

March 1978

This document describes the system dependent features of BASIC-11/RT-11. In conjunction with the *BASIC-11 Language Reference Manual* (DEC-11-LIBBB-A-D), this document provides the information required to write and run a BASIC program under the RT-11 operating system.

## **BASIC-11/RT-11 User's Guide**

Order No. DEC-11-LIBUA-A-D, DN1

<b>SUPERSESSION/UPDATE INFORMATION:</b>	This document in conjunction with the <i>BASIC-11 Language Reference Manual</i> (DEC-11-LIBBB-A-D) completely supersedes the <i>BASIC/RT-11 Language Reference Manual</i> (DEC-11-LBACA-E-D), published October 1976. This document includes Update Notice No. 1.
<b>OPERATING SYSTEM AND VERSION:</b>	RT-11 V03
<b>SOFTWARE VERSION:</b>	BASIC-11/RT-11 V2

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

First Printing, September 1977  
Revised, March 1978

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

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

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

Copyright © 1977, 1978 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECTape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

## CONTENTS

	Page
PREFACE	v
DOCUMENTATION CONVENTIONS	vi
CHAPTER 1 GETTING STARTED WITH BASIC-11/RT-11	1-1
1.1 OPTIONAL FEATURES	1-1
1.2 STARTING BASIC	1-2
1.2.1 Running BASIC With the Single Job Monitor or as the Background Job	1-2
1.2.2 Running BASIC As the Foreground Job	1-4
1.2.3 Running BASIC From an Indirect File	1-5
1.3 STOPPING BASIC PROGRAMS (CTRL/C COMMAND)	1-6
1.4 TERMINATING THE SESSION (BYE COMMAND)	1-7
1.5 FLOATING POINT NUMBER PRECISION	1-7
1.6 SYSTEM DEPENDENT ERROR MESSAGES	1-8
CHAPTER 2 FILES	2-1
2.1 FILE SPECIFICATION	2-1
2.2 THE OPEN STATEMENT - SYSTEM DEPENDENT FEATURES	2-3
2.3 LISTING YOUR FILE DIRECTORY	2-3
CHAPTER 3 UTILITY FUNCTIONS	3-1
3.1 BASIC UTILITY FUNCTIONS	3-1
3.2 SETTING THE TERMINAL MARGIN (TTYSET FUNCTION)	3-1
3.3 CANCELING THE EFFECT OF CTRL/O (RCTRL/O FUNCTION)	3-2
3.4 DISABLING CTRL/C (RCTRLC AND CTRLC FUNCTIONS)	3-3
3.5 TERMINATING YOUR PROGRAM (ABORT FUNCTION)	3-4
3.6 SYSTEM FUNCTIONS	3-5
3.6.1 Single Character Input	3-6
3.6.2 Terminating BASIC	3-6
3.6.3 Checking for CTRL/C	3-7
3.6.4 Enabling Lower Case Support	3-7
CHAPTER 4 USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC	4-1
4.1 INTRODUCTION TO ASSEMBLY LANGUAGE ROUTINES	4-1
4.2 FORMAT OF THE ASSEMBLY LANGUAGE ROUTINE	4-2
4.3 ACCESSING THE ARGUMENTS - THE ARGUMENT LISTS	4-4
4.3.1 Numeric Arrays	4-7
4.3.2 Strings and String Arrays	4-7
4.4 USING ROUTINES PROVIDED BY BASIC	4-9
4.4.1 Error Handling and Message Routines	4-9
4.4.2 Mathematical Operation and Function Routines	4-11
INDEX	Index-1

## CONTENTS (Cont.)

			Page
FIGURES			
FIGURE	4-1	User Routine Name Table and Routine Name Formats	4-2
	4-2	Assembly Language Routine Argument Lists	4-5
	4-3	Format of the Argument Descriptor Word	4-6
	4-4	Format of Array and String Argument Descriptors	4-7
	4-5	State of Stack for Threaded Code Routines	4-14
	4-6	Argument List for Supplied Single-Precision Routines	4-16
	4-7	Argument List for Supplied Double-Precision Routines	4-16

TABLES			
TABLE	2-1	RT-11 Device Names	2-1
	2-2	Default File Names	2-2
	2-3	Default File Types	2-2
	3-1	Summary of System Functions	3-6
	4-1	Using String Access Routine	4-10
	4-2	BASIC Mathematical Operations	4-13
	4-3	BASIC Mathematical Functions	4-13

## PREFACE

Before reading this manual, you should be familiar with the BASIC-11 language and the RT-11 system. If necessary, read the following manuals before reading this User's Guide:

- BASIC-11 Language Reference Manual (DEC-11-LIBBB-A-D)
- Introduction to RT-11

or

- RT-11 System User's Guide

Most features of BASIC-11/RT-11 V2 are the same as in other versions of BASIC-11. (DIGITAL's name for a family of BASICs for the PDP-11). These features are described in the BASIC-11 Language Reference Manual (DEC-11-LIBBB-A-D).

This guide describes the system dependent features of BASIC-11/RT-11. They are:

- Procedure for starting BASIC
- Effect of the CTRL/C key command
- Accuracy of storing numbers
- Format of error messages
- Format of the file specification
- Effects of parameters in the OPEN statement
- Procedure for checking files
- Effect of superseding files
- Effects of the utility functions
- Procedure for using assembly language routines
- Procedure for terminating BASIC

All BASIC users should read this guide excluding only Chapter 4. Only users who are adding assembly language routines to BASIC need to read Chapter 4. Chapter 4 assumes that you are an experienced RT-11 MACRO programmer.

This guide assumes that you have linked BASIC according to the procedure described in the BASIC-11/RT-11 Installation Guide (DEC-11-LIBTA-A-D).

## DOCUMENTATION CONVENTIONS

This section describes the documentation conventions, notations, and symbols used throughout this manual.

The following symbols denote special terminal keys that you will use frequently when using BASIC.

Symbol	Meaning
<code>CTRL/x</code>	While pressing the CTRL key, type the letter indicated after the slash.
<code>RET</code>	Type the RETURN key.
<code>ESC</code>	Type the ESCAPE key (ALTMODE on some terminals).
<code>DEL</code>	Type the DELETE key (RUBOUT on some terminals).

In addition, this manual uses certain conventions when describing the format of statements, functions, and commands.

These are:

Convention	Meaning
<code>[ ]</code>	The enclosed elements are optional. For example:  <code>[LET] variable=expression</code>
<code>{ }</code>	A choice of one element among two or more possibilities, for example:  IF relational expression { THEN statement THEN line number } GO TO line number }
...	Preceding element can be repeated as indicated. For example:  line number CLOSE#expr1,#expr2,...
Items in capital letters and special symbols	Type these elements exactly as they appear in the format, for example:  LET RUN #
	Items in capital letters are called keywords.
Items in lower case letters	Replace these elements according to the description provided in text. See below for list of commonly used lower case items.

This list describes some lower case items commonly used in format descriptions. The general meaning of each item is given. Unless a specific format description places restrictions on an item, its general meaning applies. See the BASIC-11 Language Reference Manual for more information on these items.

Lower Case Item	Abbreviation	Meaning
expression	expr	Any valid BASIC-11 expression. It is always a numeric expression unless the description specifically states that it can be a numeric or string expression. For example: (5*SIN(X))^Y
file specification	---	A file specification as described in Section 2.1
integer	int	Any positive integer number constant or any positive numeric constant that could be an integer if a percent sign were put after it. For example: 5%, 3%, 2, 7
line number	---	Any valid line number. For example: 10, 100, 32767
string	---	Any string expression. For example: "ABC", C\$+SEG\$(A\$,3,4)
variable	var	A floating point, integer or string variable.

If more than one lower case word appears in a format, the words are numbered 1, 2, 3, etc. For example:

```
CLOSE #expr1,#expr2,#expr3,...
```

Throughout this manual, the term BASIC means BASIC-11 or BASIC-11/RT-11.

To differentiate between what BASIC prints and what you type, the user type-in is printed in red ink. For example:

```
RUNNH
WHAT NUMBERS? 5,10
THE SUM IS 15
READY
```

All user type-in is terminated by the RETURN key unless the text indicates a different terminator.

## CHAPTER 1

### GETTING STARTED WITH BASIC-11/RT-11

#### 1.1 OPTIONAL FEATURES

BASIC-11/RT-11 has numerous optional features. If you include all optional features, any feature described in the BASIC-11 Language Reference Manual or in this guide is available. By excluding some or all optional features, you can increase the amount of memory available for programs or have faster program execution, or both.

BASIC-11/RT-11 has available the following optional features:

##### Statements

CALL  
PRINT USING

##### Commands

SUB  
RESEQ

##### Functions

SQR	SYS	ABS	SEG\$
SIN	RCTRL0	SGN	VAL
COS	ABORT	BIN	TRM\$
ATN	TTYSET	OCT	STR\$
LOG	CTRLC	LEN	PI
LOG10	RCTRLC	ASC	INT
EXP	TAB	CHR\$	DAT\$
	RND	POS	CLK\$

##### Miscellaneous

- Double precision arithmetic
- Long error messages
- Exponentiation (e.g., the expression  $A^B$ )
- Ability to run BASIC as foreground or background job
- Features affecting program space availability and program execution speed

You must specify the inclusion or exclusion of some optional features at BASIC linking time. Others you select at BASIC run time. The features you can choose when you link BASIC are:



## GETTING STARTED WITH BASIC-11/RT-11

- All optional statements
- All optional commands
- SQR, SIN, COS, ATN, EXP, LOG, and LOG10 functions
- All miscellaneous optional features

The features you can choose at run time are the following optional functions:

SYS	ABS	SEG\$
RCTRLO	SGN	VAL
ABORT	BIN	TRM\$
TTYSET	OCT	STR\$
CTRLC	LEN	PI
RCTRLC	ASC	INT
TAB	CHR\$	DAT\$
RND	POS	CLK\$

Before using BASIC you must link a version with the optional features you want. See the BASIC-11/RT-11 Installation Guide for instructions to link BASIC and for information about allowed program size and speed of execution tradeoffs.

### 1.2 STARTING BASIC

You can use BASIC with either the single-job (SJ), foreground/background (FB), or extended memory (XM) RT-11 V3 monitor. When using the FB or XM monitor, you can run BASIC as either the foreground or background job.

Before starting BASIC, you must bootstrap RT-11 and enter the DATE and TIME commands. See the Introduction to RT-11 for a description of these procedures.

#### 1.2.1 Running BASIC With the Single Job Monitor or as the Background Job

To run BASIC with the SJ monitor or as the background job, enter either the BASIC or the RUN command. The BASIC command runs the file BASIC.SAV on your system device. To enter the BASIC command, type:

```
.BASIC
```

To use another version of BASIC, type:

```
.RUN file specification
```

where:

file specification specifies the file containing the version of BASIC that you want.

