94-CC3/C71/22

January 1981

This manual describes the use of the BASIC–PLUS–2 Compiler on the RSX, IAS, and VMS operating systems. It includes descriptions of compiler commands, resident libraries, file operations, utilities, and system-specific usage.

BASIC–PLUS–2
RSX/IAS/VMS User's Guide

Order No. AA-0157C-TC

Including AD-0157C-T1, T2

OPERATING SYSTEM AND VERSION:	RSX–11M	V3.2
	RSX–11M PLUS	V1.0
	IAS	V3
	VMS	V1.6
SOFTWARE VERSION:	BASIC–PLUS–2	V1.6

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

digital equipment corporation · maynard, massachusetts

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.

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

The following are trademarks of Digital Equipment Corporation:

DEC	FOCAL
DECnet	IAS
DECsystem-10	MASSBUS
DECSYSTEM-20	PDP
DECtape	RSX
DECUS	UNIBUS
DIBOL	VAX
DIGITAL	VMS

Contents

Chapter 2 Memory Resident Libraries

Chapter 3 Files

Chapter 4 Program Segmentation

Chapter 5 BASIC–PLUS–2 Utilities

Chapter 6 BASIC–PLUS–2 on RSX–11M

Chapter 7 BASIC–PLUS–2 on IAS

Chapter 8 BASIC–PLUS–2 on VMS (Compatibility Mode)

Chapter 9 BASIC–PLUS–2 on RSX–11M PLUS

Chapter 10 BASIC–PLUS–2 on TRAX

Appendix A BASIC–PLUS–2 Language Elements

Appendix B Compile-Time Error Messages

Appendix C RUN-Time Error Messages

Appendix D ASCII Codes and Data Representation

Appendix E Reserved Words in BASIC–PLUS–2

Appendix F Program and Subprogram Coding Conventions

Figures

Tables

Commercial Engineering Publications typeset this manual using DIGITAL's TMS–11 Text Management System.

157ALL

Preface

Document Objective

The *BASIC–PLUS–2 RSX/IAS/VMS User's Guide* describes how to use the BASIC–PLUS–2 Language Processor on the following operating systems:

RSX–11M V3.2
RSX–11M PLUS V1.0
IAS V3
VMS V1.6

Intended Audience

This manual is not a tutorial. You should be familiar with the RSX/IAS operating system and the BASIC–PLUS–2 language before reading the User's Guide.

Document Structure

Chapters 6 through 10 describe system-specific usage of BASIC–PLUS–2. Read the chapter pertaining to your operating system before you read Chapters 1 through 5.

Chapter 1	describes the BASIC–PLUS–2 Compiler, compiler commands, command formats, and the building of programs for execution as task images.
Chapter 2	describes the BASIC–PLUS–2 memory resident library.
Chapter 3	explains device specific (RSX) and Record Management Services (RMS) file handling.
Chapter 4	explains subprogram calling conventions and linkage.
Chapter 5	describes the BASIC–PLUS–2 utilities: the Translator, Resequencer, and Cross-reference program.
Chapter 6	describes system-specific usage of BASIC–PLUS–2 on RSX.
Chapter 7	describes system-specific usage of BASIC–PLUS–2 on IAS.

Chapter 8 describes system-specific usage of BASIC–PLUS–2 on VMS (Compatibility Mode).

Chapter 9 describes system-specific usage of BASIC–PLUS–2 on RSX–11M PLUS.

Chapter 10 describes system-specific usage of BASIC–PLUS–2 on TRAX.

The manual also contains six appendices:

Appendix A summarizes BASIC–PLUS–2 commands, statements, functions, and operators.

Appendix B lists compile-time error messages.

Appendix C lists run-time error messages.

Appendix D summarizes data formatting and character representation.

Appendix E lists reserved keywords.

Appendix F is a coding template for BASIC programs.

Associated Documents

Users unfamiliar with RSX should read the RSX System documentation. Those unfamiliar with BASIC–PLUS–2 should read the *PDP–11 BASIC–PLUS–2 Language Reference Manual*, which explains language elements, commands, statements and functions.

Conventions

This manual uses the following conventions:

RET represents a carriage return/line feed.

^ (circumflex) represents a control character. For example, ^C represents pressing the keyboard "Control" character and the letter C simultaneously. Depending on the context, the circumflex could also indicate exponentiation.

color indicates information that you must type into sample programs.

[brackets] enclose an optional portion of a general format. When they enclose vertically stacked entries, brackets indicate that you can select one of the enclosed elements.

{braces} enclose a mandatory portion of a general format. When they enclose vertically stacked entries, braces indicate that you must choose one of the enclosed elements.

Chapter 1
Using the BASIC–PLUS–2 Compiler

The RSX family of operating systems supports a BASIC–PLUS–2 language processor. The processor contains a compiler and a memory-resident library/disk library combination. This chapter describes the BASIC–PLUS–2 compiler, commands and source line input to the compiler, program editing, and debugging aids. Chapter 2 explains the memory-resident library.

NOTE

Throughout this manual, "BASIC" refers to "BASIC–PLUS–2" unless otherwise indicated.

The BASIC–PLUS–2 Compiler:

- Checks each program line for syntax errors and returns error messages.
- Generates linkable object modules from your source programs. You then task build these modules. Task building is described in the *RSX–11M Task Builder Reference Manual.*

To invoke the compiler, type:

```
RUN $BASIC2 (RET)
```

Depending on installation options, you can also use the short form "BP2." An identification line and a prompt then appear on your terminal, and the compiler awaits your input. If this command does not work, ask the system manager how your system invokes BASIC–PLUS–2.

Input to the BASIC–PLUS–2 Compiler can be a command or a source program line. Section 1.1 describes BASIC commands. Section 1.2 describes BASIC source programs.

1.1 Compiler Commands

BASIC-PLUS-2 supports the commands listed in Tables 1-1 and 1-2. BASIC returns the error message "?What?" for all other commands. Note that commands and their arguments have no line numbers. "Filespec" refers to the standard RSX file specification. File specifications have the format:

dev:[group,member]filename.ext;version

where:

dev:	is the device designator. Use two ASCII characters optionally followed by a one- or two-digit octal unit number. The default is SY0:.
group,member	is the octal group number and member number associated with the user file directory (UFD). The default is the current account.
filename	is a one- to nine-character file name. The default is NONAME.
.ext	is a one- to three-character file extension. The default is .B2S.
;version	is an octal number indicating the generation number of the file. The default is the most recent version of the file.

Table 1-1: BASIC-PLUS-2 Commands

Command	Function
APPEND	Merges the specified program with the program currently in memory.
BRLRES	Enables you to specify a resident library to be linked to your program during task building.
BUILD	Creates an indirect command file and an overlay description language file to specify input to the task builder. See Table 1-2 for BUILD command options.
COMPILE	Converts a BASIC source program into an object module with a default extension of OBJ. See Table 1-2 for COMPILE command options.
DELETE	Erases a specified line or lines from a BASIC source program.
DSKLIB	Enables you to select BASIC2 or your own disk library to link to your program during task building.
EXIT	Clears memory and returns control to the operating system.
LIBRARY	Enables you to select a BASIC-PLUS-2 memory resident library to be linked to your program.
IDENTIFY	Causes BASIC-PLUS-2 to print an identification message on terminal.
LIST	Prints a copy of the current program or specified lines on the terminal.

(continued on next page)

Table 1-1: BASIC-PLUS-2 Commands (Cont.)

Command	Function
LOCK/sw	Allows you to specify default switches (sw) for later commands. These switches are for: • Task extend size • Scale factor • Precision • MAP/NOMAP • DUMP/NODUMP • CHAIN/NOCHAIN • DEBUG/NODEBUG • Output type • File organizations
NEW	Clears memory for the creation of a new program.
ODLRMS	Enables you to select an RMS-11 ODL file when you build the program.
OLD	Loads a specified program into memory.
RENAME	Changes the name of the program currently in memory.
REPLACE	Stores a new program in place of an old one of the same name.
RMSRES	Enables you to specify the RMS-11 resident library that will provide code for RMS-11 file and record operations.
SAVE	Copies a source program from memory to a specified device.
SCALE	Controls the scale factor for double-precision (4-word) floating-point format.
SEQUENCE	Enters program line numbers beginning at a number you specify. You can also specify the increments between line numbers.
SHOW	Prints the current compiler defaults on your terminal.
UNSAVE	Deletes a specified file.

All switches in Tables 1-1 and 1-2 have two forms: /sw and /NOsw. You can shorten these commands and switches to a minimum of three letters each: for example, COM/DEB and COM/NODEB.

You can use the commands listed in Tables 1-1 and 1-2 by typing them at your terminal, or you can include them in an indirect command file you create. An indirect command file allows you to execute a series of commands with a single command file. The file specification format is:

@filename.CMD

Table 1-2: BASIC-PLUS-2 BUILD and COMPILE Switches

Command/switch	Function
BUILD/BRLRES:filespec	Enables you to specify a resident library to be linked to your program during task building.
BUILD/DSKLIB:filespec	Enables you to select BASIC2, or your own disk library to link to your program during task building.
BUILD/DUMP	Instructs the operating system to generate a binary dump of memory contents in the event of an abnormal exit from your program.
BUILD/EXTEND:n	Increases run-time program storage by a minimum of n words.
BUILD/LIBR:filespec	Allows you to select the memory resident library to link to the executable file image.
BUILD/IND	Links in the code necessary to use RMS-11 indexed files.
BUILD/MAP	Causes the task builder to create a memory allocation map file with a default extension of .MAP.
BUILD/ODLRMS:filespec	Allows you to select an RMS-11 ODL file when you build the program.
BUILD/REL	Links in the code necessary to use RMS-11 relative files.
BUILD/RMSRES:filespec	Instructs the task builder to link your program to the specified RMS-11 resident library.
BUILD/SEQ	Links in the code necessary to use RMS-11 sequential files.
BUILD/VIR	Links in the code necessary for virtual and block I/O files.
COMPILE/DEBUG	Converts a BASIC source program to object code and enables you to use the debugging aid.
COMPILE/DOUBLE	Converts a BASIC program to object code and enables you to use the double-precision math package.
COMPILE/LINE	Converts a BASIC source program to object code and retains line information.
COMPILE/MACRO	Converts a BASIC source program into a MACRO source file with a default extension of .MAC.
COMPILE/OBJECT	Converts a BASIC source program into an object module with a default extension of .OBJ.

For example, if you create a file called COMP1.CMD that contains these commands:

```
OLD MOD1
COM/MAC
OLD MOD2
COM
OLD MOD3
COM/DEB
```

the BASIC Compiler creates MOD1.MAC, MOD2.OBJ, and MOD3.OBJ; MOD3.OBJ includes the debugging aid.

A switch in an indirect command file affects its corresponding command only. There are four ways to specify switches. In their order of precedence, they are:

1. Switches specified inside individual command file commands (for example, COM/MAC). These switches operate only when executing the indirect command file.

These override:

2. Switches specified in a LOCK command inside a command file. These remain in force until you: (1) exit from BASIC-PLUS-2, (2) specify another LOCK, or (3) finish processing the command file.

These override:

3. Switches specified in command file specifications outside a command file (for example, @file/sw). These switches operate only when executing the indirect command file.

These override:

4. Switches specified in a LOCK command outside the command file. Switches set by LOCK remain in force until you exit from BASIC-PLUS-2 or specify another LOCK.

NOTE

Switches set outside the command (.CMD) file are global; using them saves time when you must specify options for files containing many commands. However, make sure these switches match those you specify inside the command file, or they will be overridden.

1.1.1 APPEND Command

The APPEND command (APP) merges an existing BASIC source program with a program currently in memory. It does not change the name of the program in memory. To use APPEND, type:

APPEND filespec RET

where:

filespec is the file specification of the program to be appended.

The compiler opens the specified source program as secondary input and reads it into memory. It then adds the contents to the current program, based on line number order. If both programs contain identical line numbers, the appended program line replaces the current program line.

If you type APPEND without a filespec, the compiler prompts:

```
APPEND FILE NAME-
```

and awaits the name of the program you want to merge. If you type a carriage return, the compiler searches for a source program called NONAME.B2S. If no program is found, BASIC prints:

```
?Can't find file or account
```

If both programs are on the current system disk, one of them must be brought into memory before giving the APPEND command. You do this with the OLD command.

In this example, two programs, named AP1 and AP2, are on disk:

Program AP1

```
10    LET B=5
20    LET C=2
30    LET A=B^C
40    PRINT A
50    END
```

Program AP2

```
35 LET D=A^C
40 PRINT A;D
```

An OLD command brings program AP1 into memory and sets the program name. An APPEND command merges AP1 with AP2, and the output is:

```
OLD AP1   (RET)

BASIC2

APPEND AP2   (RET)

BASIC2

LISTNH   (RET)

10    LET B=5
20    LET C=2
30    LET A=B^C
35    LET D=A^C
40    PRINT A;D
50    END
```

The name of the program remains AP1. Line 40 of the appended program replaces line 40 of the program in memory; line 35 is merged sequentially.

1.1.2 BRLRES Command

The BASIC shared Resident Library (BRL) command enables the compiler to generate the proper task builder commands to link your program to a resident library. You can access the distributed BASIC2 memory resident library with either the BRLRES or LIBR commands.

To link to a resident library, type:

```
BRLRES {User Name}
       {NONE     }
```

where:

User Name	is your own resident library. Only experienced programmers should create resident libraries.
NONE	is an option of no BASIC–PLUS–2 resident library.

If you enter BRLRES without the file specification, BASIC prompts for one. If none is supplied, BASIC retains the current default value.

Enter BRLRES before BUILD. The command remains in effect until: (1) you give a new library command, or (2) you exit from the compiler, at which time the system returns to the original default value. You can override BRLRES with a switch added to the BUILD command. This new value remains in effect for one BUILD only, however. The system then returns to the previous default value.

If a resident library you select is not available, the task builder returns an error message. Your system manager can provide information about the resident libraries available to you.

1.1.3 BUILD Command

The BUILD (BUI) command generates a command (CMD) and an overlay description language (ODL) file for the specified program. The command file is a set of instructions for the task builder. The command file takes the name of the main program and a default extension of .CMD. The ODL file specifies how segments of the task-built program will overlay when you run it. The BUILD command has the format:

```
BUILD MAIN [,SUB1, SUB2,...][/sw]
```

where:

MAIN	represents the name of an object module (a previously compiled program).
SUB1, SUB2	represent the names of one or more previously compiled subprograms. Separate these names with commas.
/sw	is one or more BUILD command switches.

NOTE

The command line must fit on a single, 80–character line.

In addition to object module names and BUILD switches, BUILD accepts defaults from previously specified BRLRES, DSKLIB, LIBR, ODLRMS, or RMSRES commands.

A BUILD switch permits special operations on object modules at task building; for example, RMS I/O or linking to libraries. You can use any combination of these switches on the command line, depending on the content of the modules. Table 1–3 summarizes BUILD Switches.

Table 1-3: BUILD Switches

Command/switch	Function
/BRLRES:filespec	Enables you to specify your own resident library to be linked to your program at task building.
/DSKLIB:filespec	Enables you to select BASIC2 or your own disk library to link to your program at task building.
/DUMP	Instructs the operating system to generate a dump of memory contents if there is an abnormal exit from your program.
/EXTEND:n	Increases the program's storage space by a minimum of n words, where n is rounded up to the nearest 32 word boundary. The maximum allowable program space is 32K words.
/LIBR:filespec	Enables you to select the memory resident library to link to the executable file image.
/IND	Links in the code needed for indexed file operations. This switch sets automatically when you compile a program containing an organization INDEXED clause in an OPEN statement.
/MAP	Generates a memory allocation map of the resulting task.
/ODLRMS:filespec	Enables you to select an RMS or user-created ODL file when building your program.
/REL	Links in the code needed for relative file operations. This switch sets automatically when you compile a program containing an organization RELATIVE clause in an OPEN statement.
/RMSRES:filespec	Allows you to specify an RMS resident library to link to your program to provide code for RMS-11 record and file operations.
/SEQ	Links in the RMS-11 code needed for sequential file operations. This switch sets automatically when you compile a program containing an organization SEQUENTIAL clause in an OPEN statement.
/VIR	Links in the RMS-11 code needed for virtual array and block I/O file operations. This switch is set automatically when you compile a program containing an organization VIRTUAL clause in the OPEN statement.

BUILD generates CMD and ODL files using the name of the main object module. For example:

```
BUILD MAIN, SUB1, SUB2
```

creates two files—MAIN.CMD and MAIN.ODL. By specifying the indirect command file (MAIN.CMD) to the task builder, you create: (1) an executable task image file with a default extension of .TSK, and (2) an optional memory allocation map with a default extension of .MAP. Creating a map file is a BASIC-PLUS-2 installation option; if NOMAP is the default, you must use the BUILD/MAP command to generate a memory allocation map.

The BUILD command also generates an overlay description language file (MAIN.ODL). The ODL file has this form if you perform device specific I/O:

```
          .ROOT USER
USER:     .FCTR SY:NONAME - LIBR
LIBR:     .FCTR LB:[1,1]BASIC2/LB
          .END
```

This ODL supports INPUT/PRINT operations from TI: (the terminal). When you use virtual, sequential, relative, or indexed files on disk, the ODL form is:

```
          .ROOT BASIC2-RMSROT-USER-RMSALL
USER:     .FCTR SY:NONAME - LIBR
LIBR:     .FCTR LB:[1,1]BASIC2/LB
@LB:[1,1]BASICn
@LB:[1,1]RMS11S
          .END
```

In this ODL file:

.ROOT	defines the task linkages.
BASIC2	is a factor label in the specified ODL file (@LB:[1,1]BASICn).
RMSROT, RMSALL	are factor labels in the RMS ODL file (@LB:[1,1]RMS11S).
USER	links in your modules.

The USER line ends with a LIBR reference line. All lines you add to an ODL should end with a reference to the LIBR to retrieve OTS modules from the BASIC library.

The BASIC ODL line (@LB:[1,1]BASICn) tells the task builder to retrieve modules needed to support file OPEN statements from BASRMS.ODL. You set the value of "n": (1) implicitly by specifying a file organization in the OPEN statement, or (2) explicitly by specifying BUILD switches. Table 1-4 summarizes the values of n and their meanings.

Table 1-4: BASIC ODL Values

Value	ODL File Name	File Types Supported
0	BP2IC0.ODL	Virtual
1	BP2IC1.ODL	Sequential
2	BP2IC2.ODL	Relative
3	BP2IC3.ODL	Sequential, Relative, Indexed
4	BP2IC4.ODL	Indexed
5	BP2IC5.ODL	Sequential, Indexed
6	BP2IC6.ODL	Relative, Indexed
7	BP2IC7.ODL	Sequential, Relative, Undefined

The ODL line (@LB:[1,1]RMS11S) is controlled by the ODLRMS command to the compiler. You can edit this file to generate program overlay segments. The *RSX-11M Task Builder Reference Manual* describes the overlay syntax.

Output from BUILD/DUMP helps you to check programs. DUMP instructs the system to generate a dump of memory contents if there is an abnormal exit from your program during execution. The system generates a file with an extension of PMD. See your Task Builder Manual for more information on PMD files.

1.1.4 COMPILE Command

The COMPILE (COM) command converts a source program into a linkable object module or a MACRO source program. The default output (.OBJ or .MAC) is set by your system manager at installation.

You must bring a program into memory to COMPILE it. The COMPILE command does not execute a program; it converts the source program to object format and writes that file to disk. For example:

```
COMPILE (RET)
```

compiles the program currently in memory. You can also type:

COMPILE filespec (RET)

where:

filespec — is an RSX file specification. The filespec defaults to the program name on the current system disk and account.

The command compiles the current program, assigns the specified name, and appends the .OBJ or .MAC extension. Table 1–5 summarizes the COMPILE switches.

In most cases, you can combine COMPILE command switches. The exceptions are /OBJECT and /MACRO switches. For example:

```
COMPILE/OBJECT/DEBUG/DOUBLE
```

is a valid specification, but:

```
COMPILE/OBJECT /MACRO
```

is not.

You can use the LOCK command to specify any valid combination of compiler switches. These then become the defaults for COMPILE commands. Thus, you prevent having to respecify switches for each compilation. Use the LOCK command without arguments to disable all switches. Note that a COMPILE command with no arguments retains the current defaults.

For example:

```
LOCK/OBJECT/NOLINE/NODEBUG

OLD PROG1

BASIC2

COMPILE

BASIC2

OLD PROG2

BASIC2

COMPILE

BASIC2

OLD PROG3

BASIC2

COMPILE/MAC

BASIC2

LOCK

OLD PROG4

BASIC2

COMPILE/MACRO

BASIC2
```

In this example, the OLD command brings four programs into memory. The first LOCK sets the /OBJECT, /NOLINE, and /NODEBUG compiler switches as the defaults. The compiled PROG1 and PROG2 become object modules with /NOLINE and /NODEBUG enabled. When you compile PROG3, however, you specified the creation of a macro file. This overrides the /OBJECT default and creates a compiled program with a .MAC extension. The /NOLINE and /NODEBUG switches remain enabled. Finally, the LOCK command with no arguments disables all defaults, and PROG4 is compiled as a MACRO file with default switches. The result of these four compilations is:

PROG1.OBJ (NOLINE and NODEBUG enabled)
PROG2.OBJ (NOLINE and NODEBUG enabled)
PROG3.MAC (NOLINE and NODEBUG enabled)
PROG4.MAC (default switches enabled)

You could also specify:

```
LOCK/OBJECT/NOLINE/NODEBUG
   .
   .
   .
LOCK/MACRO
BASIC2
OLD PROG4
COMPILE
```

and generate the same results. The LOCK command, with or without switches, supersedes previous LOCK commands.

Table 1-5: COMPILE Switches

Command	Function
COMPILE/DEBUG (COM/DEB)	Converts the program into object code and enables the BASIC-PLUS-2 debugging aid. The /DEBUG switch significantly increases the size of the module being compiled. Therefore, when the module has been debugged, recompile it without /DEBUG to reduce the module's size. You cannot specify /NOLINE with a /DEBUG option.
COMPILE/DOUBLE (COM/DOU)	Converts the program into object code and uses double-precision format (4-word) for all floating-point operations. If an object task contains both single and double-precision formats, you receive the error "?Wrong math package" (ERR=125). The BASIC default is set at installation time.
COMPILE/MACRO (COM/MAC)	Converts the program into MACRO source code and saves it with a .MAC default extension.
COMPILE/NOLINE (COM/NOLIN)	Is an installation option that disables program line headers in memory and reduces program memory needs by: — Two words per line. — Four words per function definition. — Two words per DIM statement. — Four words per FOR NEXT, WHILE, or UNTIL NEXT loop or clause. Do not specify /NOLINE when using an ERL function, a debugging program, or a RESUME statement without a line number specification. In each case, a warning message will indicate that /NOLINE is overridden.
COMPILE/OBJECT (COM/OBJ)	Converts a program into object code, saves it as an object module, and adds an OBJ extension to the program name. You must task build the object modules to create a runnable program.

1.1.5 DELETE Command

The DELETE (DEL) command removes specified lines from the program currently in memory. To delete a series of lines, specify the line numbers and separate them with commas. To delete a consecutive group of lines, type the first and last line number of the group, separated by a hyphen.

For example:

`DEL 50`	removes line 50 from the program.
`DEL 50,80`	removes lines 50 and 80 from the program.
`DEL 50-80`	removes lines 50 through 80 from the program.
`DEL 50,60,90-110`	removes lines 50, 60, and 90 through 110 from the program.

If you do not specify a line in the DELETE command, BASIC returns an error message ("Illegal Delete Command"). If you specify a range of lines, BASIC removes all lines in that range. Thus, if you type DELETE 50–80, BASIC erases all of the lines between 50 and 80, inclusive. An invalid line specification such as DELETE 80–50 returns the error message "Bad Line Number Pair."

1.1.6 DSKLIB Command

The Disk Library (DSK) command makes a disk-resident library of object modules available to your program at task build time. Every system must have a disk library default set at installation. To select a library, type one of the following:

```
DSKLIB { LB:[1,1]BASIC2 [.OLB] }
       { dev:name [.OLB]       }
```

where:

LB:[1,1]BASIC2	is a disk object library of run-time routines used with BASIC2.
dev:name	is the name of a user-created library. User names need full file specifications.
[.OLB]	is an optional object library file extension.

If you enter DSKLIB without a file specification, BASIC prompts for one. If none is supplied, BASIC retains the current default value.

Enter DSKLIB before BUILD. The command remains in effect until: (1) you give a new disk library command, or (2) you exit from the compiler, at which time the system returns to the original default value. You can override the DSKLIB command with a switch added to the BUILD command, but this remains in effect for one BUILD only.

If the library you select is not available, the task builder will return an error message. Your system manager can provide information about the disk libraries available to you.

1.1.7 EXIT Command

The EXIT (EXI) command ends access to the BASIC–PLUS–2 Compiler and returns control to the operating system.

1.1.8 IDENTIFY Command

The IDENTIFY (IDE) command prints a header containing the BASIC-PLUS-2 name and version number. This header is displayed only if you have invoked the compiler.

For example:

```
IDENTIFY (RET)

PDP-11 BASIC-PLUS-2      V1.6  BL-01.60

BASIC2

EXIT  (RET)

>

> IDENTIFY  (RET)

MCR - Illegal function

>
```

If the response to IDENTIFY is "BASIC-PLUS-2", BASIC-PLUS-2 is available. The MCR error message indicates a return to the RSX monitor.

1.1.8A INQUIRE Command

INQUIRE is an installation option on all systems. During installation, you must answer YES to the "Generate Help Files" BASBLD prompt. The INQUIRE command is automatically installed on TRAX.

The INQUIRE command displays information about BASIC–PLUS–2 commands, statements, and functions. The format is:

INQUIRE topic(s)/sw

where:

topic(s)	is one or more BASIC–PLUS–2 commands, statements, or functions. Separate multiple topics with commas.
/sw	is one of the optional switches, /TEX[T] or /EXA[MPLE].

NOTE

The INQUIRE command must fit on a single 80–character line.

If you do not specify a topic, BASIC returns a list of topics for which information is available. To obtain information for all the topics in the INQUIRE list, type an asterisk after the INQUIRE command.

If you use neither of the INQUIRE command switches, BASIC returns both explanatory text and examples for the topics you specify. To display only the text, append the /TEXT switch to the command line. To display only the examples, use the /EXAMPLE switch. The switch modifies all the topics in the command line.

For example:

```
INQ MAP,ARRAYS/TEXT
```

1.1.9 LIBRARY Command

The LIBRARY (LIBR) command enables you to select the memory resident library to be linked to your program. To select a library, type one of the following:

```
{BASIC2}
{NONE  }
```

where:

BASIC2	is an 8KW memory resident library of BASIC–PLUS–2 routines.
NONE	disables the memory resident library linkage.

You can also specify your own memory resident library. User names require full file specifications.

For example:

```
LIBR (RET)
NAME [BASIC2] - NONE (RET)
ACCOUNT [LB:[1,1]] - (RET)
```

In this example, the LIBR command displays the current resident library (BASIC2). Specifying NONE disables linking to BASIC2.

The BASIC2 library has these advantages:

1. Maximum shareable code. The entire 8KW is shareable among all users. This minimizes the memory needs of your system.
2. Faster linking. Because most of the modules are in the LIBR, the task builder accesses the disk library less often.

However, because the allowable system space is set by the system manager, you must see him to determine if the library is available.

1.1.10 LIST Command

The LIST (LIS) command displays a copy of the program currently in memory, with line numbers correctly sequenced.

If you type:

```
LIST (RET)
```

BASIC prints the program, along with a header containing the program name, the current time, and the date. To suppress this information and print the program only, type:

```
LISTNH (RET)
```

where NH specifies no header.

You can print single lines by specifying their line numbers. For example:

LIST 30, 70	prints a header and lines 30 and 70 on your terminal.
LISTNH 30–70	suppresses the header and prints lines 30 through 70.

1.1.11 NEW Command

The NEW command allows you to create programs and assign names to them. Type the command followed by the name. For example, typing:

```
NEW PROG1
```

assigns the name PROG1 to your program. A program name can be a maximum of six alphanumeric characters.

If you do not specify the program name, the system prompts:

```
NEW (RET)
NEW FILE NAME-
```

You can then specify the name or enter a carriage return. The program also becomes the file name. File names permit up to nine characters; program names permit only six. Therefore, if you assign a program name longer than six characters, BASIC truncates it. NONAME is the default.

NOTE

When you type the NEW command, any source code currently in memory is lost.

1.1.12 ODLRMS Command

The Overlay Description Language (ODL) command enables you to select an ODL file to describe the RMS overlay structure used when your program is task built. The default for ODLRMS is the installation option. See the *RMS-11 User's Guide* for more information. To select an ODL file, type one of the following:

```
ODLRMS{ LB:[1,1]RMSRLS[.ODL]
      | LB:[1,1]RMSRLX[.ODL]
      | LB:[1,1]RMS11S[.ODL]
      | LB:[1,1]RMS11X[.ODL]
      | LB:[1,1]RMS12X[.ODL]
      | User-Created
      | NONE                 }
```

where:

RMSRLS	is an ODL file structured to overlay the 8KB of the RMSSEQ resident library. You need not have RMS-11K available on your system.
RMSRLX	is an ODL file structured to overlay the 48KB of the RMSRES resident library. You must have RMS-11K installed on your system.
RMS11S	is an ODL file structured to add a little more than 8KB to task size. RMS11S features only sequential and relative file organization routines in 19 overlay segments. RMS11S does not require a resident library, and RMS-11K need not be available on your system.
RMS11X	is an ODL file structured to add a little more than 9KB to task size. RMS11X features sequential, relative, and indexed file organization routines in 57 overlay segments. RMS11X does not require an RMS resident library, but you must have RMS-11K available on your system.
RMS12X	is an ODL file structured to add no more than 12KB to task size. RMS12X features sequential, relative, and indexed file organization routines in 17 overlay segments. RMS12X does not require an RMS resident library, but you must have RMS-11K available on your system.
User-Created	is an option of defining your own overlay structure for RMS code.

NONE is an option of no RMS-11 code.

[.ODL] is an optional file extension.

NOTE

RMS ODL file names can change. The RMS distribution kit will indicate if any ODLs have new names.

If you enter the ODLRMS command without a file specification, BASIC prompts for one. If none is supplied, BASIC retains the current default value.

Enter ODLRMS before BUILD. The ODL you select remains in effect until: (1) you give a new ODLRMS command, or (2) you exit from the compiler, at which time the system returns to the original default value. You can override the ODLRMS command with an ODL switch added to the BUILD command. This remains in effect for one BUILD only.

If the file you select is not available, the task builder will return an error message. Your system manager can provide information about the files available to you.

1.1.13 OLD Command

The OLD command allows you to bring a previously created source program into memory. Type the command followed by the file name. For example, typing:

```
OLD PROG1
```

brings PROG1 into memory.

If you do not provide a file name, the system prompts:

```
OLD (RET)
OLD FILE NAME-
```

You can then enter the name or a carriage return; when you enter a carriage return, the system searches for a source program called NONAME.B2S, where .B2S is the system default extension.

If you specify a program that does not exist, the system returns the error message:

```
?Can't find file or account
```

NOTE

When you type the OLD command, any source code currently in memory is lost. Also, the system does not check the syntax of the program it brings into memory.

1.1.14 RENAME Command

The RENAME (REN) command changes the name of a program currently in memory. For example, if you bring a saved program named FILE1 from disk to memory and type:

```
RENAME FILE2 (RET)
```

the program becomes FILE2 in memory, but retains the name FILE1 on disk.

1.1.15 REPLACE Command

The REPLACE (REP) command replaces a program on the current system disk or on a specified device with the one in memory. For example, if a program named FILE.B2S needs modification, you can bring it into memory, change it, and type:

```
REPLACE (RET)
```

This replaces the contents of the original program named FILE.B2S with the contents of the newly edited program.

You can also specify a new name for the edited program in the REPLACE command. For example:

```
REPLACE FILE1 (RET)
```

saves the edited version of FILE under the name FILE1.B2S.

If the program named FILE is in memory and there are no other programs with that name, REPLACE performs the same function as SAVE.

1.1.16 RMSRES Command

The RMS Resident Library command causes the compiler to generate the command line so the task builder will link your program to an RMS-11 resident library; this library supplies code for RMS-11 file and record operations. An RMS library saves physical memory space only if multiple tasks using that library run at the same time. The default for RMSRES is an installation option. RMSRES sets defaults for later BUILDs.

To select a resident library, type one of the following:

```
RMSRES { LB:[1,1]RMSRES }
       { LB:[1,1]RMSSEQ }
       { dev: name      }
       { NONE           }
```

where:

LB:[1,1]RMSRES	is an RMS-11 resident library that contains code for sequential, relative, and indexed file operations. You must use the RMSRLX.ODL file when using the RMSRES Resident Library.
LB:[1,1]RMSSEQ	is an RMS-11 resident library that contains code for sequential file operations only. You must use the RMSRLS.ODL file when using the RMSSEQ Resident library.
dev: name	is a user-created resident library. User names need full file specifications.
NONE	is an option of no RMS resident library.

