I: CHARACTER(n)NONVARYING

The PL/I data type CHARACTER(n)NONVARYING, usually declared as CHARACTER(n), and the F77 data type CHARACTER*n both consist of *n* characters. These data types can be declared in PMA as DATA C'*xxx*...', or passed as literals. Either item should be *n* characters long.

PMA: 16-bit integer
PL/I: BIT(1) ALIGNED

PL/I programs that expect arguments of this type should not be called from PMA unless the argument is declared in PL/I as BIT(1) ALIGNED. If the argument is declared as BIT(1) ALIGNED, it can be treated as a 16-bit integer, with a value of -1 for false.

APPENDICES

A FORTRAN Internal Subroutines

INTERNAL SUBROUTINES

The following subroutines are used internally by the FORTRAN compiler. They may be of some value to the PMA user and are briefly described. For calling sequence and further information, refer to the compiler or library source listings.

Table A-1
Subroutines Internal to FORTRAN

Subroutine	Function
F$A1	Input/output 16-bit integer.
F$A2	Input/output single-precision floating-point.
F$A3	Input/output logical.
F$A5	Input/output complex.

Table A-1 (continued)
Subroutines Internal to FORTRAN

Subroutine	Function
F$A6	Input/output double-precision floating-point.
F$A7	Input/output long integer.
F$AT	FORTRAN R-mode argument transfer subroutine.
F$ATI	FORTRAN argument transfer subroutine for PROTECTED subroutine.
F$BKSP	Backspace statement processor.
F$BN	Rewind logical device specified.
F$CB	End of READ/WRITE statement.
F$CG	FORTRAN computed GOTO processor.
F$CLOS	Close statement processor.
F$DE	Decode statement processor.
F$DEX	Decode statement processor with ERR=.
F$DN	Close (END-FILE) logical device specified.
F$EN	Encode statement processor.
F$END	Endfile statement processor.
F$FN	Provide backspace function to FORTRAN runtime programs.
F$IBR	Initialize unformatted read.
F$IBW	Initialize unformatted write.
F$IFR	Initialize formatted read.
F$IFW	Initialize formatted write.
F$ILDR	Initialize list-directed read.
F$ILDW	Initialize list-directed write.
F$INQF	Inquire by file statement processor.

Table A-1 (continued)
Subroutines Internal to FORTRAN

Subroutine	Function
F$INQU	Inquire by unit statement processor.
F$INR	Initialize namelist read.
F$IO77	Read and write variable-length records in default case of F$IO.
F$IOBF	F$IO buffer definition (up to 128 halfwords, for R mode and nonshared V mode; up to 16K-1 halfwords in shared V-mode library).
F$IOFTN	Read and write records in manner compatible with F$IO.
F$OPEN	Open statement processor.
F$PAUS	Pause statement processor.
F$RA	Read ASCII, no alternate returns.
F$RAX	Read ASCII, with ERR= and END= alternate returns.
F$RB	Read BINARY, no alternate returns.
F$RBX	Read BINARY with ERR= and END= alternate returns.
F$REW	Rewind statement processor.
F$RN	Read with no alternate returns.
F$RNX	Read with ERR= and END= alternate returns.
F$RTE	FORTRAN RETURN statement processor.
F$RX	COMMON read handler.
F$STOP	Stop statement processor.
F$TR	Perform the function of the FORTRAN TRACE routine.
F$WA	Write ASCII, no alternate returns.

Table A-1 (continued)
Subroutines Internal to FORTRAN

Subroutine	Function
F$WAX	Write ASCII with ERR= and END= alternate returns.
F$WB	Write BINARY, no alternate returns.
F$WBX	Write BINARY, with ERR= and END= alternate returns.
F$WN	Write with no alternate returns.
F$WNX	Write with ERR= alternate return.
F$WX	COMMON write handler.

INTRINSIC FUNCTIONS

The following subroutines are the FORTRAN library intrinsic function handlers:

Subroutine	Function
F$LS	Left shift
F$LT	Left truncate
F$OR	Inclusive OR
F$RS	Right shift
F$RT	Right truncate
F$SH	General shift

FLOATING-POINT EXCEPTIONS

The FLEX (or F$FLEX) subroutine is invoked by the compiler or system. This subroutine is the floating-point exception-interrupt processor. It determines the exception type, and returns a message as follows:

DE	Exponent underflow, store exception
DZ	Divide by 0
RI	Real-integer exception
SE	Exponent overflow

For further information on floating-point exception (FLEX), refer to the System Architecture Reference Guide.