For example, if you have a version of BASIC on device DX1: with file name BAS8K, and you want that version instead of the one in BASIC.SAV, you should enter:

```
.RUN DX1:BAS8K
```

If you specify a file that does not exist, RT-11 prints the message:

```
?KMON-F-File not found
```

## GETTING STARTED WITH BASIC-11/RT-11

If there is not enough room to run BASIC, one of the following messages is printed:

```
NOT ENOUGH MEMORY FOR BASIC
```

or

```
?KMON-F--Not enough memory
```

This error often results from a large foreground job that has not been unloaded.

If there are no errors, BASIC prints an identifying message and inquires whether you want the optional functions that are selectable at run time.

```
.BASIC
BASIC-11/RT-11 V02-xx
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)?
```

To include all of the optional functions, type an A. To exclude all of the optional functions, type an N. (You must always terminate input to BASIC with the RETURN key.) In response to your A or N, BASIC includes or excludes all the functions and then prints the READY message. For example:

```
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)?A
READY
```

Typing only the RETURN key in response to the optional functions request is equivalent to typing A.

If you want to choose among the optional functions individually, type an I. BASIC then prints an inquiry for each function individually. To include a function type a Y; otherwise type an N. Typing only the RETURN key in response to the function request is equivalent to typing Y. If you type anything else, BASIC repeats its request. After you have typed a Y or an N in response to each function inquiry, BASIC prints the READY message. For example:

```
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? I
SYS? N
RCTRL? N
ABORT? N
TTYSET? N
CTRLC & RCTRL? N
TAB? Y
RND? Y
ABS? Y
SGN? Y
BIN? Y
OCT? Y
LEN? N
ASC? N
CHR$? N
POS? N
SEG$? N
VAL? N
TRM$? N
STR$? N
PI? N
```

## GETTING STARTED WITH BASIC-11/RT-11

```
INT? Y
DAT? N
CLK? N
```

READY

### 1.2.2 Running BASIC As the Foreground Job

To run BASIC as the foreground job, use the FRUN command. Type:

```
.FRUN file specification /N:number.
```

where:

file specification	specifies the file containing BASIC
number	is the size of the user area (i.e., the number of words to be reserved). It must be 1000. or greater. The decimal point identifies the number as decimal, not octal.

You must specify the user area size, or else no area will be reserved and BASIC will not be able to run.

The user area will actually be approximately 100 words more than you request. For example, the following command reserves approximately 3100 words.

```
.FRUN BASIC/N:3000.
```

If the file specified does not exist, RT-11 prints the message:

```
?KMON-F-File not found
```

If the number of words requested in the FRUN command is not large enough, BASIC prints the message:

```
NOT ENOUGH MEMORY FOR BASIC
```

If there are no errors, RT-11 prints a dot and the F> message to indicate that the next message is printed by the foreground job. BASIC then prints an identifying message and inquires whether you want the optional functions. For example:

```
.FRUN BASIC/N:3000.
*
F>
BASIC-11/RT-11 V02-xx
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)?
```

Type a CTRL/F and then answer the optional function inquiry as described in the previous section.

#### NOTE

To use a device other than the system device, you must load the handler before you run BASIC in the foreground. See the RT-11 System User's Guide for more information about foreground jobs.

## 1.2.3 Running BASIC From an Indirect File

You can run BASIC and answer the initial dialogue by using an indirect file. You can only run BASIC in this way as the background job or in the single job monitor. This technique is useful when you select the optional functions individually.

You cannot enter any BASIC command, program line, or immediate mode statement through an indirect file.

To create the indirect file, direct the editor to create a file with a file type .COM that contains all anticipated responses to system queries. For example:

```
.R EDIT
*EWMINRUN.COM (ESC) (ESC)
*IR BASIC
I
N
N
N
N
N
N
Y
Y
Y
Y
N
N
N
N
N
N
N
N
N
N
Y
N
N
(ESC) (ESC)
*EX (ESC) (ESC)
*
```

To start BASIC, type an @ ("at" sign) followed by the file name. The complete initial dialogue is printed on the terminal. For example:

```
.*@MINRUN

.R BASIC
BASIC-11/RT-11 V02--XX
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? I
SYS? N
RCTRL? N
ABORT? N
TTYSET? N
CTRLC & RCTRL? N
TAB? N
```

```

RND? Y
ABS? Y
SGN? Y
BIN? Y
OCT? Y
LEN? N
ASC? N
CHR$? N
POS? N
SEG$? N
VAL? N
TRM$? N
STR$? N
PI? N
INT? Y
DAT$? N
CLK$? N

```

```
READY
```

See the RT-11 System User's Guide for more information on using indirect files.

### 1.3 STOPPING BASIC PROGRAMS (CTRL/C COMMAND)

To stop execution of a BASIC program, use the CTRL/C command. If you type one CTRL/C, BASIC interrupts your program the next time it requests input. If you type two consecutive CTRL/C's, BASIC interrupts your program immediately. After BASIC interrupts your program, it prints:

```

STOP AT LINE xxxxx
READY

```

where:

```

xxxxx      is the number of the line that BASIC was executing when
            the CTRL/C command halted the program.

```

However, if you were not executing a program line, BASIC prints:

```

STOP
READY

```

When you type CTRL/C, the system prints ^C. For example:

```

10 GO TO 10
RUNNH
^C ^C
STOP AT LINE 10
READY

```

#### NOTE

CTRL/C does not return control to the RT-11 monitor. You must type the BYE command (see Section 1.4) to return control to RT-11.

#### 1.4 TERMINATING THE SESSION (BYE COMMAND)

To terminate a session with BASIC, type the BYE command. The BYE command returns control to the RT-11 monitor, which prints its prompting period. For example:

```
BYE
```

\*

Once you have entered the BYE command you cannot use the RT-11 REENTER command to return to BASIC. Instead, you must restart BASIC as described in Section 1.2. If you want to reuse your BASIC program, save it before entering the BYE command.

If you ran BASIC as the foreground job, you must unload it after you enter the BYE command. Type:

```
.UNLOAD FG
```

\*

#### 1.5 FLOATING POINT NUMBER PRECISION

You can use BASIC with either single or double precision arithmetic. Single precision arithmetic allows floating point numbers to seven digits of precision. Thus, single precision BASIC stores the numbers 1.000001 and 1.000000 (seven digits) differently but stores 1.0000001 and 1.0000000 (eight digits) as the same number. Double precision arithmetic allows you to specify floating point numbers to 15 digits of precision.

If you need more than seven digits of precision, you should use BASIC with double precision arithmetic. However, double precision BASIC has two disadvantages.

1. It allows less BASIC program space, because BASIC itself requires more memory and because all floating point constants, variables, and arrays require twice the memory that single precision would need.
2. Arithmetic operations and functions run more slowly with double precision than with single precision.

The PRINT statement only prints six digits even when you are using double precision arithmetic. Consequently, if you want to print a number with more than six digits, you must use the PRINT USING statement or the STR\$ function. The following example was run using double precision arithmetic.

```
LISTNH
10 X=4.237194237
20 Y=6.9090909
30 PRINT X*Y
40 PRINT USING "###.#####",X*Y
50 PRINT STR$(X*Y)
```

```
READY
```

## GETTING STARTED WITH BASIC-11/RT-11

RUNNH

29.2752  
29.2751601  
29.275160144389

READY

Double precision compiled BASIC uses the default file type .BAX while single precision compiled BASIC programs have the default file type .BAC. The different default file types are necessary because double precision BASIC cannot read a program compiled by single precision BASIC and vice versa. If you are using double precision BASIC and specify the file type of a program compiled by single precision BASIC or vice versa, the results are unpredictable.

### 1.6 SYSTEM DEPENDENT ERROR MESSAGES

Some of the error messages listed in the BASIC-11 Language Reference Manual either have special meaning in BASIC-11/RT-11 or are not produced by it. These error messages are

?CANNOT DELETE FILE (?CDF)

BASIC-11/RT-11 does not produce this message.

?ERROR CLOSING CHANNEL (?ECC)

BASIC-11/RT-11 does not produce this error message. If an error occurs when BASIC-11/RT-11 is trying to close a channel, BASIC-11/RT-11 prints the ?CHANNEL I/O ERROR (?CIE).

?FILE ALREADY EXISTS (?FAE)

BASIC-11/RT-11 does not produce this message.

?FILE PRIVILEGE VIOLATION (?FPV)

BASIC-11/RT-11 does not produce this message.

?FILE TOO SHORT (?FTS)

The file is too small to contain the output. If the error occurs in a data file, specify a larger FILESIZE. If the error occurs in a program file, delete unused files with the UNSAVE command and then retry.

?ILLEGAL DEF (?IDF)

BAISC-11/RT-11 does not produce this message.

?ILLEGAL FILE LENGTH

The FILESIZE specified was less than -1 (see Section 2.2).

?ILLEGAL RECORD SIZE (?IRS)

BASIC-11/RT-11 does not produce this message.

?NOT A VALID DEVICE (?NVD)

BASIC-11/RT-11 does not produce this message.

?NOT ENOUGH ROOM (?NER)

There is not enough room for the FILESIZE specified. Delete unused files with the UNSAVE command.

## CHAPTER 2

### FILES

#### 2.1 FILE SPECIFICATION

BASIC uses the standard RT-11 file specification. Its format is:

[device:] [filename] [.type]

where:

device is the device name. It can be any device name listed in Table 2-1 or any assigned device name (see the RT-11 User's Guide).

filename is the one- to six-character name of the file.

type is the zero- to three-character type of the file.

Table 2-1  
RT-11 Device Names

Code	Device
CR:	Card Reader
CTn:	Cassette
DLn:	RL01 Disk
DMn:	RK06 Disk
DPn:	RP02 Disk
DSn:	RJS03/4 Disk
DTn:	DEctape
DXn:	RX11 Diskette
LP:	Line Printer
MMn:	TJU16 Magtape
MTn:	TM11 Magtape
PC:	Combined high-speed paper tape reader and punch
RF:	RF11 Disk

(continued on next page)



## FILES

Table 2-1 (Cont.)  
RT-11 Device Names

Code	Device
RKn:	RK05 Disk
TT:	Console Terminal Keyboard/Printer
SYn:	System device (the volume from which the monitor was bootstrapped)
DK:	The default storage volume

If you do not specify any of the elements of the file specification, BASIC uses a default value.

The default device is DK:. The default for the file name and file type depends on the statement or command in which the file specification appears. Table 2-2 shows the file name defaults, and Table 2-3 shows the file type defaults.

Table 2-2  
Default File Names

Statement or Command	Default
SAVE, REPLACE, COMPILE	the current program name
OLD, APPEND, CHAIN OVERLAY	the file name NONAME
UNSAVE, OPEN, KILL NAME	no default but prints the ?ILLEGAL FILE SPECIFICATION (?IFS) error message instead.

Table 2-3  
Default File Types

Statement or Command	Single precision BASIC Default	Double precision BASIC Default
OPEN, KILL, NAME	.DAT	.DAT
SAVE, REPLACE, UNSAVE APPEND	.BAS	.BAS
COMPILE	.BAC	.BAX
RUN, OLD	.BAC (and if a .BAC cannot be found .BAS)	.BAX (and if a .BAX cannot be found .BAS)

## FILES

When you create a file whose file specification is the same as an existing file, the older file will be deleted (superseded) when the new file is closed. You can avoid unwanted deletions by using the SAVE command to save new files. If a SAVE command specifies a file name that already exists, BASIC-11 prints the following error message:

```
?USE REPLACE (TRPL)
```

This gives you an opportunity to decide whether you want to supersede the old file, or store the file under a different file specification.