NOTE

Do not use RMSSEQ, RMSRES, or RMS ODL files if your program does not perform RMS-11 operations. For more information on RMS Resident Libraries, see the *RMS-11 User's Guide*.

If you enter the command without a file specification, BASIC prompts for a new one. If none is supplied, BASIC retains the current default value.

Enter RMSRES before BUILD. The command remains in effect until: (1) you give a new library command, or (2) you exit from the compiler, at which time the system returns to the original default value. You can override the RMSRES command with a switch added to the BUILD command. This switch remains in effect for only one BUILD, however.

If you specify an RMS resident library that is not available, the task builder will return an error message. Your system manager can advise you on the availability and names of RMSRES options.

1.1.17 SAVE Command

The SAVE (SAV) command transfers a source program from memory into a file on a specified device or the current system disk. For example, if you have a program in memory and type:

```
SAVE RET
```

BASIC-PLUS-2 sequentially orders the line numbers of the program. It then stores the program on the current system disk as source code under the current name with a .B2S extension. To specify a storage device, extension, or program name, type:

SAVE filespec RET

where:

filespec is a RSX-11M file specification.

If you have not specified a name for the program currently in memory, the program becomes NONAME.B2S.

You cannot save a program with the same file specification as one already saved or the system prints the error message:

```
?File exists - RENAME or REPLACE
```

This error prevents erasing a program by mistake.

1.1.18 SCALE Command

The SCALE (SCA) command controls the scaled arithmetic features of BASIC-PLUS-2 on double-precision systems. It can overcome accumulated round-off and truncation errors in fractional computations performed on systems with floating point format. SCALE enables the system to maintain the decimal accuracy of fractional computations to the number of places you specify.

To specify a scale factor, type:

SCALE [int] RET

where:

int is an integer in the range 0 to 6. If you do not supply a value, BASIC displays the current SCALE factor.

For example:

```
SCALE 2

BASIC2
```

The SCALE 2 command sets the current scale factor to 2. All programs then compiled for that job have floating point numbers: (1) multiplied by 100, or (2) divided by 100, where required. If you do not compile all program modules with the same scale factor, you receive the error message:

```
"?Scale factor interlock",
```

The scale factor you specify remains in effect until: (1) you exit from the compiler, or (2) you specify a different SCALE factor. A SCALE command with no factor specification causes the system to print the current scale factor.

All modules in a program must have the same scale factor.

1.1.19 SEQUENCE Command

The SEQUENCE (SEQ) command automatically enters line numbers as you type a BASIC–PLUS–2 program. The format is:

SEQUENCE [n [,m]]

where:

n is the starting line number. The default is 10.

m is the increment between line numbers. The default is 10.

For example:

```
SEQUENCE 1000,50
```

numbers your program in increments of 50 starting at line 1000.

The SEQUENCE command remains in effect until: (1) you enter a null line (a carriage return only), or (2) you enter an END or SUBEND statement, or (3) the next line number would exceed 32767. The compiler does not check program syntax when you use the SEQUENCE command.

1.1.20 SHOW Command

The SHOW (SHO) command allows you to display the current compiler defaults. To use this command, type:

```
SHOW (RET)
```

after the BASIC prompt. In this example, sample SHOW output is on the left, and alternate values you can specify are on the right:

```
EXAMPLE                                 ALTERNATE VALUES

SHOW
RSX-11M Basic Plus 2 V1.6   BL-01.60 Using EIS -19-
Installed on 20-DEC-79 at 12:46PM
Library for BUILD is BASIC2             NONE
BASIC+2 Resident Library is NONE        User-created
Disk Library is LB:[1,1]BASIC2
RMS ODL file is LB:[1,1]RMS11X          RMSRLS, 11S, 12X, RLX,
                                        NONE
RMS Resident Library is NONE            RMSRES, RMSSEQ
Task extend size = 512                  Any integer
Scale factor = 0                        Integers from 0 to 6
Precision: Single                       Double
Switch Settings:                        Switches can be on ( :LINE)
       NO:MAP                           or off (NO:MAP)
       NO:DUMP
       NO:CHAIN
         :LINE
       NO:DEBUG
   Output:OBJ                           MAC
File Orgs:Sequential                    Sequential (SEQ),
                                        Relative (REL), Indexed
                                        (IND), Virtual (VIR) or
                                        device specific.
```

1.1.21 UNSAVE Command

The UNSAVE (UNS) command deletes a file from the current system disk. For example, if you type:

```
UNSAVE    RET
```

the file associated with the source program currently in memory is deleted from your account. If you type:

```
UNSAVE filespec    RET
```

the specified file is deleted from the current system disk or specified device whether or not it is in memory. You can use this command to erase unwanted files.

If the source program specified in UNSAVE is not found, the system prints the error message:

```
?Can't find file or account
```

To delete a compiled or non-source program, you must type the program's name and extension. For example:

```
UNSAVE FILE.TSK    RET
```

1.2 Editing, Debugging, and Running Source Programs

Programs are ready for execution when you have edited, debugged, and compiled them. The following sections describe these procedures.

1.2.1 Editing

You can edit BASIC program lines once the program is in memory. This section describes how to remove or replace unwanted lines.

To edit an incorrect line, retype the same line number followed by a corrected version of the line. This deletes the old, incorrect line from memory and replaces it with the new one. In this example:

```
10 LAD A = 7 \ B = 9 \ C = SQR(144)    RET
?Misspelled keyword at line 10 statement 1
```

the carriage return enters an incorrect line into memory, causing the compiler to print an error message. Typing:

```
10 LET A = 7 \ B = 9 \ C = SQR(144)    RET
```

erases the previous line 10 from memory and replaces it with the corrected version.

You can also delete a line currently in memory by typing the line number with no text. For example, you can delete:

```
10 LET D = A + B**C  RET
```

from the source program by typing:

```
10  RET
```

You can also use the DELETE command to perform this function.

1.2.2 Debugging

BASIC provides interactive debugging commands to help you locate run-time errors in your program. These commands allow you to check program operation and make corrections. They are:

BREAK	EXIT	STATUS
CONTINUE	FREE	STEP
CORE	I/O BUFFER	STRING
ERL	LET	TRACE
ERN	PRINT	UNBREAK
ERR	RECOUNT	UNTRACE

You can use these commands only on programs or subprograms that have been compiled with the /DEBUG switch. After you have debugged the program and edited the source file, recompile the program without the /DEBUG switch. This saves memory. You can DEBUG individual subprograms in a main program by specifying /DEBUG for that subprogram only.

When you run a program, execution stops the first time you enter a module compiled with the /DEBUG switch. After execution stops, the debugging aid prints an identifying message and prompt:

```
DEBUG: module name
#
```

where:

module name	is the name of the program or subprogram compiled with the /DEBUG switch.
#	signals you to enter debugging aid commands.

Then, to continue the program and execute the command, type the CONTINUE (CON) command. For example:

```
DEBUG: ARCTR
#BREAK 10  RET
#CON  RET
```

In this example, the CON command resumes program execution until line 10.

Following the successful execution of a debugging command, a message identifies your current position in the program or subprogram:

command AT LINE n [,name]

where:

command	is the last executed debugging command (for example, BREAK or STEP) that stops execution.
n	is your current line number in the program or subprogram.
name	is the name of the currently executing subprogram. This name is not displayed if you are executing the main program.

After this message, the debugger gives the # prompt, and you can enter any valid debugger command.

1.2.2.1 BREAK, UNBREAK, and BREAK ON Commands — The BREAK command allows you to stop program execution before a specified item in a program or subprogram compiled with the /DEBUG switch. Type the command in response to the debugger prompt:

BREAK [arg]

where:

arg is an optional command argument or breakpoint. See Table 1–6.

Table 1–6: BREAK Command Formats

Command	Meaning
BREAK	Specifies a command with no arguments that sets a breakpoint at each program line number. BASIC stops execution at each line number and reissues the # prompt.
BREAK n	Stops execution at line n and gives the prompt when it finds that line number in any module with debugging enabled.
BREAK n;	Specifies that line number n is a breakpoint only in the currently executing program or subprogram that has debugging enabled.
BREAK n;name	Specifies that line number n is a breakpoint only in the named program or subprogram.

You can specify a maximum of ten breakpoints as arguments in the BREAK command. If you specify several arguments, separate them with semicolons or commas. For example:

```
# BREAK 10, 300; 310;PROC, 60  RET
```

stops execution at these points:

1. Line 10 when found in a /DEBUG enabled program, regardless of whether it is the main program or a subprogram.
2. Line 300 in the currently executing program.
3. Line 310 in the program named PROC.
4. Line 60 when found in a /DEBUG enabled program, regardless of whether it is the main program or a subprogram.

If you specify more than ten breakpoints, BASIC prints the error message:

```
No room
```

To disable the breakpoints, use the UNBREAK command. Table 1–7 summarizes UNBREAK formats.

Table 1–7: UNBREAK Command Formats

Command	Meaning
UNBREAK	Disables all breakpoints.
UNBREAK n	Disables the breakpoint set at all lines numbered n.
UNBREAK n;	Disables the breakpoint set at line number n in the current program or subprogram.
UNBREAK n;name	Disables the breakpoint set at line number n in the named program.

In addition to line number breakpoints, the BREAK command allows you to specify a stop on CALL statements, user-defined functions, and loops. The BREAK ON arguments for these stops are CALL, DEF, and LOOP. Their formats are:

BREAK ON $\left\{\begin{matrix}\text{CALL}\\\text{DEF}\\\text{LOOP}\end{matrix}\right\}$

where:

CALL stops execution each time BASIC executes a CALL statement to a subprogram compiled with the /DEBUG switch. The stop occurs immediately before execution of the subprogram's first statement.

DEF stops execution each time BASIC enters a user-defined function in a module compiled with the /DEBUG switch. The stop occurs immediately before the execution of the first statement in the function.

LOOP stops execution each time BASIC finds a FOR, WHILE, or UNTIL statement or modifier. Stops occur after the loop is initialized (but before execution of the loop body) and after exit from the loop.

The BREAK ON command allows you to specify one argument only. You can combine it with other breaks. For example:

```
# BREAK 45, ON CALL, 330;  RET
```

stops execution at these points:

1. Line 45 when found in a /DEBUG enabled program, regardless of whether it is the main program or a subprogram.
2. After a CALL to any subprogram compiled with the /DEBUG switch and immediately before the execution of the subprogram's first statement.
3. Line 330 in the currently executing program.

1.2.2.2 STEP Command — The STEP command causes execution on a statement-by-statement basis. You type the command in response to the debugger prompt:

STEP [n]

where:

STEP executes the next statement in the current program or subprogram.

n is an optional argument that specifies the statements to be executed. It must be a positive integer in the range 1 to 32767.

Like other debugging commands, STEP has effect only on programs or subprograms compiled with the /DEBUG switch. Therefore, the first statement executed by the STEP command is the first statement found in a /DEBUG enabled routine.

NOTE

Typing a carriage return in response to a # debugger prompt is the same as typing STEP without an argument.

1.2.2.3 PRINT and LET Commands — The PRINT and LET commands allow you to test and change the contents of variables in programs and subprograms compiled with the /DEBUG switch.

The PRINT command has the form:

PRINT var

where:

var is the variable you want to test. You can specify only one variable or argument. This command prints the current contents of the variable.

The LET command has the form:

LET var=value

where:

var is the variable you want to change. You can specify only one variable or argument. The PRINT and LET debugging commands allow constants or variables as arguments, but not expressions.

You cannot set string variables to null string with the LET command. You can, however, set a variable to the null string in your source program (for example: NUL$ = " "). Then, while the debugger runs, set another string variable equal to the null string. For example: LET A$ = NUL$.

1.2.2.4 TRACE and UNTRACE Commands — The TRACE command allows you to track by line number the execution of a program or subprogram compiled with the /DEBUG switch. You type the command in response to the debugger prompt:

TRACE

where:

TRACE causes the number of each line to print as the line executes.

To disable the TRACE command, type UNTRACE after the prompt.

1.2.2.5 ERR Command — The ERR command allows you to display the error number of the last error found. Type the command in response to the debugger prompt:

ERR

where:

ERR returns the number of the last error in the format:

ERR = nn

where nn is the decimal error number.

Refer to Appendix C for a list of errors and their numbers.

1.2.2.6 ERL Command — The ERL command allows you to display the line number of the last error found. Type the command in response to the debugger prompt:

ERL

where:

ERL displays the line number of the last error in the format:

ERL = nn

where nn is the line number containing the error.

1.2.2.7 ERN Command — The ERN command allows you to display the name of the module that was executing when the latest error was found. Note that ERN does not return a value unless an error has occurred. Type the command in response to the debugger prompt:

ERN

where:

ERN returns the name of the module containing the last trapped error, in the format:

ERN = mod nam

where mod nam is the one-to-six character module name.

1.2.2.8 EXIT Command — The EXIT command returns you from the debugger to the monitor level. Type the command in response to the debugger prompt:

EXIT

1.2.2.9 RECOUNT Command — The REC(OUNT) command allows you to display the number of characters transferred by the latest input operation. Type the command in response to the debugger prompt:

RECOUNT

where:

RECOUNT displays the number of characters returned by the last input statement, in the format:

RECOUNT = nn

where nn is the number of characters, including terminators, from the last input statement.

1.2.2.10 STATUS Command — The STA(TUS) command allows you to display the status word reflecting: (1) characteristics of the last opened file, or (2) additional RMS file information. Type the command in response to the debugger prompt:

STATUS

where:

STATUS returns a word that indicates the device containing the last opened file's characteristics. The format is:

STATUS = nn

where nn is encoded as described in the STATUS description in Chapter 3.

1.2.2.11 I/O BUFFER Command — The I/O Buffer command returns the number of words currently allocated for I/O buffer space. The format is:

I/O BUFFER

See "Memory Allocation" in Chapter 3 for more information on the allocation of I/O buffer space.

1.2.2.12 STRING Command — The STRING command returns the number of words currently allocated for string space. The format is:

STRING

See "Memory Allocation" in Chapter 3 for more information on the allocation of string space.

1.2.2.13 FREE Command — The FREE command returns the number of words currently available in free space. The format is:

FREE

See "Memory Allocation" in Chapter 3 for more information on how BASIC allocates free space.

1.2.2.14 CORE Command — The CORE command returns the number of words currently allocated for your task. The format is:

CORE

See "Memory Allocation" in Chapter 3 for more information on how BASIC allocates space for your task.

1.2.3 Running the Program

The following sections explain how to BUILD, compile, task build, and execute programs on the RSX family of operating systems. Because you cannot generate a task image directly from the source program, you must:

- Create one or more object modules with the COMPILE command.
- Generate a command file for the task builder with the BUILD command.
- Specify the command file in response to the task builder prompt.

For example:

```
RUN $BASIC2

PDP-11 BASIC-PLUS-2 V1.6 BL-01.60

NEW
NEW FILE NAME-SORT02

BASIC2

10   DIM SORT(100)                              ! MAX NUMBER OF ELEMENTS
20   INPUT "NUMBER OF ENTRIES"; CNT%            ! GET NUMBER OF ELEMENTS     &
\    IF CNT% < 2% OR CNT% > 100%                ! CHECK CORRECT NUMBER       &
         THEN PRINT 'LIMITS - 2 TO 100'         ! WRONG - INFORM USER        &
\            GO TO 20                           ! TRY AGAIN                  &
         ELSE INPUT SORT(I%) FOR I% = 1% TO CNT%
30   REM                                                                     &
              B U B B L E   S O R T                                          &
                                                                             &
     CHECK EACH PAIR OF ELEMENTS                                             &
     IF IN WRONG ORDER, SWITCH THEM                                          &
     SORT.FLG IS SET TO FALSE (0) WHEN A SWITCH IS MADE                      &
     PASS OVER THE ENTIRE LIST UNTIL NO SWITCH IS MADE                       &

31   .FLG% = 1%                          ! SET TO TRUE INITIALLY            &
\    WHILE SORT.FLG% <> 0%               ! LOOP UNTIL SORT .FLG IS FALSE    &
\         SORT.FLG% = 0%                 ! SET TO FALSE BEFORE PASS         &
\         FOR I%=1% TO CNT%-1%           ! LOOP THROUGH ENTIRE LIST         &
\              IF SORT(I%)<=SORT(I%+1%)  ! CHECK A PAIR                     &
                   THEN SORT.FLG%=-1%    ! IF WRONG-FORCE ANOTHER PASS      &
\                       T=SORT(I%)        ! SWAP ELEMENTS                    &
\                       SORT(I%)=SORT(I%+1%)
\                            SORT(I%+1%)=T                                  &
40               NEXT I%
50          NEXT
60          PRINT SORT(I%) FOR I%=1% TO CNT%     ! PRINT ELEMENTS IN ORDER
32767       END

SAVE

BASIC2

COMPILE

BASIC2

BUILD

BASIC2

EXIT

TKB @SORT02

RUN SORT02

NUMBER OF ENTRIES? 6
? 0
? -5.5
? 10
? 20
? -5.6
? -100
20            10          0          -5.5          -5.6          -100
```

The program accepts up to 100 numbers as input, sorts them, and prints them in descending order. Table 1–8 summarizes the command sequence.

Table 1–8: Command Sequence

Command	Explanation
NEW NEW FILE NAME—SORT02	Clears a space in the temporary buffer for creation of the source program. When you type NEW, you lose any source code then in memory. Typing SORT02 assigns that name to your program.
BASIC2	Indicates that the previous command has been successfully executed and that the compiler is ready to accept input.
SAVE	Copies the program and saves it as a source file with the extension B2S.
COMPILE	Converts your program into object code and adds the default extension .OBJ to the program name.
BUILD	Creates command (CMD) and ODL files that reference the libraries and options required for the task builder.
EXIT	Returns control to the operating system.
TKB @SORT02	Uses the indirect command file to create an executable task image of the program with the extension .TSK.
RUN	Executes the program.

The BUILD command file contains instructions for task builder input. For example:

```
SY:SORT02/CP=SY:SORT02/MP
LIBR=BASIC2:RO
UNITS = 14
ASG = TI:13
ASG = SY:5:6:7:8:9:10:11:12
EXTTSK= 832
//
```

The BUILD command also generates an ODL file that describes the program's overlay structure:

```
        .ROOT BASIC2-RMSROT-USER,RMSALL
USER:   .FCTR SY:SORT02-LIBR
LIBR:   .FCTR LB:[1,1]BASIC2/LB
@LB:[1,1]BP2ICO
@LB:[1,1]RMS11X
        .END
```

All operating systems follow a similar method for compiling, building, and linking, and running a program. Table 1–9 summarizes system-specific differences.

Table 1–9: System Differences and Program Execution

System	Specific Implementation
IAS	You invoke the compiler with: `RUN LB:[1,1]BASIC2`
IAS	Do not invoke the task builder explicitly (TKB). The first line of the command file has a line that calls for linking. You enter "@SORT02".
VMS	The command file has no LIBR line. VMS–Compatibility Mode does not support resident libraries.
VMS	Include the MCR command before the TKB command. For example: `MCR TKB @SORT02`
TRAX	Use the command LINK @SORT02 or LINK/BASIC SORT02 to create an executable task image. Do not use the TKB command.

Chapter 2
Memory Resident Libraries

Memory resident libraries let all BASIC-PLUS-2 users share OTS code. Shared code saves system memory space.

This chapter describes the BASIC2 Resident Library and its related object library.

2.1 BASIC2 Resident Library

The BASIC2 memory resident library enables all BASIC-PLUS-2 users to share 8KW of the BASIC-PLUS-2 Object Time System (OTS).

The BASIC2 library has these advantages:

- Maximum shareable code. The entire 8KW is shareable among all users, which minimizes the memory needs of your system.
- Faster linking. Because many of the modules are in the library, the task builder accesses the disk library less often.

The LIBRARY command enables you to use the memory resident library. For example:

```
LIBR
Name [BASIC2] --
Account [LB:[1,1]]--
```

The LIBRARY command shows that the memory resident library, BASIC2, is the current default. In addition to BASIC2, you can select NONE (for no resident library) or your own resident library. User names require full file specifications. See Chapter 1 for more information.

2.2 BASIC–PLUS–2 Object Libraries

An object library is a disk-resident collection of object modules the task builder links to your program. The task builder links only those modules needed for program execution, or those explicitly referenced in the ODL command file.

BASIC–PLUS–2 provides two object libraries: BASIC2.OLB and BASRMS.OLB. The task builder uses these libraries to locate routines not found in a memory resident library (if you use one). Use the DSKLIB command to specify the object library you want to link with. For example:

```
DSKLIB
File spec [LB:[1,1]BASIC2]--
```

2.2.1 BASIC2 Object Library

The BASIC2 object library contains all BASIC–PLUS–2 routines except those that interface to RMS.

2.2.2 BASRMS Object Library

The BASRMS object library provides software routines to interface BASIC–PLUS–2 and RMS. Adding /VIR, /SEQ, /REL, or /IND to the BUILD command references BASRMS indirectly. When using the BASRMS library, you should also specify an RMSODL file to match program requirements and the RMS options available to you. For example:

```
ODLRMS
File spec [LB:[1,1]RMS11X]--
```

In this case, the BUILD command references the BASRMS object library to provide BASIC–PLUS–2 access to RMS code for all RMS file organizations. The RMS11X.ODL file overlays code from the RMS disk library (RMSLIB.OLB) and makes its routines available for file and record operations. The task builder can now extract the needed RMS modules.

Chapter 3
Files

This chapter explains BASIC–PLUS–2 file organizations and operations. For a thorough understanding of file organization and file and record operations, see the *RMS–11 User's Guide*.

3.1 Introduction to BASIC–PLUS–2 Files

BASIC supports RSX and RMS–11 file organizations. RSX provides support for device-specific file I/O. RMS–11 provides these file organizations:

- Terminal-format
- Block I/O
- Virtual array
- Sequential
- Relative
- Indexed

BASIC associates each file with a distinct channel number when you OPEN it. These channel numbers are integers between one and twelve. Your terminal is always channel zero. After you open the file, you make all references to it with the channel numbers.

BASIC does not check the file designation you specify. Before opening the file, check all file specification syntax for your operating system, including device names, upper case letters, and so forth.

BASIC stores data in physical records, or blocks. A block is the smallest number of bytes BASIC transfers in a read or write operation. On disk, a block is 512 bytes. On magnetic tape, it is between 18 and 8192 bytes; 512 is the default.

BASIC stores one or more data records (logical records) in each block. A data record is a group of fields your program treats as a unit. Data records can be less than, equal to, or greater than blocks.

3.1.1 Native File Organizations

BASIC-PLUS-2 works with the RSX operating system to provide an interface for device-specific I/O. Device-specific I/O is used for:

- Non-file structured data input and output (QIO interface).
- Accessing terminals (for example, opening a terminal as a file).

3.1.2 RMS File Organizations

3.1.2.1 Terminal-Format Files — Terminal-format files store ASCII characters sequentially in variable-length records with the implied carriage return (RET) attribute. You access terminal-format files with INPUT and PRINT statements.

3.1.2.2 Block I/O Files — Block I/O files are sequential or random access files that contain a series of blocks. BASIC treats each block as one record containing a stream of characters (data). You access the data with GET or PUT statements. Your program specifies the location and format of data in the block with MAP and MOVE statements.

3.1.2.3 Virtual Array Files — Virtual arrays provide a simple disk storage structure for small data bases. You can access them like arrays in memory—randomly or sequentially. Virtual arrays can contain integer, real, or string data referenced by array name and subscript.

3.1.2.4 Sequential Files — A sequential file contains logically contiguous records stored in the order that they were written. Your program specifies the record format, and GET and PUT statements access data. You usually read a sequential file from beginning to end only. Therefore, sequential files are most useful when you access the data in the file sequentially each time you use it.

3.1.2.5 Relative Files — A relative file contains a series of cells. Each cell contains a single record. For fixed length records, the length of each cell equals the record length plus one byte. For variable length records, the length of the cell equals the maximum record size plus three bytes. Your program: (1) determines the record format, and (2) interprets the data and its divisions inside each record. You access data with GETs and PUTs, either randomly by cell number, or sequentially by omitting cell numbers.

Relative files are most useful when: (1) randomly accessed, and (2) record contents correspond to cell numbers (for example, when inventory numbers correspond to cell numbers).

3.1.2.6 Indexed Files — Indexed files contain data records sorted in ascending order by primary index key value. They can also contain one or more alternate indexes. Your program determines the format of data records, and GETs and PUTs access the data randomly by specific key value, or sequentially according to ascending key value.

Indexed files are most useful when: (1) randomly accessed, and (2) you want to access the records in more than one way. For example, you can specify index keys to access a file by one of several record fields.

3.1.3 Record Format Types

The record format determines how RMS stores a record in the block. You specify record format in the OPEN statement. Select one of three formats:

1. Fixed length
2. Variable length
3. Stream

Fixed-length records are all the same length. RMS stores fixed-length records as they appear in the record buffer, including padded spaces after the record. Processing these records involves less overhead than other record formats. However, this format uses storage space less efficiently than variable-length records.

Variable-length records can have different lengths. No record can exceed a maximum size you set for the file. RMS adds a one-word record length header to each record. This count gives the length of the record in bytes. It is transparent to your program, and the record buffer does not include it.

While variable-length records usually make more efficient use of storage space, the record length headers generate processor overhead.

Specifying variable-length format for relative files does not save disk space. BASIC always writes a fixed length record to the file. Specifying variable length records will, however, help prevent unwanted data from being displayed when you access a record.

A stream-format file contains a series of contiguous ASCII characters. The following terminators end a stream-format record:

- Carriage return-line feed
- CTRL/Z
- Escape
- Form feed
- Line feed

Use stream format records on RMS sequential files only.

Stream format files use storage space efficiently because they store each character contiguously. However, they generate the most processing overhead, because the system must test each character to see if it terminates the record.

The file organization you select determines the record formats available to you. Sequential files permit all three formats. Relative and indexed files do not permit stream format; terminal-format files are in sequential variable format.

NOTE

RSX recognizes the carriage return (<CR>) and escape (<ESC>) line terminators only.

3.1.4 Opening a File (OPEN Statement)

Opening a file with the OPEN statement: (1) creates new files, and (2) makes records available for processing. This section presents the general format of the OPEN statement. Specific syntax appears with the description of each file type.

The OPEN statement defines all aspects of a file OPEN operation, including the structure of the file and its file-sharing restrictions. OPEN syntax includes keywords that describe file attributes. These attributes are usually followed by a name or numeric expression and separated by commas.

BASIC supplies no file specification defaults. If you do not specify a device name, the channel remains assigned to: (1) the device used in a previous OPEN on that channel, or (2) SY: as specified in the task builder command file. In addition, BASIC does not recognize logical UICs, file names, or file types, and all file names must be in upper case.

The format for the OPEN statement is:

```
OPEN filespec-exp[ FOR INPUT  ] AS FILE [#]num-exp%
                 [ FOR OUTPUT ]

,[ORGANIZATION] SEQUENTIAL [ FIXED    ]
                RELATIVE   [ VARIABLE ]
                INDEXED    [ STREAM   ]
                UNDEFINED
                VIRTUAL

[,ACCESS READ   ]
        WRITE
        MODIFY
        SCRATCH
        APPEND  ]

[,ALLOW NONE    ]
        READ
        WRITE
        MODIFY  ]

[,MAP mapname]
[,RECORDSIZE num-exp]
[,SPAN ]
[,NOSPAN]
[,BLOCKSIZE num-exp%]
[,FILESIZE num-exp%]
[,BUCKETSIZE num-exp%]
[,TEMPORARY]
[,WINDOWSIZE num-exp]
[,NOREWIND]
[,CONTIGUOUS]
[,CONNECT num-exp%]
[,BUFFER num-exp%]
[,MODE num-exp%]
[,PRIMARY [KEY] name[ DUPLICATES   ]]
                    [ NODUPLICATES ]

[,ALTERNATE [KEY] name[ [DUPLICATES] [CHANGES]   ]]
                      [ [NODUPLICATES] [NOCHANGES] ]
```

NOTE

The ORGANIZATION clause must be the first attribute specified. If you specify no ORGANIZATION clause, you can perform native mode I/O operations only.

where:

filespec-exp	is an RSX file specification.
FOR INPUT	OPENs an existing file for modify operations.
FOR OUTPUT	creates a new file with the name you specify, or destroys an existing data file of the same name.

NOTE

The FOR INPUT and FOR OUTPUT clauses have no affect on how your program can use the file or how others can share it.

[#]num-exp%	associates the file with a channel number between 1 and 12. Channel number 0 is your terminal, and cannot be opened.
,[ORGANIZATION] VIRTUAL	specifies a block I/O or virtual array file.
,[ORGANIZATION] SEQUENTIAL	specifies an RMS file of sequential records stored in the order they were written.
,[ORGANIZATION] RELATIVE	specifies an RMS file of fixed-length record cells that stores records by physical location.
,[ORGANIZATION] INDEXED	specifies an RMS file of records sorted in ascending order by primary key value and one or more indexes that point into the records.
,[ORGANIZATION] UNDEFINED	specifies an existing file whose organization is not known. You must open the file FOR INPUT only.
[FIXED]	specifies fixed-length records.
[VARIABLE]	specifies variable-length records. This is the default format for all RMS file organizations.
[STREAM]	specifies ASCII stream records for RMS sequential files.
[,ACCESS {READ / WRITE / MODIFY / SCRATCH / APPEND}]	specifies the record operations you can perform on the file. These are summarized in Table 3–1.

Table 3-1: File Access Specifications

Access Mode	RMS Sequential Files	RMS Relative Files	RMS Indexed Files
Unspecified	GET PUT UPDATE	GET PUT UPDATE DELETE	GET PUT UPDATE DELETE
READ	GET	GET	GET
WRITE	PUT	PUT	PUT
MODIFY	GET PUT UPDATE	GET PUT UPDATE DELETE	GET PUT UPDATE DELETE
SCRATCH	GET PUT UPDATE TRUNCATE	None	None
APPEND	PUT EOF (End-of-File)	None	None

[,ALLOW { NONE | READ | WRITE | MODIFY }]

specifies what other users can do to the file when you are using it. NONE and READ specify a file that others cannot write to. This is the default for relative and indexed files. WRITE and MODIFY allow shared read and write access. You cannot share RMS sequential files for writing.

[,MAP mapname]

references a MAP statement. The map you reference declares the primary divisions of data in the record (by type and size) and the size of the record buffer.

[,RECORDSIZE num-exp]

defines the maximum record size (in characters) in the file. You must specify RECORDSIZE if you do not specify a MAP clause.

NOTE

If you specify both a MAP and a RECORDSIZE, RECORDSIZE overrides the record buffer size BASIC calculates from the MAP. If the RECORDSIZE is larger than the MAP, a fatal run-time error occurs.

[,SPAN] [,NOSPAN]	signifies whether records in an RMS sequential file can cross block boundaries. The default is SPAN.
[,FILESIZE num-exp%]	allocates an integer number of disk blocks when you create a file.
[,TEMPORARY]	opens a temporary file that is deleted when you close the file.
[,BLOCKSIZE num-exp%]	defines the number of records in a block on magnetic tape.
[,BUCKETSIZE num-exp%]	specifies the number of records in each bucket. RMS relative and indexed files only.
[,NOREWIND]	specifies that the magnetic tape volume containing the file is not to be rewound before you open or create the file.
[,CONTIGUOUS]	specifies physically adjoining disk blocks for file storage.
[,WINDOWSIZE num-exp]	specifies the number of retrieval pointers to be used for the file.
[,CONNECT num-exp%]	establishes additional record access streams that allow your program to process (in parallel) more than one record of a file. Each stream represents an independent and concurrently active sequence of record operations. The numeric argument (num-exp%) is the original channel number for the file. Each connect established in an OPEN statement uses an I/O channel. Because there are 12 I/O channels available, you can have a maximum of 11 connects to a file. RMS relative and indexed files only.
[,BUFFER num-exp%]	specifies the number of I/O buffers maintained for indexed file keys.
[,MODE num-exp%]	specifies MODE values for files. BASIC ignores this attribute except for magnetic tape files.
[,PRIMARY [KEY]]	specifies a MAP statement field as the Primary Key for an indexed file. The primary key appears first in the MAP statement, and is key 0 in the file. Use strings only. For primary keys, you can specify DUPLICATES, but not CHANGES.
[,ALTERNATE [KEY]]	allows you to define the names of 1 to 254 alternate keys for indexed files. Alternate keys must also be strings.

[NODUPLICATES] [DUPLICATES]	allows an indexed file to contain more than one record with the same key value. NODUPLICATES is the default.
[CHANGES] [NOCHANGES]	allows you to change the key value for alternate keys only. NOCHANGES is the default. The combination CHANGES and NODUPLICATES is invalid.