B
Arithmetic Routines Callable From PMA

INTRODUCTION

Calls to the routines that perform mathematical calculations are generated by the FORTRAN compiler when arithmetic operations are specified in a FORTRAN program. They should not be called explicitly by a FORTRAN program, but may be called in a PMA program.

All of these subroutines are callable in 32R mode or 64R mode and are contained in FTNLIB. The subset of the subroutines that are necessary in 64V mode are in PFTNLB.

FORMAT AND ARGUMENTS

Subroutine names are of the form p$xy or F$pxy. p is a prefix; x is the first argument (argument-1); y is the second argument (argument-2).

The prefix specifies the action of the subroutine. (See Table B-1.) argument-1 is a number specifying the register in which the first argument is stored. (See Table B-2.) argument-2 is a number specifying the type of the second argument pointed to by a DAC (R mode) or AP (V mode) following the subroutine call. (See Table B-2.)

Table B-1
Subroutine Prefix Explanations

Prefix	Meaning	Number of Arguments
A	Addition	2
C	Conversion	1
D	Division	2
E	Exponentiation	2
H	Store complex number	1
L	Load complex number	1
M	Multiplication	2
N	Negation	1
S	Subtraction	2
Z	Zero double-precision exponent	1
	FORTRAN Support Subroutines (F$)	
DI	Positive difference	2
MA	Maximum	2
MI	Minimum	2
MO	Remainder (modulus)	2
SI	Magnitude of first times sign of second	2

Table B-2
Data Type Codes

Type Code	Register	Type
1	A	16-bit integer (INTEGER*2)
2	FAC	Single-precision floating-point number (REAL or REAL*4)
5	AC1-AC4	Complex number (COMPLEX)
6	DFAC	Double-precision floating-point number (DOUBLE PRECISION or REAL*8)
7	A+B	Long integer (INTEGER*4)
8	--	Exponent part of a double-precision number

<u>Keys</u>

A	A register
FAC	Floating-point accumulator
AC1-AC4	Complex accumulator addresses AC1 to AC4
DFAC	Double-precision floating-point accumulator
A+B	Concatenated A and B registers

<u>Note</u>

Some long-integer subroutines may need to be entered or exited in DBL mode (R mode only); this is noted with the description of these subroutines.

Note

In subroutines with only one argument, argument-2 has a slightly different meaning. This is discussed under the specific subroutines.

The following are examples of formats:

A$22	Adds two single-precision floating-point numbers (two arguments).
C$12	Floats a 16-bit integer to a single-precision floating-point number (one argument).

A complete list of subroutines of this type follows. In the rest of this appendix, the discussion is divided into subroutines with one argument and subroutines with two arguments.

A$21	C$26	D$51	E$27	F$DI11	F$SI11	M$77
A$51	C$27	D$52	E$51	F$DI71	F$SI71	
A$52	C$51	D$55	E$52	F$DI77	F$SI77	N$55
A$55	C$52	D$57	E$55			N$77
A$61	C$57	D$61	E$57	F$MA11	H$55	
A$62	C$61	D$62	E$61	F$MA22		S$21
A$77	C$62	D$67	E$62	F$MA77	L$55	S$51
	C$67	D$71	E$66			S$52
C$12	C$75	D$77	E$67	F$MI11	M$21	S$55
C$15	C$76		E$71	F$MI22	M$51	S$61
C$16	C$77	E$11	E$77	F$MI77	M$52	S$62
C$21		E$21			M$55	S$77
C$21G	D$21	E$22	F$CL	F$MO71	M$61	
C$25	D$27	E$26		F$MO77	M$62	Z$80

SINGLE-ARGUMENT SUBROUTINES

Each of these subroutines takes a single argument stored in the appropriate register, operates on it, and stores the result in the same or another register.

Conversion

C$xy

Converts the type of the argument in the register identified by _x_ to the type of the argument identified by _y_ and stores it in the proper register for _y_-type variables. For example, C$75 converts a long integer in the A+B register into the real part of a complex number in the complex accumulator (imaginary part is 0). See Table B-3 for a complete list.

Complex Number Manipulation

H$55

Stores the contents of the complex accumulator (AC1 to AC4) at the address specified by the DAC or AP following the call.

L$55

Loads the complex accumulator (AC1 to AC4) from the four halfwords pointed to by the DAC or AP following the call.