### 2.2 THE OPEN STATEMENT - SYSTEM DEPENDENT FEATURES

The format of the OPEN statement is:

```
OPEN string [ { FOR INPUT } ] AS FILE [#] expr1 [DOUBLE BUF]
            [,RECORDSIZE expr2] [,MODE expr3] [,FILESIZE expr4]
```

where:

string	is a file specification as described in Section 2.1.
expr1	is the channel number of the file. It can have any value between 1 and 12.
DOUBLE BUF	causes the file to be double buffered. Double buffering increases the speed of some file operations but requires additional memory for the second buffer.
RECORDSIZE expr2	is ignored if specified.
MODE expr3	is ignored if specified.
FILESIZE expr4	if positive, specifies the maximum number of 256-word blocks the file can occupy. If FILESIZE is missing or expr4 equals 0, it requests the standard BASIC-11/RT-11 file allocation (that is, either half the largest free area or all of the second-largest free area, whichever is larger). If expr4 equals -1, it requests the absolute largest free area. If expr4 is less than -1, the error message ?ILLEGAL FILE LENGTH appears.

The elements of the OPEN statement described above are the system dependent elements. The other elements of the OPEN statement are described in the BASIC-11 Language Reference Manual.

### 2.3 LISTING YOUR FILE DIRECTORY

You must return control to the RT-11 monitor before listing your file directory. First save your current BASIC program (if you wish to reuse it later) and then enter the BYE command. The monitor prints the dot prompt. For example:

## FILES

SAVE TEMP

READY

BYE

Following the prompt, type the RT-11 DIRECTORY command. A simplified format of the RT-11 directory command (see the RT-11 System User's Guide for a complete description) is:

```
DIRECTORY [ /PRINTER ] file specification
```

where:

/PRINTER specifies that the directory is to be printed on the line printer. (If omitted, the directory is printed on the terminal.)

file specification specifies the files that you want listed. If you omit the file specification, all files are listed.

The DIRECTORY command wildcard feature allows you to specify files with similar file names, or similar file types, or both. If you substitute an asterisk for the file name but specify a file type, all files with that file type are listed. For example, the following command lists all BASIC source programs on the line printer:

```
.DIRECTORY/PRINTER *.BAS
```

Similarly, if you substitute an asterisk for the file type, but specify a file name, all files with that file name are listed, regardless of file type. For example, the following command lists all files with the file name TEST:

```
.DIRECTORY/PRINTER TEST.*
```

If you specify a percent sign in place of any characters in a file name or file type (for example, TEST%.BAS), then all the files whose specifiers match the other characters in the specification are listed (TESTAB.BAS, TEST01.BAS, and TESTER.BAS would be listed, if they exist, for the specification TEST%.BAS).

To list all the BASIC programs and compiled BASIC programs, type:

```
.DIRECTORY *.BAZ
```

Note that this command also lists files with the file type .BAK and .BAT. Because the specification /PRINTER is absent, listing occurs on the terminal.

After listing your directory, you can return to BASIC by using the BASIC command, then restore your saved program with the OLD command, and finally, delete the temporary file. For example:

```
.BASIC
BASIC-11/RT-11 V02-xx
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)? A
```

```
READY
OLD TEMP
```

```
READY
UNSAVE TEMP
```

```
READY
```

CHAPTER 3  
UTILITY FUNCTIONS

**3.1 BASIC UTILITY FUNCTIONS**

BASIC has utility functions to:

- Change the terminal width (TTYSET)
- Cancel the effect of CTRL/O (RCTRL/O)
- Disable CTRL/C (CTRLC and RCTRLC)
- Terminate your program (ABORT)
- Input a single character from your terminal (SYS)
- Terminate BASIC (SYS)
- Check if a CTRL/C has been typed (SYS)
- Enable lower case support (SYS)

In the following sections, BASIC-11 utility functions are shown in the context of a LET statement with a dummy target variable, as follows:

```
[LET]variable = utility function
```

where:

variable                    is the target variable.

utility function        is one of the functions described in this chapter.

Actually, utility functions can appear in any arithmetic expression. The LET statement format is recommended because it is the simplest statement, and consequently, produces easier-to-read programs.

**3.2 SETTING THE TERMINAL MARGIN (TTYSET FUNCTION)**

Use the TTYSET function to set your terminal's right margin. BASIC prints on a line until a number or string would extend past the margin you set. BASIC then prints a return and line feed on the current line and prints the string or number on the next line.

The format of the TTYSET function is:

```
[LET] variable=TTYSET(255%,expression)
```

## UTILITY FUNCTIONS

where:

variable	is the target variable and contains an undefined value after the statement is executed.
255%	is either a numeric constant (as specified in format) or an expression with an integer value of 255 (for compatibility with other versions of BASIC).
expression	specifies the right margin of the terminal. The margin is set to the value of the expression minus 1. If the expression equals 0, BASIC does not change the previous margin.

For example, to set BASIC to print to the full width of an LA36 DECwriter II (132 columns), type:

```
A=TTYSET(255%,133%)
```

To set BASIC to print to the full width of a VT50 display terminal (80 columns), type:

```
A=TTYSET(225%,81%)
```

If you do not specify the TTYSET function, BASIC assumes a terminal with 72 columns.

Ensure that the system's margin for your terminal is equal to or greater than the margin you specify in TTYSET.

If the value of the expression is less than 0, equal to 1, or greater than 256, BASIC prints the ?ARGUMENT ERROR (?ARG) message. If the first argument has a value other than 255, BASIC prints the same message.

### 3.3 CANCELING THE EFFECT OF CTRL/O (RCTRL0 FUNCTION)

BASIC stops terminal output when the CTRL/O key is typed; however, the RCTRL0 function causes BASIC to resume printing. Use the RCTRL0 function to ensure that certain data is printed on the terminal even if a CTRL/O has been typed.

The format of the function is:

```
[LET] variable=RCTRL0
```

where:

variable	is the target variable and contains an undefined value after the statement is executed.
----------	---

Consider the following example:

## UTILITY FUNCTIONS

```
LISTNH
10 REM PROGRAM TO INPUT DATA
20 REM FROM FILE AND PRINT SUM
30 OPEN "NUMBR" FOR INPUT AS FILE #1
40 PRINT "DATA IN FILE:"
50 IF END #1 THEN 100
60 INPUT #1,D
70 PRINT D
80 T=T+D
90 GO TO 50
100 A=RCTRL0
110 PRINT
120 PRINT "SUM=";T
```

```
READY
RUNNH
```

```
4
16
147
26
CTRL/O
SUM= 4172
```

```
READY
```

While BASIC executes the loop from line 50 to line 90 it prints out numbers. If CTRL/O is typed BASIC stops printing. But when BASIC executes line 100, BASIC resumes printing.

### 3.4 DISABLING CTRL/C (RCTRLC AND CTRLC FUNCTIONS)

In certain parts of the program you may need to override CTRL/C interrupts from the terminal. The RCTRLC function disables CTRL/C and prevents it from stopping the BASIC program. The CTRLC function enables the CTRL/C key command.

The format of the functions are:

```
[LET] variable=RCTRLC
[LET] variable=CTRLC
```

where:

variable is the target variable; it contains an undefined value after the statement is executed.

After BASIC executes the RCTRLC function, typing CTRL/C on the terminal does not stop the program.

After BASIC executes the CTRLC function, typing CTRL/C stops the program. BASIC does not save any CTRL/C that is typed while CTRL/C is disabled. If the program encounters a CTRL/C function, and no prior RCTRLC function is in effect, the CTRL/C function has no effect.

When BASIC prints the READY message, it automatically enables the CTRL/C key command.

For example:

## UTILITY FUNCTIONS

```
LISTNH
1000 REM DO NOT ALLOW INTERRUPTS
1010 A=RCTRLC
1020 PRINT "NO INTERRUPTS"
1030 FOR I= 1 TO 1000 \ S=S+I \ NEXT I
1100 REM NOW ALLOW INTERRUPTS
1110 A=CTRLC
1120 PRINT "INTERRUPTS OKAY"
1130 FOR I = 1 TO 1000 \ S=S+I \ NEXT I
32767 END
```

```
READY
RUNNH
```

```
NO INTERRUPTS
```

```
CTRL/C
```

```
INTERRUPTS OKAY
```

```
CTRL/C
```

```
STOP AT LINE 1130
```

```
READY
```

For information on a system function that determines if CTRL/C has been typed while CTRL/C is disabled, see Section 3.6.3.

### NOTE

Once CTRL/C is disabled it is not possible to interrupt BASIC. Do not disable CTRL/C until your program is debugged.

### 3.5 TERMINATING YOUR PROGRAM (ABORT FUNCTION)

If you want a program to delete itself from memory when it terminates, use the ABORT function. The ABORT function is equivalent to an END statement except that ABORT can optionally delete your program from memory and change the program name to NONAME (equivalent to the SCR command).

The format of the ABORT function is:

```
[LET] variable=ABORT(expression)
```

where:

variable	is the target variable; it contains an undefined value after the statement is executed.
expression	determines if the program is to be deleted from memory. If expression equals 0, BASIC does not delete the program. If expression equals 1, BASIC deletes the program.

Consider the following examples:

## UTILITY FUNCTIONS

Delete from memory when program completed	Do not delete when program completed
LIST	LIST
ABORT 21-JUN-77 14:52:45	ABORT 21-JUN-77 14:54:00
10 PRINT "123"	10 PRINT "123"
20 A=ABORT(1)	20 A=ABORT(0)
30 PRINT "456"	30 PRINT "456"
READY	READY
RUNNH	RUNNH
123	123
READY	READY
LIST	LIST
NONAME 21-JUN-76 14:53:30	ABORT 21-JUN-76 14:54:30
READY	10 PRINT "123"
	20 A=ABORT(0)
	30 PRINT "456"
	READY

### 3.6 SYSTEM FUNCTIONS

System functions perform system-dependent operations.

The formats of the system functions are:

```
[LET] variable= SYS(expression1 ,expression2 )
```

where:

variable is the target variable.

expression1 determines the function to be performed.

expression2 is an optional argument used in some system functions.

Table 3-1 summarizes the functions performed according to the specified value of expression1. Any value of expression1 other than those specified causes BASIC to print the ?ARGUMENT ERROR (?ARG) message.



## UTILITY FUNCTIONS

Table 3-1  
Summary of System Functions

Value of expression1	Function Performed
1	Processes input one character at a time. Target variable contains the ASCII value of the next character typed at the terminal.
4	Terminates BASIC and returns control to system monitor (equivalent to the BYE command).
6	Determines if CTRL/C has been typed while CTRL/C is disabled by RCTRLC function. Target variable equals 1 if CTRL/C has been typed and equals 0 if CTRL/C has not been typed.
7	Enables or disables lower case input from your terminal. If expression2 equals 0, lower case character input is allowed. If expression2 equals 1, lower case character input is converted to the equivalent upper case character input.