3.1.5 File Operations

3.1.5.1 Completing File I/O (CLOSE Statement) — All programs should close files before terminating. However, BASIC automatically closes files:

- While executing a CHAIN statement
- At an END statement
- When it completes the highest numbered line in the program
- When you open another file on the same channel

BASIC does not close files after executing a STOP or SUBEND statement.

The CLOSE statement has the format:

CLOSE [#] file-expressions

where:

file-expressions	are one or more channel numbers separated by commas. If you do not specify expressions, BASIC returns a syntax error.

For example:

```
10 CLOSE #1%                                ! Closes the file associated with file #
20 B% = 4%
30 CLOSE #2%, B%, 6% + 1%                   ! Closes file numbers 2, 4, and 7
40 CLOSE #I% FOR I% = 12% to 1% STEP-1%     ! Closes all file numbers
```

The CLOSE statement closes files and disassociates these files and their buffers from the file numbers. Because BASIC allocates I/O buffer space from the dynamic area, you should close files in reverse order: close the last opened file first, and so forth. This returns buffer space to the free space area and makes it available for string or I/O use. See "Memory Allocation" for more information.

3.1.5.2 Renaming Files (NAME AS Statement) — You can change the name of a file with the NAME AS statement if the protection code permits. The format is:

NAME "string1" AS "string2"

where:

string1 is the old file name.

string2 is the new file name.

For example:

```
10 NAME "MONEY.DAT" AS "ACCNTS.DAT"
```

changes the name of the file named MONEY.DAT to ACCNTS.DAT.

Do not omit file extensions. There is no default. If you use the NAME AS statement on an open file, the new name does not take effect until you close the file.

Because BASIC uses the NAME AS statement as a native-mode operation, it is subject to the operating system file naming rules. The *System User's Guide* explains these rules.

3.1.5.3 Deleting a File (KILL Statement) — You can delete a file with the KILL statement if the protection code permits. The KILL statement has the format:

KILL filespec

where:

filespec is the file specification of the file you want deleted. You can delete only one file at a time. Do not omit file extensions. There is no default.

For example:

```
610 KILL "TEST.BP2"
```

deletes the file TEST.BP2. KILL takes effect when all programs that have opened the file close it. You cannot open or access a file after you have deleted it. However, others using the file when you KILL it can continue to use it.

Because BASIC-PLUS-2 uses KILL as a native mode file operation, it is subject to operating system restrictions. Refer to the *System User's Guide*.

3.1.5.4 Truncating Records (SCRATCH Statement) — The SCRATCH statement deletes RMS sequential file records from the current record to the file's end. You must open the file with ACCESS SCRATCH. SCRATCH has the format:

SCRATCH [#]file-number

where:

file-number is the channel number of an open RMS sequential file.

For example:

```
50 SCRATCH #7%
```

deletes all records beginning with the current record. The file's physical length remains the same after a SCRATCH.

3.1.5.5 Restoring Files (RESTORE # Statement) — The RESTORE # statement returns the current record pointer to the beginning of the file. RESTORE does not change the file. The RESTORE # statement has the format:

RESTORE #file-expression [,KEY # num-exp]

where:

file-expression	is the channel number of the file you want to restore.
,KEY # num-exp	resets an RMS indexed file by key of reference. KEY # 0 is the primary key, KEY # 1 is the first alternate, and so forth. When you RESTORE an indexed file, specify the key you want restored. The default is the primary key.

For example:

```
70 RESTORE #3%, KEY #2%
```

All RMS file organizations can use the RESTORE statement.

RESTORE without a channel number resets the data pointer for READ and DATA statements and does not affect any files.

3.2 Terminal-Format Files

Terminal-format files are RMS sequential variable files that store ASCII characters in a record size you specify. The default record size equals the terminal width. Each record ends with an implicit [CR] as the line terminator, and is stored exactly as printed on the terminal.

3.2.1 Opening a Terminal-Format File

This syntax opens a terminal-format file:

OPEN filespec-exp [FOR INPUT / FOR OUTPUT] AS FILE [#]num-exp%

[,ACCESS { READ / WRITE / MODIFY }]

[,WINDOWSIZE num-exp]

[,FILESIZE num-exp%]

[,CONTIGUOUS]

[,RECORDSIZE num-exp]

Filespec follows the rules for full RSX file specifications. See your *System User's Guide* for more information.

3.2.2 Record Operations

You write records to terminal-format files with PRINT # and PRINT # USING statements. You read records with INPUT #, INPUT LINE #, and LINPUT # statements.

3.2.2.1 Writing Records to the File (PRINT # and PRINT # USING)

3.2.2.1.1 PRINT # — Use PRINT # to write single records (lines of data) to the file. The program:

```
10 OPEN "TEXT.FIL" AS FILE #2%
20 WHILE A$<>"FINI"
30 LINPUT A$
40 PRINT #2%, A$
50 NEXT
60 CLOSE #2%
70 END
```

prompts you for a line of text and stores it in the file. The loop at line 50 repeats the prompt until "FINI" signifies the end of file.

3.2.2.1.2 PRINT # USING — PRINT # USING stores formatted data. For example:

```
10 OPEN "FILE.DAT" AS FILE #1%
60 PRINT #1% USING "###.#", 456
80 PRINT #1% USING "# # #", 1, 2, 3
90 CLOSE #1%
99 END
```

stores the data like this:

```
 456.0
 1 2 3
```

The program line:

```
60 PRINT #2% USING "'LLLLL", "CONTINUE"
```

prints "CONTINUE", left justified and truncated beyond six characters. See the *BASIC-PLUS-2 Language Reference Manual* for information on PRINT USING formats.

Every PRINT operation sets a counter for the CCPOS function; CCPOS marks the character position of an output line in the record buffer. See section 3.12.4 for information on CCPOS.

3.2.2.2 Reading Records from the File — INPUT #, INPUT LINE #, LINPUT#, MAT INPUT #, and MAT LINPUT # access data sequentially in terminal-format files. Table 3–2 summarizes these statements.

Table 3-2: Terminal-Format File Input Statements

Statement	Function
INPUT #	Reads a record and assigns it to a specified program variable.
INPUT LINE #	Reads a line of text, including the line terminator, and assigns it to a string variable.
LINPUT #	Reads a line of text, but without the line terminator, and assigns it to a string variable.
MAT INPUT #	Reads records and assigns them to elements of an array.
MAT LINPUT #	Reads lines of text, without the line terminator, and assigns them to elements of a string array.

Every input operation sets the RECOUNT variable. RECOUNT contains the number of characters read by the last input. See section 3.12.3 for more information.

When the program accesses a record, you can tell BASIC to print that record at your terminal. For example, the program:

```
10  OPEN "FI.DAT" FOR OUTPUT AS FILE #1%
20  PRINT #1%, 23;",";"STRINGB"
30  PRINT #1%, "STRINGA" + CHR$(13%)
40  PRINT #1%, "STRINGC"
50  CLOSE #1%
60  OPEN "FI.DAT" FOR INPUT AS FILE #1%
70  INPUT #1%, A,B$\PRINT A,B$
80  INPUT LINE #1% A$\PRINT A$
90  LINPUT #1%, C$\PRINT C$
100 CLOSE #1%
110 END
```

writes three variable-length records to the file, accesses them, and outputs them to the terminal. The file looks like this:

```
 23 ,STRINGB<RET><LF>
STRINGA<RET><RET><LF>
STRINGC<RET><LF>
```

and the printed output like this:

```
 23             STRINGB
STRINGA

STRINGC
```

In this program, BASIC includes the comma (line 20) in the file. However, the INPUT statement at line 70 interprets the comma as a line terminator, and so BASIC prints the record as:

```
23             STRINGB
```

The MAT LINPUT # statement reads string data from the file until a specified string array is filled. You can then print the array and display the data. For example:

```
70  DIM TEST.SITES$(125%)
80  OPEN "EXP.FIL" FOR INPUT AS FILE #6%
90  MAT LINPUT #6%, TEST.SITES$
100 MAT PRINT TEST.SITES$,
110 CLOSE #6%
120 END
```

reads in lines of string data to the array TEST.SITES$ and prints the list in separate print zones.

Terminal-format files enable you to read records one at a time:

```
10 OPEN "TEXT.FIL" FOR INPUT AS FILE #4%
20 LINPUT #4%, A$ \ PRINT A$
30 INPUT "NEXT RECORD";B$
40 IF B$ = "YES" THEN 20
50 CLOSE #4%
60 END
```

or output an entire file:

```
 5 ON ERROR GOTO 40
10 OPEN "TEXT.FIL" FOR INPUT AS FILE #4%
20 LINPUT #4%, A$ \ PRINT A$
30 GO TO 20
40 IF ERR = 11% THEN RESUME 50 ELSE ON ERROR GOTO 0
50 CLOSE #4%
60 END
```

3.3 Block I/O Files

Block I/O files access data in units of one or more 512 character records. These blocks are a series of logically contiguous records, accessed sequentially or randomly by record (block) number.

Your program must define data and block and deblock records. For more information, see Section 3.8.

3.3.1 Opening a Block I/O File

This syntax opens a block I/O file:

OPEN filespec-exp [FOR INPUT | FOR OUTPUT] AS FILE #num-exp%

,[ORGANIZATION] VIRTUAL

[,ACCESS {READ | MODIFY}]

[,ALLOW {NONE | MODIFY}]

[,RECORDSIZE num-exp]

[,WINDOWSIZE num-exp]

[,MAP mapname]

[,CONTIGUOUS]

[,FILESIZE num-exp%]

[,MODE num-exp%]

Filespec follows the rules for full RSX-11M file specifications, as defined in the *RSX-11M System User's Guide.*

3.3.2 Record Operations

Block I/O files use PUT and GET statements to write and access data.

3.3.2.1 Writing Data to the File (PUT) — The PUT statement writes data from the record buffer to the file. Use PUT with the channel number and record number. For example:

```
PUT #12%, RECORD 18%
```

writes the contents of the record buffer into the 18th record of the file.

For sequential entry into the file, omit the RECORD clause. Successive PUTs write sequential records. PUT overwrites existing records, and BASIC does not return an error message if a record already exists.

3.3.2.2 Reading Data from the File (GET) — The GET statement reads data from the disk into the record buffer. Type GET with the channel number and the record number. For example:

```
100 GET #3%, RECORD 7%
```

reads the seventh record (record 7%) into the buffer. For sequential access, omit the RECORD clause. Successive GETs read successive records.

After a successful GET, the RECOUNT variable contains the number of characters read. See Section 3.12.3 for information on RECOUNT.

3.4 Virtual Array Files

Virtual arrays exist only on disk, and allow array operations when an array is too large to fit in memory. You can store data in array elements and access that data as you would an array in memory.

Virtual array files assume these definitions:

- A "list" is a one dimensional array.
- A "matrix" is a two dimensional array.
- A "row" is data arranged horizontally.
- A "column" is data arranged vertically.

3.4.1 Opening a Virtual Array File

You must use the DIM # statement with virtual array files. The DIM # statement:

- Associates the array with a channel number.
- Defines the types of data you store.
- Defines where the record is located in the block, and therefore how you can block or deblock records.

The OPEN statement defines the file attributes.

This syntax opens a virtual array file:

```
DIM # num-exp%, array(s)[ = number]
OPEN filespec-exp[{FOR INPUT  }] AS FILE #num-exp%
                 [{FOR OUTPUT}]
,[ORGANIZATION] VIRTUAL
[,ACCESS{READ  }]
        {MODIFY}
[,ALLOW{NONE  }]
       {MODIFY}
[,WINDOWSIZE num-exp]
[,CONTIGUOUS]
[,FILESIZE num-exp%]
```

Filespec follows the rules for full RSX file specifications, as defined in the *System User's Guide.*

3.4.2 Dimensioning the Array (DIM # Statement)

When opening a virtual array file, you must describe the arrays in the file with the DIM # statement. Place the DIM # statement before the OPEN statement in your program. You cannot specify a DIM # statement as part of a conditional expression. The DIM # format is:

```
DIM # num-constant, array(s)[ = number]
```

where:

num-constant	is the channel number associated with the virtual array file.
array(s)	is a 1 or 2 dimensional subscripted array. Separate multiple arrays with commas.
number	is the maximum length of a string array. This value must be a power of 2. The default is 16 characters.

The "num-constant" associates the DIM # statement with the file. For example, the array:

```
DIM #6, A(750)
```

dimensions this virtual array file:

```
20 OPEN 'VIR.DAT' AS FILE #6%    &
        ,ORGANIZATION VIRTUAL
```

The maximum size for a list is 32766; the maximum size for a matrix is 32766 by 32766.

The DIM # statement can also provide multiple arrays for the same file. For example:

```
10 DIM #2%, A(15,20), B(50), C$(18) = 8%
```

dimensions three arrays: (1) matrix A, with space for 336 numeric elements, (2) list B, with space for 51 numeric elements, and (3) Array C$, with space for 19 string elements, each 8 characters long.

The array dimensions in the DIM statement start at the beginning of the opened file. Therefore:

```
100 DIM #1%, A(100), B(100)
```

and

```
10 DIM #1%, A(100)
20 DIM #1%, B(100)
```

do not perform the same function. Line 100 allocates two arrays of 101 elements (the 100 shown plus the 0 element) on channel number 1. Lines 10 and 20 together allocate one array of 101 elements on file number 1. That array, however, has two names. You can reference the elements with either name.

You can specify a string length in virtual arrays. The default is 16 characters. If you specify a length, it should be a power of 2 (2, 4, 8, 16,...). BASIC rounds other numbers up to the next power of two and truncates strings longer than the actual length. Virtual array strings are left justified and null-filled.

BASIC does not pre-initialize virtual arrays. You can initialize an array with a program similar to:

```
10 DIM #1% A$(32766) = 32%
15 OPEN "PARTS.DAT" FOR OUTPUT AS FILE #12%
20 OPEN "ARR.FIL" FOR OUTPUT AS FILE #1%
30 A$(I%) = SPACE$(32%) FOR I% = 32766% TO 0% STEP -1%
40 PRINT "ARRAY INITIALIZED"
50 END
```

Initializing a file in reverse order forces all file overhead to occur at one time. The file system allocates space for the entire file, and no future extensions are necessary.

3.4.3 Record Operations

Virtual array files allow you to input values and access them as you do an array in memory. In addition, you can access array elements across subprograms. The following sections describe these operations.

3.4.3.1 Writing Data to the File — You store data in virtual array files by assigning numeric or string values to the array elements.

3.4.3.1.1 Assigning Single Array Elements (LET) — Use the LET statement to assign single array values. For example:

```
30 LET C%(3%,177%) = 485%
```

assigns the value 485 to element (3,177).

Because LET overwrites existing data, you can update array elements. For example:

```
60 LET A$(4%,32%) = "JONES"
```

enters JONES in element (4,32) and replaces the previous value.

3.4.3.1.2 Justifying Array Elements (LSET) and RSET) — You can also use the LSET and RSET statements when updating single array elements. These statements left justify (LSET) or right justify (RSET) string elements, and limit their size to a specified length. For example:

```
50 DIM #4%, C$(5%,10%)
60 INPUT "NAME"; B$
70 LSET C$(3%,7%) = B$
```

writes a new record (B$) into array element (3,7). The string B$ is left justified.

3.4.3.1.3 Assigning Values to All or Part of an Array — You can use a FOR/NEXT loop to write to selected array elements. For example:

```
10 DIM #12%, F%(6%,10%)
20 FOR I% = 5% TO 6%
30    FOR J% = 7% TO 10%
40       INPUT "PART NUMBER"; PART.NUM%
45       LET F%(I%,J%) = PART.NUM%
50    NEXT J%
60 NEXT I%
70 CLOSE #12%
80 END
```

writes data you input to elements (5,7), (5,8), (5,9), (5,10), (6,7), (6,8), (6,9) and (6,10). The other elements are unchanged.

A FOR/NEXT loop also writes data to the entire array. For example:

```
10 DIM #5%, C(4,225)
20 OPEN "ID.NUM" FOR OUTPUT AS FILE #5%
30 FOR BLDG.NUM% = 0% TO 4%
40    FOR TIME.CARD.NUM% = 0% TO 225%
50       INPUT "EMPLOYEE NAME"; Z$
55       LET C(BLDG.NUM%,TIME.CARD.NUM%) = Z$
60    NEXT TIME.CARD.NUM%
70 NEXT BLDG.NUM%
80 CLOSE #5%
90 END
```

requests employee names and stores them in row order. Because of BASIC's storage methods, accessing by row is more efficient.

You can read string records from a terminal-format file and write it to a string array with the MAT LINPUT # statement. For example:

```
90 MAT LINPUT #4%, RECORD.DAT$
```

reads string data from a terminal-format file open on channel 4 and reads the elements into array RECORD.DAT$.

3.4.3.2 Reading Data from the File — When you open a virtual array file FOR INPUT, the DIM # statement must specify the same data type and subscript values as those in the program that created the file. For example:

```
10 DIM #2%, F$(15%,50%)
20 OPEN "VRTARY.DAT" FOR INPUT AS FILE #2%     &
       ,ORGANIZATION VIRTUAL
```

opens the file "VRTARY.DAT" and associates it with channel #2. The DIM statement specifies the same subscripts used to create the file.

3.4.3.2.1 Reading Single Array Elements (LET Statement) —

You access array elements by assigning them to a variable, and accessing this variable in the program. For example:

```
60 LET EMP.NAME$ = A$(7%,12%)
```

assigns the array element (7,12) to the variable EMP.NAME$. You can then use the PRINT statement to display the value of the variable on your terminal:

```
70 PRINT EMP.NAME$
80 PRINT A$(7%,12%)
```

3.4.3.2.2 Reading All or Part of an Array (Loops) — You can use the LET statement in a FOR/NEXT loop to access all or part of an array. For example:

```
20 DIM #2%, A$(15%,25%) = 64%
30 FOR I% = 5% TO 10%
40    FOR J% = 0% TO 25%
50    PRINT A$(I%,J%)
55 LET A$(I%,J%) = ""
60    NEXT J%
70 NEXT I%
```

accesses and prints the 6th to 11th rows of array A$ and sets the original data to null. By changing line 30:

```
30 FOR I% = 0% TO 15%
```

you can access and print the entire array.

You can display the contents of an array on your terminal with the MAT PRINT statement. Similarly, you can write the array to a terminal format file with the MAT PRINT # statement.

3.4.4 Using Multiple Arrays

You can use one array to index data in another. This permits faster access to records. For example, the following program accesses employee badge numbers stored in one virtual array file, and the corresponding employee's master record stored in another. Searching the badge number file locates records faster than searching the master file, because the badge number records are shorter.

```
10    DIM #1, THE.BADGE.NUMBER.ON.FILE%(1000)
20    DIM #2, THE.MATCHING.EMPLOYEE$(1000) = 32%
30    OPEN "BADGE.VRT" FOR INPUT AS FILE #1%, VIRTUAL
40    OPEN "EMPL.VRT" FOR INPUT AS FILE #2%, VIRTUAL
50    CANT.FIND.EMPLOYEE% = 1%
60    PRINT "What is the badge number";
70    INPUT THE.BADGE.NUMBER.I.WANT%
80    FOR I% = 1% WHILE I%< 1001%
90         IF THE.BADGE.NUMBER.ON.FILE%(I%) =          &
                 THE.BADGE.NUMBER.I.WANT%              &
      THEN PRINT "The employee's name is ";            &
                 THE.MATCHING.EMPLOYEE$(I%)            &
\                        I% = 1001%                    &
\                        CANT.FIND.EMPLOYEE% = 0%
100   NEXT I%
110   IF CANT.FIND.EMPLOYEE%                           &
           THEN PRINT "No such employee"
120   CLOSE #2%,1%
32767      END
```

The program requests a badge number. It then searches the array dimensioned in line 10 until it finds a matching badge number. The program then prints the master file record that corresponds to the subscripts of the badge number record. If the badge number is not found, the program prints the message: "No such employee."

3.4.5 Accessing Virtual Arrays across Subprograms

The DIM # statement declares the array and allows compile-time definitions of the elements. Because you compile subprograms separately, each must make data definitions and associate itself with the file.

Once you open a virtual array file in a main program, you can reference that array in subprograms. For example:

```
10 DIM #2%, A$(10%)
20 OPEN "VATEST.DAT" AS FILE #2%
30 A$(I%) = SPACE$(10%) FOR I% = 10% TO 0% STEP -1%
40 CALL SUBVAT
50 PRINT A$(3%)
32767 END
```

dimensions the array A$ and opens the file "VATEST.DAT". Line 30 initializes the array, and line 40 calls the subprogram SUBVAT. The SUBVAT program:

```
10 SUB SUBVAT
20 DIM #2%, B$(10%)
30 B$(3%) = "SUNK"
40 SUBEND
```

- Restates the DIM statement at line 20.
- Assigns the string SUNK to array element B$(3%)—which is actually A$(3%).
- Returns control to the main program.

You must specify the DIM # statement in all subprograms accessing the virtual array. Otherwise, statements will reference an array in memory.

3.5 RMS Sequential Files

Sequential files contain virtually contiguous records stored in the order in which they were written. You can specify fixed, variable, or stream format records.

Sequential files permit BASIC dynamic buffering or user buffering. See Section 3.8.

3.5.1 Opening an RMS Sequential File

This syntax opens a sequential file:

```
OPEN filespec-exp{FOR INPUT  }{AS FILE   #num-exp%
                 {FOR OUTPUT }

,[ORGANIZATION] SEQUENTIAL[{FIXED   }]
                          {VARIABLE}
                          {STREAM  }

[,ACCESS{READ  }]
        {WRITE }
        {MODIFY}
        {SCRATCH}
        {APPEND}]

[,MAP mapname]

[,RECORDSIZE num-exp]

[,WINDOWSIZE num-exp]

[,BLOCKSIZE num-exp%]

[,SPAN ]
[,NOSPAN]

[,NOREWIND]

[,CONTIGUOUS]

[,FILESIZE num-exp%]

[,TEMPORARY]
```

You cannot share sequential files for writing, but you can share them for reading.

The following program opens a sequential file with user-controlled buffering. The record pointers point to the end of the file:

```
20  MAP (MAP1) NA.ME$ = 32%, DEPT.NUM%, SSN
30  OPEN "CASE.DAT" FOR INPUT AS FILE #5%     &
          ,ORGANIZATION SEQUENTIAL VARIABLE   &
          ,ACCESS APPEND, ALLOW NONE          &
          ,MAP MAP1
```

The MAP clause at line 30 references the MAP statement at line 20. The MAP statement defines the data types and declares the record size. The data record is one string, one integer, and one real number. The record size is the total of these fields, or 38 bytes. BASIC statically allocates a record buffer 38 bytes long; all record operations use this buffer for I/O to the file.

The following program opens a sequential file for reading only (ACCESS READ), with a dynamically assigned record buffer:

```
10 OPEN "CASE.DAT" FOR INPUT AS FILE #1%    &
       ,ORGANIZATION SEQUENTIAL VARIABLE    &
       ,ACCESS READ                         &
       ,RECORDSIZE 100%
```

BASIC allocates an area for the record buffer out of the program's free space. Your program must then use MOVE TO and MOVE FROM statements to move data elements to and from the record buffer.

3.5.2 Record Operations

BASIC permits four record operations on sequential files: PUT, FIND, GET and UPDATE.

3.5.2.1 Writing Records to the File (PUT) — The PUT statement transfers data from the record buffer to the file. Use PUT to write records for the first time. You can change records after writing (and then reading) them with the UPDATE statement. Because sequential files store records in the order in which they were written, you can write new records only at the end of the file. You can go directly to the end of the file by specifying ACCESS APPEND in the OPEN statement, or by executing FIND or GET operations until you receive the "?End of File" error message.

To add records, type PUT with the channel number. For example:

```
110 PUT #3%
```

writes the next record. If you are not at the end of the file, you receive the error message "?Not at end of file" (ERR = 149). After a PUT operation, there is no current record. The next record pointer is set to the end-of-file.

When processing variable-length records, you can use the COUNT clause to specify the number of bytes to be written. For example:

```
110 PUT #3%, COUNT 60%
```

writes a 60-byte record to the file opened on channel 3. Without the COUNT clause, BASIC writes a record equal to the MAP or RECORDSIZE clause.

3.5.2.2 Locating Records in the File (FIND) — FIND locates records but does not move them into the record buffer. You can use FIND to check if a record exists, and adjust the current record pointer so you can GET or UPDATE that record. Successive FINDs locate successive records. For example:

```
10 OPEN "EXAM.PLE" FOR INPUT AS FILE #8%     &
        ,SEQUENTIAL VARIABLE
20 FIND #8% FOR I% = 1% TO 50%
30 GET #8%
```

locates the 50th record and GETs it. An error message indicates that you have reached the end-of-file (EOF).

FIND locates records faster than GET. Although both locate the record and update the current and next record pointers, GET also moves the data into the record buffer.

3.5.2.3 Reading Records from the File (GET) — The GET statement reads a record from the file into the record buffer. Type GET with the channel number. For example:

```
30 GET #7%
```

reads the record specified by the next record pointer unless the previous record operation was a successful FIND. If you found the record, GET reads that record.

A successful GET sets the current record pointer to the record read and the next record pointer to the current record plus 1. Successive GETs read successive records.

Your program can read a file and end the program after the last record. For example:

```
00010     ON ERROR GO TO 19000
00015     MAP (DAT) PROD.NAM$ = 30%, NUM%, REQ.CODE$
00020     OPEN "RST.DAT" FOR INPUT AS FILE #1%          &
          ,ORGANIZATION SEQUENTIAL, MAP DAT
00030     GET #1%
00040     PRINT "THE PRODUCT NAME IS", PROD.NAM$
00050     PRINT "THE CALL NUMBER IS", NUM%
00060     PRINT "THE REQUISITION CODE IS", REQ.CODE$
00070     GO TO 30
19000     IF (ERR = 11%) AND (ERL = 30%)                &
          THEN RESUME 19010                             &
          ELSE ON ERROR GO TO 0
19010     PRINT "END OF FILE"
19020     CLOSE #1%
32767     END
```

3.5.2.4 Replacing Records in the File (UPDATE) — The UPDATE statement replaces an existing record at the position indicated by the current record pointer. However:

- The file containing the record must be on disk.
- The new record must be the same size as the one it is replacing.
- The record format cannot be STREAM.
- A successful FIND or GET must position the target record before an UPDATE. The error message "?Record not found" (ERR = 155) indicates that the record you specified does not exist.

For example:

```
10      ON ERROR GOTO 19000
30      MAP (AAA) L.NAME$ = 60%, F.NAME$ = 20%, RM.NUM$ = 8%
40      OPEN "STU.DAT" FOR INPUT AS FILE #9%,                    &
        SEQUENTIAL, MAP AAA
50      INPUT "LAST NAME"; SEARCH.NAME$
60      SEARCH.NAME$ = SEARCH.NAME$+SPACE$(60%-LEN(SEARCH.NAME$))
70      GET #9%
80      GOTO 70 IF SEARCH.NAME$<>L.NAME$
90      INPUT "ROOM NUMBER"; RM.NUM$
100     UPDATE #9%
110     GOTO 50
19000   RESUME 19010
19010   CLOSE #9%
19020   PRINT "UPDATE COMPLETE"
19030   END
```

reads the third record into the record buffer and updates it with a new record field, L.NAME$.

The current record pointer is destroyed during the UPDATE operation. The next record pointer is unchanged.

NOTE

When updating records, include error trapping in your program to make sure you complete FINDs and GETs successfully.

3.5.3 Stream Format Records in Sequential Files

A stream format record is a contiguous series of ASCII characters terminated by a line terminator.

Table 3–3 summarizes valid line terminators.

The record length is the data's length — up to and including the line terminator. This length is limited only by the buffer size, as defined by a MAP statement or RECORDSIZE clause.

Table 3-3: Valid Stream Format Record Line Terminators

Terminator	Symbol
Carriage Return	CR
Control Z	^Z
Escape	ESC
Form Feed	FF
Line Feed	LF
Vertical Tab	VT

3.5.3.1 Writing Records to a Stream Format File — You can write records with the PRINT # and PUT statements:

- If the last character of the record is a valid terminator, the record (including the terminator) is written to the file.
- If the last character is not a valid terminator, BASIC adds a carriage return/line feed combination to the end of the record and writes it to the file.

For example:

```
10 OPEN "TI:" FOR OUTPUT AS FILE #1%,   &
   SEQUENTIAL STREAM, RECORDSIZE 132%
20 INPUT A$
30 A$ = A$ + LF  !ADD A LINE FEED FOR TERMINATOR
40 INPUT LINE B$ ![ALREADY HAS TERMINATOR]
50 PRINT #1%, A$
60 PRINT #1%, B$
70 PRINT "DONE
80 END

RUN

? A1F
? B2G

A1F
   B2G

DONE
```

NOTE

A record that: (1) is the same length as the buffer, and (2) has no terminator, becomes longer than the buffer when BASIC adds the carriage return/line feed combination. This returns the error "?Line too long" (ERR = 47) when you read the record from a file.

3.5.3.2 Reading Records from a Stream Format File — You can read records with GET, INPUT, INPUT LINE, and LINPUT statements. BASIC reads the record with or without line terminators, depending on the statement you use.

3.5.3.2.1 GET — When executing a GET, BASIC scans each character in the record to see if it is a line terminator. During this process, BASIC discards nulls and:

- Reads the record with the terminator if the terminator is a:
  - Control Z (^Z)
  - Escape (ESC)
  - Form Feed (FF)
  - Line Feed (LF)
  - Vertical Tab (VT)
- Checks the next character if the string scan finds a carriage return. If the next character is a:
  - Null — both the carriage return and the null are discarded, and the search for a valid terminator continues. The record read is the data between valid terminators.
  - Line feed — the record is read and returned without the carriage return and line feed.
  - Another character — the carriage return is included in the record, and the search continues for a valid terminator. The record read is between valid terminators.

After BASIC reads the record, you can access the data with a MAP statement if the record fields were predefined at compile time. If the record fields were not predefined, access the data with MOVE FROM.

3.5.3.2.2 INPUT — When executing an INPUT statement, BASIC reads the record and discards the line terminator.

Each INPUT statement reads the next record from the file. Make sure you specify enough variables to equal the number of fields in the record. For example:

```
90 INPUT #4%, STRG.DATA$, REAL.NUM, INTDATITEM%
```

reads the first three fields of the next record. The variables in the INPUT statement must match the data types in the record fields.

If the record does not contain enough data to assign a value to each variable in the INPUT statement, BASIC issues the error message "?Not enough data in record" (ERR = 59). BASIC does not wait for additional data. For example:

```
10     ON ERROR GOTO 19000
20     OPEN 'STV.DAT' FOR OUTPUT AS FILE #1%,                    &
               SEQUENTIAL STREAM, RECORDSIZE 132%
30     PRINT #1%, 'AA,BB,CC'      ! 1 RECORD; 3 FIELDS
40     PRINT #1%, 'DD,EE,FF'      ! WRITE 2ND RECORD
50     PRINT #1%, 12%,',',24%     ! RECORD 3 HAS 2 FIELDS
60     PRINT #1%, 144%            ! RECORD 4 HAS 1 FIELD
70     PRINT #1%, 288%            ! LAST RECORD HAS 1 FIELD
80     CLOSE #1%
90     OPEN 'STV.DAT' FOR INPUT AS FILE #2%,                     &
               SEQUENTIAL STREAM, RECORDSIZE 132%
100    INPUT #2%, A$              ! GET 1ST RECORD, 1ST FIELD
110    INPUT #2%, D$,E$,F$        ! GET RECORD 2
120    INPUT #2%, DOZ%,DOZ2%      ! GET RECORD 3
130    PRINT A$,D$,E$,F$,DOZ%,DOZ2%
140    INPUT #2%, GROSS%,GROS2%  ! TRY FOR MORE DATA, RECORD 4
150    GO TO 32767
19000  IF ERR = 59% AND ERL = 140%                               &
               THEN PRINT 'EXPECT ERROR 59 WHEN YOU TRY THIS' &
               ELSE PRINT 'ERROR';ERR;'AT LINE';ERL
19010  RESUME 32767
32767  END

RUN

AA            DD            EE            FF 12  24
EXPECT ERROR 59 WHEN YOU TRY THIS
```

3.5.3.2.3 INPUT LINE and LINPUT — When executing a LINPUT statement, BASIC reads the record and discards the line terminator. When executing an INPUT LINE statement, BASIC reads the record and includes the line terminator. For example:

```
10    OPEN 'TEST.DAT' FOR OUTPUT AS FILE #1%,                &
          SEQUENTIAL STREAM
20    PRINT #1%, 'ALL LINES ARE WRITTEN TO THE FILE'
30    PRINT #1%, 'WITH TERMINATORS IF NONE IS SPECIFIED,'
40    PRINT #1%, 'A CR/LF IS APPENDED TO'
50    PRINT #1%, 'THE LAST RECORD' + LF
60    CLOSE 31%
70    END

100   OPEN 'TEST.DAT' FOR INPUT AS FILE #1%,                 &
          SEQUENTIAL STREAM
110   LINPUT #1%, A$   !NO TERMINATOR RETURNED HERE
120   INPUT LINE #1%, B$   !CR/LF RETURNED ON RECORD
130   INPUT LINE #1%, C$   !CR/LF RETURNED ON END OF RECORD
140   LINPUT #1%, D$   !TERMINATOR NOT RETURNED
150   PRINT A$ \ PRINT B$
160   PRINT C$ \ PRINT D$
170   CLOSE #1%
32767     END

RUN

ALL LINES ARE WRITTEN TO THE FILE
WITH TERMINATORS. IF NONE IS SPECIFIED

A CR/LF IS APPENDED TO

THE LAST RECORD
```

3.5.3.3 Optimizing Stream Format Record Operations — If your records contain distinct data fields accessed as separate items, use the RMS GET, PUT, and UPDATE operations. In contrast, use PRINT, INPUT, INPUT LINE, and LINPUT statements when the entire record is treated as a single data item; the statements execute faster than GETs, PUTs, and UPDATES. For example:

Retrieving Stream Records from a Mapped Buffer

```
50 MAP (BUFF) Z$ = 8%, B%, C%, F$, X, Y
60 GET #1%, \ A$ = Z$
70 GET #1%, \ B$ = Z$
```

Retrieving Stream Records with INPUT and PRINT Statements

```
50 INPUT LINE #1%, A$
60 LINPUT #1%, B$
```

3.5.3.4 Stream Format File Compatibility — BASIC supports stream format files for BASIC-PLUS compatibility. Because they generate processor overhead, consider their use carefully. For non-file-structured device-specific I/O (for example, paper tape), it is an efficient method to store records.