Negation

N$xx

Negates the value of the argument in the register specified by _x_, and stores it in that same register. (See Table B-3.)

Zeroing

Z$80

Clears the exponent part of the double-precision floating-point accumulator (DFAC). This is for R mode only.

Table B-3

Single-argument Subroutines
(Negation and Conversion)

x	y	N$ (Negation)	C$ (Conversion)
1	1		n/a
1	2	n/a	R
1	5	n/a	R,V
1	6	n/a	R
2	1	n/a	R (2)
2	2		n/a
2	5	n/a	R,V
2	6	n/a	R
2	7	n/a	R
5	1	n/a	R,V
5	2	n/a	R,V
5	5	R,V	n/a
5	7	n/a	R,V
6	1	n/a	R
6	2	n/a	R
6	6		n/a
6	7	n/a	R,V
7	2	n/a	
7	5	n/a	R
7	6	n/a	R,V
7	7	R (1)	R

Keys

n/a	Not applicable
R	Used in R mode only
R,V	Used in R mode or V mode
x	Argument type (See Table B-2.)
y	Result type (See Table B-2.)

Notes

1. Exit mode is DBL (R mode).

2. There is also a subroutine C$21G (R mode only), which performs the same functions as C$21 without the use of any floating-point instructions.

TWO-ARGUMENT SUBROUTINES

These subroutines perform arithmetic operations (addition, subtraction, and so on.) on two arguments. If the arguments do not have the same data type, the data type of the result is that of the higher. The data types, in descending order are:

```
COMPLEX or DOUBLE PRECISION
REAL
LONG INTEGER (INTEGER*4)
16-BIT INTEGER (INTEGER*2)
```

There are no operations that combine COMPLEX and DOUBLE PRECISION numbers (no "56" or "65" subroutines). The result of a two-argument subroutine is stored in the appropriate register for its data type. (See Table B-2.) For example:

```
R mode

CALL A$21
DAC I
```

floats the 16-bit integer I and adds it to the contents of the Floating Point Accumulator (FAC).

```
V mode

CALL F$MI11
AP I2,SL
```

loads I2 into the A register if I2 is less than the current contents of the A register.

Addition

A$xy

Adds argument of type y, pointed to by the DAC or AP following the call, to an argument of type x in the appropriate register. See Table B-4 for a complete list.

Division

D$xy

Divides the argument of type x in the appropriate register by the argument of type y, pointed to by the DAC or AP following the call. See Table B-4 for a complete list.

Exponentiation

E$xy

Raises the argument of type x in the appropriate register to the power specified by the argument of type y pointed to by the DAC or AP following the call. A complete list is given in Table B-4.

Note

In all modes, zero to the zero power is one.

Multiplication

M$xy

Multiplies the argument of type x in the appropriate register by the argument of type y pointed to by the DAC or AP following the call. See Table B-4 for a complete list.

Subtraction

S$xy

Subtracts the argument of type y, pointed to by a DAC or AP following the call, from an argument of type x in the appropriate register. See Table B-4 for a complete list.

Positive Difference

F$DIxy

Subtracts the argument of type y, pointed to by the DAC or AP following the call, from the argument of type x in the appropriate register. If the result is less than 0, the register is cleared. See Table B-5 for a complete list.

Maximum

F$MAxx

Places the maximum of the register, specified by type x, and the value of the argument of type x, pointed to by the DAC or AP, into the specified register. See Table B-5 for a complete list.

Table B-4

Two-argument
Arithmetic Subroutines (First Group)

x	y	A$ Addition	S$ Subtraction	M$ Multiplication	D$ Division	E$ Exponentiation
1	1					R,V
2	1	R	R	R	R,V	R,V
2	2					R,V
2	6					R,V
2	7				R,V	R,V
5	1	R,V	R,V	R,V	R,V	R,V
5	2	R,V	R,V	R,V	R,V	R,V
5	5	R,V	R,V	R,V	R,V	R,V
5	7				R,V	R,V
6	1	R	R	R	R,V	R,V
6	2	R	R	R	R,V	R,V
6	6					R,V
6	7				R,V	R,V
7	1				R,V	R,V
7	7	R(1)	R(1)	R(1)	R(1)	R,V(1)

Keys

R	Used in R mode only
R,V	Used in R mode or V mode
x	First argument, stored in appropriate register
y	Second argument, pointed to by DAC (R mode) or AP (V mode)

Note

Exit mode is DBL (R mode).