### 3.6.1 Single Character Input

Use the single character input system function, SYS(1), to process input one character at a time.

SYS(1) returns the seven-bit ASCII value of any character typed on the terminal except CTRL/C. (See the BASIC-11 Language Reference Manual for a list of the ASCII values.) If CTRL/C is typed when BASIC is executing SYS(1) and CTRL/C is enabled, then BASIC prints the STOP and READY messages. If CTRL/C is disabled, then BASIC continues executing SYS(1) and waits for another character. BASIC cannot process the character until you type the RETURN key.

```
LISTNH
10 PRINT "TYPE A CHARACTER: ";
20 A=SYS(1)
40 PRINT "THE ASCII VALUE OF ";CHR$(A);" IS";A

READY
RUNNH
TYPE A CHARACTER: Z
THE ASCII VALUE OF Z IS 90

READY
```

### 3.6.2 Terminating BASIC

To terminate BASIC from a BASIC program, use system function SYS(4). It is equivalent in effect to the BYE Command.

For example:

## UTILITY FUNCTIONS

```
LISTNH
10 PRINT "GOODBYE"
20 A=SYS(4)
```

```
READY
RUNNH
GOODBYE
.
```

### 3.6.3 Checking for CTRL/C

If you have disabled CTRL/C with the RCTRLC function and want to check if CTRL/C has been typed, use system function SYS(6). The function returns a 1 if CTRL/C has been typed and a 0 if it has not been typed.

For example:

```
LISTNH
10 A=RCTRLC \ REM Disable CTRL/C.
30 B=SYS(6) \ REM Check for CTRL/C.
40 IF B=1 THEN 100
50 PRINT "STILL EXECUTING"
60 GO TO 30
100 PRINT "PROGRAM TERMINATING"
110 A=CTRLC REM Reenable CTRL/C.
120 A=ABORT(1)
```

```
READY
RUNNH
```

```
STILL EXECUTING
STILL EXECUTING
CTRL/C CTRL/C
STILL EXECUTING
PROGRAM TERMINATING
```

```
READY
```

### 3.6.4 Enabling Lower Case Support

If you want to enter lower case characters at your terminal, use the system function SYS(7,expr2). The RT-11 system usually converts all lower case alphabetic characters to upper case. Executing the function SYS(7,0) causes RT-11 to stop converting lower case characters and to pass them unchanged. To cause RT-11 to resume converting lower case characters, you must execute the function SYS(7,1). After you exit from BASIC, the monitor continues to process characters as it did before BASIC was active.

The following example demonstrates how to enable and disable lower case. The program is first run to enable lower case by causing the function SYS(7%,0%) to be executed. After this the program is modified to allow the user to enter a lower case response. Finally, the modified form of the program is run; this disables lower case. The modified program is then saved.

## UTILITY FUNCTIONS

```
LISTNH
10 REM PROGRAM TO CHANGE LOWER CASE CONVERSION
20 PRINT "DO YOU WANT TO ENTER LOWER CASE CHARACTERS (Y OR N)";
30 INPUT A$
40 IF A$="Y" THEN 100
50 IF A$<>"N" THEN 20
60 A=SYS(7Z,1Z) \ REM DISABLE LOWER CASE
70 GO TO 32767
100 A=SYS(7Z,0Z) \ REM ENABLE LOWER CASE
32767 END
```

READY  
RUNNH

DO YOU WANT TO ENTER LOWER CASE CHARACTERS (Y OR N)? Y

```
READY
45 if a$="y" then 100 \ rem Check for lower case y
sub 50 @20@if a$<>"n" then 20 \ Rem Check for lower case n
50 IF A$<>"N" THEN if a$<>"n" then 20 \ Rem Check for lower case n
```

```
READY
listnh
10 REM PROGRAM TO CHANGE LOWER CASE CONVERSION
20 PRINT "DO YOU WANT TO ENTER LOWER CASE CHARACTERS (Y OR N)";
30 INPUT A$
40 IF A$="Y" THEN 100
45 IF A$="y" THEN 100 \ REM Check for lower case y
50 IF A$<>"N" THEN IF A$<>"n" THEN 20 \ REM Check for lower case n
60 A=SYS(7Z,1Z) \ REM DISABLE LOWER CASE
70 GO TO 32767
100 A=SYS(7Z,0Z) \ REM ENABLE LOWER CASE
32767 END
```

READY  
runnh

DO YOU WANT TO ENTER LOWER CASE CHARACTERS (Y OR N)? n

READY  
SAVE LOWCHM

READY

If you type lower case letters when lower case is disabled, they are echoed as upper case.

Note that BASIC converts lower case keywords and variable names to upper case characters but leaves string constants, strings entered at the terminal, and remarks unchanged.

## CHAPTER 4

### USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

#### 4.1 INTRODUCTION TO ASSEMBLY LANGUAGE ROUTINES

BASIC-11 allows you to add assembly language routines (ALRs) to expand or extend BASIC's capabilities. For example, you can write routines for communication with special devices (such as laboratory equipment) or to manipulate arrays. Once added to BASIC, such routines can be executed in immediate mode or in programs, by means of the CALL statement. (See the BASIC-11 Language Reference Manual.) applications programs. There are several advantages to doing this rather than doing all your programming in assembly language. They are:

- Only the programmer writing the routine need know assembly language. The application programmers need only to know BASIC.
- It is easier to write, debug, and modify BASIC programs than assembly language programs. You can write, execute, debug, and modify your program without leaving BASIC.
- You can execute ALRs without writing a program, using immediate mode CALL statements.

#### NOTE

This chapter assumes that you are an experienced MACRO-11 programmer and that you are familiar with your operating system and its utility programs (editors, MACRO assemblers, task builders, linkers, etc.)

This chapter describes:

- ALR format.
- The procedure to access arguments.
- Use of auxiliary routines provided by BASIC.

See the BASIC-11/RT-11 Installation Guide for the procedure to add the routines to BASIC.

ALRs that use the FORTRAN IV call interface (as defined in RT-11 FORTRAN IV User's Guide) can be called from either FORTRAN IV or RT-11 BASIC. However, these ALRs must not access any routines or global locations in FORTRAN IV itself.



USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

NOTE

You should edit the items printed in red in this listing into the file BSCLI.MAC. The items printed in black are already in the file.

The locations, INITNM, ADDNM, and CHKNM should be at the beginning of the INITIT, ADDER, and CHCKST, respectively. For example:

```

; THE INIT ROUTINE
.GLOBAL INITNM
INITNM: .BYTE 6 ;NUMBER OF CHARACTERS IN NAME
        .ASCII "INITIT"
        .EVEN
        .WORD INITST
INITST: ;START OF ROUTINE
        *
        *
        *
    
```

An alternative method is to add the routine name and starting address after the routine name table. In this case the starting addresses of the routines should be globals. Using the same examples as above, the routine name table should be:

```

        .GLOBAL FTABI
        .GLOBAL INITST, ADDST, CHKSST
FTABI:  .WORD FTBL
FTBL:   .WORD INITNM
        .WORD ADDNM
        .WORD CHKSNM
INITNM: .WORD 0
        .BYTE 6 ;NUMBER OF CHARACTERS IN NAME
        .ASCII "INITIT"
        .EVEN
        .WORD INITST
ADDNM   .BYTE 5
        .ASCII "ADDER"
        .EVEN
        .WORD ADDST
CHKSNM: .BYTE 6
        .ASCII "CHKSTA"
        .EVEN
        .WORD CHKSST
        *
        *
        *
        .END
    
```

Each ALR should start with the global address specified. For example:

```

; THE INITIT ROUTINE
.GLOBAL INITST
INITST: ;START OF ROUTINE
        *
        *
        *
    
```

You should use this alternative method when you are adding an ALR written for FORTRAN IV to BASIC.

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

All the examples in this chapter use the recommended method (where the routine name packet is at the start of the routine).

Once you have defined the name and starting address of the routine, you can write the routine itself. The ALR can use the stack but it must ensure that the stack limit is not exceeded. BASIC puts the stack limit in R4 before transferring control to the ALR. If you use any of the mathematical operations or function routines provided by BASIC, ensure that there is enough free space on the stack before executing the routine (15 free words for single precision routines and 30 free words for double precision routines). The ALR must end with an RTS PC instruction with the stack unchanged from its original state. The format of the INITIT routine is:

```
      ; THE INIT ROUTINE
      .GLOBL      INITNM
INITNM: BYTE      6
      .ASCII     "INITIT"
      .EVEN
      .WORD      INITST
INITST:          ;START OF ROUTINE
      ;
      ;
      ;
      ; MAIN BODY OF ROUTINE
      ;
      ;
      ;
      RTS PC      ;END OF ROUTINE
```

### 4.3 ACCESSING THE ARGUMENTS - THE ARGUMENT LISTS

When BASIC executes the CALL statement, it evaluates the arguments and provides the routine with two lists. One contains pointers to the evaluated arguments and the other contains descriptors of the argument types. An assembly language routine (ALR) should ensure that the list contains the expected number and the right type of arguments.

Argument checking ensures that errors in a BASIC program will not cause a fatal error in the ALR or in BASIC itself. If no argument checking is done and a CALL statement contains an incorrect data type, the ALR produces unpredictable results. For example, if the ALR expects an integer array and the CALL statement contains a string expression, the ALR could overwrite sections of the stack. If the ALR checks arguments for errors, it can protect itself from errors in BASIC programs. (There is no protection from errors in the ALR itself.)

A FORTRAN-compatible ALR cannot check arguments unless it first checks and determines that the language calling it is BASIC, because FORTRAN does not provide an argument descriptor list.

Before BASIC transfers control to the ALR, it evaluates the arguments in the CALL statement. It creates a list of pointers to the arguments and a list of argument descriptors. Figure 4-2 shows the argument descriptor lists that BASIC creates before it transfers control to the ALR.

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

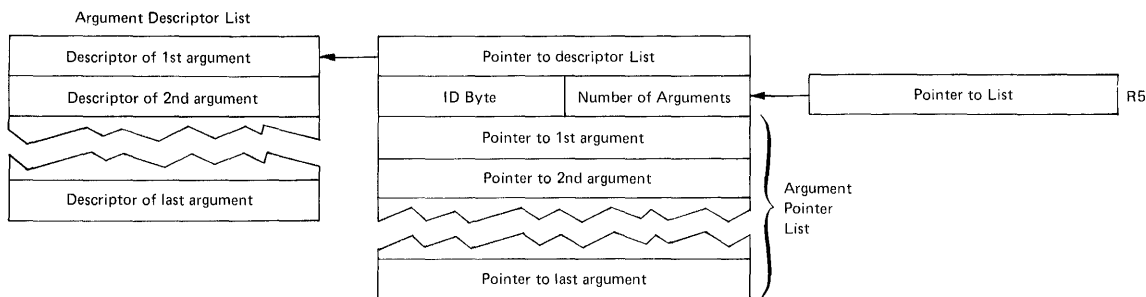


Figure 4-2 Assembly Language Routine Argument Lists

As shown in Figure 4-2, R5 points to a word that specifies the number of arguments in the CALL statement and identifies the language calling the ALR. The argument pointer list starts at the next word and the pointer to the argument descriptor list is stored in the previous word.