In contrast, the use of stream files for file-structured I/O differs for each operating system. This hinders their transportability. On the RSTS/E system, stream files are a native file operation, and are compatible with system utilities.

On RSX systems, they are non-native, and therefore are not compatible with system utilities (for example, PIP). RSX system utilities require the carriage control file attributes supplied by BASIC with the FIXED and VARIABLE keywords.

3.5.4 Truncating Sequential Files (SCRATCH)

Although you cannot delete single records from a sequential file, you can delete (truncate) all records starting with the current record. To do this, you must specify ACCESS SCRATCH in the OPEN syntax.

To truncate the file, position the current record pointer with either FIND or GET after the last desired record and then issue a SCRATCH. For example:

```
10 OPEN "MMM.DAT" AS FILE #2%                &
       ,SEQUENTIAL FIXED, ACCESS SCRATCH
30 FIND #2%, FOR I% = 1% TO 33%
50 SCRATCH #2%
60 CLOSE #2%
70 END
```

locates the 33rd record and truncates the file beginning with that record. SCRATCH does not change the physical size of the file, however, and you can PUT records immediately after a SCRATCH.

3.6 RMS Relative Files

Relative files store records in fixed-length record cells. You can access an individual record sequentially or randomly according to its position in the file. Each cell contains one record or is empty.

You cannot OPEN a relative file without defining the maximum record length. The MAP option implicitly declares the record length and uses static (user) buffering. The RECORDSIZE option explicitly declares the record length (for example, RECORDSIZE 128%) and uses dynamic buffering.

The cell length for fixed-length records equals the record length plus one byte. The cell length for variable length records equals the record length plus three bytes. Stream-format records are not permitted.

3.6.1 Opening an RMS Relative File

This syntax opens a relative file:

```
OPEN filespec-exp[FOR INPUT ]AS FILE #num-exp%
                 [FOR OUTPUT]
,[ORGANIZATION] RELATIVE[FIXED   ]
                        [VARIABLE]

[,ACCESS {READ  }]
         {WRITE }
         {MODIFY}

[,ALLOW {NONE  }]
        {READ  }
        {WRITE }
        {MODIFY}

[,MAP mapname]

[,RECORDSIZE num-exp]

[,FILESIZE num-exp%]

[,BUCKETSIZE num-exp%]

[,WINDOWSIZE num-exp]

[,CONTIGUOUS]

[,TEMPORARY]

[,CONNECT num-exp%]
```

This program opens a relative file with user controlled buffering:

```
110     MAP (TEST) PART.NUMBER%, INV.NAME$, UNIT.PRICE
120     OPEN "RELATV.FIL" FOR OUTPUT AS FILE #1%        &
        ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY,    &
        ,ALLOW READ, MAP TEST
```

The MAP clause at line 120 references the MAP statement at line 110. The MAP statement defines the data types and declares the record size. The data record is one integer, one string, and one real number. The record size is the total of these fields, or 22 to 26 bytes, depending on the system's precision. At OPEN time, BASIC statically allocates space for the required record buffer. All record operations use this buffer for I/O to the file.

This program opens a relative file with a record buffer dynamically assigned by BASIC:

```
110     OPEN "RELATV.FIL" FOR OUTPUT AS FILE #1%          &
        ,ORGANIZATION RELATIVE FIXED, ACCESS MODIFY, &
        ,ALLOW READ, RECORDSIZE 220%
```

BASIC allocates an area for the record buffer out of the program's free space. Your program must then use MOVE TO and MOVE FROM statements to move data record elements to and from the record buffer.

After an OPEN statement, there is no current record pointer. The next record pointer is set to the first record.

3.6.2 Record Operations

Relative files permit sequential and random PUT, FIND, and GET operations. In addition, you can UPDATE, DELETE, and UNLOCK records.

3.6.2.1 Writing Records to the File (PUT) — PUT writes records sequentially or randomly. For sequential PUTs, type PUT with the file's channel number:

```
90 PUT #10%
```

BASIC writes the new record in the location specified by the next record pointer. If a record already exists, you receive the error message "?Record already exists" (ERR = 153).

A sequential PUT destroys the current record pointer. The next record pointer is set to the new record plus 1.

For random PUTs, specify the record cell number:

```
90 PUT #10%, RECORD 134%
```

BASIC writes the new record in the location specified by the relative record number. If a record already exists, you receive the error message "?Record already exists" (ERR = 153).

A random PUT destroys the current record pointer and leaves the next record pointer unchanged. For example, in the program:

```
310 PUT #1%
320 PUT #1%, RECORD 20%
```

the PUT in line 320 does not change the next record pointer set in line 310.

When processing variable length records, you can use the COUNT clause to specify the number of bytes written to the file. For example:

```
90 PUT #10%, RECORD 134%, COUNT 56%
```

writes a 56-character data record into logical record (block) 134. The number specified in the COUNT must not exceed the size in the MAP or RECORDSIZE clause. Setting the record length prevents unwanted data when retrieving the record. See the RECOUNT variable and CCPOS function for more information.

3.6.2.2 Locating Records in the File (FIND) — FIND locates records sequentially or randomly, and updates the current and next record pointers. For sequential FINDs, type FIND with the channel number:

```
70 FIND #5%
```

BASIC finds the record specified by the next record pointer. If the record does not exist, you receive the error message "?Record not found" (ERR = 155). A sequential FIND sets the current record pointer to the record found. The next record pointer is set to the current record plus 1.

For random FINDs, specify the record cell number:

```
70 FIND #5%, RECORD 26%
```

If no record exists, you receive ERR = 155, and the record pointers are undefined. Random finds set the current record pointer to the record found. The next record pointer is unchanged.

FIND is useful: (1) to determine if the record exists or (2) if the next operation is GET, DELETE, or UPDATE. For example, the program:

```
100 FIND #1%, RECORD 20%
110 GET #1%
120 UPDATE #1%
```

updates RECORD 20.

Use FIND instead of GET to save time. It is especially useful when:

- Using a loop to skip over records in a file
- Determining the existence of a record for a GET or UPDATE
- Establishing the current record for an UPDATE or DELETE

3.6.2.3 Reading Records from the File (GET) — GET reads records sequentially or randomly. For sequential GETs, type GET with the channel number:

```
40 GET #1%
```

GET reads the record specified by the next record pointer unless the last record operation was a successful FIND. In that case, GET reads the record you found. Successive GETs read successive records.

After a sequential GET, BASIC sets the record pointers depending on any previous FIND operations:

- A GET with no preceding FIND sets the current record pointer to the new record. The next record pointer is set to the new record plus 1.
- A GET preceded by a FIND leaves the current record pointer unchanged. The next record pointer is set to the current record plus 1.

Your program can read all records sequentially and then close the file when finished. For example:

```
10      ON ERROR GOTO 19000
20      MAP (CRE) STATE$, MAIN.OFF$, NUM.SPL%, SCODE
30      OPEN "REC.DAT" FOR INPUT AS FILE #1%,                    &
        ORGANIZATION RELATIVE, MAP CRE
40      FOR I% = 1% TO 32767%
50        GET #1%
60        PRINT "REPORT FOR", STATE$
70        PRINT "THE OFFICE FOR"; STATE$; "IS", MAIN.OFF$
80        PRINT MAIN.OFF$; "HAS", NUM.SPL%; "EMPLOYEES"
90        PRINT "THE SALES AREA CODE IS", SCODE
100     NEXT I%
19000   IF (ERR = 11%) AND (ERL = 50%)                           &
          THEN RESUME 32000 ELSE ON ERROR GOTO 0
32000   PRINT "END OF FILE"
33000   CLOSE #1%
32767   END
```

For random GETs, specify the record cell number:

```
40 GET #1%, RECORD 88%
```

or use a variable to access a record:

```
10  MAP (BEC) VEH.NUM%, SERIAL.NUM$ = 22%, OWNER$ = 30%
20  OPEN "VEH.IDN" FOR INPUT AS FILE #2%,                        &
        ORGANIZATION RELATIVE FIXED, MAP BEC
30  INPUT "WHICH RECORD DO YOU WANT";A%
40  GET #2%, RECORD A%
50  PRINT "THE VEHICLE NUMBER IS", VEH.NUM%
60  PRINT "THE SERIAL NUMBER IS", SERIAL.NUM$
70  PRINT "THE OWNER OF VEHICLE"; VEH.NUM% "IS", OWNER$
80  INPUT "NEXT RECORD=i";A%
90  IF A% = 0% THEN 100 ELSE 40
100 CLOSE #2%
110 END
```

Random GETs change the value of the current record pointer to that of the record read. The next record pointer is set to the current record plus 1. In the program:

```
10 GET #2%, RECORD 10%
20 GET #2%
```

line 10 GETs record 10, and line 20 GETs record 11.

3.6.2.4 Replacing Records in the File (UPDATE) — UPDATE writes a new record at the location indicated by the current record pointer. You can UPDATE records only after a successful FIND or GET. Therefore, UPDATE needs no RECORD clause. The error message "?Record not found" (ERR = 155) indicates that the FIND or GET was unsuccessful. For example:

```
10 MAP (UPD) ENRDAT$ = 8%, INVOC%, SH.NUM%, COST
20 OPEN "REC.ING" FOR INPUT AS FILE #8%,  &
        RELATIVE FIXED, MAP UPD
30 INPUT "WHICH RECORD TO UPDATE";A%
40 FIND #8%, RECORD A%
50 INPUT "REVISED COST IS";COST
60 UPDATE #8%
70 INPUT "NEXT RECORD";A%
80 IF A%>5000 THEN 90 ELSE 40
90 CLOSE #8%
100 END
```

UPDATEs the records you specify.

The current record pointer is destroyed after an UPDATE operation. The next record pointer is unchanged.

You can use the COUNT clause to specify the size of the new record. For example:

```
50 GET #3%
60 INPUT "NEW DATA"; NEW.DATA%
70 UPDATE #3%, COUNT 80%
```

writes a record with a length of 80 bytes. You can specify a length equal to the latest input operation with the RECOUNT variable. For example:

```
50 GET #8%, RECORD 404%
60 INPUT "NEW DATA"; NEW.DATA%
70 UPDATE #8%, COUNT RECOUNT
```

writes a record exactly equal to the length of NEW.DATA%. In this example:

```
50 INPUT "TYPE IN NEW NAME"; EMP.NAME$
60 GET #8%, RECORD 591%
70 UPDATE #8%, COUNT RECOUNT
```

the recount variable is set to the length of record 591, and the UPDATE operation writes EMP.NAME$ with a length equal to that record.

3.6.2.5 Deleting Records from the File (DELETE) — You can delete records in a relative file. You must successfully FIND or GET the record before deleting it. The error message "?Record not found" (ERR = 155) indicates that the FIND or GET was unsuccessful. For example:

```
40 GET #1%, RECORD 67%
50 DELETE #1%
```

locates record 67 and deletes it. Since the cell itself is not deleted, you can PUT a new record after deleting an old one. For example:

```
PUT #1%, RECORD 67%
```

writes a new record in place of the old one.

There is no current record after a delete operation. The next record pointer is unchanged.

3.6.2.6 Locking Buckets — To protect file integrity, BASIC causes RMS to perform bucket locking on files that are write shared (opened with ALLOW WRITE or ALLOW MODIFY). When your program FINDs or GETs a record, BASIC locks the bucket containing the record to prevent other programs from using the same record. BASIC unlocks a bucket when:

- Your program performs another record operation.
- Your program explicitly unlocks it.

For example:

```
60 UNLOCK #8%
```

unlocks the bucket associated with channel #8%. Use UNLOCK if two or more users must simultaneously WRITE to the file.

NOTE

Locking records with the LOCK clause is a TRAX only feature.

3.7 RMS Indexed Files

Indexed files contain data records stored in ascending order by key value. They also contain one or more indices that provide access paths to these records by primary and alternate key values. When indexed files are created you must specify: (1) a record buffer, (2) the primary key and, and (3) all duplicate keys. When opening an existing indexed file, specify either: (1) no keys or (2) all keys.

Although indexed organization is the most versatile, it generates the most overhead in disk space and I/O.

3.7.1 Opening an RMS Indexed File

This syntax opens an indexed file:

```
OPEN filespec-exp [FOR INPUT  ] AS FILE #num-exp%
                  [FOR OUTPUT ]
,[ORGANIZATION] RELATIVE [FIXED   ]
                         [VARIABLE]
```

```
[,ACCESS {READ  }]
         {WRITE }
         {MODIFY}
[,ALLOW {NONE  }]
        {READ  }
        {WRITE }
        {MODIFY}
[,MAP mapname]
[,RECORDSIZE num-exp]
[,FILESIZE num-exp%]
[,BUCKETSIZE num-exp%]
[,WINDOWSIZE num-exp]
[,CONTIGUOUS]
[,TEMPORARY]
[,CONNECT num-exp%]
,PRIMARY [KEY] name[{DUPLICATES  }]
                   {NODUPLICATES}
[,ALTERNATE [KEY] name[{[DUPLICATES ] [CHANGES ]}]
                       {[NODUPLICATES] [NOCHANGES]}
```

For example:

```
10 OPEN " INVEN.TOR" FOR INPUT AS FILE #1%      &
         ,ORGANIZATION INDEXED, PRIMARY KEY A$ &
         ,MAP PRT, ALLOW NONE, ACCESS READ
```

After an OPEN statement, the next record pointer is set to the first record.

3.7.2 Creating and Using Index Keys

BASIC requires one primary key and permits up to 254 alternate keys for multiple access paths. A separate index is created for each defined key. The key assignments appear in the OPEN FOR OUTPUT and accompany a MAP statement defining the key fields.

You must declare the primary and all alternate keys when the file is created. When you open an existing file, specify either: (1) no keys, or (2) all keys declared when you created the file.

3.7.2.1 Assigning Key Names — Primary and alternate keys must be mapped string variables. You specify keys in the OPEN statement. For example:

```
10 MAP (INV) RECNO$, INV.NAME$ = 4%, PROJECT.NUM$ = 8%  &
        ,JOB.SUPERVISOR$ = 32%
20 OPEN "INDEX.DAT" FOR OUTPUT AS FILE #8%              &
        ,ORGANIZATION INDEXED FIXED                     &
        ,PRIMARY KEY RECNO$                             &
        ,ALTERNATE KEY INV.NAME$                        &
        ,ALTERNATE KEY PROJECT.NUM$                     &
        ,ALTERNATE KEY JOB.SUPERVISOR$                  &
        ,MAP INV
```

BASIC assumes that the primary key has no duplicate values. If you allow duplicates, specify that attribute in the OPEN statement. For example:

```
,PRIMARY KEY RECNO$ DUPLICATES
```

Alternate keys can also have duplicates. In addition, you can allow key values to change by specifying CHANGES with DUPLICATES:

```
,ALTERNATE KEY JOB,SUPERVISOR$ DUPLICATES CHANGES
```

3.7.2.2 Creating Data Fields (MAP) — The MAP statement creates a record buffer for input and output, defines data fields in the records, and positions these fields in each record. Each key in the file open must have a corresponding field in the MAP statement. You cannot open an indexed file without defining the MAP.

For example:

```
50  OPEN "ACCNT.DAT" FOR INPUT AS FILE #4%     &
        ,ORGANIZATION INDEXED VARIABLE         &
        ,ACCESS MODIFY,ALLOW NONE              &
        ,PRIMARY KEY B$, ALTERNATE KEY WAGES$  &
        ,MAP BUFF1
```

opens a file ACCNT.DAT and names the MAP BUFF1. You then define the MAP. Specify lengths for string fields. The default is 16 characters. For example:

```
60 MAP (BUFF1) NA.ME$ = 26%, EMP.NUM%, AGE%
```

You can use FILL items to reserve space in a MAP; for example:

```
60 MAP (BUFF1) NA.ME$ = 26%, EMP.NUM%, FILL, FILL$, AGE%
```

reserves space for a floating point number (FILL) and a string of sixteen characters (FILL$).

3.7.3 Record Operations

BASIC indexed files permit five record operations: PUT, sequential and random FINDs and GETs, UPDATE, and DELETE.

3.7.3.1 Writing Records to the File (PUT) — BASIC stores records in order of ascending key value. Therefore, you do not specify a random or sequential PUT. Type PUT with the channel number. For example:

```
20 MAP (XXX) R.NUM$, DEPT.NAME$, PUR.DAT$
              .
              .
              .
30 INPUT "REQUISITION NUMBER"; R.NUM$
40 INPUT "DEPARTMENT NAME"; DEPT.NAME$
50 INPUT "DATE OF PURCHASE"; PUR.DAT$
60 PUT #2%
```

The error message "?Record already exists" (ERR = 153) indicates that a record with a matching primary key field already exists.

3.7.3.2 Locating Records in the File (FIND) — FIND determines if a record exists, so you can GET, UPDATE, or DELETE it. FIND locates records in the file but does not move them into the record buffer.

To FIND records sequentially, type FIND and the file's channel number. For example:

```
70 FIND #6%
```

locates the record with the next highest key value.

```
70 FIND #5%, KEY #2%
```

locates the record with the next highest key value according to the second alternate key. Successive FINDs locate successive records.

A sequential FIND updates the current record pointer and sets the next record pointer to the record that logically follows the current record. The "next logical record" depends on the key you are using.

For random FINDs, specify a target key value. BASIC then searches the records until it locates the one that matches your specification. The format for a random FIND is:

FIND #num-exp%, KEY # num-exp% $\left\{\begin{matrix}\text{EQ}\\\text{GT}\\\text{GE}\end{matrix}\right\}$ string expression

where:

KEY # num-exp%	is the reference number of the key. Zero is the reference number of the primary key, 1 is the reference number of the first alternate key, and so forth.
EQ	specifies a search for the first record with a key value equal to "string expression."
GT	specifies a search for the first record with a key value greater than that of "string expression."
GE	specifies a search for the first record with a key value equal to or greater than "string expression."
string expression	is a string field. You can specify an expression or a variable.

With the exact key match (EQ), RMS looks for the first record in the file that equals the key value given in string-expression. For example:

```
150 FIND #3%, KEY #0% EQ "KATHY HARPER"
```

If no match is possible, BASIC returns the error message "?Record not found" (ERR = 155).

With the greater than (GT) key match, BASIC searches for the record with the next highest key value specified by "string expression." If no GT record exists, you receive the EOF error message.

If you specify greater than or equal to (GE), and an exact key match is possible, BASIC locates the first record that equals the key value in the string expression. For example:

```
90 FIND #3%, KEY #2% GE "S547H2A"
```

If you specify GE and no exact match is possible (that is, no record in the file has a value for KEY # num-exp% that equals string expression), BASIC locates the first record whose key value is higher than the string expression. For example:

```
40  FIND #5%,   KEY #0% GE "JONES"
50  FIND #5%,   KEY #0% GT "ABRAMSON"
```

If the file contains the names ABELL, ABRAMSON, ADAMS, HOTCHKISS, JONES, KNIGHT, and SMITH, statements 40 and 50 find the records JONES and ADAMS. Line 40 identifies the key as the primary key (#0) and searches for the first record whose primary key is equal to or greater than JONES. JONES is the first record to meet this condition. A sequential FIND after line 40 locates Knight and Smith. Line 50 also uses the primary key and searches for a record greater than ABRAMSON. ADAMS is the first record to meet this condition.

The string expression can contain fewer characters than the key of the record you hope to find. The statements on lines 40 or 50 could have specified values of fewer characters, such as "JO" in line 40 or "ABR" in line 50. Note, however, that if line 50 specified "AB", ABELL is the target record. This process is called "generic key searching."

A random FIND updates the current record pointer but leaves the next record pointer unchanged.

3.7.3.3 Reading Records from the File (GET) — You can read records sequentially by specifying GET with the file's channel number. For example:

```
90 GET #1%
```

reads the record with the next higher key value. Successive GETs read records sequentially.

Your program can read through an entire file and CLOSE after printing all the records. For example:

```
10 ON ERROR GOTO 300
20 MAP (CRE) STATE$, MAIN.OFF$, NUM.SPL%, SCODE
30 OPEN "REC.DAT" FOR INPUT AS FILE #1%,  &
     ORGANIZATION INDEXED, MAP CRE,  &
     PRIMARY KEY STATE$, ALTERNATE MAIN.OFF$
40 FOR I% = 1% TO 32766%
50 GET #1%
60 PRINT "REPORT FOR", STATE$
70 PRINT "THE OFFICE FOR"; STATE$; "IS", MAIN.OFF$
80 PRINT MAIN.OFF$; "HAS", NUM.SPL%; "EMPLOYEES"
90 PRINT "THE SALES AREA CODE IS", SCODE
100 NEXT I%
300 IF (ERR = 11%) AND (ERL = 50%)   &
        THEN RESUME 400 ELSE ON ERROR GOTO 0
400 PRINT "END OF FILE"
500 CLOSE #1%
600 END
```

Specifying another key reads successive records according to that key value. For example:

```
90 GET #1%, KEY #1% GE "ARK"
```

reads the next record with the next highest value according to the current key of reference.

If the last record operation was not a FIND, GET positions the current record pointer to the record read. The next record pointer is set to the record logically following the current record in the key of reference.

If the last operation was a FIND, the current record pointer is unchanged, and the next record pointer is set to the record logically following the current record in the key of reference.

For random searches, you must specify a target key value. The format for GET is:

GET #num-exp%, KEY #num-exp% $\left\{\begin{matrix}\text{EQ}\\\text{GT}\\\text{GE}\end{matrix}\right\}$ string expression

See FIND for a description of generic key searching. You can read records randomly by specifying the target string:

```
560 GET #4%, KEY 0% GT "COLUMBUS"
```

or use a variable to read a number of different records:

```
10 MAP (BEC) OWNER$ = 30% VEH.NUM%, SERIAL.NUM$ = 22%
20 OPEN "VEH.IDN" FOR INPUT AS FILE #2%,  &
       ORGANIZATION INDEXED, MAP BEC
30 INPUT "WHICH RECORD DO YOU WANT";A$
40 GET #2%, KEY #0% EQ A$
50 PRINT "THE VEHICLE NUMBER IS", VEH.NUM%
60 PRINT "THE SERIAL NUMBER IS", SERIAL.NUM$
70 PRINT "THE OWNER OF VEHICLE"; VEH.NUM% "IS", OWNER$
80 INPUT "NEXT RECORD";A$
90 IF A$ = "DONE" THEN 100 ELSE 40
100 CLOSE #2%
110 END
```

A random GET sets the current record pointer to the record read. The next record pointer is set to the record logically following the current record in the index of reference.

3.7.3.4 Replacing Records in the File (UPDATE) — UPDATE writes a new record at the location indicated by the current record pointer. Because UPDATE depends on a previous successful FIND or GET, your program does not need to specify a record number or key value with UPDATE. The error message "?Record not found" (ERR = 155) indicates the record you specified does not exist. Use UPDATE with the file's channel number. For example:

```
30 INPUT "WHAT IS THE EMPLOYEE NAME"; TARGET.NAME$
40 GET #2%, KEY #1% EQ TARGET.NAME$
50 INPUT "TYPE IN THE NEW ID"; EMP.ID.NUMBER$
60 UPDATE #2%
```

enters a new identification number for the employee name you specify.

When the file permits duplicate primary keys, the new record must be the same length as the old one. When the program does not permit duplicate primary keys, the new record:

- Can be no longer than the maximum record size.
- Must include at least the primary key field.

If the new record omits one of the old record's alternate key fields, the OPEN statement must specify CHANGES for that key field.

After an UPDATE operation, there is no current record. The next record pointer is unchanged.

3.7.3.5 Deleting Records from the File (DELETE) — A successful FIND or GET must precede the DELETE operation. These operations make the target record available for deleting. The error message "?Record not found" (ERR = 155) indicates that the FIND or GET was unsuccessful. For example:

```
30 GET #2%, KEY #0 EQ "421-56-9012"
40 DELETE #2%
```

deletes the record with the primary key value equal to 421-56-9012.

An error message indicates the record you specified does not exist.

After a DELETE operation, there is no current record. The next record pointer is unchanged.

3.7.3.6 Locking Buckets — To protect file integrity, BASIC performs bucket locking on files that are write shared (opened with ALLOW WRITE or ALLOW MODIFY). When your program FINDs or GETs a record, BASIC locks the bucket containing the record to prevent other programs from using the same record. BASIC unlocks a bucket when:

- Your program performs another record operation.
- Your program explicitly unlocks it.

For example:

```
60 UNLOCK #8%
```

unlocks the bucket associated with channel #8%. Use UNLOCK if two or more users must simultaneously WRITE to the file.

NOTE

Locking records with the LOCK clause is a TRAX only feature.

3.7.4 Restoring an Indexed File (RESTORE)

Because an indexed file can have alternate access paths, you must specify the access path (key) you want restored. You can restore the primary key or an alternate key. For example:

```
90 RESTORE #3%, KEY 0%
```

restores the file's primary key only. You can restore an alternate key by naming that key in the RESTORE statement. For example:

```
220 RESTORE #4%, KEY 8%
```

restores the 8th alternate key.

After a RESTORE, there is no longer a current record pointer, and the next record pointer points to the first record in the file.

If you have been processing records sequentially according to 1st alternate key value, you continue to FIND and GET records according to that key after restoring it. To begin record operations on another key, you must specify that key. For example:

```
GET #4%, KEY 0%
```

begins record operations on the primary key. Specifying no key restores the primary key.

3.8 Buffer Control and File Optimization

Controlling the I/O and record buffers helps you to optimize file handling and program size. This section describes (1) the OPEN statement keywords affecting the I/O buffer size, (2) mapping the record buffer, and (3) record blocking.

3.8.1 OPEN Statement Keywords

3.8.1.1 BLOCKSIZE — A block on magnetic tape is between 18 to 8192 bytes. With RMS sequential tapes, you specify this size in the BLOCKSIZE clause as a positive integer divisible by four.

You must specify BLOCKSIZE as an integer number of records. For example, a file on tape with 126 byte records has a block size between 1 and 64. The default is 512 bytes. For example, in the OPEN statement:

```
10 OPEN "MM0:[100,100]TEST.SEQ" FOR OUTPUT AS FILE #12% &
        ,ORGANIZATION SEQUENTIAL, RECORDSIZE 90% &
        ,BLOCKSIZE 12%
```

the RECORDSIZE attribute defines the size of the largest record in the file as 90 bytes, and BLOCKSIZE defines the size of a block as 12 records (1080 bytes). Thus your program contains an I/O buffer of 1080 bytes. Each physical read or write moves 1080 bytes of data. Every 12th GET or PUT causes a physical read or write. The previous GETs or PUTs only move data into or out of the I/O buffer. Specifying a BLOCKSIZE larger than the default can reduce overhead by eliminating some physical reading and writing to the tape. In the example, a BLOCKSIZE of 12 saves time by accessing the tape only after the 12th record operation.

3.8.1.2 BUCKETSIZE — A bucket is a logical storage structure that BASIC uses to build and maintain files on disk devices. It contains from one to fifteen blocks. Although a bucket contains blocks, you define a bucket by the number of records it contains. For example:

```
,BUCKETSIZE 12%
```

specifies a bucket of 12 records.

If you specify a BUCKETSIZE other than that originally assigned to the file, you receive the error message "File attributes not matched" (ERR = 160) when re-opening that file.

NOTE

> Although you specify buckets in terms of records, the RMS DISPLAY utility returns bucketsize in terms of the number of blocks.

Your program cannot change the length of a block on disk. It is always 512 bytes. A bucket, however, is a logical structure that you can tailor to file requirements.

Records cannot span bucket boundaries. Therefore, when you specify bucket size in your program, you must consider the size of the largest record in the file.

There are two ways to establish the number of blocks in a bucket. The first is to use the BASIC default. The second is to specify the number of records you want in each bucket. BASIC then calculates a BUCKETSIZE based on that number.

The default BUCKETSIZE assigned to relative and indexed files is as small as possible. A small bucket minimizes memory buffer space and the number

of records locked when a file is shared. However, it also forces more disk transfers, especially when you are accessing records sequentially.

BASIC selects a default bucket size depending on:

- The record length.
- The file organization (relative or indexed).
- The record format (fixed or variable).

If you do not define the BUCKETSIZE clause in the OPEN statement, BASIC:

- Assumes that there is a minimum of one record in the bucket.
- Calculates a size.
- Assigns the number of blocks.

If you define BUCKETSIZE and specify the number of records, BASIC uses formulas to derive the necessary number of blocks.

BASIC determines the default bucket size for relative files from these formulas:

Fixed-length records without BUCKETSIZE specification:

```
Bnum = (1+Rlen)/512
```

Fixed-length records with BUCKETSIZE specified:

```
Bnum = ((1+Rlen)*Rnum)/512
```

Variable-length records without BUCKETSIZE specification:

```
Bnum = (3+Rmax)/512
```

Variable-length records with BUCKETSIZE specified:

```
Bnum = ((3+Rmax)*Rnum)/512
```

where:

Bnum	is the number of blocks for each bucket, rounded up to an integer.
Rlen	is the length in bytes of the file's fixed-length records, as defined in the RECORDSIZE clause.
Rmax	is the length in bytes of the largest variable-length record in the file, as defined in the RECORDSIZE clause.
Rnum	is the number of records you want in each bucket, as defined in the BUCKETSIZE clause.
1	is the byte RMS uses to determine deleted records in the file.
3	represents the existence byte plus two bytes that indicate the count field.

Table 3–4 shows the default bucket sizes selected by BASIC when the bucket contains the default of one record.

Table 3–4: Relative File Default Bucket Size

Number of Blocks	Record Length, Fixed Length Records	Maximum Record Size, Variable Length Records
1	1–511	1–509
2	512–1023	510–1021
3	1024–1535	1022–1533
4	1536–2047	1534–2045
5	2048–2559	2046–2557
6	2560–3071	2558–3069
7	3072–3583	3070–3581
8	3584–4095	3582–4093
9	4096–4607	4094–4605
10	4608–5119	4606–5117
11	5120–5631	5118–5629
12	5632–6143	5630–6141
13	6144–6655	6142–6653
14	6656–7167	6654–7165
15	7168–7679	7166–7677

BASIC derives the default bucket size for indexed files from the following formulas:

Fixed-length records without BUCKETSIZE specified:

Bnum = (22+Rlen)/512

Fixed-length records with BUCKETSIZE specified:

Bnum = (((7+Rlen)*Rnum)+15)/512

Variable-length records without BUCKETSIZE specified:

Bnum = (24+Rmax)/512

Variable-length records with BUCKETSIZE specified:

Bnum = (((9+Rmax)*Rnum)+15)/512

where:

Bnum — is the number of blocks for each bucket, rounded up to an integer.

Rlen — is the length in bytes of the file's fixed-length records, as defined in the RECORDSIZE clause.

Rmax — is the length in bytes of the largest variable-length record in the file, as defined in the RECORDSIZE clause.

Rnum — is the number of records you want in each bucket, as defined in the BUCKETSIZE clause.

22 — is a 15–byte RMS bucket overhead plus 7 bytes for the fixed-format record header length. When you define BUCKETSIZE, BASIC allocates 7 bytes to each record in the bucket and 15 bytes to the complete bucket.

24 — is a 15–byte RMS bucket overhead plus 9 bytes for the variable-format record header length. When you define BUCKETSIZE, BASIC allocates 9 bytes to each record in the bucket and 15 bytes to the complete bucket.