Minimum

F$MIxx

Places the minimum of the register specified by type x and the value of the argument of type x, pointed to by the DAC or AP, into the specified register. See Table B-5 for a complete list.

Remainder

F$MOxy

Divides an argument of type x in the appropriate register by an argument of type y, pointed to by the DAC or AP. The remainder is placed in the appropriate register. See Table B-5 for a complete list.

Sign and Magnitude

F$SIxy

Multiplies the argument of type x in the appropriate register by the sign of the argument of type y pointed to by the DAC or AP and stores the result in the register for type x. See Table B-5 for a complete list.

Comparison (R mode Only)

F$CL

Compares the long integer L1 in the concatenated A and B registers with the long integer L2, pointed to by a DAC following the call. Control passes as follows:

L1>L2	Next location
L1=L2	Skip one location
L1<L2	Skip two locations

The A and B registers are not modified. For example:

```
    CALL F$CL
    DAC L2
    ...return here if L1>L2
    ...return here if L1=L2
    ...return here if L1<L2
```

Table B-5

Two-argument
Arithmetic Subroutines (Second Group)

x	y	F$MO Remainder	F$SI Sign and Magnitude	F$DI Positive Difference	F$MA Maximum	F$MI Minimum
1	1		R,V	R,V	R,V	R,V
2	2				R,V	R,V
7	1	R,V	R,V	R,V		
7	7	R,V	R,V	R,V	R,V	R,V

Keys

R	Used in R mode only
R,V	Used in R mode or V mode
x	First argument, stored in appropriate register
y	Second argument, pointed to by DAC (R mode) or AP (V mode)

C
Data Type Equivalents

To call a subroutine from a program written in any Prime language, you must declare the subroutine and its parameters in the calling program. Therefore, you must translate the PL/I data types expected by the subroutine into the equivalent data types in the language of the calling program.

The table that follows shows the equivalent data types for the Prime languages BASIC/VM, C, COBOL 74, FORTRAN IV, FORTRAN 77, Pascal, and PL/I. The leftmost column lists the generic storage unit, which is measured in bits, bytes, or halfwords for each data type. Each storage unit matches the data types listed to the right on the same row. The table does not include an equivalent data type for each generic unit in all languages. However, with knowledge of the corresponding machine representation, you can often determine a suitable workaround. For instance, to see if you can use a left-aligned bit in COBOL 74, you could write a program to test the sign of the 16-bit field declared as COMP. In addition, if a subroutine parameter consists of a structure with elements declared as BIT(n), it can be declared as an integer in the calling program.

For more information on matching data types for a particular language, see the chapter on that language in this volume. The data type tables in these chapters may suggest additional ways of declaring the data types expected by subroutines or functions.

<u>Note</u>

> The term PL/I refers both to full PL/I and to PL/I Subset G (PL/I-G).

Table C-1
Data Type Equivalents

Generic Unit	BASIC/VM SUB FORTRAN	C	COBOL 74	FORTRAN IV	FORTRAN 77	Pascal	PL/I
16-bit integer	INT	short enum	COMP PIC S9(1)- PIC S9(4)	INTEGER INTEGER*2 LOGICAL	INTEGER*2 LOGICAL*2	INTEGER Enumerated	FIXED BIN FIXED BIN(15)
32-bit integer	INT*4	int long	COMP PIC S9(5)- PIC S9(9)	INTEGER*4	INTEGER INTEGER*4 LOGICAL LOGICAL*4	LONGINTEGER	FIXED BIN(31)
64-bit integer			COMP PIC S9(10)- PIC S9(18)				
32-bit float single precision	REAL	float	COMP-1	REAL REAL*4	REAL REAL*4	REAL	FLOAT BIN FLOAT BIN(23)
64-bit float double precision	REAL*8	double	COMP-2	REAL*8	REAL*8	LONGREAL	FLOAT BIN(47)
128-bit float quad precision					REAL*16		
1 bit		short					BIT BIT(1)
1 left-aligned bit		short				BOOLEAN	BIT(1) ALIGNED