Each byte of the word pointed to by R5 is meaningful. The low-order byte contains the number of arguments. The high order byte identifies the language. If the calling language is BASIC, the high order byte has a value of 202. If the calling language is FORTRAN IV, the high-order byte has a value of 0.

The pointers in the argument pointer list specify the location of the evaluated arguments. There are two exceptions, pointers for null arguments and pointers for string array arguments.

If an argument is null then its pointer does not point to that argument but instead contains a value of -1. A CALL statement argument list with two adjacent commas or a terminating command produces a null argument. For example, CALL "INITIT" (A, B,, D,) produces the following arguments: A, B, null, D, and null.

If the argument is a string array, then the pointer does not point to that argument but instead contains a value needed to access the string array. (See Section 4.3.2.) If the argument is an unsubscripted string or an element of a string array, the pointer specifies the location of the first character of the string.

The argument descriptor list specifies the data type of each argument. It also indicates whether the argument is an array or not and whether the ALR can return a result in the argument.

BASIC provides additional information for strings and arrays. In these cases the word in the argument descriptor list is a pointer to the descriptor word, which has the additional information after it. Figure 4-3 describes the format of the descriptor word. BASIC indicates if a word in the list is a pointer or a descriptor word by the value of the 0 bit. If the 0 bit is clear, then the word in the descriptor list is a pointer. If the 0 bit is set, then the word in the descriptor list is the descriptor word. Note that the descriptor word for strings and arrays has a value of 0 in the 0 bit.

### NOTE

All numbers in this chapter that specify the contents of a word or a section of a word are octal numbers not decimal numbers.



## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

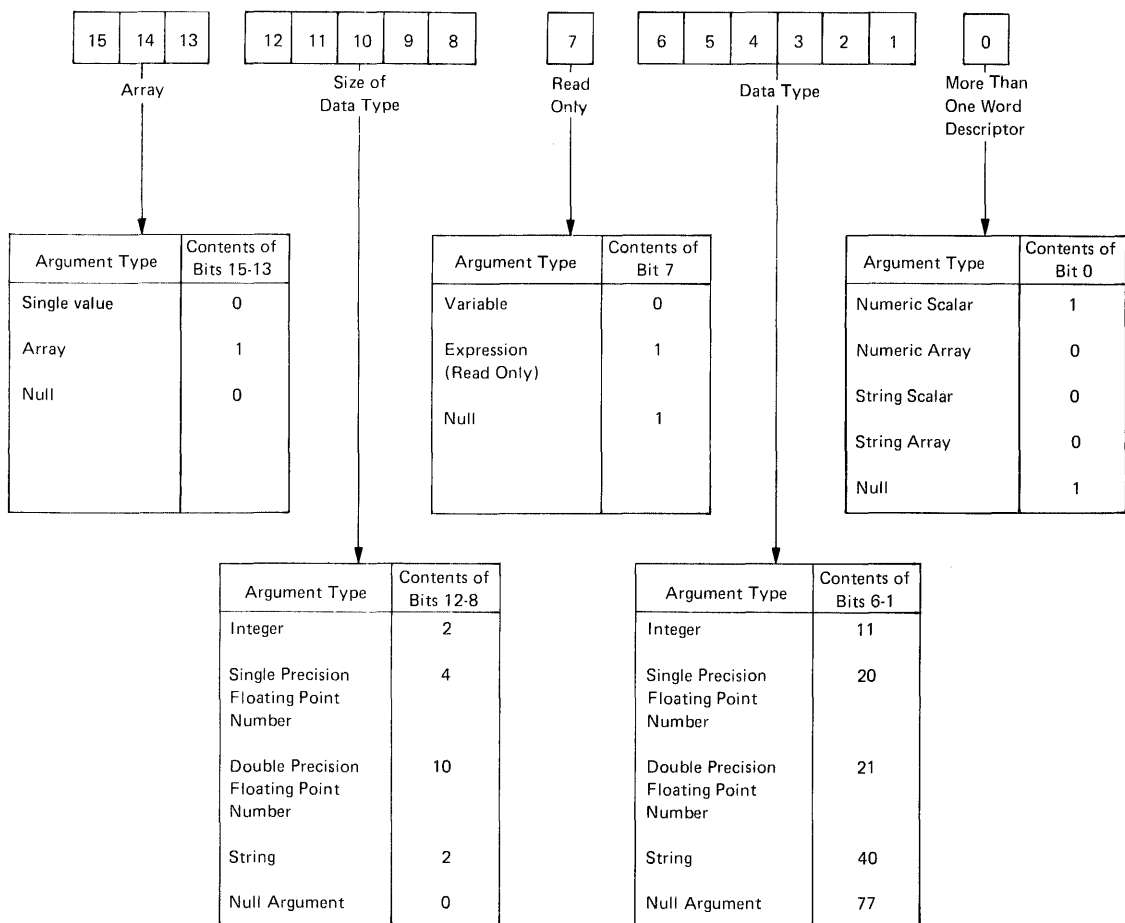


Figure 4-3 Format of the Argument Descriptor Word

The ALR can return arguments only to variables and arrays. If the argument is an expression, constant, or element of a virtual array, the seventh bit of the argument descriptor word is set and the ALR must not return a value to that argument.

Bits 12 through 8 of the argument descriptor word specify the size of the data type. The ALR does not need to check this information because each argument type (specified in bits 6 through 1) has a fixed size. The contents of bits 12 through 8 for a string argument can be ignored.

BASIC provides additional information for array and string arguments. BASIC specifies the total number of bytes in the array, the number of subscripts, the high limit of the first subscript, and the high limit of the second subscript (if there are two subscripts). BASIC also provides a string reference pointer for string arguments. This pointer is used by routines provided by BASIC to access the string arguments. See Section 4.3.2 for a description of these routines. Figure 4-4 describes the format of array and string descriptors.

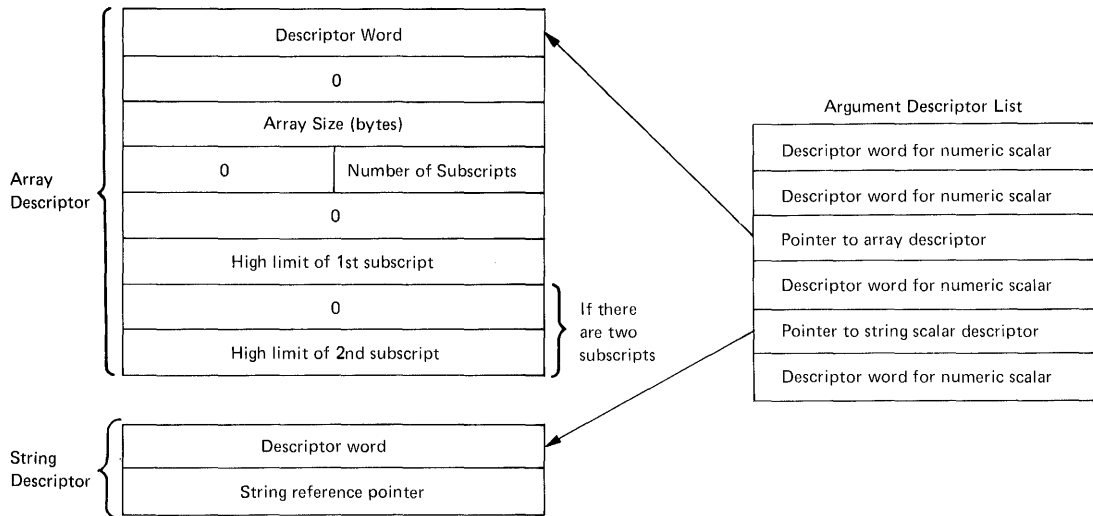


Figure 4-4 Format of Array and String Argument Descriptors

#### 4.3.1 Numeric Arrays

If the CALL statement specifies an element of a numeric array, for example A(10), BASIC considers it a 1-dimensional array starting with the specified element and ending with the last element of the array. BASIC considers it a one-dimensional array even if the entire array is two-dimensional.

BASIC and FORTRAN IV store arrays differently. BASIC array subscripts start at 0, but FORTRAN array subscripts start at 1. In BASIC arrays, the second subscript varies faster, but in FORTRAN IV arrays the first subscript varies faster. If you are designing a routine to be called from either BASIC or FORTRAN IV, you must consider these differences in the ALR.

#### 4.3.2 Strings and String Arrays

This section describes the routines BASIC provides to allow the assembly language routine (ALR) to access strings. It also describes some example routines which use these string access routines. BASIC allows dynamic-length strings, whose length can change during program execution. The BASIC string access routines keep track of the location and size of strings. Consequently, an ALR cannot change a BASIC string without using the string access routines.

The procedures for accessing strings and for accessing elements of string arrays are different. Note that if the CALL statement specifies an element of a string array (for example, A\$(10)), BASIC considers it a string scalar. Only if the entire array is passed (for example, A\$()), does BASIC consider it a string array.

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

The ALR must locate and retrieve the string reference pointer word and pass it to the string access routines. For a string argument, the string reference pointer is the word following the descriptor word. For a string array argument, The ALR must calculate the string reference pointer to access any element of the array. The string reference pointer is a word whose value is determined by the following formula:

$$\text{string reference pointer} = 2 * \text{offset} + \text{argument pointer}$$

where:    offset                    is the position of the element in the array.

          argument pointer        is the value for the string array in the list of argument pointers. (Note the argument pointer for a string array does not point to the argument itself.)

The offset for an element of a one-dimensional array is equal to the value of its subscript. The offset for an element of a two-dimensional array is defined by this formula:

$$\text{offset} = \text{subscript1} * (\text{maximum value of subscript2} + 1) + \text{subscript2}$$

For example, consider two arrays A\$(10) and B\$(3,5) with argument pointers of A and B respectively. (NOTE: All numbers in the following list are decimal.)

Element	2*offset+argument pointer	string reference pointer
A\$(0)	2*0+A	A
A\$(4)	2*4+A	8+A
B\$(0,5)	2*(0*6+5)+B	10+B
B\$(1,5)	2*(1*6+5)+B	22+B
B\$(2,0)	2*(2*6+0)+B	24+B

The string access routines use the string reference pointer that the ALR provides to find and manipulate the string.

BASIC provides four string access routines:

```
$FIND
$ALC
$STORE
$DEALC
```

The \$FIND routine returns the length of a string and a pointer to the first character. The \$ALC routine allocates a temporary string. An ALR can only write characters directly to strings created by \$ALC. The \$STORE routine assigns the value of one string to a second string and changes the first string to a null string. The \$DEALC routine deallocates space used by the temporary string on the stack.

The ALR should use the following general procedure to manipulate a string argument and then return the resultant string. First, the ALR accesses the string argument by using the \$FIND routine. Then it creates a temporary string with the \$ALC routine. It then reads the characters of the string argument, manipulates them in the desired way, and writes the characters out to the temporary string. After this the ALR uses the \$STORE routine to copy the temporary string to a string argument (which can be the original string). Finally, it uses the \$DEALC routine to remove data placed on the stack by the \$ALC routine.