Table 3–5 shows the bucket sizes selected by BASIC when the number of records is undefined.

Table 3–5: Indexed File Default Bucket Size

Number of Blocks	Record Length, Fixed Length Records	Maximum Record Size, Variable Length Records
1	1–490	1–488
2	491–1002	489–1000
3	1003–1514	1001–1512
4	1515–2026	1513–2024
5	2027–2538	2025–2536
6	2539–3050	2537–3048
7	3051–3562	3049–3560
8	3563–4074	3561–4072
9	4075–4586	4073–4584
10	4587–5098	4585–5096
11	5099–5610	5097–5608
12	5611–6122	5609–6120
13	6123–6634	6121–6632
14	6635–7146	6633–7144
15	7147–7658	7145–7656

For example, if a BASIC-PLUS-2 program OPENs an indexed file with fixed-length records of 100 bytes and a bucketsize of 5, the run-time system makes these calculations when creating the file:

```
100 bytes for the data: for example, MAP (ABC) A$ = 100
+ 7 bytes for the record header
-----

107 bytes for the complete record

535 bytes for the records in a bucket (5*107 = 535)
+15 bytes for the bucket overhead
-----

550 bytes specified for each bucket
```

BASIC requires that buckets be an integral number of blocks; therefore, the bucketsize for this file is two blocks instead of one. This means that each bucket can hold at least five records. BASIC continues to fill the bucket with as many records as possible.

When you specify a bucket size for files in your program, keep in mind the space versus speed trade-offs. A large bucket size increases file processing speed because a greater amount of data is available in memory at one time. However, it also increases the memory space needed for buffer allocation. Likewise, a small bucket size minimizes buffer requirements, but can also decrease the speed of operations.

In addition to record buffer space, your program needs overhead space for internal control structures. These control structures (FAB, RABs, buffer descriptor blocks, and so forth) are allocated when the file is opened. See Section 3.9.4 for more information.

3.8.2 Statically Allocating Buffer Space (MAP)

MAP statements statically allocate record buffer space and define data at compile time. The MAP referenced in the OPEN statement also defines record length. Define the MAP before: (1) the OPEN statement, and (2) you reference any mapped variables.

The format for the MAP statement is:

```
MAP (map name) argument list
```

where:

MAP	is a required keyword.
map name	is one to six characters beginning with a letter. You must include this name in the OPEN statement also. You cannot specify a MAP as part of a conditional expression.
argument list	defines the name, size, and data type of each record in the file. The length of the record buffer equals the sum of all fields in the argument list.

For example:

```
110 MAP (ASA) NA.ME$ = 60%, ADD.RESS$ = 60%
```

defines two string fields of 60 characters and a record size of 120 bytes. Later MAPs cannot exceed this record size.

NOTE

Avoid using a MAP and a RECORDSIZE specification in the same OPEN statement. Because RECORDSIZE overrides MAP, it is possible to define a record size and cause a record operation to overwrite mapped areas.

You can use additional MAP statements to redefine the fields in the record. These statements assign new variables to the same fields, and allow, for example, the variable "ADD.RESS$" to include the fields "STREET$", "CITY$", "STATE$" and "ZIP$". You cannot, however, assign the same variable name to more than one field.

Because MAP statements define data at compile time, execution time is faster. In addition, record buffers are in the program space. This can use less space than buffers assigned dynamically with the RECORDSIZE attribute. MAPs have two other advantages:

- Because MAPs point into the buffer but do not actually move data, you eliminate the overhead of the MOVE statement.
- You do not, as in FIELD statements, have to change integers and real numbers to string format before moving them to the buffer. MAP statements define all data types.

3.8.2.1 Single MAP Statements — When all records in the file are alike in content and format, you can specify one MAP statement to define the record buffer. You name the MAP in the OPEN statement and describe it elsewhere in the program. For example:

```
100  MAP (ASA) F.NAME$ = 30%, L.NAME$ = 30%, ADD.RESS$ = 60%
110  OPEN "AAA.DAT" FOR INPUT AS FILE #3%     &
          ,ORGANIZATION SEQUENTIAL            &
          ,MAP ASA, ACCESS APPEND
```

You can then input data and associate that data with variables in the argument list. For example:

```
110 MAP (ASA) F.NAME$ = 30%, L.NAME$ = 30%, ADD.RESS$ = 60%
120 INPUT "FIRST NAME"; F.NAME$
130 INPUT "LAST NAME"; L.NAME$
140 INPUT "ADDRESS"; ADD.RESS$
150 PUT #1%\GOTO 120
```

inputs data for the three data fields defined in the OPEN statement and writes the record in the file.

Because the OPEN statement associates the MAP with a channel number, you can FIND, GET, PUT, UPDATE, and DELETE records without any further reference to the buffer.

3.8.2.2 Multiple MAP Statements — Because multiple MAPs permit multiple record definitions at the same time, you can redefine the record buffer. This can: (1) optimize space in the file, (2) permit different types of records in the same file, or (3) allow for variations in record fields. For example:

```
500 MAP (ASA) NA.ME$ = 60%, ADD.RESS$ = 60%, EMP.INFO$ = 38%
```

defines a record with three fields with a total length of 158 characters. You can redefine these fields with later MAPs. For example:

```
1060 MAP (ASA) F.NAME$ = 28%, FILL$ = 2%, L.NAME$ = 30%    &
     ,STREET$ = 34%, FILL$ = 1%, CITY$ = 15%, FILL$ = 1% &
     ,STATE$ = 2%, FILL$ = 1%, ZIP$ = 7%, FILL$ = 2%      &
     ,SALARY.REVIEW.DATE$ = 8%, FILL$ = 2%, SUPER.VISOR$ = 22%
```

defines the same data as the MAP in line 500 with 14 fields. You can then reference any variable in either statement without further reference to the MAPs:

```
1070 INPUT "FIRST NAME"; F.NAME$
1080 INPUT "LAST NAME"; L.NAME$
1090 PRINT "EMPLOYEE NAME IS";NAME$
1100 INPUT "IS SPELLING CORRECT"; B$
1110 IF B$ = "NO" GOTO 1070
1120 INPUT "STREET ADDRESS"; STREET$
1130 INPUT "CITY"; CITY$
1140 INPUT "STATE"; STATE$
1150 INPUT "ZIP CODE"; ZIP$
1160 PRINT "ADDRESS IS";ADD.RESS$
1170 INPUT "IS THAT CORRECT"; B$
1180 IF B$ = "NO" GOTO 1120
1190 INPUT "SALARY REVIEW DATE"; SALARY.REVIEW.DATE$
1200 INPUT "SUPERVISOR"; SUPER.VISOR$
1210 PUT #1%
```

Make sure the MAP with the largest actual size is referenced by the OPEN statement. You should include all MAPs before the variables they reference.

3.8.2.3 FILL Items — FILL items mask parts of the record buffer and enable you to: (1) access portions of a record, (2) skip over record fields, and (3) reserve space within or between data elements.

FILLs are available for all data types. Table 3–6 summarizes their formats and allocations.

Table 3–6: FILL Item Formats, Representations, and Allocations

FILL Format	Representation	Bytes Used
FILL	Real (single or double precision)	4 or 8
FILL(n)	n real elements	4n or 8n
FILL%	Integer	2
FILL%(n)	n integer elements	2n
FILL$	String	16
FILL$(n)	n string elements	16n
FILL$ = m	String	m
FILL$(n) = m	n string elements, each m bytes long	m * n

NOTE

In the applicable formats of FILL, n represents a repeat count, not an array subscript. FILL(n), for example, represents n real elements, not n+1.

For example:

```
540 MAP (ASA) NA.ME$, FILL$ = 60%, SALARY.REVIEW.DATE$ = 8%,  &
              FILL$ = 20%
550 PRINT NA.ME$, SALARY.REVIEW.DATE$
```

accesses only two fields in the record, and prints a listing of the employee name and salary review date. The field FILL$ = 60% skips over the employee's address.

The FILL field in:

```
STATE$ = 2%, FILL$ = 2%, ZIP$ = 5%
```

reserves a two-character space between the state and zip code fields.

The FILL field in:

```
STATE$ = 2%, FILL%(5%), ZIP$ = 5%
```

reserves space for five integers (ten bytes) between the STATE$ and ZIP$ fields.

3.8.3 Dynamically Allocating Buffers (RECORDSIZE)

The RECORDSIZE option makes dynamic reallocation of record buffer space when data definitions cannot be made until run-time. For example, the program:

```
50 OPEN "SEQ.DAT" FOR OUTPUT AS FILE #1%,  &
          SEQUENTIAL VARIABLE, RECORDSIZE 100%
60 INPUT "PROJECT MANAGER"; PROJ.MGR$
70 A% = LEN(PROJ.MGR$)
80 IF A%>30% THEN PRINT "NAME TOO LONG"\GOTO 60
90 INPUT "PROJECT NAME"; PROJ.NAM$
100 B% = LEN(PROJ.NAM$)
110 IF B%>68% THEN PRINT "NAME TOO LONG"\GOTO 90
120 INPUT "PROJECT NUMBER"; PROJ.NUM%
130 PROJ.MGR$ = PROJ.MGR$+SPACE$(30%-A%)
140 PROJ.NAM$ = PROJ.NAM$+SPACE$(68%-B%)
150 MOVE TO #1%, PROJ.MGR$ = 30%, PROJ.NAM$ = 68%, PROJ.NUM%
160 PUT #1%, RECORD PROJ.NUM%
170 INPUT "MORE RECORDS"; C$
180 IF C$ = "YES" THEN 60
190 CLOSE #1%
200 END
```

assigns field lengths at run-time based on data you input.

In contrast, MAP statements define all data locations and resolve data definitions at compile time. They have the disadvantage of fixing record buffer space in the program, and not allowing for dynamic redefinition. However, you can overcome this disadvantage by re-using the record buffer area for other files and connecting to more than one record stream.

When opening files with dynamic buffering, BASIC allocates record buffers from the free space in your program area. This free space is the difference, in words, between your task image and:

- 32K words (the maximum program size).

or

- The maximum program size set by your system manager.

BASIC uses free space for both dynamic string handling and I/O buffers.

The amount of space reserved for the dynamic record buffer depends on the record size, block and bucketsize, file organization, record type, and the device. The RECORDSIZE clause allocates more record buffer space by specifying an even integer number of bytes.

Use RECORDSIZE with MOVE TO, MOVE FROM or FIELD statements to perform run-time association of data elements with record buffer positions. RMS indexed files cannot use dynamic buffering because you must map their key variables.

3.8.4 Record Blocking

You can reduce disk accessing by filling disk blocks completely. This also makes more records available for processing at one time.

For example, a file of 128 byte records can store four records in each virtual block. One disk access makes four records available for processing. In contrast, a file with a separate block for each record requires four times as much disk activity.

Through RMS, BASIC-PLUS-2 performs all blocking and deblocking on sequential, relative, and indexed files. You can, however, do your own blocking and deblocking with FIELD and MOVE statements on:

- RMS sequential files.
- Terminal-format files.
- Block I/O files.

The individual sections for each file type explain the valid record operations you can perform.

3.8.4.1 MOVE Statement — The MOVE statement defines data fields and moves them to and from the dynamic record buffer. The format of the MOVE statement is:

MOVE {FROM / TO} [#]num-exp%, I/O list

where:

FROM	moves the data from the record record buffer associated with the channel number and places it in the variables in the I/O element list.
TO	moves the data from the variables in the I/O element list and places them in the record buffer associated with the channel number.
num-exp%	is the file number associated with the opened file.
I/O list	is a list of the variables you move. Separate them with commas.

For example:

```
50 MOVE FROM #9%, A$, COST, NA.ME$, ID.NUM%
```

moves a record with four data fields from the record buffer to the elements in the I/O list. This includes: a string field with a default length of 16 characters (A$), a real number field (COST), a second 16-character string field (NA.ME$), and an integer field (ID.NUM%).

You can input values to the I/O list in your program. For example:

```
20 INPUT "NAME"; A$
30 INPUT "COST PER UNIT"; COST
```

Valid I/O list variables are:

- Scaler variables
- Arrays
- Array elements
- FILL items

You can specify string length in the I/O list. For example:

```
NA.ME$ = 30%
```

Because BASIC dynamically assigns space for string variables, the default string length during a MOVE TO is the length of the string; the default for MOVE FROM is 16 characters.

An array specified in a MOVE statement must have the format:

A( ), A%( ) or A$( ) for a list
A(,), A%(,) or A$(,) for a matrix

For example:

```
60 MOVE FROM #5%, A$( ), C(,), D%(,), LIST$(,), NUM( )
```

moves two lists (A$ and NUM) and three matrices from the buffer into the specified variables.

NOTE

The MOVE statement moves the contents of row zero and column zero.

You specify an array element by naming the array and the subscripts of that element; for example, A$(25) or B(3,2).

Successive MOVE statements to or from the buffer start at the beginning of the record buffer. If a MOVE TO only partially fills the buffer, the rest of the buffer is unchanged.

Use the GET statement to read a record from the file. Then MOVE the data FROM the buffer to assign the data values to the variables in the I/O list, and reference the variables in your program.

A MOVE TO transfers data from the variables into the I/O buffer. A PUT or UPDATE statement then moves the data from the buffer to the file.

For example:

```
 5 DIM B%(3,3)
10 OPEN "MOV.DAT" AS FILE #1% &
        ,RELATIVE VARIABLE        &
        ,ACCESS MODIFY, ALLOW NONE &
        ,RECORDSIZE 100%
20 GET #1%
25 Q = Q + 1
30 MOVE FROM #1%, A, B%(,), C$ = 10%
40 A = A + Q\ B%(3,3) = 128%      &
   \     C$ = "NEW RECORD"
50 MOVE TO #1%, A, B%(,), C$ = 10%
60 UPDATE #1%
70 CLOSE #1%
80 END
```

opens file MOV.DAT, reads the first record into the buffer, modifies part of the buffer, and moves the data from the buffer into the variables specified in the MOVE FROM statement. The string length of C$ (line 50) is set to 10 characters.

The MOVE TO statement moves the data from the named variables into the buffer. The UPDATE statement writes the record back into file #1 (MOV.DAT). Line 70 closes the file.

FILL items are valid elements in MOVE statements. They mask parts of the record buffer and enable you to: (1) access portions of a record, (2) skip over fields, and (3) reserve space in or between data elements. FILLs are available for all data types. Table 3–6 summarizes their formats and allocations.

3.8.4.2 FIELD Statement — The FIELD statement associates string names with all or part of a record buffer.

NOTE

BASIC-PLUS-2 supports the FIELD statement for BASIC-PLUS compatibility only and it is not recommended for new program development.

The FIELD statement has the format:

FIELD num-exp%, expression AS string variable

[,expression AS string variable...]

where:

num-exp%	is the channel number of the file.
expression	is an integer that represents the length of the data field.
string variable	is a unique string variable name.

A previous GET of the record associates the variables with fields in the record buffer. For example:

```
40 GET #2%
50 FIELD #2%, 10% AS A$, 20% AS B$, 3% AS F$
```

associates three contiguous strings in the record buffer, A$, B$, and F$, with lengths of 10, 20, and 3 characters. The total number of characters represented is 33. This total must be less than or equal to the record buffer size.

NOTE

The data in block I/O files must be in string format; therefore, you must convert integers and real numbers for string storage. The CVT functions perform these conversions.

You can then PUT the record (buffer) into the file:

```
300 PUT #4%
```

3.8.4.3 Writing Blocked Records — You write blocked records with the MOVE statement by skipping over any previous records and inserting the new record in the next available position in the block. For example:

```
30 OPEN "FILES" FOR OUTPUT AS FILE #8%  &
        ,RECORDSIZE 512%
40 INPUT "HOW MANY BLOCKS DO YOU WANT TO WRITE";N%
50 FOR I% = 1% TO N%
60     FOR J% = 0% TO 15% !16 RECORDS PER BLOCK
70     INPUT "NAME"; NA.ME$
80     INPUT "PART NUMBER"; NUMBER%
90     INPUT "COST PER UNIT"; COST
100    MOVE TO #8%, FILL$ = J%*32%, NA.ME$ = 26%, NUMBER%, COST
110    NEXT J%
120    PUT #8% !PUT 16 RECORDS
130 NEXT I%
```

Each pass through the loop increments the FILL$ field. The program skips over all previous records and writes the new record in the file.

The FIELD statement operates similarly. Your program can fill and PUT blocks automatically with a FOR/NEXT loop. For example, this program fills a block with 64 byte records and PUTs the block in the file:

```
100 OPEN "LOCATE.INV" FOR OUTPUT AS FILE #1%
110 INPUT "HOW MANY ITEMS IN THE FILE"; N%
120 FIELD #1%, 5% AS L$\LSET L$ = STR$(N%)
130 C% = 1% !INITIALIZE COUNTER
140 U% = N%/8% + 1% !U = NUMBER OF BLOCKS IN THE FILE
150 F% = 1% !F% IS THE FIRST DATA RECORD
160 FOR R% = 1% TO U%
170     FOR J% = F% TO 7%
180     PRINT "LOCATION DATA FOR PART NUMBER"; C%
190     INPUT "FLOOR, BIN, RACK"; S1%, S2%, S3%
```

```
200     PRINT "NOW THE NAME OF THE ITEM";
210     LINPUT X$
220     FIELD #1%, J% * 64% AS D$, 1% AS F$, 2% AS B$    &
          ,2% AS R$, 59% AS N$
230     LSET F$ = CHR$(S1%)
240     LSET B$ = CVT%$(S2%)
250     LSET R$ = CVT%$(S3%)
260     LSET N$ = X$
270     C% = C% + 1
280     GO TO 330 UNLESS C% < = N%
290     NEXT J%
300 PUT #1%, RECORD R%
310 F% = 0%
320 NEXT R%
330 PUT #1%
340 CLOSE #1%
350 PRINT "ALL DONE"
360 END
```

Each pass through the loop increments the dummy variable, D$, 64 bytes (the length of one record). This enables BASIC to skip over previous records and write the new record in the next available position. However, the FIELD statement generates the additional overhead of CVT conversions that the MOVE statement does not need.

The MAP statement cannot block records directly. Because the MAP statement defines the record buffer, you must use multiple MAPs to define each different record in that buffer. To do this, you must use different variable names in each MAP. For example:

```
60 MAP (ASA) A$ = 80%, FILL$ = 422%
100 MAP (ASA) FILL$ = 80%, A1$ = 80%, FILL$ = 342%
140 MAP (ASA) FILL$ = 160%, A2$ = 80%, FILL$ = 282%
```

and so on. This consumes programmer time and program space.

3.8.4.4 Reading Blocked Records — MOVE and FIELD statements can deblock records by incrementing a dummy variable or FILL field to skip over previously read records. For example, in the program:

```
730 INPUT "HOW MANY RECORDS DO YOU WANT TO READ";N%
740 FOR I% = 1% TO N%
745 GET #1%
750     FOR J% = 0% TO 31%
760     MOVE FROM #2%, FILL$ = J%*16, NA.ME$ = 26%, NUMBER%, COST
770     PRINT NA.ME$, NUMBER%, COST
780     NEXT J%
790 NEXT I%
```

BASIC (1) increments the FILL field FILL$, (2) skips over previously accessed records to remap the buffer, and (3) displays the values of the I/O list on your terminal.

The FIELD statement reads records similarly. For example, in the program:

```
130 INPUT "HOW MANY BLOCKS DO YOU WISH TO READ"; N%
140 FOR I% = 1% TO N%
145 GET #6%
150     FOR J% = 0% TO 7%
160     FIELD #6%, J%*64% AS D$, 1% AS F$, 2% AS B$  &
        2% AS R$, 59% AS N$
170     S1% = CVT$%(F$)
180     S2% = CVT$%(B$)
190     S3% = CVT$%(R$)
200     X$ = N$
210     PRINT S1%, S2%, S3%, N$
220     NEXT J%
230 NEXT I%
```

BASIC (1) increments the dummy variable in the FIELD statement, (2) skips over previously accessed records, (3) reads a new record, and (4) displays the values of the I/O list on your terminal.

The MAP statement cannot deblock records directly. You must use multiple MAPs and respecify the variables for each record. For example:

```
790 GET #2%
800 MAP (ASA) A$ = 64%, AGE%, FILL$ = 446%
840 MAP (ASA) FILL$ = 64%, A1$ = 64%, AGE1%, FILL$ = 382%
```

and so on. You cannot automatically increment FILL fields and use the MAP statement in a loop.

3.8.5 Mixing MAP and MOVE Statements

You can often process records more quickly by mixing dynamic (MOVE) and static (MAP) buffering. This is true if: (1) your records have a variety of formats, but (2) at least one group of records has an identical format.

By branching, you can:

- Process the records with identical formats through MAPs (and gain the advantage of compile-time data definitions).
- Process the records with different formats through MOVE statements (and define the data at run-time).

For example:

```
 5        ON ERROR GOTO 19000
10        MAP (MASTER) M.EMP.NUM%, REGULAR.HRS
20        MAP (WEEK) EMP.NUM%, HRS.THIS.WEEK
30        OPEN "EMP.MST" AS FILE #1%, RELATIVE, MAP MASTER
40        OPEN "EMP.WEK" AS FILE #2%, SEQUENTIAL, MAP WEEK
50        OPEN "EMP.UPD" AS FILE #3%, SEQUENTIAL
60        GET #2%
70        GET #1%, RECORD EMP.NUM%
```

```
80          IF REGULAR.HRS > HRS.THIS.WEEK    &
                 THEN MSG$ = "EMPLOYEE HAS NO OVERTIME"    &
\                      MOVE TO #3%, EMP.NUM%, MSG$, ZER%   &
                 ELSE MSG$ = "EMPLOYEE HAS OVERTIME" + SPACE$(2%)   &
\                OVER.TIME = HRS.THIS.WEEK - REGULAR.HRS   &
\                T.OVER.TIME = T.OVER.TIME + OVER.TIME     &
\                MOVE TO #3%, EMP.NUM%, MSG$, OVER.TIME
90          PUT #3%
100         GOTO 50
19000       IF ERR = 11%    &
          THEN MSG$ = "TOTAL OVERTIME HOURS" + SPACE$(4%)   &
\                MOVE TO #3%, MSG$, T.OVER.TIME    &
\                PUT #3%   &
\                RESUME 32000    &
                 ELSE ON ERROR GO TO 0
32000       CLOSE #3%, #2%, #1%
32767       END
```

3.8.6 MAP Statements vs. FIELD and MOVE

In most applications, the MAP statement has these advantages over FIELDs and MOVEs:

- Data definitions are made at compile time. Programs with MAP statements run faster than those with FIELD or MOVE statements.
- MAPs point to the record buffer, but do not move data. In contrast, MOVEs maintain data in its original format and take time to move data to or from the record buffer.
- MAPs permit all data types. When using FIELD statements, you must convert data to or from string format before your program can use it.
- When processing similar records, multiple MAPs allow redefinition of the record buffer without data conversions or MOVEs.

MAPs have two disadvantages:

- When you do your own record blocking, MAPs require constant reiteration, and consume programmer time and program space without significantly improving run time.
- If your program design does not allow other files to re-use static buffers, MAP statements can waste program space.

3.9 Advanced File Operations

You can improve file efficiency with OPEN statement keywords and by controlling file sharing. The following sections describe these operations.

3.9.1 OPEN Statement Keywords

This section explains the OPEN statement keywords that enable you to structure your file more efficiently on RSX. These keywords are: WINDOWSIZE, TEMPORARY, FILESIZE, SPAN, CONTIGUOUS, and CONNECT.

3.9.1.1 WINDOWSIZE — A window is a set of retrieval pointers BASIC maintains for virtual to logical block mapping. From the structure and content of a file's pointers, the system can equate virtual and logical blocks.

The file processor stores retrieval pointers in a file header, using enough file headers to cover the file. A file can contain up to 102 pointers. Each pointer records:

- The number of blocks the pointer maps.
- The logical block number where the group of blocks starts.

One pointer can map a maximum of 256 blocks. Therefore, a file header can map a maximum of 26,112 logical blocks.

When you open a file, the system reads in a set of retrieval pointers and uses them to create a "window" to map the file. As additional blocks of the file are read in, new pointers are read in as part of the window. This process of bringing in new pointers to the window is called "window turning." In this way, the entire file can be mapped to its physical location on the disk.

When you reduce window turning, you improve performance. In RSX, you can reduce window turning by:

- Specifying a windowsize larger than the system default with the WINDOWSIZE keyword.
- Initializing the disk volume containing the file with a windowsize greater than the default of seven pointers per window. Your system manager can help you with this procedure.
- Mounting the volume containing the file with the /WIN switch to specify a windowsize greater than the volume default. Again, your system manager can help.

3.9.1.2 TEMPORARY — Specifying TEMPORARY in the OPEN statement tells BASIC to delete the file when you close it. TEMPORARY should be used only in the creation of files.

3.9.1.3 FILESIZE — With the FILESIZE attribute, you can allocate disk space for a file when you OPEN it. For example:

```
100 OPEN "VALUES.DAT" FOR OUTPUT AS FILE #3%, FILESIZE 50%
```

allocates 50 blocks of disk space for the file "VALUES.DAT". Specify FILESIZE as an integer number of blocks.

Pre-extension has several advantages. First, the system can create a complete directory structure for the file, instead of allocating and mapping additional windows when needed. Second, you reserve the needed disk space for your application. You will not run out of space when the program is running. Third, pre-extension can make some of the file's windows contiguous. This can permit contiguous retrieval entries.

Pre-extension can be a disadvantage if it allocates disk space needed by other users, however.

3.9.1.4 SPAN — The SPAN attribute for sequential files allows records to cross block boundaries. If records cross block boundaries, BASIC packs records into the file end-to-end, allowing for control information and padding.

If you do not allow records to SPAN blocks, BASIC packs records into each block, allowing for control information and padding. You will waste space in the file if: (1) you do not allow records to span blocks, and (2) your records do not exactly fit into a block. SPAN is the default.

When block boundaries restrict records, each record must be less than 512 bytes. This can waste extra bytes at the end of the file. When records SPAN block boundaries, however, BASIC can write:

- More than one record in each block (for records shorter than 512 bytes).
- A partial record in each block (for records longer than 512 bytes).
- Records end-to-end without regard to block boundaries.

For example, with NOSPAN, only four 120-byte records fit into a disk block. When you specify SPAN, BASIC begins writing the fifth record in the block, and continues that record in the next block. This minimizes wasted disk space and increases the file's capacity.

3.9.1.5 Contiguous — The physically adjoining blocks of a contiguous file minimize disk searching, and so decrease file access time. Once the system knows where a contiguous file starts on the disk, it need not use as many retrieval pointers to locate windows in that file. Rather, it can access data by calculating the distance from the beginning of the file to the desired data. You may find, however, that not enough contiguous disk space exists for your file.

Be sure to pre-extend the file with the FILESIZE option. This enables RSX-11M to find enough contiguous space to store the file.

3.9.1.6 Connect — A record access stream can handle only one series of records at a time. However, you can connect more than one record access stream to a relative or indexed file and maintain more than one context during file processing.

Each stream represents an independent, active sequence of record operations. For example, a program can open an indexed file and connect to two record access streams. In one stream, the program can use the primary key to access records randomly; in the other, you can access records sequentially in the order specified by the alternate key.

3.9.1.7 UNDEFINED — When you do not know the attributes of a file, you can OPEN it with ORGANIZATION UNDEFINED. You must BUILD the program that opens the file with all file switches. For example:

```
BUILD UNKNON/SEQ/REL/IND/VIR
```

You then open the file FOR INPUT and with ACCESS READ. For example:

```
1000 OPEN "TEST.DAT" FOR INPUT AS FILE #%,   &
            ORGANIZATION UNDEFINED, ACCESS READ
```

Use MAP or MOVE statements to deblock records, and you read them with FIND and GET statements. You can also use the FSP$ function to return the characteristics of the file. See FSP$ for more information.

3.9.2 File Sharing

Except for sequential files on non-disk devices, programs can share read access to all files. The ACCESS and ALLOW keywords control file sharing. The OPEN FOR INPUT and OPEN FOR OUTPUT clauses have no effect. Table 3-7 summarizes file sharing.

Table 3-7: File Sharing

File Type	User Access: Reading	User Access: Writing
Sequential Non-disk	Single User	Single User
Sequential Disk Device	Single/Multiple Users	Single User
Relative	Single/Multiple Users	Single/Multiple Users
Indexed	Single/Multiple Users	Single/Multiple Users

The ALLOW attribute in the OPEN statement specifies the types of operations you permit other programs to perform on the file while you have it open. The specifications you can make in the ALLOW attribute, and the operations they permit other users to perform, are:

READ or NONE — permits others to READ the file only. READ is the default.

MODIFY — permits full access to other users.

WRITE — permits others to WRITE to the file, but prevents UPDATE operations.

NOTE

Multiple programs that share the same file must specify ALLOW MODIFY or ALLOW WRITE. Also, you cannot extend any file opened with ALLOW MODIFY.

The ACCESS attribute in the OPEN statement specifies the record operations you perform on the file. The specifications you can make in the ACCESS attribute, and the operations they refer to, are:

READ	specifies GET and FIND operations on records in the file.
WRITE	specifies PUT operations on the file.
MODIFY	specifies GET, FIND, PUT, and UPDATE operations on records in sequential, relative, and indexed files; it specifies DELETE operations on records in relative and indexed files.
SCRATCH	specifies GET, FIND, PUT, UPDATE, and SCRATCH operations on records in sequential files on disk.
APPEND	specifies PUT operations at the end of an existing sequential file on disk.

FIND and GET operations on relative and indexed files lock the bucket containing the accessed record. This ensures that other programs do not disrupt the modifications you make to a record. The lock remains in effect until you specify another record operation.

You can explicitly unlock a bucket after a FIND or GET by specifying an UNLOCK statement. For example:

```
70 UNLOCK #1%
```

makes the last record accessed by a GET or FIND on channel 1 accessible to other users.

If another program attempts an operation on a locked bucket, the operation fails. BASIC returns ERR = 154:

```
?Record/bucket locked
```

If you plan to extend a relative or indexed file after creating it, specify READ as the ALLOW attribute. You cannot extend a shared file. You generate the "Protection violation" error message if you try to extend a file that specifies a WRITE or MODIFY clause in the ALLOW attribute.

NOTE

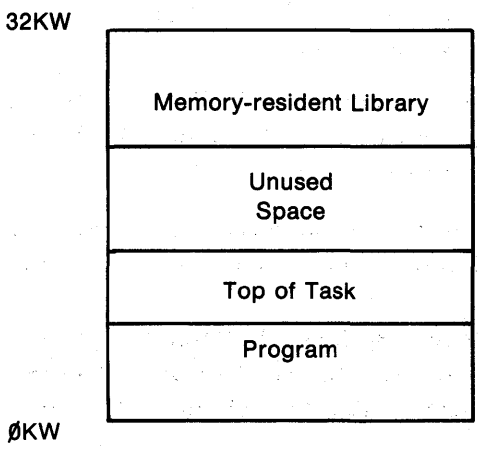
On RSX-11M, if the first program has specified ACCESS WRITE and ALLOW READ, other programs with ACCESS READ, allow no-write declarations can still open the file. However, the reading programs are not protected from changes being made by the writing program.

See the *RMS-11 User's Guide* for more information.

3.10 Memory Allocation

You can improve program design by understanding how BASIC allocates internal buffers and memory. In general, your program's task image has three parts when first loaded into memory, as shown in Figure 3–1:

Figure 3–1: Memory Allocation



The memory-resident library contains the OTS routines shared by all users. Because the library is an installation option, see your system manager to see if it is available on your system.

The program area contains: (1) compiled BASIC code (threads), (2) the BASIC Object-Time System (OTS), and (3) record buffer space, if your program includes a MAP statement.

During execution, BASIC uses any unused space at load time for string space and to establish buffers for I/O operations.

The EXTEND TASK system service permits your program to grow by adding the unused space to the program space. BASIC divides the unused space into two areas: (1) I/O space, allocated from the top of the program toward the memory-resident library, and (2) string space, allocated from the top of the currently mapped memory downwards. This mapping creates free space between the I/O and string space. See Figure 3–2.

BASIC allocates free space when the program needs dynamic space for either I/O or string operations. If there is not enough free space, the strings are compressed to create the needed area. If the string compression fails, BASIC issues another EXTEND TASK directive to extend the task. BASIC cannot extend the task beyond 32KW. In this case, the program aborts with a "Maximum memory exceeded" error message. If the expansion succeeds, string space moves to the top of the newly extended task, thereby creating a larger free space area. BASIC then allocates this area for I/O or string use and the program continues.

3.10.1 I/O Allocation

For each open file, BASIC allocates I/O buffers and control blocks for input and output. BASIC allocates these areas from dynamic space.

3.10.1.1 Record Buffer — If your program includes a MAP statement, the record buffer comes from the program space. In contrast to buffers in dynamic space, buffers in the program area (user-buffers) eliminate overhead in the BASIC space manager. In addition, this arrangement allows easier interface to MACRO subroutines.