Table C-1 (continued)
Data Type Equivalents

Generic Unit	BASIC/VM SUB FORTRAN	C	COBOL 74	FORTRAN IV	FORTRAN 77	Pascal	PL/I
Bit string		unsigned int				SET	BIT(n)
Fixed-length character string	INT	char;	DISPLAY PIC A(n) PIC X(n) FILLER		CHARACTER *n	CHAR PACKED ARRAY[1..n] OF CHAR	CHAR(n)
Fixed-length digit string			DISPLAY PIC 9(n)				PICTURE
Fixed-length digit string, 2 digits per byte			COMP-3				FIXED DECIMAL
Varying-length character string		char NAME[n]; char *NAME;				STRING[n]	CHAR(n) VARYING
32-bit pointer		Pointer (32IX-mode)					POINTER OPTIONS (SHORT)
48-bit pointer		Pointer (64V-mode)				Pointer	POINTER

Notes

For a discussion of possible workarounds for some of the empty boxes in this table as well as a description of generic units for PMA, refer to the appropriate language chapter in the *Subroutines Reference Guide, Volume I.*

The BASIC/VM column lists FTN data types to be declared in the SUB FORTRAN statement in a BASIC/VM program.

INDEXES

Index of Subroutines by Name

AT$ABS	Set the attach point to a specified top-level directory and partition.	II	3-6
AT$ANY	Set the attach point to a specified top-level directory on any partition.	II	3-8
AT$HOM	Set the attach point to the home directory.	II	3-10
AT$LDEV	Set the attach point by top-level directory and logical disk number.	II	3-11
AT$OR	Set the attach point to the login directory.	II	3-13
AT$REL	Set the attach point relative to the current directory.	II	3-15
ATCH$$	Set the attach point to a specified directory.	II	A-2
ATTDEV	Change a device assignment temporarily.	IV	3-6
BIN$SR	Perform binary search in ordered table.	III	6-21
BNSRCH	Binary search.	IV	17-48
BREAK$	Inhibit or enable BREAK function.	III	3-50
BUBBLE	Bubble sort.	IV	17-50
C$xy series	FORTRAN compiler conversion functions.	I	B-5
C$A01	Control functions for user terminal.	IV	6-5
C$M05	Control functions for 9-track tape.	IV	E-5
C$M10	Control functions for 7-track tape.	IV	E-5
C$M11	Control functions for 7-track tape (BCD).	IV	E-5
C$M13	Control functions for 9-track tape (EBCDIC).	IV	E-5
C$P02	Control functions for paper tape.	IV	6-12
C1IN	Read a character.	III	3-5
C1IN$	Read a character.	III	3-7
C1NE$	Read a character, suppressing echo.	III	3-9
CALAC$	Determine whether an object is accessible for a given action.	II	2-17
CASE$A	Convert between upper- and lowercase.	IV	14-2
CAT$DL	Delete an access category.	II	2-19
CE$BRD	Return caller's maximum command environment breadth.	II	6-3
CE$DPT	Return caller's maximum command environment depth.	II	6-4
CH$FX1	Convert string (decimal) to 16-bit integer.	III	6-3
CH$FX2	Convert string (decimal) to 32-bit integer.	III	6-5
CH$HX2	Convert string (hexadecimal) to 32-bit integer.	III	6-7
CH$MOD	Change the open mode of an open file.	II	4-6
CH$OC2	Convert string (octal) to 32-bit integer.	III	6-9

CTIM$A	Return CPU time since login.	IV	12-2
CV$DQS	Convert binary date to quadseconds.	III	6-12
CV$DTB	Convert ASCII date to binary format.	III	6-13
CV$FDA	Convert binary date to ISO format.	III	6-15
CV$FDV	Convert binary date to "visual" format.	III	6-17
CV$QSD	Convert quadsecond date to binary format.	III	6-19
D$xy series	FORTRAN compiler division functions.	I	B-8
D$INIT	Initialize disk.	IV	5-13
DATE$	Return current date and time.	III	2-8
DATE$A	Return today's date, American style.	IV	12-3
DELE$A	Delete a file.	IV	15-3
DIR$CR	Create a new directory.	II	4-15
DIR$LS	Search for specified types of entries in a directory open on a file unit.	II	4-17
DIR$RD	Read sequentially the entries of a directory open on a file unit.	II	4-24
DIR$SE	Return directory entries meeting caller-specified selection criteria.	II	4-29
DISPLY	Update sense light settings.	III	10-3
DKGEO$	Register disk format with driver.	IV	5-18
DLINEQ	Solve a system of linear equations (double precision).	IV	18-7
DMADD	Matrix additions (double precision).	IV	18-9
DMADJ	Calculate adjoint matrix (double precision).	IV	18-11
DMCOF	Calculate signed cofactor (double precision).	IV	18-13
DMCON	Set matrix to constant matrix (double precision).	IV	18-16
DMDET	Calculate determinant (double precision).	IV	18-18
DMIDN	Set matrix to identity matrix (double precision).	IV	18-20
DMINV	Calculate inverted matrix (double precision).	IV	18-22
DMMLT	Matrix multiplication (double precision).	IV	18-24
DMSCL	Multiply matrix by a scalar (double precision).	IV	18-26
DMSUB	Matrix subtraction (double precision).	IV	18-28
DMTRN	Calculate transpose matrix (double precision).	IV	18-30
DOFY$A	Return today's date as day of year (Julian).	IV	12-4
DS$AVL	Return data about a disk partition.	III	2-51
DS$ENV	Return data about a process's environment.	III	2-53
DS$UNI	Return data about file units.	III	2-57
DTIM$A	Return disk time since login.	IV	12-5