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

Table 4-1, "Using String Access Routines", describes the four string access routines. It describes the initial setup, including the format of the subroutine jump (JSR) instruction required to invoke the string access routine. It also describes the expected results and how to interpret them. (In particular, it indicates how to determine whether or not you made a correct initial setup in preparation for the string access routine.)

If the ALR calls \$FIND, \$ALC, \$STORE, and \$DEALC, it must specify them as global locations.

Before calling any of these routines, you must ensure that R5 contains its initial value, the value it had when BASIC transferred control to the ALR. That is, R5 must point to the word identifying BASIC and specifying the number of arguments.

### NOTE

These routines require that a register contain the same value in bits 6-1 as an argument descriptor word for a string argument. You can ensure this by moving a value of 100 into the specified register (puts a value of 40 in bits 6-1) or by moving an argument descriptor word in the specified register.

## 4.4 USING ROUTINES PROVIDED BY BASIC

BASIC provides routines that handle error conditions, print messages on the terminal and perform mathematical operations and functions.

### 4.4.1 Error Handling and Message Routines

BASIC provides two error handling routines (\$ARGER and \$BOMB) and two message printing routines (\$MSG and \$CHROT). The \$ARGER routine produces the fatal ?ARGUMENT ERROR (?ARG) message. The ALR should call \$ARGER when it detects an incorrect argument. The \$BOMB routine allows the ALR to specify its own fatal message. The \$MSG routine prints any message on the terminal and then returns control to the ALR. The \$CHROT routine prints any single character on the terminal and then returns control to the ALR.

If the ALR calls \$ARGER, \$BOMB, \$MSG, or \$CHROT, it must specify them as global locations.

Call the \$ARGER routine by executing the instruction:

```
JMP $ARGER
```

The \$ARGER routine prints the error message on the terminal in one of the following formats:

```
?ARGUMENT ERROR AT LINE xxxxx  
?ARG AT LINE xxxxx
```

where:

xxxxx is the line number of the CALL statement.

Table 4-1  
Using String Access Routine

Routine	Program Setup	Result With No Errors Detected	Result With Errors Detected
\$FIND (return location and length of string)	R0+string reference pointer R1+100 R5+initial value Execute: JSR PC, \$FIND	R0 = address of first string character R1 = length of string R2 = 100 R3,R4,R5 unchanged C-bit = 0 (char) Z-bit = 1 if a null string (R1=0)	R0 contains error code: if R0=1, R1 did not equal 100 if R0=2, R5 did not contain correct initial value  R3,R4,R5 unchanged C-bit = 1
\$ALC (allocate temporary string)*	R0+required string length R1+100 R5+initial value Execute: JSR PC, \$ALC	R0 = address of first string character R1 = length of string R2 = 100 R3,R4,R5 unchanged C-bit = 0 Z-bit = 1 if a null string (R1=0) SP = string reference pointer stack contains several words of internal pointers. Remove these words from the stack by the \$DEALC routine	R0 contains error code: if R0=0, indicates insufficient free space for requested string if R0=1, R1 did not equal 100 if R0=2, R5 did not contain correct initial value  R3,R4,R5 unchanged C-bit = 1
\$STORE (store value of a string in a second string, make first string null)	R0+string reference pointer of string to be copied R1+string reference pointer of receiving string R2+100 R5+initial value Execute: JSR PC, \$STORE	R0,R1,R2,R3,R4,R5 unchanged C-bit = 0 string whose pointer was in R0 is null string whose pointer was in R1 contains former value of the other string	R0 contains error code: if R0=1, R2 did not equal 100 if R0=2, R5 did not contain correct initial value  R1,R2,R3,R4,R5 unchanged C-bit = 1
\$DEALC (remove from stack the internal pointers produced by \$ALC routine)*	Return stack to the state that it was immediately following \$ALC routine. Do this by removing any words you have added to the stack since calling the \$ALC routine; this ensures that the string reference pointer is in the SP. R2+100 R5+initial value Execute: JSR PC, \$DEALC	R0,R1,R2,R3,R4,R5 unchanged C-bit = 0 Stack returned to the state that existed before \$ALC was called	R0 contains error code: if R0=1, R2 did not equal 100 if R0=2, R5 did not contain correct initial value  R1,R2,R3,R4,R5 unchanged C-bit = 1 Stack

\*Any temporary string created by \$ALC must be removed by \$DEALC before the ALR ends.

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

If the CALL statement was an immediate mode statement, then AT LINE xxxxx is not printed. Control then returns to BASIC, which prints the READY message.

Call the \$BOMB routine by executing the following instruction:

```
JSR      R1,$BOMB
.ASCIIZ  'message'
.EVEN
```

where:

message is the string of characters that you wish to print.

The \$BOMB routine prints the error message on the terminal in the form:

```
?error message AT LINE xxxxx
```

where:

xxxxx is the line number of the CALL statement.

If the CALL statement was an immediate mode statement, then AT LINE xxxxx is not printed. Control then returns to BASIC, which prints the READY message.

Call the \$MSG routine by executing the instruction:

```
JSR R1,$MSG
.ASCII 'message'
.BYTE 15,12,0           ;MUST HAVE CARRIAGE RETURN
.EVEN                  ;AND LINE FEED AND END WITH 0
```

where:

message is the string of characters that you wish to print.

The \$MSG routine prints the message you specify on the terminal, and then returns control to the instruction that follows the .EVEN instruction.

Call the \$CHROT routine as follows:

1. put the 8-bit ASCII code of the character in the low order byte of R0
2. execute the instruction:

```
JSR PC,$CHROT
```

\$CHROT prints the character specified in R0 on the terminal, and then returns control to the ALR.

### 4.4.2 Mathematical Operation and Function Routines

Assembly language routines (ALRs) can use BASIC's mathematical operation and function routine to perform operations and functions that you can use in a BASIC program. ALRs can use the same routine that BASIC itself uses to perform these operations and functions. An advantage of this is that the ALR need not duplicate routines that already exist in BASIC.

NOTE

Assembly language routines that use the FPl1 Floating Point unit are required to save and restore the FPU status. If the assembly language routine will modify the FPU status, it must preserve the FPU status on entry by executing the following instruction:

STFPS        -(SP)

and restore the status (prior to returning to the calling program) by executing the instruction:

LDFPS        (SP)+

Tables 4-2 and 4-3 describe the BASIC mathematical operations and functions. They show how each operation or function would appear in the BASIC language, and name the BASIC-provided routine that will perform it. Note that certain operations and functions require one routine for single precision arithmetic, a different routine for double precision arithmetic, and yet another for integer arithmetic.

If you are running a BASIC system designed for double precision arithmetic, either the single or double precision routine names can be used. Either routine name will execute the double-precision routine; this fact allows you to use the same code for different systems regardless of precision. However, you must still be aware of which precision you are using, and ensure that the data manipulations in the program properly reflect the BASIC configuration on which programs are running. To be compatible with FORTRAN IV, you must use only the double precision routine names to execute the double precision routines.

All routines that have a dollar sign (\$) in their name must be called in threaded code mode. To call routines in threaded code mode, first call a special subroutine, \$POLSH. After calling \$POLSH, list the names of the threaded code routines you wish to call. In threaded code mode, each routine is executed in the order listed. All arguments and results are passed on the stack. Finally, list the name of a second special subroutine, \$UNPOL, which ends threaded code mode.

You must specify \$POLSH, \$UNPOL and any routine names you specify as globals.

The call to \$POLSH is in the following format:

JSR        R4,\$POLSH

Figure 4-5 describes the state of the stack before and after each threaded code routine.

USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

Table 4-2  
BASIC Mathematical Operations

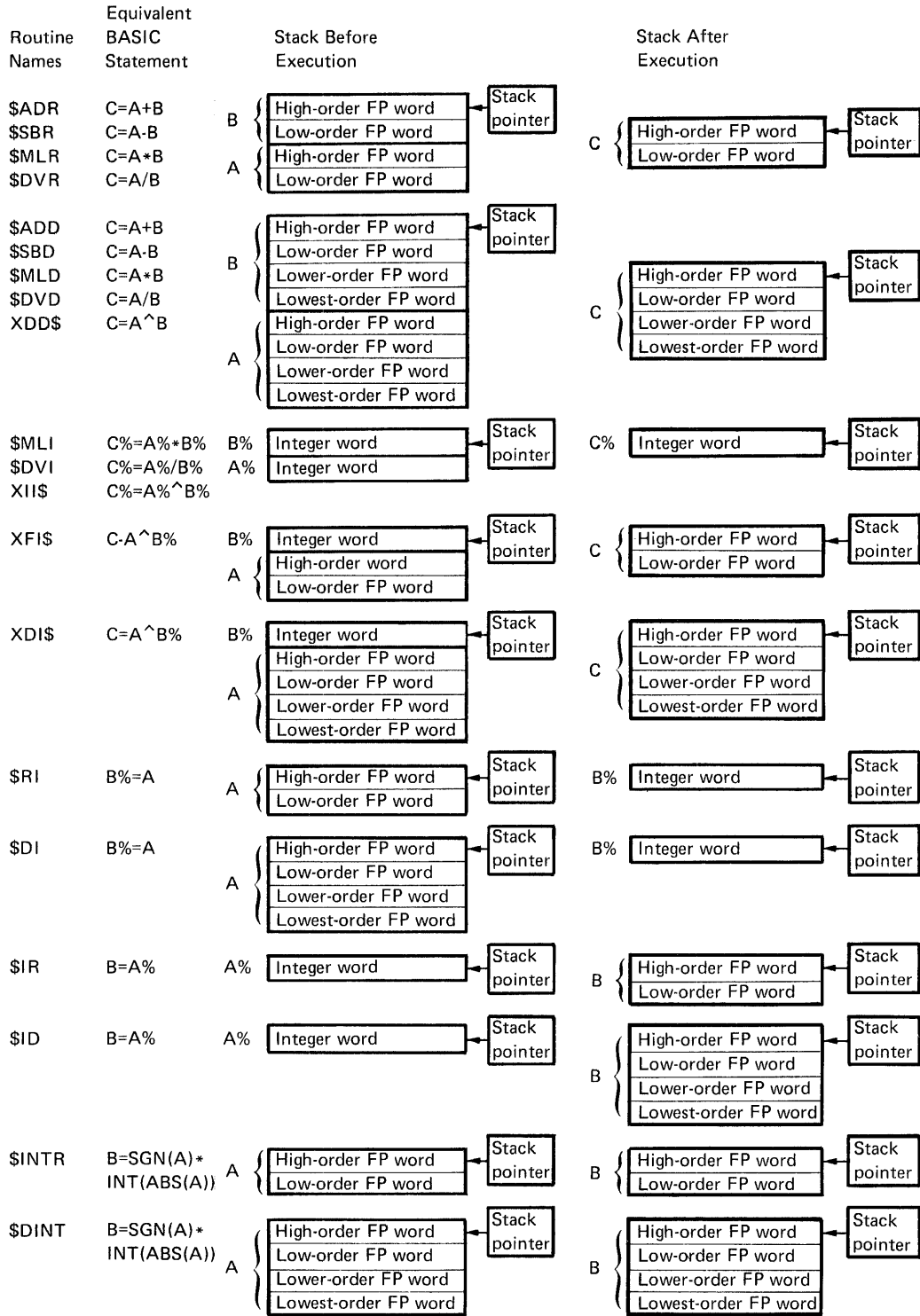
Operation	Operator	Meaning	BASIC Equivalent	Single-Precision Routine	Double Precision Routine
Addition	+	Adds two floating point numbers	$C=A + B$	\$ADR	\$ADD
Subtraction	-	Subtracts one floating point number from another	$C=A - B$	\$SBR	\$SBD
Multiplication	*	Multiplies two floating point numbers	$C=A * B$	\$MLR	\$MLD
		Multiplies two integers	$C\%=A\%*B\%$	\$MLI	\$MLI
Division	/	Divides one floating point number by another	$C=A / B$	\$DVR	\$DVD
		Divides one integer by another integer	$C\%=A\%/B\%$	\$DVI	\$DVI
Exponentiation	^	Raises a floating point number by a floating point exponent.	$C=A ^ B$	XFF\$	XDD\$
		Raises a floating point number by an integer exponent.	$C=A ^ B\%$	XFI\$	XDI\$
		Raises an integer by an integer exponent.	$C\%=A\%^B\%$	XII\$	XII\$