The compiler creates all MAPs as PSECTs. Because both BASIC and MACRO know this PSECT name, MACRO subprograms can access data in the record buffer.

If your program has no MAP statement, BASIC allocates the record buffer from the dynamic space. There is no user mapping to dynamic space.

3.10.1.2 Device Buffer — Through the device buffer, the system's file services can read and write physical records (blocks) for BASIC programs. On disk, the buffer size is 512 bytes. On magnetic tape, the size is between 18 and 8192 bytes.

Figure 3-2: Allocation of I/O Buffer and String Space

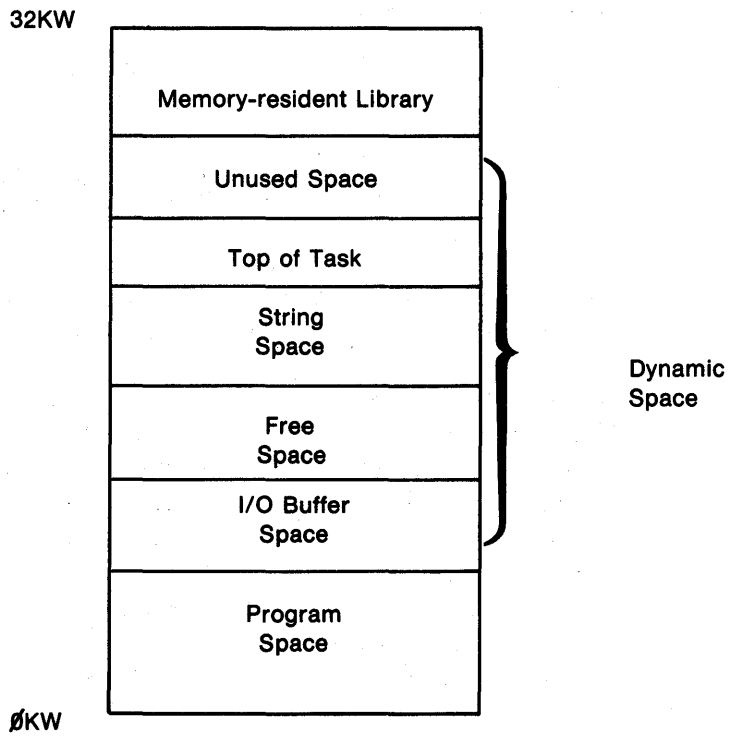
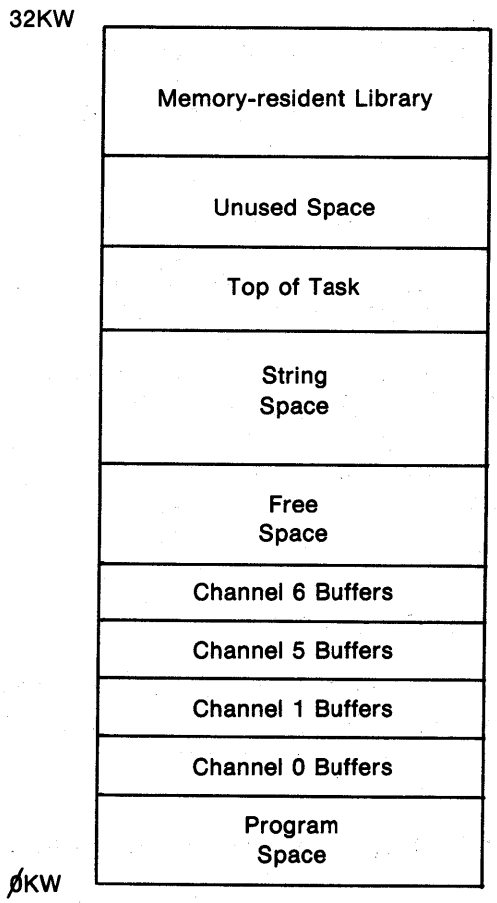


Figure 3–3 represents the memory allocation of a task that opens channels 1, 5, and 6, in that order.

Closing the last opened file first makes more free space available to your program. For example, closing channel 1 while channel 5 and 6 are open creates a second hole in the dynamic buffer space. BASIC maintains a linked list of these holes in ascending memory address order. When you attempt to open a new file, BASIC checks this list to try and satisfy the request from an existing secondary hole. If it cannot, BASIC allocates from free space.

When your program closes a file and returns the buffers to the free space, BASIC checks the linked list to see if it can return any holes below those buffers to the free space as well. For example, closing channel 5 in the previous example leaves a hole between the buffers allocated for channels 1 and 6. Closing channel 6 returns the buffer space associated with channel 6 to free space, and returns the buffer of channel 5 as well.

Figure 3–3: Order of Memory Allocation



3.10.1.3 Control Blocks — For each opened channel, BASIC maintains a control block of 156 bytes. This block is a buffer header that stores pointers and counters for the file. The pointers point to the record buffer, the device buffer, and any RMS control blocks.

3.10.1.4 RMS Control Structures — BASIC allocates dynamic space for RMS file control blocks. These blocks include:

- Buffer descriptor blocks (BDBs)
- Internal File Attribute Blocks (IFABs)
- Internal Record Attribute Blocks (IRABs)
- Extended Attribute Blocks (XABs)

3.10.1.5 Miscellaneous Allocations — Dynamic Space is also allocated for:

- A native mode file for channel 0 of your task. This file is always open, and is treated like other file I/O channels.
- A channel header for an imaginary channel used to handle READ DATA statements.
- An internal scratch area for data conversions, PRINT USING operations, and element transmissions.

3.10.2 Order of Memory Allocation

BASIC allocates buffer space in the order your program opens the files.

3.10.3 FIELD Statements

FIELD statements allow direct access to the record buffer. BASIC supports FIELD for BASIC-PLUS compatibility.

FIELDs present a major obstacle to normal memory allocation. It is possible to FIELD into a closed channel. This forces BASIC to lock the FIELDed buffer and all buffers below, preventing a return of those buffers to free space. The linked list solves some of these problems, but using FIELD can also fragment the dynamic space. If your program uses FIELD statements, try to FIELD into the first opened channel(s) only.

3.11 Magnetic Tape Operations

BASIC provides both sequential and device-specific files on magnetic tape. RMS tapes are ANSI formatted and file structured; native mode tapes are unformatted and non-file structured.

3.11.1 RMS File-Structured Magnetic Tapes

RMS magnetic tape files are sequential access only. You can read or write only one file at a time, and the files are not available to other users.

3.11.1.1 Opening an RMS Magnetic Tape FOR OUTPUT — You create and open the magnetic tape for output with the syntax:

```
OPEN "dev:filename" FOR OUTPUT AS FILE [#]num-exp%

,[ORGANIZATION] SEQUENTIAL

[,MAP mapname]
[,BLOCKSIZE num-exp%]
[,NOREWIND]
[,RECORDSIZE num-exp]
```

For example:

```
40 OPEN "MT1:PARTS.DAT" FOR OUTPUT AS FILE #2% &
          ,RECORDSIZE 256%
```

opens the file "PARTS.DAT" and writes 256 byte records. A file opened FOR OUTPUT permits WRITE access only.

3.11.1.2 Opening an RMS Magnetic Tape FOR INPUT — You open the magnetic tape for input with the syntax:

```
OPEN "dev:file name" FOR INPUT AS FILE [#]num-exp%

,[ORGANIZATION] SEQUENTIAL

[,ACCESS APPEND]
[,MAP mapname]
[,BLOCKSIZE num-exp%]
[,NOREWIND]
[,RECORDSIZE num-exp]
```

For example:

```
100 OPEN "MT2:PAYROLL.DAT" FOR INPUT AS FILE #4% &
          ,RECORDSIZE 1024%
```

opens the file "PAYROLL" and specifies 1024 byte records. A file opened FOR INPUT permits READ access only. BASIC positions the magnetic tape at the start of the file unless you specify ACCESS APPEND.

3.11.1.3 Positioning an RMS Magnetic Tape — Use NOREWIND to position the tape for reading and writing:

- Specifying NOREWIND when the file is created positions the tape at the logical end-of-tape and leaves the unit open for writing. If you omit NOREWIND, you start writing at the beginning of the tape (BOT), logically deleting all subsequent files.
- Specifying NOREWIND when the file is OPEN FOR INPUT starts a search for the file at the current position. The search continues to the logical end-of-tape. If the record is not found, BASIC rewinds and continues the search

until reaching the logical end-of-tape again. Omitting NOREWIND tells BASIC to rewind the tape and search for the file name until reaching the end-of-tape. In either case, you receive an error message if the file does not exist.

3.11.1.4 Record Operations — You write records to an RMS sequential file with PUT statements. You read records with GET statements.

3.11.1.4.1 Writing Records to the File (PUT) — The PUT statement writes sequential records to the file. For example:

```
70      OPEN "MM0:TEST.DAT" FOR OUTPUT AS FILE #2%,    &
           ORGANIZATION SEQUENTIAL, RECORDSIZE 20%,    &
           BLOCKSIZE 4%
80      INPUT "NAME";A$
90      MOVE TO #2%, A$ = 20%
100     PUT #2%
110     INPUT "WRITE ANOTHER RECORD";B$
120     IF B$ = "YES" THEN 80
130     CLOSE #2%
140     END
```

writes a record to the file. Successive PUTs write successive records.

Each PUT writes one buffer, or tape block, to the file. If your OPEN statement specifies a RECORDSIZE clause, the record buffer length equals RECORDSIZE. For example:

```
RECORDSIZE 60%
```

specifies a record length and a record buffer size of 60 bytes. You can specify a record length between 18 and 8192 bytes. The default is 512 bytes. BASIC converts any value less than 19 to 18.

If you also specify BLOCKSIZE, the buffer equals BLOCKSIZE. For example:

```
RECORDSIZE 60%, BLOCKSIZE 4%
```

specifies a record length of 60 bytes and an I/O buffer size of 240 bytes (60*4). You specify BLOCKSIZE as an integer number of records. This integer must be divisible by 4. The total I/O buffer length cannot exceed 8192 bytes. The default is a buffer (tape block) of 512 bytes.

3.11.1.4.2 Reading Records from the File (GET) — The GET statement reads one block of records into the buffer. For example:

```
240     OPEN "MM0:TEST.DAT" FOR INPUT AS FILE #5%,  &
           ORGANIZATION SEQUENTIAL, RECORDSIZE 20%, &
           BLOCKSIZE 4%
250     GET #5%
260     MOVE FROM #5%, A$ = 20%
270     PRINT A$
280     INPUT "DO YOU WANT ANOTHER RECORD";B$
290     IF B$ = "YES" THEN 250
300     CLOSE #5%
310     END
```

reads a block of records from the file on channel 5. Successive GETs read successive records.

3.11.1.5 Record Blocking — Through RMS, BASIC controls the blocking and deblocking of records. BASIC checks each PUT operation to see if the specified record fits in the tape block. If it does not, RMS fills the rest of the block with blanks and starts the record in a new block. Records cannot span blocks in magnetic tape files.

When you read blocks of records, your program can issue successive GETs until it locates the fields of the record you want. For example:

```
110 MAP (XXX) NA.ME$ = 5%, ADDRESS$ = 20%
120 OPEN "MM0:FILE.DAT" FOR INPUT AS FILE #4%,  &
       SEQUENTIAL, MAP XXX
130 GET #4%
140 IF NA.ME$ = "JONES" THEN &
       PRINT NA.ME$; "LIVES AT"; ADDRESS$  &
       ELSE 130
150 CLOSE #4%
160 END
```

finds and displays a record on the terminal. You can test the RECOUNT variable to see how many bytes were read in the GET operation.

3.11.1.6 Closing an RMS Magnetic Tape File (CLOSE) — The CLOSE statement ends I/O to the file. For example:

```
590 CLOSE #6%
```

ends input and output to the file open on channel 6.

If the file is OPEN FOR INPUT, CLOSE has no further effect. If the file is OPEN FOR OUTPUT, BASIC:

- Writes file trailer labels (2 end-of-file marks) following the last record.
- Backspaces over the last end-of-file mark.
- Releases allocated buffer space.
- Awaits further output.

BASIC does not rewind the tape.

3.11.1.7 OPEN Statement Keywords — RECORDSIZE and BLOCKSIZE control the size of the record and I/O buffers. NOREWIND controls the way BASIC positions your magnetic tape. The following sections describe these keywords.

3.11.1.7.1 RECORDSIZE — The RECORDSIZE attribute defines record length. You can specify a RECORDSIZE between 18 and 8192 bytes. The default is 512 bytes. For example:

```
120 OPEN "MT2:TEST.DAT" FOR OUTPUT AS FILE #6%   &
          ,RECORDSIZE 88%
```

opens the file TEST.DAT and allows you to write records 88 bytes long. When you omit the BLOCKSIZE attribute, RECORDSIZE defines the length of the record buffer as well.

3.11.1.7.2 BLOCKSIZE — The BLOCKSIZE attribute defines the number of records in each block. The default is one 512 byte record in each block. You specify BLOCKSIZE in the OPEN statement as an integer number of records. This BLOCKSIZE must be divisible by four. For example:

```
10 OPEN "MM0:[100,100] TEST.SEQ" FOR OUTPUT AS FILE #12% &
         ,ORGANIZATION SEQUENTIAL, RECORDSIZE 90%  &
         ,BLOCKSIZE 12%
```

opens the file "TEST.SEQ" with a RECORDSIZE of 90 bytes. BLOCKSIZE defines the size of a block as 12 records (1080 bytes). Therefore, your program contains a buffer of 1080 bytes. Every 12th GET or PUT causes a physical read or write, which moves 1080 bytes of data. The previous GETs or PUTs only move data into or out of the block buffer. The total BLOCKSIZE cannot exceed 8192 bytes.

3.11.1.7.3 NOREWIND — NOREWIND prevents BASIC from rewinding a magnetic tape when you open the file. For example:

```
10 OPEN "MT1:PAYROL.DAT" FOR OUTPUT AS FILE #1%   &
                    ,ORGANIZATION SEQUENTIAL, NOREWIND
```

opens "PAYROL.DAT" after advancing the tape to the logical end-of-tape. The default is REWIND. If you omit NOREWIND, the file opens at the beginning of the tape (BOT), logically deleting all subsequent files.

3.11.2 Native Mode Magnetic Tapes

Native mode permits non-file structured magnetic tape file operations. You OPEN a physical device and transfer data between the tape and your program.

3.11.2.1 Opening a Native Mode Tape FOR OUTPUT — You can OPEN a magnetic tape FOR OUTPUT with the syntax:

OPEN "dev:" FOR OUTPUT AS FILE [#]num-exp%

[,RECORDSIZE num-exp]
[,BLOCKSIZE num-exp%]
[,MODE num-exp%]

For example:

```
190 OPEN "MT1:" FOR OUTPUT AS FILE #1%, MODE 256%
```

opens tape drive MT1: for writing at 1600 bits per inch. A file opened FOR OUTPUT permits write access only.

3.11.2.2 Opening a Native Mode Tape FOR INPUT — You can OPEN a tape FOR INPUT with the syntax:

```
OPEN "dev:" FOR INPUT AS FILE [#]num-exp%

[,RECORDSIZE num-exp]
[,BLOCKSIZE num-exp%]
[,MODE num-exp%]
```

For example:

```
140 OPEN "MT2:" FOR INPUT AS FILE #2%
```

opens tape unit MT2: for reading. Opening a tape FOR INPUT permits READ access only.

3.11.2.3 MODE Values — MODE values in magnetic tape operations describe tape characteristics, but do not position the magnetic tape. Your program controls the position of the magnetic tape through the MAGTAPE function.

You determine the value of MODE with the formula:

MODE = E + P

where:

E (phase encoded) in bits per inch (BPI) is:

256 = 1600 BPI, phase encoded

0 = system default

P (parity) is:

0 = odd parity

1 = even parity

If you do not specify a MODE value, BASIC uses the system defaults.

3.11.2.4 Positioning the Tape (MAGTAPE Function) — The MAGTAPE function permits program control over magnetic tape files. The format of the MAGTAPE function is:

I% = MAGTAPE(F%,P%,U%)

where:

F% is the function code (1 to 9).

P% is the integer parameter.

U% is the internal channel number assigned to the selected open magnetic tape.

I% is the value returned by the function.

The function code (F%) determines the effect of the MAGTAPE function. The following sections assume that magnetic tape unit 1 is open on internal channel 2.

The explanation of each of these functions includes the word IMMEDIATE or WAIT. IMMEDIATE indicates that the monitor initiates the action and immediately returns control to the program. WAIT indicates that the program continues after completing the operation.

3.11.2.4.1 Off-Line (Rewind and Off-Line) Function

IMMEDIATE

Function code	= 1
Parameter	= unused
Value Returned	= 0

The Off-line function rewinds the specified magnetic tape and sets it to OFF-LINE (thus clearing READY). For example:

```
200 I% = MAGTAPE(1%,0%,2%)
```

rewinds and sets the magnetic tape open on channel 2 to OFF-LINE.

3.11.2.4.2 WRITE End-Of-File (EOF) Function

WAIT

Function code	= 2
Parameter	= unused
Value returned	= 0

The WRITE End-of-File function writes one EOF record at the current magnetic tape position. For example:

```
200 I% = MAGTAPE(2%,0%,2%)
```

writes an EOF on the MAGTAPE that is open on channel 2.

3.11.2.4.3 Rewind Function

IMMEDIATE

Function code	= 3
Parameter	= unused
Value returned	= 0

The Rewind function rewinds the selected magnetic tape. For example:

```
200 I% = MAGTAPE(3%,0%,2%)
```

rewinds the magnetic tape open on channel 2, but does not take the tape off-line.

3.11.2.4.4 Skip Record Function

WAIT

Function code	= 4
Parameter	= number of records to skip (1 to 32767)
Value returned	= number of records not skipped

The Skip Record function advances the magnetic tape until: (1) the specified number of records is skipped, or (2) the tape reaches the end of the file. For example:

```
200 I% = MAGTAPE(4%,50%,2%)
```

skips 50 records in the file open on channel 2.

3.11.2.4.5 Backspace Function

WAIT

Function code	= 5
Parameter	= number of records to backspace (1 to 32767)
Value returned	= number of records not backspaced

The backspace operation backspaces records until: (1) the specified number of records is skipped, or (2) the load point (BOT) is reached.

For example:

```
200 I% = MAGTAPE(5%,1%,2%)
```

backspaces one record on the magnetic tape opened on channel 2.

3.11.2.4.6 Set Density and Parity Function

IMMEDIATE

Function code	= 6
Parameter	= E+D*4+P
Value returned	= 0

where:

E (phase encoded) in bits per inch (BPI) is:

256 = 1600 BPI, phase encoded

0 = values of D and P

D (density) in bits per inch (BPI) is:

0 = 200 BPI (7 track only)

1 = 556 BPI (7 track only)

2 = 800 BPI (7 track only)

3 = 800 BPI (9 track only)

The density and parity function changes the density and/or parity of a magnetic tape drive. For example:

```
10 OPEN "MT0:" AS FILE #2%
20 I% = MAGTAPE(6%, 2%*4%+1%, 2%)
```

changes the density and parity of the 7 track magnetic tape drive to 800 BPI, even parity.

3.11.2.4.7 Tape Status Function

IMMEDIATE

Function code	= 7
Parameter	= unused
Value Returned	= status

Tape Status returns the status of the specified magnetic tape as a 16 bit integer. The bits that determine the status are summarized in Table 3–8. For example, you can test the value of I% in:

```
200 I% = MAGTAPE(7%,0%,2%)
```

to determine the status of the magnetic tape opened on channel 2.

Table 3–8: Magnetic Tape Status Word

Bit	Test	Meaning
15	I% < 0%	Last command caused an error.
14–13	(I% AND 24576%)/8192%	0 = 200 BPI 1 = 556 BPI 2 = 800 BPI, 7 Track 3 = 800 BPI, 9 Track
12	(I% AND 4096%) = 0%	9 track tape.
	(I% AND 4096%)<>0%	7 track tape.
11	(I% AND 2048%) = 0%	Odd parity.
	(I% AND 2048%)<>0%	Even parity.
10	(I% AND 1024%)<>0%	Magnetic tape is physically write locked.
9	(I% AND 512%)<>0%	Tape is beyond end-of-tape marker.
8	(I% AND 256%)<>0%	Tape is at beginning-of-tape (Load Point).
7	(I% AND 128%)<>0%	Last command detected an EOF.
6		Reserved
5	(I% AND 32%)<>0%	Unit is off-line.
4	(I% AND 16%)<>0%	Unit is TU16, TE16, TU45, or TU77
3	(I% AND 8%)<>0%	Mode is 1600 BPI phase encoded (TU16).
2–0		Reserved

3.11.2.5 Record Operations — You write and read records with PUT and GET operations.

3.11.2.5.1 Writing Records to the File (PUT) — The PUT statement writes records to the file in sequential order. For example:

```
10    OPEN "MT0:" FOR OUTPUT AS FILE #9%  !NON-FILE STRUCTURED
20    I% = MAGTAPE(3%,0%,9%)  !REWIND TAPE
30    INPUT "NAME";NA.ME$  !GET DATA FROM USER
40    MOVE TO #9%, NA.ME$  !PLACE DATA IN BUFFER
50    PUT #9%  !WRITE BUFFER TO TAPE
```

writes the contents of the buffer to the file. Successive PUTs write successive records.

The default record length (and therefore, the size of the buffer) is 512 bytes. The RECORDSIZE attribute tells BASIC to read or write records longer than 512 bytes. For example:

```
100 OPEN "MT0:" FOR INPUT AS FILE #1%, RECORDSIZE 900%
```

opens tape unit MT0: and processes records of 900 characters. You must specify an even integer larger than 512. If you specify a buffer length less than 512, BASIC uses the default of 512 bytes. If you try to PUT a record longer than the buffer, you receive the "Magtape record length error" message.

To write records shorter than the buffer, include the COUNT clause with the PUT statement. For example:

```
50 PUT #6%, COUNT 56%
```

writes a 56 character record to the file open on channel 6. If you do not specify COUNT, BASIC writes a full buffer. You can specify a minimum COUNT of 14, and a maximum COUNT equal to the buffer size.

3.11.2.5.2 Reading Records from the File (GET) — The GET statement reads records into the buffer. For example:

```
50      OPEN "MM1:" FOR INPUT AS FILE #1%
60      GET #1%
70      MOVE FROM #1%, A$
80      PRINT A$
90      I% = MAGTAPE(9%,0%,1%)
100     CLOSE #1%
```

reads a record into the buffer, prints a string field, and rewinds the file when closed. Successive GETs read successive records.

3.11.2.6 Closing a Native Mode Magnetic Tape — The CLOSE statement ends I/O to the file. For example:

```
300 CLOSE #12%
```

ends input and output to the file open on channel 12.

If the file is OPEN FOR INPUT, CLOSE has no further effect. If the file is OPEN FOR OUTPUT, BASIC:

- Writes trailer end-of-file records following the last record.
- Backspaces one record.
- Releases allocated buffer space.
- Positions the magnetic tape for further output.

The tape is not rewound unless you specified MAGTAPE(9%,0%,n%) in your program.

3.12 File Related Functions

3.12.1 STATUS Function

The STATUS function accesses: (1) the status word containing characteristics of the last opened file, or (2) additional RMS file information. The STATUS function has the following format:

```
A% = STATUS
```

where:

A% is:

(1) the RMS "STV" variable, as described in the *RMS-11 User's Guide*, Appendix B.

(2) an FSP$ device variable.

(3) The RMS "DEV" variable immediately after the file open. See the *RMS-11 MACRO Manual* for more information.

The STATUS variable after an FSP$ function is set depending on the presence of a device in the file name string.

- When the string contains no device name, the STATUS is undefined. This condition exists when bit 12 of flag word 2 equals 0.
- When the device name is logical and untranslatable (an actual device is not assigned), STATUS is undefined. This condition exists when bits 12, 13, and 14 of flag word 2 test as not equal to 0, and bit 15 tests as (S1%<0%).
- When the device name is either an actual device name or is logical and untranslatable (an actual device is assigned), STATUS is set for the device. This condition exists when bit 12 tests as not equal to 0, and bit 15 tests as equal to 0 (S1%> = 0%).

3.12.2 COUNT Clause

The COUNT clause specifies the number of bytes written in a PUT or UPDATE operation. The default is the maximum record size (MRS). The format is:

PUT num-exp% [,RECORD num-exp1] ,COUNT num-exp2

UPDATE num-exp% [,RECORD num-exp1] ,COUNT num-exp2

where:

num-exp	is the channel number associated with the file.
num-exp1	is the record (block) number of the data record.
num-exp2	is the number of bytes written in the operation.

For example:

```
PUT #5%, RECORD 62%, COUNT 122%
```

writes a record 122 bytes long into block number 62 of the file opened on channel 5. The COUNT clause must:

- Equal the record size in fixed-length records.
- Be less than or equal to the record size in variable- or stream-format records.

You can use COUNT on any RMS file that has fixed- or variable-length records, and on RMS sequential files that have stream format records. COUNT guarantees a true variable record by writing only the length you specify. BASIC automatically blocks and deblocks variable-length records.

Omitting COUNT defeats the purpose of variable-length records. BASIC writes the record, but also includes any data past the record up to the maximum record size.

3.12.3 RECOUNT Function

The RECOUNT function returns the number of characters transferred by the latest input operation. This equals the size of that record.

The format is:

A% = RECOUNT

where:

A%	is the number of characters from the last input operation.

The input operations are:

- INPUT and INPUT #
- INPUT LINE and INPUT LINE # (RECOUNT includes a terminator)
- LINPUT and LINPUT #
- GET
- Matrix input operations

Your program must execute RECOUNT immediately after an input operation. For example:

```
10      ON ERROR GO TO 100
20      OPEN "INPUT.FIL" FOR INPUT AS FILE #1% ,SEQUENTIAL
30      OPEN "OUTPUT.FIL" FOR OUTPUT AS FILE #2% ,SEQUENTIAL
40      GET #1%
50      PUT #2%, COUNT RECOUNT
60      GO TO 40
100     IF ERL = 40%, AND ERR = 11%      &
        THEN RESUME 110 ELSE ON ERROR GO TO 0
110     CLOSE #2%, #1%
120     PRINT "FILE COPIED"
32767   END
```

You can access RECOUNT with the PRINT statement:

```
55 PRINT RECOUNT
```

The RECOUNT variable is set for all input operations. For RMS files, RECOUNT equals the size of the record in the record buffer. For native files, RECOUNT equals the size of the last physical read. Your program must then deblock the records. If you are using a dynamic buffer and MOVE FROM and MOVE TO statements, you can use the CCPOS function to determine the current character position in the buffer.

3.12.4 CCPOS Function

CCPOS returns the current character position on an output line. The format is:

CCPOS(X)

where:

X is a file I/0 channel.

Specifying CCPOS(0%) returns the character position of a line output on your terminal. For example:

```
40 PRINT "TEST LINE";CCPOS(0%)
```

returns:

```
TEST LINE 8
```

because LINE ends at the eighth character position on your terminal.

Specifying a file channel number returns the current character position in the record buffer. For example:

```
410 GET #4%
420 MOVE TO #4%, LOCATION$, NUM%, JOBBER$ = 60%
430 PRINT CCPOS(4%)
```

returns the current character position in the record buffer.

3.12.5 FSP$ Function

The function FSP$ returns file organization data for an opened file. This function is intended for files OPENed as ORGANIZATION UNDEFINED, and your program must execute it immediately after the OPEN statement. The syntax of the FSP$ function is:

X$ = FSP$(channel-number)

In this program:

```
10  MAP (A) A$ = 32
20  MAP (A) A%(15)
30  OPEN "FIL.DAT" FOR INPUT AS FILE #1%,    &
    ORGANIZATION UNDEFINED, ACCESS READ
40  A$ = FSP$(1%)
50  REM A%(0%) = FILE CHARACTERISTICS
```

FSP$ generates the following values:

- A%(0%), which returns file characteristics:

  High Byte contains the RMS Record Attributes (RAT) field.

  Low byte contains the RMS Organization (ORG) and Record Format (RFM) Fields.

- A%(1) returns the RMS maximum record size (MRS) field.
- A%(2) and A%(3) return the RMS allocation quantity (ALQ) field.
- A%(4) returns the RMS bucketsize (BKS) field for disk files, or the RMS blocksize (BLS) field for magnetic tape files.
- A%(5) returns the number of keys.
- A%(6) and A%(7) return the RMS maximum record number (MRN) if the file is relative.
- A%(8) and A%(9) return the current block/record number.

A file opened with ORGANIZATION UNDEFINED must: (1) be open FOR INPUT only, and (2) include switches for all file types accessed by the program. For example, BUILD/SEQ/REL/IND.

Presence of the FSP$ and FSS$ functions in the same module generates an "Ambiguous symbols" error when you task-build the module.

3.12.6 FSS$ Function

FSS$ performs a filename scan on the argument string. The format for FSS$ is:

```
FSS$(A$,B%)
```

where:

A$ is the filename string.

B% is the starting position of the scan in the string.

The output is a 30-character string encoded as shown in Tables 3–9 through 3–11. On RSX: (1) omitting file name fields or (2) using wildcards returns a zero value for those fields.

Presence of the FSP$ and FSS$ functions in the same module generates an "Ambiguous symbols" error when you task-build the module.

Table 3–9: File Name String: Flag Word Bytes 1–30

Byte	Meaning
1	Job number multiplied by two
2	Version Number. If the version number is undefined, the byte returns zero. RSX only.
3–4	Seventh through ninth characters of the file name. RSX only.
5–6	Project and programmer number.
7–10	File name in RADIX-50 format.
11–12	File extension name in RADIX-50 format.
13–14	FILESIZE switch specification. RSTS/E only.
15–16	CLUSTERSIZE switch specification. RSTS/E only.
17–18	MODE specification. RSTS/E only.
19–20	Undefined
21	Zero—unless a protection code is specified or default exists. RSTS/E only.
22	Protection code if byte 21 is non-zero.
23–24	Device name if specified.
25	Unit number of device. If no unit device is given, byte 25 returns a zero.
26	255 if the unit number was specified.
27–28	Flag Word 1. See Table 3–10.
29–30	Flag Word 2. See Table 3–11.

Chapter 4
Program Segmentation

BASIC offers two ways to divide large tasks into smaller, more manageable modules: subprogramming and chaining. In subprogramming, control passes from a calling program to one or more subprograms within a single, executable image. In chaining, control passes from one executable program to another executable program. This chapter explains subprogramming and chaining.

4.1 Subprogramming

Subprogramming allows you to write frequently used procedures as small modules. You create and compile these modules separately, then build them into a single task image. Thus, subprogramming gives you the execution speed of the same task, while also giving you the coding and debugging advantages of modular construction.

BASIC programs or subprograms can call subprograms written in BASIC, MACRO, or COBOL. BASIC programs cannot call FORTRAN subprograms nor can programs written in other languages call a BASIC subprogram.

4.1.1 BASIC to BASIC Subprogramming

BASIC programs or subprograms can call BASIC subprograms. BASIC does not allow recursion: that is, a subprogram cannot call itself nor can a called subprogram call the subprogram that called it. For example:

Main Program

```
10 CALL SUB1
.
.
.
32767 END
```

(continued on next page)

Subprogram 1

```
10 SUB SUB1
20 CALL SUB2
.
.
.
32767 SUBEND
```

Subprogram 2

```
10 SUB SUB2
20 CALL SUB1
.
.
.
32767 SUBEND
```

The compiler returns the error message:

```
?Recursive subroutine call
```

4.1.1.1 Calling a BASIC Subprogram — BASIC transfers control from a calling program (or subprogram) to a subprogram by executing the CALL statement. Its format is:

CALL name [(param1,...param8)]

where:

name	is the subprogram name. The name must be a unique, one- to six-character string. A subprogram name cannot be the same as another subprogram, a COMMON, or a MAP within a single task image.
param1...param8	represent one to eight optional parameters BASIC passes from the calling program to the subprogram. The parameters must agree in data type and number with the parameters you define in the SUB statement of the subprogram. These parameters can be referred to as actual parameters.

A parameter is a value that can be passed from one routine to another. A routine can be a program, subprogram, or function. The value can be any numeric data or string data except for virtual arrays. See Section 4.1.1.2 for more information on parameters.

For example:

Calling Program

```
10 A% = 5%
20 B% = 10%
30 C% = 15%
40 CALL SUBPRG (B%)
50 PRINT A%, B%, C%
32767 END
```

Subprogram

```
10 SUB SUBPRG (B%)
20 B% = 5%
32767 SUBEND
```

When executed, the task returns:

```
5         5         15
```

The subprogram name can be either a quoted or an unquoted string. For example, these are valid subprogram names:

```
10 CALL "SUBPRG"
10 CALL 'SUBPRG'
10 CALL SUBPRG
```

You can include a dollar sign ($) or a period (.) in a subprogram name. However, if either of these is the first character in the name, the name must be enclosed in quotes.