SR$ADDB	Add a rule to the start of a search list or before a specified rule within the list.	II	7-20
SR$ADDE	Add a rule to the end of a search list or after a specified rule within the list.	II	7-23
SR$CREAT	Create a search list.	II	7-26
SR$DEL	Delete a search list.	II	7-28
SR$DSABL	Disable an optional search rule enabled by SR$ENABL.	II	7-30
SR$ENABL	Enable an optional search rule.	II	7-33
SR$EXSTR	Determine if a search rule exists.	II	7-36
SR$FR_LS	Free list structure space allocated by SR$LIST or SR$READ.	II	7-40
SR$INIT	Initialize all search lists to system defaults.	II	7-42
SR$LIST	Return the names of all defined search lists.	II	7-44
SR$NEXTR	Read the next rule from a search list.	II	7-48
SR$READ	Read all of the rules in a search list.	II	7-53
SR$REM	Remove a rule from a search list.	II	7-57
SR$SETL	Set the locator pointer for a search rule.	II	7-60
SR$SSR	Set a search list via a user-defined search rules file.	II	7-63
SRCH$$	Open, close, delete, or verify existence of an object.	II	4-99
SRSFX$	Search for a file with a list of possible suffixes.	II	4-108
SRTF$S	Sort several input files.	IV	17-16
SS$ERR	Signal an error in a subsystem.	III	5-11
SSTR$A	Shift string left or right.	IV	10-28
SSUB$A	Shift substring left or right.	IV	10-30
SSWTCH	Test sense switch settings.	III	10-14
ST$SGS	Return maximum number of static segments.	III	4-26
STR$AL	Allocate user-class dynamic memory.	III	4-5
STR$AP	Allocate process-class dynamic memory.	III	4-7
STR$AS	Allocate subsystem-class dynamic memory.	III	4-8
STR$AU	Allocate user-class dynamic memory.	III	4-10
STR$FP	Free process-class dynamic memory.	III	4-11
STR$FR	Free user-class dynamic memory.	III	4-12
STR$FS	Free subsystem-class dynamic memory.	III	4-13
STR$FU	Free user-class dynamic memory.	III	4-14
SUBSRT	Sort file on ASCII key. (V mode)	IV	17-10
SUBSRT	Sort file on ASCII key. (R mode)	IV	17-40
SUSR$	Test if current user is supervisor.	III	2-39
T$AMLC	Communicate with AMLC driver.	IV	8-23
T$CMPC	Input from MPC card reader.	IV	7-28
T$LMPC	Move data to LPC line printer.	IV	7-6

VALID$	Validate a name against composite identification.	III	2-48
WILD$	Return a logical value indicating whether a wildcard name was matched.	II	4-118
WRASC	Write ASCII.	IV	4-3
WRBIN	Write binary to any output device.	IV	4-7
WRECL	Write disk record.	IV	5-17
WTLIN$	Write a line of characters to a compressed ASCII file.	II	4-120
YSNO$A	Ask question and obtain a yes or no answer.	IV	11-7
Z$80	Clear double-precision exponent.	I	B-5

Index

SURVEY

READER RESPONSE FORM

Subroutines Reference Guide Volume I DOC10080-2LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   ____
   ____
   ____
   ____
   ____
   ____

3. What faults or errors in this manual gave you problems?

   ____
   ____
   ____
   ____
   ____
   ____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   ____
   ____

Name:____________________ Position:____________________

Company:______________________________

Address:______________________________

____

____________________ Postal Code:____________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760