Table 4-3  
BASIC Mathematical Functions

Function	Description	BASIC Equivalent	Single-Precision Routine	Double Precision Routine
Data type conversion	Converts floating point number to integer	$B\% = A$	\$RI	\$DI
	Converts integer to floating	$B = A\%$	\$IR	\$ID
Truncation	Truncates a floating point number to a floating point whole number	$B=SGN(A) * INT(ABS(A))$	\$INTR	\$DINT
Sine	Finds the sine of a radian value	$B=SIN(A)$	SIN	DSIN
Cosine	Finds the cosine of a radian value	$B=COS(A)$	COS	DCOS
	Finds the arctangent in radians of a number	$B=ATN(A)$	ATAN	DATAN
Logarithm	Finds the natural log (base e) of a number	$B=LOG(A)$	ALOG	DLOG
	Finds the common log (base 10) of a number	$B=LOG10(A)$	ALDG10	DLOG10
Square root	Finds the square root of a number	$B=SQR(A)$	SQRT	DSQRT
Exponential	Finds the value of e raised to a number	$B=EXP(A)$	EXP	DEXP



USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC



Note: FP stands for Floating Point

Figure 4-5 State of Stack for Threaded Code Routines

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

As examples, consider the following segments of routines:

Segment 1 divides an integer stored in TEMP1 by an integer stored in TEMP2 and stores the quotient in RESULT.

```

‡ SEGMENT 1

        .GLOBL    $POLSH,$UNPOL,$DVI
        MOV      TEMP1,-(SP)          ‡SET UP THE
        MOV      TEMP2,-(SP)          ‡STACK
        JSR      R4,$POLSH           ‡ENTER THREADED CODE MODE
        .WORD    $DVI                ‡SPECIFY ROUTINE NAME
        .WORD    $UNPOL              ‡LEAVE THREADED CODE MODE
        MOV      (SP)+,RESULT        ‡STORE RESULT
        .
        .
TEMP1:  .WORD    0
TEMP2:  .WORD    0
RESULT: .WORD    0
    
```

Segment 2 multiplies two single-precision floating point numbers, FLOATA and FLOATB, and stores the product in FLOATC.

```

‡SEGMENT 2

        .GLOBL    $POLSH,$UNPOL,$MLR
        MOV      FLOATA+2,-(SP)      ‡PUT FLOATA
        MOV      FLOATA,-(SP)        ‡ON STACK
        MOV      FLOATB+2,-(SP)     ‡PUT FLOATB
        MOV      FLOATB,-(SP)       ‡ON STACK
        JSR      R4,$POLSH           ‡ENTER THREADED CODE MODE
        .WORD    $MLR                ‡SPECIFY ROUTINE NAME
        .WORD    $UNPOL              ‡LEAVE THREADED CODE MODE
        MOV      (SP)+,FLOATC        ‡STORE RESULT
        MOV      (SP)+,FLOATC+2     ‡IN FLOATC
        .
        .
        .
FLOATA: .WORD    0,0
FLOATB: .WORD    0,0
FLOATC: .WORD    0,0
    
```

Segment 3 converts a double-precision floating point number stored at FLOAT to an integer and stores it at INTMDW.

```

‡SEGMENT 3

        .GLOBL    $POLSH,$UNPOL,$DI
        MOV      FLOAT+6,-(SP)      ‡PUT FLOAT
        MOV      FLOAT+4,-(SP)      ‡ON STACK
        MOV      FLOAT+2,-(SP)      ‡KEEP DOING IT
        MOV      FLOAT,-(SP)        ‡DONE
        JSR      R4,$POLSH           ‡ENTER THREADED CODE MODE
        .WORD    $DI                ‡SPECIFY ROUTINE NAME
        .WORD    $UNPOL              ‡LEAVE THREADED CODE MODE
        MOV      (SP)+,INTMDW        ‡STORE RESULT
        .
        .
        .
FLOAT:  .WORD    0,0,0,0
INTMDW: .WORD    0
    
```

Although the foregoing examples have only one routine name after each call to \$POLSH, you can specify any number of routine names. You must always follow the last of routine name with the \$UNPOL routine.

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

The sine, cosine, arctangent, logarithm, square root, and exponential routines each use an argument list similar to the BASIC CALL argument list. An ALR must establish the argument list before calling the routine. The format of the argument list for the single-precision routines, SIN, COS, ATAN, ALOG, ALOG10, SQRT, and EXP, is:

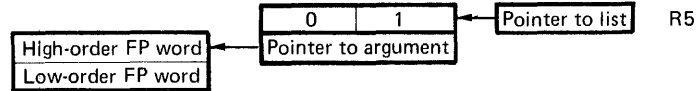


Figure 4-6 Argument List for Supplied Single-Precision Routines

The format of the argument list for the double-precision routines, DSIN, DCOS, DATAN, DLOG, DLOG10, DSQRT, and DEXP is:

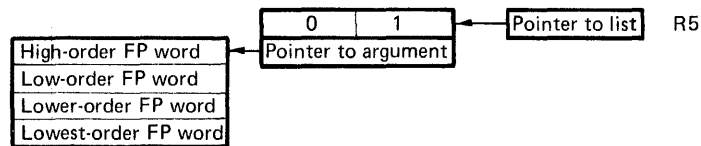


Figure 4-7 Argument List for Supplied Double-Precision Routines

In both cases, the routines are called by the instruction:

```
JSR PC, routine name
```

The single-precision routines return the result in R0 and R1; the high-order word is in R0 and the low order word is in R1.

The double-precision routines return the result in R0, R1, R2, and R3. The high-order word is in R0 and the low, lower, and lowest order words are in R1, R2, and R3, respectively.

You must specify as global any routine name that you call.

These routines do not preserve any registers.

### NOTE

You should save the initial value of R5 before loading the pointer to the argument for these routines. You will need the saved value to execute any threaded code routine to access arguments.

Consider the following segment of a routine that finds the square root of a single-precision floating point number, NUM1, and stores the result in NUM2:

## USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

‡SEGMENT WHICH FINDS SQUARE ROOT

```

        .GLOBL    SQRT
        MOV      R5, TEMP5           ‡SAVE OLD VALUE OF R5
        MOV      R1, TEMP1           ‡SAVE ANY OTHER REGISTER
        MOV      R0, TEMPO
        MOV      ‡ARG, R5            ‡SET UP R5
        JSR      PC, SQRT            ‡CALL ROUTINE
        MOV      R0, NUM2            ‡STORE HIGH ORDER RESULT
        MOV      R1, NUM2+2          ‡STORE LOW ORDER RESULT
        MOV      TEMP5, R5           ‡RESTORE SAVED
        MOV      TEMP1, R1           ‡REGISTERS
        MOV      TEMPO, R0
        *
        *
        *
ARG:    .WORD    1
        .WORD    NUM1
TEMP5:  .WORD    0
TEMP1:  .WORD    0
NUM1:   .FLT2   4
NUM2:   .FLT2   0
    
```

The following example is a complete assembly language routine. This routine can be called by the following statement:

```
CALL HYPOT(A,B,C,C%)
```

The routine calculates the value of the expression  $\text{SQR}(A^2+B^2)$ , assigns the value to C, and assigns the truncated value to C%.

```

        .TITLE   HYPOT
        .PSECT  SUBRS,RO,I
        *
        .GLOBL  HYFTAB
HYFTAB: .BYTE   5
        .ASCII  'HYPOT'
        *
        .EVEN
        .WORD   HYPOT
        *
        .GLOBL  ‡ARGER,‡BOMB,‡POLSH,‡UNPOL
        .GLOBL  ‡MLR,‡XFI‡,‡ADR,SQRT,‡RI
        *
HYPOT:  CMPB    (R5)+,‡4             ‡ARE THERE 4 ARGUMENTS?
        BEQ    20‡                 ‡YES.
10‡:    JMP     ‡ARGER              ‡NO, ISSUE ARGUMENT ERROR.
20‡:    CMPB    (R5)+,‡202          ‡ARE WE BEING CALLED BY BASIC-11
        ‡WITH ARGUMENT DESCRIPTORS?
        BNE    60‡                 ‡NO.
        ‡YES, CHECK THAT THERE IS ENOUGH
        ‡STACK SPACE. 30. BYTES SHOULD BE
        ‡SUFFICIENT.
        MOV    SP,R3
        SUB    ‡30.,R3
        CMP    R3,R4                ‡IS IT BELOW THE LIMIT?
        BHS   30‡                 ‡NO.
        JSR   R1,‡BOMB              ‡YES, ISSUE MESSAGE.
        .ASCII 'STACK OVERFLOW IN HYPOT'
        .EVEN
30‡:    MOV     -4(R5),R4            ‡GET THE POINTER TO THE FIRST ELEMENT
        ‡IN THE ARGUMENT DESCRIPTOR LIST.
        JSR   PC,GETDSC             ‡GET THE DESCRIPTOR OF THE 1ST ARGUMENT.
        BIC   ‡160201,R3           ‡IS IT A 2 WORD REAL VALUE?
        CMP   ‡2040,R3
        BNE   10‡                 ‡NO.
        JSR   PC,GETDSC             ‡YES, GET THE DESCRIPTOR OF THE 2ND ARGUMENT
    
```