You cannot use string variables to call a subprogram. BASIC interprets the string variable as the actual subprogram name. In this example:

```
10 NAM$ = "SUBPRG"
20 CALL NAM$
```

BASIC tries to call the subprogram named NAM$.

When BASIC executes a CALL statement it:

- Transfers control from the calling program to the SUB statement in the subprogram
- Passes the parameters you define in the CALL statement to the subprogram

The SUB statement must be the first statement of a BASIC subprogram. Its format is:

SUB name [(param1...param8)]

where:

name	is the same one- to six-character name you use in the CALL statement.
param1...param8	represent one to eight optional parameters BASIC passes from the calling program to the subprogram. The parameters must agree in data type and number with the parameters you define in the CALL statement. These parameters can be referred to as formal parameters.

BASIC suprograms must begin with the SUB statement and must end with the SUBEND statement.

The SUBEND statement tells BASIC to return control to the statement immediately following the CALL statement in the calling program. SUBEND must be the highest-numbered statement in the subprogram. For example:

```
10 SUB SUBPRG (STRI,NG$,RE,AL)
20 PRINT "THE VALUES OF THE FORMAL PARAMETERS ARE:"
30 PRINT STRI,NG$;RE,AL
32767 SUBEND
```

The SUBEXIT statement transfers control to the calling program. SUBEXIT is equivalent to an unconditional branch to the SUBEND statement. For example:

```
10 SUB SUB1 (A%)
20 A% = INT(10% * RND)
30 IF A% < 5% THEN SUBEXIT
40 A% = A% + 4%
32767 SUBEND
```

In line 30, if A% is less than 5, BASIC transfers control to line 32767 and line 40 is not executed.

4.1.1.2 Passing Data to a BASIC Subprogram — You can pass data from the calling program to the subprogram as parameters in a CALL statement or share data between program modules as elements of a COMMON or MAP or as records within a file.

Parameters can be modifiable or nonmodifiable. If the parameter is modifiable, the value you assign to the parameter in the subprogram replaces the value you assign in the calling program.

Modifiable parameters include:

- Entire arrays
- Simple string variables

- Simple numeric variables
- COMMON or MAP elements

The term "simple" means unsubscripted.

If the parameter is nonmodifiable, the value you assign the parameter in the subprogram does not replace the value assigned in the calling program.

Nonmodifiable parameters include:

- Constants
- Expressions
- User-defined and system-defined functions
- Individual array elements

You can force a modifiable parameter to be nonmodifiable by enclosing the parameter in parentheses.

In the following example, the calling program passes modifiable parameters to the first subprogram, and nonmodifiable parameters to the second subprogram.

Calling Program

```
 5  DIM Y%(10%,10%), X$(10%,10%)
10  PRINT "BEGIN CALLING PROGRAM"
20  A$ = "FIRST VALUE ="
30  B% = 124%
40  PRINT A$; B%
50  CALL SUB1(A$,B%)
60  PRINT A$; B%
70  X$(1%,1%) = A$
80  Y%(5%,5%) = B%
90  CALL SUB2(X$(1%,1%),Y%(5%,5%))
100 PRINT X$(1%,1%); Y%(5%,5%)
32767 END
```

Subprogram 1

```
10 SUB SUB1(A$,B%)
20 PRINT "SUBPROGRAM 1"
30 A$ = "SECOND VALUE ="
40 B% = 567%
32767 SUBEND
```

Subprogram 2

```
10 SUB SUB2(A$,B%)
20 PRINT "SUBPROGRAM 2"
30 A$ = "THIRD VALUE ="
40 B% = 742%
32767 SUBEND
```

When executed, the task returns:

```
BEGIN CALLING PROGRAM
FIRST VALUE = 124
SUBPROGRAM 1
SECOND VALUE = 567
SUBPROGRAM 2
SECOND VALUE = 567
```

In the preceding example:

1. The calling program prints the values of the variables A$ and B%, then calls the first subprogram and passes the variables to it.
2. The first subprogram changes the values of the variables and returns those values to the calling program for printing.
3. The calling program redefines the variables as array elements and passes them to the second subprogram.
4. The subprogram prints the string "SUBPROGRAM 2" and changes the values of A$ and B%; however, the values are not returned to the calling program because array elements are nonmodifiable.
5. When the calling program prints the values of A$ and B% for the final time, the values you assigned in the first subprogram are reprinted.

4.1.1.2.1 Passing Array Elements and Arrays — Single array elements are nonmodifiable when passed to the subprogram as parameters in the CALL statement. However, if you pass an entire array as a parameter, you can change one or all of the elements in that array.

NOTE

While BASIC allows you to pass an entire array, you cannot pass a virtual array as a parameter.

To pass arrays as parameters, specify the array name followed by a set of parentheses. Include a comma in the parentheses if the array is two-dimensional. For example:

```
10 CALL SUB1 (ARRAY.NAM(,))     !Passes a two-dimensional array
10 CALL SUB1 (ARRAY.NAM())      !Passes a list
```

In Figure 4–1, the calling program passes two string arrays, A$ and B$, to the subprogram. BASIC allows you to modify array A$ when you pass the entire array A$(,) as a parameter to the subprogram. BASIC does not allow you to modify array B$ when you pass a single array element B$(1%,1%) as a parameter to the subprogram.

Figure 4–1: Passing Array Elements and Arrays to BASIC Subprograms

Calling Program

```
10 DIM A$(5%,5%),B$(5%,5%)
20 FOR I% = 1% TO 5%
30 FOR J% = 1% TO 5%
40 A$(I%,J%) = "AAAA"
50 B$(I%,J%) = "ZZZZ"
60 NEXT J%
70 NEXT I%
80 PRINT "HERE ARE THE INITIAL VALUES OF THE ARRAYS:"
90 PRINT "ARRAY A"
100 MAT PRINT A$, \ PRINT
110 PRINT "ARRAY B"
120 MAT PRINT B$, \ PRINT
130 CALL SUB1(A$(,),B$(1%,1%))
140 PRINT "BACK TO THE CALLING PROGRAM"
150 PRINT "HERE ARE THE VALUES AFTER THE CALL:"
160 PRINT \ PRINT "ARRAY A"
170 MAT PRINT A$, \ PRINT
180 PRINT "ARRAY B"
190 MAT PRINT B$, \ PRINT
32767 END
```

Subprogram

```
10 SUB SUB1(DUM$(,),STR.DUM$)
20 PRINT "BEGIN SUBPROGRAM"
30 STR.DUM$ = "ARRAY ELEMENT B$(1%,1%) DOES NOT CHANGE"
40 PRINT STR.DUM$
50 FOR I% = 1% TO 5%
60 FOR J% = 1% TO 5%
70 DUM$(I%,J%) = "NEW"
80 NEXT J%
90 NEXT I%
32767 SUBEND
```

When executed, the task returns:

```
HERE ARE THE INITIAL VALUES OF THE ARRAYS:
ARRAY A
AAAA  AAAA  AAAA  AAAA  AAAA

AAAA  AAAA  AAAA  AAAA  AAAA

AAAA  AAAA  AAAA  AAAA  AAAA

AAAA  AAAA  AAAA  AAAA  AAAA

AAAA  AAAA  AAAA  AAAA  AAAA
```

(continued on next page)

```
ARRAY B
ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

BEGIN SUBPROGRAM
ARRAY ELEMENT B$(1%,1%) DOES NOT CHANGE
BACK TO THE CALLING PROGRAM
HERE ARE THE VALUES AFTER THE CALL:

ARRAY A
NEW  NEW  NEW  NEW  NEW

NEW  NEW  NEW  NEW  NEW

NEW  NEW  NEW  NEW  NEW

NEW  NEW  NEW  NEW  NEW

NEW  NEW  NEW  NEW  NEW

ARRAY B
ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ

ZZZZ  ZZZZ  ZZZZ  ZZZZ  ZZZZ
```

4.1.1.2.2 Passing Virtual Arrays — DIGITAL strongly recommends you do not pass virtual arrays as parameters in the CALL statement. Passing virtual arrays as parameters can cause unpredictable results. Instead, you can share the data in a virtual array between a calling program and a subprogram by opening a virtual file in either program and dimensioning the array in both programs using the same channel number.

NOTE

It is good programming practice to dimension a virtual array before opening the corresponding virtual file.

The two programs need not call the virtual array by the same name or dimensions but using the same dimensions reduces the risk of error. Any array redimensioned in a subprogram is redimensioned in the main program as well.

You cannot close the file before exiting in one program module if you want to access the data without opening the file in another program module.

In the following example, the calling program cannot access the virtual arrays on channel 2 until the subprogram opens the virtual file on channel 2 and returns control to the calling program. However, the subprogram can access the arrays on channel 1 because the calling program has opened the virtual file containing the array before transferring control to the subprogram:

Calling Program

```
10 DIM #1%, A$(11%), X%(15%)
20 DIM #2%, B%(12%), Y$(15%)
30 OPEN "VIRFIL.DAT" FOR OUTPUT AS FILE #1%, VIRTUAL
40 A$(11%) = "1112ST"
50 CALL VSUB1
60 B%(12%) = 12%
70 CLOSE #1%, #2%
32767 END
```

Subprogram

```
10 SUB VSUB1
20 DIM #1%, X$(11%), Z%(15%)
30 DIM #2%, CNT%(12%), ADR$(15%)
40 OPEN "OLDFIL.DAT" FOR INPUT AS FILE #2%, VIRTUAL
50 X$(3%) = ADR$(3%)
60 FOR I% = 1% TO 12%
70 Z%(I%) = CNT%(I%)
80 NEXT I%
32767 SUBEND
```

4.1.1.3 Sharing Data — There are three ways to share data between the calling program and the subprogram:

- Data in COMMONs
- Data in MAPs
- Data in files

COMMONs should be used to share data, whereas MAPs should be used for I/O operations and string manipulation. Use files to share data between programs when accessing a large data base.

4.1.1.3.1 COMMONs and MAPs — COMMON and MAP statements enable you to share data between the calling program and subprograms. These statements define a named area of memory called a program section (PSECT) containing data which may be shared between a BASIC program and subprogram.

There are advantages to using a COMMON or MAP to exchange data rather than passing parameters in a CALL statement. This is because: (1) BASIC can access the data more quickly, and (2) you can share a larger amount of data.

The COMMON statement has the format:

COMMON (name) var1,...varn

where:

name	is a one- to six-character name you assign to the COMMON. COMMONs cannot have the same name as a subprogram within a single task image. COMMONs can have the same name as a MAP provided they are not defined in the same program segment.
var1,...varn	represent the variables whose values are stored in the COMMON.

Define the COMMON or MAP area in your main program and include the same COMMON or MAP statement in your subprogram to access the data. For example:

Main Program

```
10 COMMON (RESERV) STRI.NG$ = 44%,RE.AL
20 STRI.NG$ = "HERE IS THE VALUE IN THE CALLING PROGRAM:"
30 RE.AL = 123
40 PRINT STRI.NG$;RE.AL
50 CALL SUB1
60 PRINT STRI.NG$;RE.AL
32767 END
```

Subprogram

```
10 SUB SUB1
20 COMMON (RESERV) STRI.NG$ = 44%,RE.AL
30 STRI.NG$ = "HERE IS THE VALUE AFTER THE CALL:"
40 RE.AL = 345
32767 SUBEND
```

When executed, the task returns:

```
HERE IS THE VALUE IN THE CALLING PROGRAM: 123
HERE IS THE VALUE AFTER THE CALL: 345
```

The MAP statement has the format:

MAP (name) var1,var2...varn

where:

name	is a one- to six-character name you assign to the MAP. MAPs cannot have the same name as a subprogram within a single task image. MAPs can have the same name as a COMMON provided they are not defined in the same program segment.
var1,var2...varn	represent the variables whose values are stored in the MAP.

For example:

Calling Program

```
10 MAP (RESERV) STRI.NG$,RE.AL
20 CALL SUB1
```

Subprogram

```
10 SUB SUB1
20 MAP (RESERV) STRI.NG$,RE.AL
```

The variables in a MAP or COMMON statement can be:

- Simple numeric variables
- Simple string variables
- Arrays
- FILL items

See Table 3–6 in Section 3.8.2.3 for more information on FILL items.

In both COMMONs and MAPs, simple numeric variables reserve: (1) two bytes of storage for integer values, (2) four bytes of storage for single-precision floating-point variables, and (3) eight bytes for double-precision floating-point variables.

NOTE

Examples and explanations in this section assume single-precision, floating-point variables are used.

String variables reserve fixed amounts of storage. The default amount is 16 bytes. You can reserve more or less space by defining lengths for the string variables in the MAP or COMMON statement. For example:

```
10 COMMON (RESERV) A$ = 10%,B$,C%
```

In this example, BASIC reserves a total of 28 bytes for the COMMON named RESERV: 10 bytes for A$, 16 bytes for B$, and 2 bytes for C%.

You can redefine the area of a COMMON or MAP between program modules. For example:

Calling Program

```
10 COMMON (RESERV) A$ = 10%,B$,C%
```

Subprogram

```
10 COMMON (RESERV) A1$ = 4%, A2$ = 6%,B$,C%
```

In the calling program, A$ is a 10–character string. In the subprogram, A$ is subdivided into A1$ which contains the first 4 characters and A2$ which contains the next 6 characters.

Each numeric variable in a COMMON or MAP should start on a word boundary. If the total storage allocation preceding the numeric variable is an odd number of bytes, use the FILL$ keyword to align the numeric variable on a word boundary. For example:

```
COMMON (RESERV) A$ = 9%,FILL$ = 1%,B%,C
```

String variables in a COMMON or MAP can start on any byte boundary. However, when numeric variables do not start on a word boundary, as in the following example:

```
10 MAP (RESERV) A$ = 3%,X%
```

The compiler returns the error message:

```
%Unaligned COM or MAP variable X% in (RESERV)
```

There are different ways of allocating space for multiple COMMONs and MAPs of the same name when they are in the same program module. BASIC concatenates the data stored in multiple COMMONs of the same name, whereas the data stored in multiple MAPs of the same name are overlaid.

The size of a COMMON PSECT containing multiple COMMONs of the same name is the total of the lengths of each COMMON area. The size of a MAP PSECT containing multiple MAPs of the same name is the length of the longest single MAP area. The order of variables in the COMMON and the order of multiple COMMONs of the same name determine the order of values in the shared area. For example:

Program with COMMON

```
10 COMMON (RESER1) A$ = 10%
20 COMMON (RESER1) A%,B%,C%,D%,E%
```

Program with MAP

```
10 MAP (RESER2) A$ = 10%
20 MAP (RESER2) A%,B%,C%,D%,E%
```

These COMMON statements reserve 20 bytes of storage: 10 bytes for string A$ and 2 bytes for each of 5 integers. The MAP statements reserve a total of 10 bytes: 10 bytes for string A$, then those same 10 bytes for each of 5 integers (2 bytes for each integer).

In Figure 4–2, the calling program and the subprogram access an array stored in a COMMON named ALPHA. The subprogram changes one of the

elements in the array, then returns the changed value to the calling program. You do not have to access the data in the subprogram using the same variable names or lengths that you specify in the calling program:

Figure 4–2: Sharing Data in COMMONs

Calling Program

```
10     COMMON(ALPHA) A%(5%,5%)
20     FOR I% = 1% TO 5%
30     FOR J% = 1% TO 5%
40     Y% = Y% + 1%
50     A%(I%,J%) = Y%
60     NEXT J%
70     NEXT I%
75     PRINT "HERE ARE THE INITIAL VALUES OF THE CALLING PROGRAM:"
80     MAT PRINT A%, &
         \ PRINT
90     PRINT "NOW TO THE SUBPROGRAM"
100    CALL SUB1
110    PRINT
120    PRINT "THE CHANGED VALUE OF ARRAY ELEMENT (3,3) IS:";A%(3%,3%)
32767  END
```

Subprogram

```
10   SUB SUB1
20   COMMON(ALPHA) C%(5%,5%)
30   PRINT "HERE ARE THE VALUES AFTER THE CALL TO THE SUBPROGRAM:"
40   C%(3%,3%) = 0%
50   MAT PRINT C%, \ PRINT
32767 SUBEND
```

When executed, the task returns:

```
HERE ARE THE INITIAL VALUES OF THE CALLING PROGRAM:
1             2             3             4             5
6             7             8             9             10
11            12            13            14            15
16            17            18            19            20
21            22            23            24            25
NOW TO THE SUBPROGRAM
HERE ARE THE VALUES AFTER THE CALL TO THE SUBPROGRAM:
1             2             3             4             5
6             7             8             9             10
11            12            0             14            15
16            17            18            19            20
21            22            23            24            25

THE CHANGED VALUE OF ARRAY ELEMENT (3,3) IS: 0
```

4.1.1.3.2 Files — You can also share data between the calling program and the subprogram by opening a file in either program. The following conditions apply:

- If you open the file in the calling program, you do not need to reopen the file in the subprogram to access the data. Files remain open until: (1) you

open another file on that channel, (2) you close the file, or (3) the END statement is executed.

- The file data can be accessed either statically with the MAP statement or dynamically with the FIELD statement. For more information see Sections 3.2 and 3.3.
- The file pointer for a channel is the same for both the calling program and subprogram. Each time you sequentially access the file, whether it be in the calling program or the subprogram, you get the next record.

In Figure 4–3, the main program: (1) defines the MAP, (2) opens a sequential file, and (3) writes three records to the file. The subprogram redefines the MAP, then writes records to the file based on user input. Note that the subprogram does not reopen the file.

Figure 4–3: Sharing Data in Files

Main Program

```
10    MAP (BUF) ID$ = 9%, NAME$ = 26%
20    PRINT "BEGIN MAIN PROGRAM"
30    OPEN "DATA.FIL" FOR OUTPUT AS FILE #1%, SEQUENTIAL, MAP BUF
40    READ TEMP$
50    WHILE TEMP$ <> "DONE"
60    ID$ = TEMP$
70    READ NAME$, TEMP$
80    PUT #1%
90    NEXT
100   CALL SUB1(SUCCESS%)
110   IF SUCCESS% = -1% THEN    &
      PRINT "ERROR IN SUB1"
120   CLOSE #1%
10000 DATA 1009-2222, PETER FINKLE
10001 DATA 2223-1234, LEE DAUGHT
10002 DATA 8712-3940, PHIL ERUP
10003 DATA DONE
32767 END
```

Subprogram

```
10     SUB SUB1(INFO%)    &
       \ ON ERROR GO TO 19000
20     MAP (BUF) ID$ = 9%, FIRST.NAME$ = 8%, LAST.NAME$ = 18%
30     INFO% = 0%
40     WHILE 1%
50     PRINT
60     INPUT "DO YOU WISH TO SUPPLY MORE RECORDS [Y/N]";ANS$
70     IF ANS$ <> "Y" THEN SUBEXIT
80     LINPUT "SUPPLY 9 CHARACTER ID"; ID$
90     LINPUT "SUPPLY FIRST NAME IN <= 8 CHARACTERS";FIRST.NAME$
100    LINPUT "SUPPLY LAST NAME IN <= 18 CHARACTERS";LAST.NAME$
110    PUT #1%
120    NEXT
19000 INFO% = -1%
19010 RESUME 32767
32767 SUBEND
```

When executed, the task returns:

```
BEGIN MAIN PROGRAM

DO YOU WISH TO SUPPLY MORE RECORDS [Y/N]? Y
SUPPLY 9 CHARACTER ID? 1234-5678
SUPPLY FIRST NAME IN <= 8 CHARACTERS? JOE
SUPPLY LAST NAME IN <= 18 CHARACTERS? SMITH

DO YOU WISH TO SUPPLY MORE RECORDS [Y/N]? Y
SUPPLY 9 CHARACTER ID? 9999-8888
SUPPLY FIRST NAME IN <= 8 CHARACTERS? JILL
SUPPLY LAST NAME IN <= 18 CHARACTERS? JONES

DO YOU WISH TO SUPPLY MORE RECORDS [Y/N]? N
```

The output file "DATA.FIL" now contains the following records:

1009–2222 PETER FINKLE
2223–1234 LEE DAUGHT
8712–3940 PHIL ERUP
1234–5678 JOE SMITH
9999–8888 JILL JONES

4.1.1.4 Functions — A function can be defined in either the calling program or the subprogram, but it is local to the program that defines it. The value of a function can be passed to a subprogram as a parameter in the CALL statement. You can change that value in the subprogram; however, you cannot return that value to the calling program. For example:

Calling Program

```
10 DEF FNB%(A%)
20 FNB% = A% * 2%
30 FNEND
40 CALL SUB1(FNB%(3%))
50 PRINT "THE VALUE IN THE MAIN PROGRAM IS";FNB%(3%)
32767 END
```

Subprogram

```
10 SUB SUB1(A%)
20 A% = A% * 2%
30 PRINT "THE SUBPROGRAM CHANGES THE VALUE TO";A%
32767 SUBEND
```

When executed, the task returns:

```
THE SUBPROGRAM CHANGES THE VALUE TO 12
THE VALUE IN THE MAIN PROGRAM IS 6
```

You cannot use a variable name in the function definition if it is one of the SUB statement's parameters. For example:

```
10 SUB SUBWDF(A$,B%,C)
    .
    .
    .
15000 DEF FNDUM(A$,B%,C)
    .
    .
    .
15020 FNEND
32767 SUBEND
```

The parameters in the function definition attempts to reallocate the storage space that has been set aside for the parameters in the SUB statement and results in the error message:

```
?Illegal dummy argument at line 15000
```

4.1.1.5 DATA and READ Statements — You can use DATA and READ statements in both the calling program and the subprogram. DATA statements are local to the program module that contains them. READ statements in either program module do not affect the data pointer of the other program module. Each time the calling program calls a subprogram, the data pointer returns to the beginning of the DATA sequence in that subprogram. For example:

Calling Program

```
5 C% = 0%
10 READ A$
20 PRINT A$
30 CALL SUB1
40 C% = C% + 1%
50 IF C% = 3% GO TO 32767 ELSE GO TO 10
60 DATA MAINDAT1, MAINDAT2, MAINDAT3
32767 END
```

Subprogram

```
10 SUB SUB1
20 READ B$
30 PRINT B$
40 DATA SUBDAT1, SUBDAT2, SUBDAT3
32767 SUBEND
```

When executed, the task returns:

```
MAINDAT1
SUBDAT1
MAINDAT2
SUBDAT1
MAINDAT3
SUBDAT1
```

4.1.1.6 Handling Errors — In BASIC, you can process errors using either: (1) the system or (2) user-defined error-handling routines. If you do not specify an ON ERROR statement, the default is ON ERROR GOTO 0. This means that when an error occurs, BASIC prints an error message describing the nature of the error and returns to command level without completing execution.

The user can define error-handling routines in either the calling program or subprogram. The ON ERROR GO TO 0 statement is the default for both the calling program and the subprogram. ON ERROR GO TO <line-number> lets you specify an error handler's starting line number. When an error occurs, BASIC transfers control to your error handler. User-written error handlers are terminated with a RESUME statement.

In subprograms, you can handle errors with the ON ERROR GO BACK statement. The ON ERROR GO BACK statement returns control to the calling program's error handler when an error occurs. Remember, the calling program can either be the "main" program or another subprogram. In the case of an error occurring in one subprogram that was called by another subprogram, the ON ERROR GO BACK statement in the called subprogram transfers control to the error handler in the subprogram that called it.

If an error occurs in a subprogram that does not have an error handler or an ON ERROR GO BACK statement, the system defaults to ON ERROR GOTO 0 and causes execution to abort. For more information on error-handling routines see Section 4.8 in the *PDP–11 BASIC–PLUS–2 Language Reference Manual.*

In the following example, an error in the subprogram causes BASIC to transfer control to line 19000 and evaluate the error, then either: (1) transfer control to the calling program's error handler or (2) abort the task.

```
10 SUB ERRTRP(X,Y)
20 ON ERROR GOTO 19000
30 INPUT 1%, G.LIN$
          .
          .
          .
19000 PRINT ERR,ERL
19010 IF (ERR = 11%) AND (ERL = 30%) THEN    &
          RESUME 32767                       &
          ELSE                               &
          ON ERROR GOTO 0
32767 SUBEND
```

In applications where handling errors is crucial, put the error handler on the same line as the SUB statement. This procedure minimizes the time between the start of the subprogram and the setting up of the error-handling routine. For example:

```
10 SUB ERRTRP(X,Y)    &
   \  ON ERROR GO BACK
```

4.1.1.7 Building the Task — To create an executable subprogram, you must: (1) compile the calling program and the subprograms separately and (2) build the program into a single task image.

1. Compile your programs separately. For example:

```
OLD MAIN
COMPILE

OLD SUB1
COMPILE

OLD SUB2
COMPILE
```

   The compile command generates an object module for each program:

   MAIN.OBJ
   SUB1.OBJ
   SUB2.OBJ

2. Use the BUILD command to combine your main program and subprograms, placing the name of the main program first:

```
BUILD MAIN,SUB1,SUB2
```

   This command generates two indirect files: the command (CMD) file and the Overlay Descriptor Language (ODL) file which the Task Builder uses.

   MAIN.CMD
   MAIN.ODL

3. Task-build the command file. Check with your system manager for the command for your operating system. For example:

```
TKB @MAIN.CMD
```

   Task-building your command file generates an executable image:

   MAIN.TSK

4. Now you can execute your task:

```
RUN MAIN
```

The Task Builder:

- Combines the object modules generated by the COMPILE command into a single, executable task image
- Searches the library to resolve global references made by the program
- Allocates virtual address space needed by the task

The system imposes restrictions on the size of a task that can be placed in memory at once. If the task image defined by the ODL file is too large to handle at once, you can accommodate it by changing its overlay structure.

BASIC creates an Overlay Descriptor Language (ODL) file that allows the user to describe the overlay structure of a task. The overlay structure is the way program code is brought into memory as the program executes. Changing the overlay structure of a task enables you to decrease the amount of memory space for your task.

The ODL file defines the root and branches in the task image. The root is the portion of the task that remains in memory throughout task execution. It includes the code for the main program, data local to the main program, data shared between program modules, and the object library modules needed to resolve the symbols in the generated code. The branches are the region of memory which contains the program code for subprograms, data local to the subprogram, and the object library modules needed to resolve symbols not already resolved in the main program.

In the following example, the main program calls two subprograms, SUB1 and SUB2, and SUB2 calls SUB3.

Main Program

```
10 PRINT "THIS IS THE MAIN PROGRAM"
20 CALL SUB1 (STRI.NG$,RE.AL,INTE.GER%)
30 CALL SUB2 (A$,B,C%)
32767 END
```

Subprogram 1

```
10 SUB SUB1 (STRI.NG$,RE.AL,INTE.GER%)
20 STRI.NG$ = "THIS IS SUBPROGRAM 1"
32767 SUBEND
```

Subprogram 2

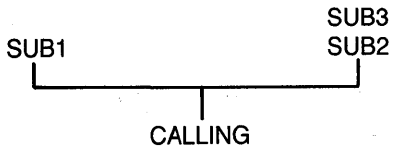
```
10 SUB SUB2 (D$,E,F%)
20 A$ = "THIS IS SUBPROGRAM 2"
30 CALL SUB3
32767 SUBEND
```

Subprogram 3

```
10 SUB SUB3
20 A$ = "THIS IS SUBPROGRAM 3"
32767 SUBEND
```

The following illustration shows one possible relationship between the calling program or root segment and the subprograms or branch segments.

Figure 4–4: Tree Figure Representing the Overlay Structure



Each branch of the tree represents a program segment. Parallel branches at the same level represent program segments whose instructions and data are overlaid in memory as the program executes.

Compare the amounts of memory space the task needs (1) if the four programs are included in the root of the ODL file and (2) if the subprograms are included in the branches of the ODL file.

Figure 4–5: Nonoverlay and Overlay Memory Requirements

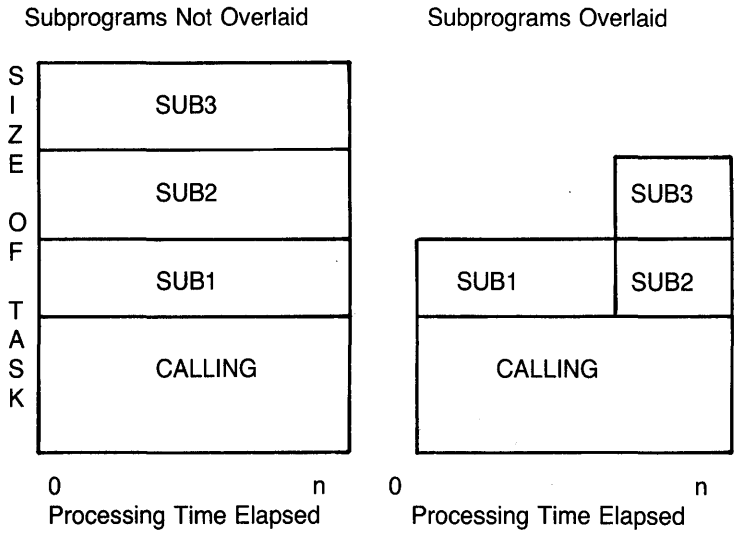


Figure 4–5 shows you how large the task is at a given time during processing, and what parts of the program are in memory at that time. Comparing them, you can see the nonoverlaid version needs more memory than the overlaid version. In the figure that shows subprograms overlaid, SUB2 and SUB3 overlay the memory reserved for SUB1 as the program executes.

To change the overlay structure defined for a task, you must change the contents of the ODL file before you build your task. To do this, edit the ODL file. See your system's Task Builder manual for more information about ODL files.

The ODL file for the task illustrated above where the subprograms are not overlaid looks like this:

```
            .ROOT USER
USER:       .FCTR SY:MAIN-SUB1-SUB2-SUB3-LIBR
LIBR:       .FCTR LB:[1,1]BASIC2/LB
            .END
```

You can edit your ODL file to overlay your subprograms, reducing the amount of memory space required for the task:

```
            .ROOT USER
USER:       .FCTR SY:MAIN-LIBR-*(BR1,BR2)
BR1:        .FCTR SY:SUB1-LIBR
BR2:        .FCTR SY:SUB2-SUB3-LIBR
LIBR:       .FCTR LB:[1,1]BASIC2/LB
            .END
```

The ODL file defines:

- The root portion of the task:

```
USER:   .FCTR SY:MAIN-LIBR-*(BR1,BR2)
```

- Two overlaid branches:

```
.FCTR SY:SUB1-LIBR
.FCTR SY:SUB2-SUB3-LIBR
```

The main program is concatenated to the BASIC disk library and to the two subprograms: BR1 and BR2. Check your file to make sure the autoload operator (*) is included in the file. The autoload operator tells the Task Builder to generate autoload code to automatically load the appropriate program segment into memory as the program executes. See your system's Task Builder manual for more information on the autoload operator.

Check your ODL file to make sure that each branch and root segment is concatenated to the BASIC disk library so that the Task Builder can resolve program code. Failure to concatenate object modules with the disk library may cause the Task Builder to return the error message:

```
?Undefined symbol
```

If you intend to use MAPs or COMMONs to share data between main programs and subprograms, make sure the COMMON or MAP is defined in the main program. The main program is located in the root of the ODL file and the data contained within is not overlaid.

For more information on task-building and overlay procedures, see your system's Task Builder manual.

4.1.2 BASIC to MACRO Subprogramming

Any BASIC program or subprogram can call a MACRO subprogram. The MACRO subprogram can be either system-supplied or user-created.

MACRO subprograms have these advantages over BASIC subprograms:

- You can pass more parameters to a MACRO subprogram than to a BASIC subprogram.
- You can use the CALL BY REF statement to pass parameters to a MACRO subprogram.
- Some MACRO subprograms run faster than comparable BASIC subprograms.
- MACRO subprograms enable you to perform tasks that are difficult or impossible with BASIC.

However, BASIC does not support the following operations in MACRO subprograms:

- Calling BASIC subprograms
- Performing I/O or monitor operations
- Accessing virtual arrays or other kinds of files
- Creating strings, or altering the lengths of existing strings

These operations may overwrite portions of the main program with subprogram instructions and data. They can, therefore, cause the main program to abort or generate unpredictable results. For the same reason, BASIC programs cannot call executive directives that require null parameters.

4.1.2.1 Calling a MACRO Subprogram — To transfer control from a BASIC main program to a MACRO subprogram, use either the CALL or the CALL BY REF statement.

The CALL statement passes parameters to the subprogram either by reference or by descriptor, depending on the type of parameter. (For information on parameter passing, see Section 4.1.1.2.) The format for the CALL statement is:

```
CALL name (param1,param2...paramn)
```

where:

name	is the one- to six-character name of the MACRO subprogram. A subprogram cannot have the same name as any other subprogram, MAP, or COMMON within the same task. The name can be a quoted or unquoted string. It cannot be a string variable.

(continued on next page)

param1,param2...paramn	represent zero to n optional arguments, or parameters, passed from the main program to the subprogram. The number of parameters you can include is limited by the size of the temporary storage area (stack) allocated in your task image.