USING ASSEMBLY LANGUAGE ROUTINES WITH BASIC

```

BIC      #160201,R3      ;IS IT ALSO A 2 WORD REAL?
CMP      #2040,R3
BNE      10$            ;NO.
JSR      PC,GETDSC      ;GET THE DESCRIPTOR OF THE 3RD ARGUMENT.
BIC      #160001,R3      ;IS IT A 2 WORD REAL WITH WRITING ALLOWED?
CMP      #2040,R3
BNE      10$            ;NO.
JSR      PC,GETDSC      ;GET THE DESCRIPTOR OF THE 4TH ARGUMENT.
BIC      #160001,R3
CMP      #1022,R3       ;IS IT AN INTEGER WITH WRITING ALLOWED?
BNE      10$            ;NO.
60$:    MOV      (R5)+,R3      ;PUSH THE 1ST ARGUMENT ON THE STACK.
        MOV      2(R3),-(SP)  ;NOTE: LOW ORDER IS PUSHED FIRST.
        MOV      (R3),-(SP)
        MOV      2(R3),-(SP)  ;PUSH IT AGAIN BECAUSE WE WILL DO
        MOV      (R3),-(SP)  ;A*A TO GET A^2.
        JSR      R4,$POLSH
        $MLR
        $UNPOL
        MOV      (R5)+,R3      ;PUSH THE 2ND ARGUMENT.
        MOV      2(R3),-(SP)
        MOV      (R3),-(SP)
        MOV      #2,-(SP)     ;PUSH A 2 BECAUSE WE WILL USE REAL
                                ;TO INTEGER EXPONENTIATION.

        JSR      R4,$POLSH
        XFI$
        $ADR
                                ;SQUARE THE 2ND ARGUMENT.
                                ;ADD SQUARE OF 2ND ARGUMENT TO SQUARE
                                ;OF FIRST ARGUMENT.

        $UNPOL
                                ;NOW CREATE ON THE STACK THE ARGUMENTS
                                ;REQUIRED BY SQRT.
        MOV      R5,-(SP)     ;SAVE R5.
        MOV      SP,R5
                                ;CREATE POINTER TO VALUE ON THE STACK.
        TST      (R5)+
        MOV      R5,-(SP)
        MOV      #1,-(SP)    ;SHOW ONLY 1 ARGUMENT TO SQRT
        MOV      SP,R5
        JSR      PC,SQRT      ;GET THE SQUARE ROOT.
        CMP      (SP)+,(SP)+  ;REMOVE OLD ARGUMENTS FROM THE STACK.
        MOV      (SP)+,R5     ;RESTORE R5.
        MOV      (R5)+,R3     ;POINT TO THE 3RD ARGUMENT.
        MOV      R0,(R3)+     ;STORE THE REAL RESULT IN THE
        MOV      R1,(R3)      ;3RD ARGUMENT.
                                ;NOTE: SQRT RETURNED ITS RESULT IN R0 & R1.
        MOV      R1,2(SP)     ;REPLACE THE SUM OF THE SQUARES
        MOV      R0,(SP)      ;WITH ITS SQUARE ROOT.
        JSR      R4,$POLSH
        $RI
        $UNPOL
        MOV      (SP)+,@(R5)+  ;STORE THE INTEGER RESULT IN
                                ;THE 4TH ARGUMENT.
        RTS      PC           ;RETURN TO THE CALLER.

```

;GETDSC RETURNS THE NEXT ARGUMENT'S DESCRIPTOR WORD.

;INPUTS:

; R4 POINTS TO THE WORD IN THE DESCRIPTOR LIST.

;OUTPUTS:

; R3 CONTAINS THE DESCRIPTOR WORD FOR THE CURRENT ARGUMENT.  
; R4 IS UPDATED TO POINT TO THE NEXT ELEMENT IN THE LIST.

```

GETDSC: MOV      (R4)+,R3      ;GET THE DESCRIPTOR.
        BIT      #1,R3        ;IS IT A POINTER?
        BNE      10$          ;NO.
        MOV      (R3),R3      ;YES, GET THE ACTUAL DESCRIPTOR.
10$:    RTS      PC
        .END

```

Ready

## INDEX

- ABORT function, 3-4
- \$ALC routine, 4-8, 4-9, 4-10
- ALR, advantages of, 4-1
- ALR format, 4-2
- ALR, FORTRAN-compatible, 4-4
- \$ARGER routine, 4-9
- Argument checking, 4-4
- Argument descriptor list, 4-4
- Argument descriptor word, 4-6
- Argument list, 4-4, 4-5
- Argument list, double precision, 4-16
- Argument list, single precision, 4-16
- Argument pointer, 4-8
- Argument pointer list, 4-4
- Array, numeric, 4-7
- Arrays, string, 4-7
- Assembly language routine, 4-1
- Assembly language routine, FORTRAN-compatible, 4-4
  
- .BAC file type, 1-8
- Background job, 1-2
- BASIC software kit, 4-2
- BASIC termination, 3-6
- .BAX file type, 1-8
- \$BOMB routine, 4-9, 4-11
- BYE command, 1-7
  
- CALL statement, 4-1, 4-4
- Canceling CTRL/O, 3-2
- Checking for CTRL/C, 3-7
- \$CHROT routine, 4-9, 4-11
- Command, BYE, 1-7
- Command, CTRL/C, 1-6
- Command, CTRL/F, 1-4
- Command, DIRECTORY, 2-4
- Command, FRUN, 1-4
- Command, RUN, 1-3
- CTRL/C checking, 3-7
- CTRL/C command, 1-6
- CTRL/C disabling, 3-3
- CTRLC function, 3-3
- CTRL/F command, 1-4
- CTRL key, vi
- CTRL/O canceling, 3-3
  
- Data type, 4-5
- \$DEALC routine, 4-8, 4-9, 4-10
  
- Default device, 2-2
- Default file name, 2-2
- Default file type, 2-2
- DEL key, vi
- Descriptor list, argument, 4-4
- Descriptor, string argument, 4-7
- Device, default, 2-2
- Device names, 2-1
- DIRECTORY command, 2-4
- Disabling CTRL/C, 3-3
  
- Enabling lower case, 3-7
- Error handling routines, 4-9
- Error messages, 1-8
- ESC key, vi
  
- File directory listing, 2-3
- File name, default, 2-2
- File specification, 2-1
- File type, default, 2-2
- \$FIND routine, 4-8, 4-9, 4-10
- Floating point precision, 1-7
- Foreground job, 1-4
- FRUN command, 1-4
- Function, ABORT, 3-4
- Function, CTRLC, 3-3
- Function, optional, 1-2
- Function, RCTRLC, 3-3
- Function, RCTRL/O, 3-2
- Function, SYS, 3-5
- Function, TTYSET, 3-1
  
- Global address, 4-3
  
- Indirect file, 1-5
  
- LET statement, 3-1
- Link time feature selection, 1-1
- Lower case characters, 3-7
  
- Mathematical routines, 4-11, 4-12, 4-13
- Message routines, 4-9
- \$MSG routine, 4-9, 4-11

INDEX (CONT.)

Name table, user routine, 4-2  
 Numeric arrays, 4-7

Offset, 4-8  
 OPEN statement, 2-3  
 Optional features, 1-1

Pointer, argument, 4-8  
 Pointer list, argument, 4-4  
 Pointer, string reference,  
 4-6, 4-8  
 \$POLSH routine, 4-12  
 Precision, floating point, 1-7,  
 4-16  
 Program termination, 3-4

RCTRLC function, 3-3  
 RCTRLLO function, 3-2  
 RET key, vi  
 Routine, \$ALC, 4-8, 4-9, 4-10  
 Routine, \$ARGER, 4-9  
 Routine, \$BOMB, 4-9, 4-11  
 Routine, \$CHROT, 4-9, 4-11  
 Routine, \$DEALC, 4-8, 4-9, 4-10  
 Routine, \$FIND, 4-8, 4-9, 4-10  
 Routine, \$MSG, 4-9, 4-11  
 Routine, \$POLSH, 4-12  
 Routine, \$STORE, 4-8, 4-9, 4-10  
 Routine, \$UNPOL, 4-12  
 Routine name, 4-2  
 Routines, assembly language,  
 4-1  
 Routines, error handling, 4-9  
 Routines, mathematical, 4-11,  
 4-12, 4-13  
 Routines, message, 4-9  
 Routines, string access, 4-8,  
 4-9, 4-10  
 Routines, threaded code, 4-12,  
 4-14

RUN command, 1-3  
 Run time feature selection, 1-1

Scalar, string, 4-7  
 Single character input, 3-6  
 Single job monitor, 1-2  
 Software kit, BASIC, 4-2  
 Stack limit, 4-4  
 Starting address, routine, 4-3  
 Starting BASIC, 1-2  
 Statement, CALL, 4-1, 4-4  
 Statement, LET, 3-1  
 Statement, OPEN, 2-3  
 Stopping BASIC programs, 1-6  
 \$STORE routine, 4-8, 4-9, 4-10  
 String access routines, 4-8,  
 4-9, 4-10  
 String argument descriptor,  
 4-7  
 String arrays, 4-7  
 String reference pointer, 4-6  
 SYS functions, 3-5  
 System functions, 3-5

Terminal margin setting, 3-1  
 Terminating BASIC, 3-6  
 Terminating the program, 3-4  
 Threaded code routine, 4-12,  
 4-14  
 TTYSET function, 3-1

\$UNPOL routine, 4-12  
 User routine name table, 4-2  
 Utility functions, 3-1

Wildcard feature, 2-4  
 Word, argument descriptor,  
 4-6

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

---

---

---

---

---

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

---

---

---

---

---

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

---

---

---

---

---

Please indicate the type of user/reader that you most nearly represent.

- Assembly language programmer
- Higher-level language programmer
- Occasional programmer (experienced)
- User with little programming experience
- Student programmer
- Non-programmer interested in computer concepts and capabilities

Name \_\_\_\_\_ Date \_\_\_\_\_

Organization \_\_\_\_\_

Street \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip Code \_\_\_\_\_  
or  
Country

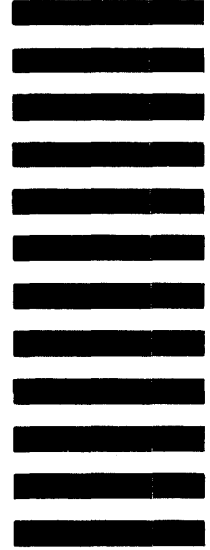


Do Not Tear - Fold Here and Tape

**digital**



No Postage  
Necessary  
if Mailed in the  
United States



**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS ML 5-5/E45  
DIGITAL EQUIPMENT CORPORATION  
146 MAIN STREET  
MAYNARD, MASSACHUSETTS 01754

Do Not Tear - Fold Here

Cut Along Dotted Line

Update Notice No. 1

BASIC-11/RT-11 User's Guide

Order No. DEC-11-LIBUA-A-DN1

March 1978

NEW AND CHANGED INFORMATION

This update includes information to reflect the release of BASIC-11/RT-11 Version 2 on the RL01 disk.

Copyright © 1978 Digital Equipment Corporation

INSTRUCTIONS

Place the following pages in the BASIC-11/RT-11 User's Guide as replacements for, or additions to, current pages. The changes made on replacement pages are indicated in the outside margin by change bars (▮) for additions and by bullets (●) for deletions.

<u>Old Page</u>		<u>New Page</u>	
Title	Page/Copyright	Title	Page/Copyright
	2-1/2-2		2-1/2-2
	3-7/3-8		3-7/3-8