For example:

```
CALL SUBPRG (A$,B%,C$(1%,1%))
```

If the subprogram name contains a dollar sign ($) or period (.) in the leftmost position, you must enclose the name in quotation marks when you use it in a CALL or CALL BY REF statement.

The CALL BY REF statement passes all parameters by reference. Its format is:

CALL name BY REF (param1,param2...paramn)

where:

name	is the one- to six-character name of the MACRO subprogram. A subprogram cannot have the same name as any other subprogram, MAP, or COMMON within the same task. The name can be a quoted or unquoted string. It cannot be a string variable.
param1,param2...paramn	represent zero to n optional arguments, or parameters. The number of parameters you can include is limited by the size of the temporary storage area (stack) allocated in your task image.

For example:

```
CALL SUBPRG BY REF (A$,B%,C$(1%,1%))
```

The name in the CALL or CALL BY REF statement must correspond to the name of the MACRO subprogram, as defined by the subprogram's global entry-point label. This can be different from the name of the file containing the code, and the name defined by the .TITLE assembly directive. However, if the subprogram includes only a single entry-point, it is a good idea to use the same name for the entry-point label and the title. For example, suppose a main program includes the statement:

```
50 CALL INSRT
```

The code of the MACRO subprogram called should include the title and label-name INSRT, as in the example below:

```
        .TITLE INSRT

; MODULE FUNCTION:
;       THIS MODULE DEMONSTRATES
;       THE FORMAT FOR MACRO SUBPROGRAMS

        .PSECT INSRT
INSRT:: ; SUBPROGRAM NAME
        RTS PC              ; RETURN TO MAIN PROGRAM
        .END
```

The RTS PC (or RETURN) instruction returns control to the main program. It corresponds to the SUBEND statement of a BASIC subprogram.

NOTE

The last statement in any MACRO program or subprogram must be the .END assembly directive.

4.1.2.2 Passing Parameters — You can pass data from a BASIC main program to a MACRO subprogram by including parameters in the CALL or CALL BY REF statement of the main program. BASIC imposes two restrictions on the kinds of parameters that main programs can pass to MACRO subprograms:

- BASIC main programs cannot pass virtual arrays to MACRO subprograms.
- MACRO subprograms cannot change the length of strings passed to them as parameters by BASIC main programs, nor can they create new strings.

If you need to access virtual arrays or to change string lengths in a MACRO subprogram, place the string and array data in a COMMON or MAP area before calling the subprogram. Once the subprogram has performed its operations on the data and has returned control to the main program, the main program can move the data back into arrays and strings. For information on using COMMONs and MAPs with MACRO subprograms, see Section 4.1.2.3.

MACRO subprograms do not have SUB statements to define the parameters they receive. Therefore, in order to access those parameters, you need to know where the main program stores them.

Like BASIC subprograms, MACRO subprograms that receive parameters from a main program receive an argument list containing information about those parameters. The information in the argument list varies, depending on whether a given parameter is passed by reference or descriptor:

- The first word of the argument list always contains, in its low-order byte, the number of arguments in the list. The high-order byte in this word is undefined.

- Each word after the first contains a pointer to one of the parameters in the CALL statement, in the same order as the parameters appeared in the CALL statement. If you pass a parameter by reference, the subprogram receives in the argument list the address at which that parameter, or a local copy of it, is located. If you pass a parameter by descriptor, the subprogram receives in the argument list the address of a descriptor block. The descriptor block contains information about the parameter or its local copy, including the address at which the parameter or copy is located, and the length of the parameter. (See Appendix D for more explanation of descriptor blocks.)

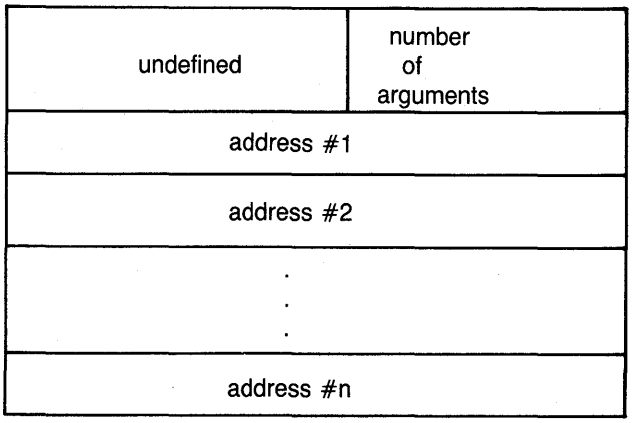
As the explanation above indicates, "pass by reference" means that the subprogram receives only the addresses of any parameters passed to it. "Pass by descriptor" means that the subprogram receives both the addresses of the parameters, and certain other information such as parameter length.

NOTE

BASIC does not support passing parameters by immediate value. That is, the argument list cannot contain the parameters themselves.

In general terms, then, argument lists look like this:

Figure 4–6: Argument List Format



Whenever BASIC encounters a CALL or CALL BY REF statement, it stores the address of the argument list's first word in general register R5. Subprograms can, therefore, express parameter addresses as offsets from the value stored in register R5.

When you use a CALL statement, BASIC passes most parameters by reference. However, it automatically passes certain parameters, such as string data and arrays, by descriptor. When you use the CALL BY REF statement, on the other hand, BASIC passes all parameters to the subprogram by reference. Therefore, all the addresses in the argument list generated by a CALL BY REF statement refer to the parameters themselves, or

to local copies of them. If the subprogram needs to know a string length, you must pass that information as a parameter. For example, a main program might contain the following CALL BY REF statement:

```
19000 CALL MACSUB BY REF (STR.NG$,LEN.STR%,A$(1%,5%))
```

This statement would generate an argument list like the following:

```
ADDRESS     VALUE

022060      000003
022062      004560      ARGUMENT LIST
022064      005632
022066      035766
```

The actual addresses, and the values stored in them, depend on your program. In this example, memory location 22060 contains the octal value 000003, since there are three parameters in the argument list. Locations 22062 and 22064 contain the addresses of string variable STR.NG$ and integer variable LEN.STR%. Location 22066 contains the address of a local copy of string array element A$(1%,5%).

General register R5 contains the value 22060, the address of the first word in the argument list. Therefore, to access the parameters, define their locations as offsets from R5. For example, use this MACRO statement to move the first two bytes of variable STR.NG$ into general register R1:

```
MOV @2(R5),R1              ;  SET R1 = STR.NG$
```

Figure 4–7 shows how a subprogram accesses and modifies parameters passed by reference from a BASIC main program. Note that the parameters passed in the CALL BY REF statement include the lengths of strings A$ and B$.

Figure 4–7: CALL BY REF with MACRO Subprogram

Main Program (BASIC)

```
10 PRINT "THIS PROGRAM WRITES SUBSTRING B$ INTO STRING A$,"   &
        \ PRINT "STARTING AT CHARACTER POSITION C."           &
        \PRINT
20 INPUT "ENTER STRING A$";A$                                 &
         \IF A$ = "DONE" GOTO 32767
30 INPUT "ENTER SUB-STRING B$";B$
40 INPUT "ENTER C ";C%
50 CALL INSRT BY REF (A$,LEN(A$),B$,LEN(B$),C%)
60 IF C% = 0% THEN PRINT "NEW VALUE OF A$ IS ";A$             &
        ELSE  &
              PRINT "ATTEMPT UNSUCCESSFUL"
70 PRINT \ PRINT "DO YOU WANT TO CONTINUE?"
80 PRINT "IF NOT, TYPE 'DONE'." /PRINT
90 GOTO 20
32767 END
```

(continued on next page)

Subprogram (MACRO)

```
        .TITLE INSRT
        .IDENT /01/

; MODULE FUNCTION:
;       THIS SUBPROGRAM WRITES SUBSTRING B$
;       INTO STRING A$, BEGINNING AT CHARACTER C%

;
;LOCAL MACROS:
;
;

; LOCAL DATA BLOCKS:
;
        .PSECT DATA,D,RW

;
; LOCAL OFFSETS:

A       =   2.
LNA     =   4.
B       =   6.
LNB     =   8.
C       =  10.

;
; FUNCTION DETAILS:
;
;       INPUTS:
;             ARG1 = ADDRESS OF A$
;             ARG2 = ADDRESS OF LENGTH OF A$
;             ARG3 = ADDRESS OF B$
;             ARG4 = ADDRESS OF LENGTH OF B$
;             ARG5 = ADDRESS OF C%

;       OUTPUTS:
;             C% = 0 IF OPERATION SUCCESSFUL
;             C% = -1 IF OPERATION UNSUCCESSFUL

        .PAGE
        .SBTTL
        .PSECT INSRT
INSRT::
        MOV     @C(R5),R2           ; R2 = C%
        BLE     ERREX               ; GOTO ERREX IF C <= 0
        ADD     @LNB(R5),R2         ; R2 = C% + LEN(B$)
        DEC     R2                  ; MAKE R2 A LENGTH
        CMP     R2,@LNA(R5)         ; DOES B$ FIT INTO A$?
        BGT     ERREX               ; IF NOT, GOTO ERREX
        MOV     A(R5),R0            ; R0 = ADDRESS OF A$
        MOV     @C(R5),R2           ; SET R2 = C%
        DEC     R2                  ; SET R2 = C% - 1
        ADD     R2,R0               ; SET R0 = ADDRESS OF A$ + C%
        MOV     @LNB(R5),R2         ; SET R2 = LEN(B$)
        BEQ     ERREX               ; GO TO ERREX IF LEN(B$) = 0
        MOV     B(R5),R1            ; SET R1 = ADDRESS OF B$
1$:
        MOVB    (R1)+,(R0)+         ; INSERT CHAR FROM B$ INTO A$
        SOB     R2,1$               ; REPEAT FOR REMAINING CHARACTER
        CLR     @C(R5)              ; SUCCESS, SET C% = 0
        RTS     PC                  ; RETURN TO MAIN PROGRAM
```

(continued on next page)

```
ERREX:
        MOV     -tC(R5)             ; FAILURE, SET C% = -1
        RTS     PC                  ; RETURN TO MAIN PROGRAM
        .END
```

Compile the main program with the BASIC compiler. Assemble the subprogram with the MACRO assembler. Then build and run them as you would any multi-segment BASIC task, by including the MACRO object module with the main program in the BUILD command.

When you run the program, it returns the following:

```
>RUN MNPROG

THIS PROGRAM WRITES SUBSTRING B$ INTO STRING A$,
STARTING AT CHARACTER POSITION C.

ENTER STRING A$? ABCDEF
ENTER SUB-STRING B$? XYZ
ENTER C? 1

NEW VALUE OF A$ IS XYZDEF

DO YOU WANT TO CONTINUE? DONE
```

NOTE

Unmatched parameter boundaries or data types in the main program and subprogram generate ?Odd address trap or ?Memory management violation error messages.

Simple variables and entire arrays are modifiable parameters. For example, the subprogram above changes A$, and then returns the changed value to the main program.

All constants, expressions, and array elements, however, are nonmodifiable. That is, when a subprogram receives these parameters, the addresses in the argument list point to local copies of the parameters rather than to the actual data. A MACRO subprogram can change local-copy values. But such changes do not affect the constants, expressions, and arrays of the main program.

You can pass local copies of simple variables and entire arrays by enclosing the individual variable and array names in parentheses. In the example below, A$ is a modifiable parameter but B$ is nonmodifiable.

```
19000 CALL MACSUB BY REF (A$,(B$))
```

Unlike the CALL BY REF statement explained above, the CALL statement passes certain parameters by descriptor instead of by reference. Table 4–1

summarizes these differences between the CALL and CALL BY REF statements. Note that certain data types cannot be used as parameters in BASIC to MACRO subprogramming.

Table 4–1: Parameter Passing with CALL and CALL BY REF

	CALL	CALL BY REF
NUMERIC DATA		
variable	passed by ref	passed by ref
constant	by ref (copy)	by ref (copy)
expression	by ref (copy)	by ref (copy)
function	by ref (copy)	by ref (copy)
array element	by ref (copy)	by ref (copy)
entire array	by desc	by ref
virtual-array element	by ref (copy)	by ref (copy)
virtual array	--------	-------
STRING DATA		
variable	by desc	by ref
constant	by desc (copy)	by ref (copy)
expression	by desc (copy)	by ref (copy)
function	by desc (copy)	by ref (copy)
array element	by desc (copy)	by ref (copy)
entire array	by desc	-------
virtual–array element	by desc (copy)	by ref (copy)
virtual array	-------	-------

When the CALL parameter is a string variable, constant, array element, or expression, the corresponding value in the argument list points to the address of a two-word descriptor block. The first word of this block contains the address of the first byte in the string. The second word expresses the length of the string, in bytes.

For example, suppose a main program includes the statement:

```
50 CALL INSRT (A$,B$,C%)
```

This generates a four-word argument list:

```
ADDRESS    VALUE

022060     000003
022062     004532      ARGUMENT LIST
022064     023462
022066     026534
```

The actual addresses, and the values stored in them, depend on your program. In this example, the second word in the argument list contains the address of the descriptor block for A$. If A$ were a six-byte string, such as "ABCDEF", the descriptor block would look like this:

```
ADDRESS   VALUE

004532    020652      DESCRIPTOR BLOCK
004534    000006
```

If you include an array in a CALL statement, the argument list contains the address of the second word of the descriptor block. This word in turn contains the address of the first element in the array. (The first word in the descriptor block is the Array Descriptor Word, which defines the array type and length. See Appendix D for more information on the Array Descriptor Word.)

Figure 4–8 shows how a MACRO subprogram can access parameters passed by descriptor.

Figure 4–8: CALL Statement with MACRO Subprograms

Main Program

```
10 PRINT "THIS PROGRAM WRITES SUBSTRING B$ INTO STRING A$"   &
        \ PRINT "STARTING AT CHARACTER POSITION C."          &
        \ PRINT
20 INPUT "ENTER STRING A$";A$                                &
         \ IF A$ = "DONE" GOTO 32767
30 INPUT "ENTER SUB-STRING B$";B$
40 INPUT "ENTER C ";C%
50 CALL INSRT (A$,B$,C%)
60 IF C% = 0% THEN PRINT "NEW VALUE OF A$ IS ";A$            &
        ELSE  &
        PRINT "ATTEMPT UNSUCCESSFUL"
70 PRINT \ PRINT "DO YOU WANT TO CONTINUE?"
80 PRINT "IF NOT, TYPE 'DONE'."
90 GOTO 20
32767 END
```

Subprogram

```
        .TITLE INSRT
        .IDENT /01/

;MODULE FUNCTION:
;       THIS SUBPROGRAM OVERWRITES SUBSTRING B$
;       INTO STRING A$, BEGINNING AT CHARACTER C%.

;
; LOCAL MACROS:
;
;
```

(continued on next page)

```
; LOCAL DATA BLOCKS:
;
        .PSECT DATA,D,RW

;
; LOCAL OFFSETS:
A = 2.
B = 4.
C = 6.

;
; FUNCTION DETAILS:
;
;       INPUTS:
;            ARG1 = ADDRESS OF A$ STRING DESCRIPTOR
;            ARG2 = ADDRESS OF B$ STRING DESCRIPTOR
;            ARG3 = ADDRESS OF C%

;       OUTPUTS:
;            C% = 0 IF SUCCESSFUL
;            C% = -1 IF UNSUCCESSFUL

        .PAGE
        .SBTTL
        .PSECT INSRT
INSRT::
        MOV     A(R5),R0            ; SET R0 = ADDRESS OF A$ DESC
        MOV     B(R5),R1            ; SET R1 = ADDRESS OF B$ DESC
        MOV     @C(R5),R2           ; SET R2 = C%
        BLE     ERREX               ; GO TO ERREX IF C <= 0
        ADD     2(R1),R2            ; SET R2 = C% + LENGTH OF B$
        DEC     R2                  ; MAKE R2 A LENGTH
        CMP     R2,2(R0)            ; DOES B$ FIT INTO A$?
        BGT     ERREX               ; IF NOT, GO TO ERREX
        MOV     (R0),R0             ; SET R0 = ADDRESS OF A$
        MOV     @C(R5),R2           ; SET R2 = C%
        DEC     R2                  ; SET R2 = C% -1
        ADD     R2,R0               ; SET R0 = ADDRESS OF FIRST CHAR
        MOV     2(R1),R2            ; SET R2 = LENGTH OF B$
        BEQ     ERREX               ; IF B$ = 0, GO TO ERREX
        MOV     (R1),R1             ; SET R1 = ADDRESS OF B$
1$:
        MOVB    (R1)+,(R0)+         ; INSERT A CHAR FROM B$ INTO A$
        SOB     R2,1$               ; REPEAT
        CLR     @C(R5)              ; SUCCESS, SET C% = 0
        RTS     PC; RETURN TO MAIN PROGRAM
ERREX:
        MOV     #-1,@C(R5)          ; FAILURE, SET C% = -1
        RTS     PC                  ; RETURN TO MAIN PROGRAM
        .END
```

In this example, the lengths of A$ and B$ are part of the descriptor blocks instead of being passed as parameters in the CALL statement.

NOTE

Use the CALL BY REF statement, rather than the CALL statement, when passing parameters from a BASIC program to an executive directive or library module.

4.1.2.3 Sharing Data: COMMON and MAP — BASIC allows you to access COMMON and MAP areas from MACRO subprograms. This enables you to share large amounts of data between your BASIC main programs and your MACRO subprograms. In addition, you can use MACRO subprograms to initialize COMMON or MAP areas in BASIC main programs.

You can rewrite the main program and subprogram in Figure 4–7 so that they share data by means of a MAP statement rather than by passing parameters in the CALL statement. Figure 4–9 show the BASIC code of the main program.

Figure 4–9: MAP Statement with MACRO Subprogram

Main Program

```
10 MAP (RESERV) A$,LNA%,B$=8%,LNB%,C%
20 PRINT "THIS PROGRAM WRITES SUBSTRING B$ INTO STRING A$" &
        \ PRINT "STARTING AT CHARACTER POSITION C."         &
        \ PRINT
30 INPUT "ENTER 16-CHARACTER STRING A$";A$                  &
         \ IF A$ = "DONE" GOTO 32767
40 INPUT "ENTER 8-CHARACTER SUB-STRING B$";B$
50 INPUT "ENTER C ";C%
55 LNA% = LEN(A$) LNB% = LEN(B$)
60 CALL INSRT
70 IF C% = 0 THEN PRINT "NEW VALUE OF A$ IS ";A$            &
        ELSE &
             PRINT "ATTEMPT UNSUCCESSFUL"
80 PRINT \ PRINT "DO YOU WANT TO CONTINUE?"
90 PRINT "IF NOT, TYPE 'DONE'."
100 GOTO 20
32767 END
```

The MAP statement in this program sets the size of the A$ and B$ strings to a pre-determined 16 and 8 bytes.

When the BASIC compiler generates object code from the BASIC source program, it creates a data storage area for every COMMON and MAP statement. You can see this most clearly if you compile the BASIC source program above with the COMPILE/MACRO command. The compiler generates the following MACRO code for the MAP statement in that program:

```
        .PSECT RESERV,RW,D,GBL,REL,OVR
RESERV::
        .PSECT RESERV
        .BLKW 15.
```

In order for the MACRO subprogram to access the data stored in MAP area RESERV, you have to define a subprogram PSECT of the same name and attributes as that created by the MAP statement of the main program. The subprogram can then assign variable names to the data in that PSECT, and use those variables as operands in its instructions. The subprogram in Figure 4–10 shows how to do this.

Figure 4–10: MACRO Code for MAP Statement

```
        .TITLE INSRT
        .IDENT /01/
        ;
        ;MODULE FUNCTION:
        ;       THIS SUBPROGRAM USES A MAPPED AREA
        ;       TO OVERWRITE SUBSTRING B$ INTO STRING A$,
        ;       BEGINNING AT CHARACTER C.

        ;
        ; LOCAL MACROS:
        ;

        ;
        ; LOCAL DATA BLOCKS:

              .PSECT RESERV RW,D,GBL,REL,OVR
        A:    .BLKB 16.
        LNA:  .BLKW
        B:    .BLKB 8.
        LNB:  .BLKW
        C:    .BLKW
        ;
        ; FUNCTION DETAILS:
        ;
        ;       INPUTS:
        ;             PSECT RESERV CONTAINS A$,LNA%
        ;             B$,LNB%, AND C%
        ;
        ;       OUTPUTS:
        ;             C% = 0 IF OPERATION WAS SUCCESSFUL
        ;             C% = -1 IF OPERATION FAILED

        ;
        ; CODE BEGINS HERE:
              .PAGE
              .SBTTL
              .PSECT INSRT
        INSRT::

        MOV     C,R2            ; SET R2 = C%
        BLE     ERREX           ; IF C <= 0, GO TO ERREX
        ADD     LNB,R2          ; SET R2 = C% + LEN(B$)
        DEC     R2              ; MAKE R2 A LENGTH
        CMP     R2,LNA          ; DOES B$ FIT INTO A$?
        BGT     ERREX           ; IF NOT, GO TO ERREX
        MOV     C,R2            ; SET R2 = C%
        DEC     R2              ; SET R2 = C% - 1
        MOV     #A,R0           ; SET R0 = ADDRESS OF A
        ADD     R2,R0           ; SET R0 = ADDRESS OF FIRST CHAR REPLACED
        MOV     LNB,R2          ; SET R2 = LENGTH OF B$
        BEQ     ERREX           ; IF B$ = 0, GO TO ERREX
        MOV     #B,R1           ; SET R1 = ADDRESS OF B$
1$:
        MOVB    (R1)+,(R0)+     ; INSERT CHARACTER FROM B$ IN A$
        SOB     R2,1$           ; REPEAT
        CLR     C               ; SUCCESS, SET C% = 0
        RTS     PC              ; RETURN TO MAIN PROGRAM
ERREX:
        MOV     #-1,C           ; FAILURE, SET C% = -1
        RTS     PC              ; RETURN TO MAIN PROGRAM
        .END
```

When you build this multi-segment task, the Task Builder defines a single area named RESERV. If the buffer allocations in the main program and subprogram differ, the Task Builder defines an area equal to the larger allocation.

The Task Builder does not check to see that the main program and subprogram define the same data types and boundaries in their common areas. If the areas do not correspond, you may receive ?Odd address trap or ?Memory protection violation error messages when you run the task. For this reason, be sure you properly align your data definitions in the main program and subprogram. Remember that COMMON statements of the same name within a single program section are concatenated. For example:

```
10 COMMON (RESERV) VR.STR$ = 30%, FX.STR$ = 30%
20 COMMON (RESERV) A$,B%
```

These statements generate a single PSECT:

```
        .PSECT RESERV,RW,D,GBL,REL,OVR
RESERV:
        .PSECT RESERV
        .BLKW 39.
```

The total area set aside for RESERV is 78 decimal bytes (39 words), the sum of the two COMMON statements. Variable A$ begins at byte location 60 of RESERV, not at location 0. If these were MAP statements, the area set aside for RESERV would equal the larger allocation (30 words). Variables VR.STR$ and A$, in that case, would both begin at byte location 0.

In using MAP or COMMON areas to share data between main programs and subprograms, remember that:

- You must check the lengths of your string and integer elements to make sure that you correctly line up the areas reserved by your main and subprograms

- If you compile your BASIC program with double precision, you must reserve eight bytes of storage for each floating-point variable in the corresponding PSECT of your MACRO subprogram

- You must assign the same name and attributes to a MAP or COMMON area of the main program, and the corresponding PSECT of the subprogram

You can use MACRO routines to initialize MAPs and COMMONs in a BASIC main program. For example, a main program could begin with COMMON statements, in which it stored data that both the main program

and subprogram want to use in printing error messages or checking maximum values. If you were to assign values to those COMMON areas by means of statements in the main program, the first lines of the BASIC source code would look like this:

```
10 COMMON (FIXSTR)          OUT.STR$    = 10%,        &
                            BAD.INFO$   = 24%,        &
                            ATLIN$      =  8%
20 COMMON (FIXDAT)          MAXNUM%,                  &
                            MAXVAL,                   &
                            BADNUM%,                  &
                            FUN.STR$    = 6%
30 OUT.STR$ = "OUTPUT IS"                             &
   BAD.INFO$ = "BAD INFORMATION SUPPLIED"             &
   ATLIN$    = " AT LINE"
40 MAXNUM%   = 100%                                   &
   MAXVAL%   = 2E6                                    &
   BADNUM%   = -1%                                    &
   FUN.STR$  = " FUNNY"
```

In this example, seven statements are executed to initialize variables in the COMMON area. In addition, each constant is allocated storage before being placed in the COMMON, and none of this storage is recovered. The following MACRO code performs the same initialization procedure as the BASIC code above:

```
     .TITLE INIT
; MODULE FUNCTION:
;     THIS MODULE INITIALIZES THE COMMON
;     AREAS OF THE MAIN PROGRAM

; INITIALIZE FIXSTR
     .ENABLE  LC                              ;ENABLE LOWER CASE
     .PSECT   FIXSTR,RW,D,GBL,REL,OVR
     .ASCII   /Output is /                    ;OUT.STR$ len = 10
     .ASCII   /Bad information supplied/      ;BAD.INFO$ len = 24
     .ASCII   / at line/                      ;ATLIN$    len = 8

; INITIALIZE FIXDAT
     .PSECT   FIXDAT,RW,D,GBL,REL,OVR
     .WORD    100.                            ;MAXNUM%
     .FLT2    2E6                             ;MAXVAL
     .WORD    -1                              ;BADNUM
     .ASCII   / FUNNY/                        ;FUN.STR$ len = 6
     .END
```

This routine is not, strictly speaking, a subprogram. The main program cannot call it, as it does not contain any executable statements. But if you build this module into your task as though it were a subprogram, you can omit statements 30 and 40 in the main program. An initialization routine like this one, therefore, can save you both time and memory space when you run the task.

NOTE

Because this routine contains no code and is not a subprogram, you cannot call it later in the main program to reinitialize values in the COMMON.

4.1.2.4 Building the Task — Follow this general procedure when you build your multisegment task:

- Compile the BASIC modules and assemble the MACRO modules.
- Include in the BUILD command all the object modules you wish to combine into a single task image, or modify the BUILD–generated ODL file to include individual subprograms in the task. If your program includes RMS file operations, use the appropriate BUILD command switch to incorporate RMS code into your task.
- Use the TKB or LINK command, depending on your operating system, to build the task image.

The ODL file generated by the BUILD command causes the Task Builder to concatenate all the modules in the root of the task. Since the operating system and hardware impose restrictions on task size, you may need to design an overlay structure for the task. For more information on overlay structures, see Section 4.1.1.8 of this chapter, and your Task Builder or Linker manual. In general, when designing an overlay structure for a task including either BASIC or MACRO subprograms, bear these considerations in mind:

BASIC or MACRO subprograms, bear these considerations in mind:

1. Think about overlay structure in the early stages of programming. Design your task to take advantage of overlay.
2. Test each module separately, writing small programs to call the modules and supply whatever data they need.
3. Be sure you know where global symbols will be resolved, and when overlays will be brought into memory when you run the task.
4. Be sure to align MAP and COMMON variables in the main program and subprograms.
5. Use co-trees (that is, overlay structures with independent root segments) only when necessary.

In most ways, designing an overlay structure for a task that includes a MACRO subprogram is the same as designing an overlay structure for a task that includes a BASIC subprogram. However, a knowledge of MACRO may enable you to take advantage of BASIC threaded code to decrease task size and enhance performance.

When the BASIC compiler translates a source program into object code, it generates threaded, rather than in-line, code. That is, the compiler generates out of the BASIC source program a series of global symbols and arguments. These symbols are names of routines that will perform the operations the user task requires. When you build the task image, the Task Builder resolves the global symbols by searching within the modules of the task itself, and within the BASIC object library and resident library, for the routines they refer to.

By compiling a BASIC program with the COMPILE/MACRO command, you can find out what global symbols the Task Builder will have to resolve. For example:

```
20 PRINT A%
```

From this line, the compiler generates the following threaded code:

```
L20:    LIN$        ,20             ; #20
        CLI$S
        IPT$
        MOI$MS      ,$IDATA+800     ; A%
        PVI$SI      ,0              ; #0
        EOL$
```

The code generated by the COMPILE/MACRO command is not the same as MACRO code you might use when programming the task. Rather, it is the MACRO equivalent of the object code generated by the compiler. In this example, L20: is a label identifying this particular block of code, while LIN$, CLI$ and so on are global symbols representing BASIC routines. The Task Builder follows a prescribed sequence to resolve these global symbols:

- It first searches within the module itself, and within other modules in the same segment, for a resolution of the symbol.
- It then searches in modules along the same branch toward the root, in the root module, and in the memory resident library if there is one.
- It then searches in modules along the same branch away from the root.
- It then searches co-trees.
- It finally searches in the BASIC object library.

When designing the overlay structure for a task, remember this resolution sequence. If you do not, your task may not be executable. For example, suppose you design an overlay structure in which one subprogram calls a second subprogram. The second subprogram contains an undefined symbol that you expected the executive to resolve by searching the BASIC object library. However, the executive resolves that symbol by bringing a third subprogram into memory and overlaying that third subprogram on the first one. When the second subprogram is finished, and attempts to return to the first subprogram, the task aborts with a ?Memory management violation or ?Odd address trap error.

You can avoid this problem, and at the same time conserve space in the task, by placing in the root any threads needed to resolve global symbols, especially potentially ambiguous ones. Inspect your task map (filename.MAP) to find the OTS module that contains the thread you need. Then edit your ODL file to put the module in the root segment. The ODL file below, for example, shows how an RSX–11M system uses the $IQNMA module to force the NAME AS (NMA$) into the root segment of the task MNPROG.

```
        ,ROOT BASIC2-RMSROT-USER,RMSALL
USER:   ,FCTR SY:MNPROG-LB:[1,1]BASIC2/LB:$IQNMA-LIBR-*(BR1,BR2)
BR1:    ,FCTR SY:SUBPR1-LIBR
BR2:    ,FCTR SY:SUBPR2-LIBR
LIBR:   ,FCTR LB:[1,1]BASIC2/LB
@LB:[1,1]BP2IC1
@LB:[1,1]RMS11S
        ,END
```

The file specifications in the ODL file vary, depending on the operating system. See your Task Builder or Linker manual for more information.

4.1.2.5 Handling Errors — MACRO subprograms should not contain error-handling routines that will abort the task. If a fatal error occurs in a MACRO subprogram, the subprogram should return control to the main program, and signal to it that an error has occurred.

You can use parameters or COMMON areas to return status information to the calling program. Or you can use ERR, ERL and ERN$ functions in the BASIC program to determine the kind and source of error. The information returned by ERN$ and ERL differs, depending on whether the BASIC program uses a CALL or CALL BY REF statement:

- If the program uses a CALL statement, ERN$ returns the name of the subprogram called and ERL returns a value of 0.

- If the program uses a CALL BY REF statement, ERN$ returns the name of the calling program and ERL returns the line number of the CALL BY REF statement.

4.1.3 BASIC to COBOL Subprogramming

BASIC can call subprograms written in PDP–11 COBOL V4.1. The name of the subprogram in the BASIC CALL statement must correspond to the name in the PROGRAM–ID line of the COBOL program. For example:

Main Program (BASIC)

```
10 PRINT "BEGIN MAIN PROGRAM"
20 CALL COBSUB
30 PRINT \ PRINT "RESUME MAIN PROGRAM"
40 END
```

Subprogram (COBOL)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBSUB.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. PDP-11.
OBJECT-COMPUTER. PDP-11.
DATA DIVISION.
PROCEDURE DIVISION.
P01-BEGIN.
    DISPLAY "ENTER SUBPROGRAM".
P02-END.
    EXIT PROGRAM.
```

Use the COBOL/SUB command to compile the COBOL subprogram. To build the task image, edit the ODL file generated by the BUILD command to include the COBOL library of object modules. For example:

```
        .ROOT USER
USER:   .FCTR SY:MAINPR-SUBPR-LIBR
SUBPR:  .FCTR SY:COBSUB-LIBC
LIBC:   .FCTR LB:[1,1]COBLIB/LB
LIBR:   .FCTR LB:[1,1]BASIC2/LB
        .END
```

ODL syntax varies, depending on the operating system. See your Task Builder or Linker manual for more information.

In general, BASIC and COBOL do not have the same type of data representation. However, BASIC string and integer data types correspond to COBOL display and computational data types. By using these data types and converting data as necessary, you can pass parameters from a BASIC program to a COBOL subprogram.

BASIC does not support I/O operations in COBOL subprograms.

NOTE

COBOL main programs cannot call BASIC subprograms.

4.2 Chaining

The CHAIN statement transfers control from the current program to another, executable program. The format is:

CHAIN string-expression

where:

string-expression — represents the file specification of the new program's task image.

When BASIC encounters a CHAIN statement, it closes all open files and then requests the new program to begin execution. If it cannot find the new program, BASIC prints the error message "?File not found". Figure 4–11 shows how you use the CHAIN statement to transfer program control.

Figure 4–11: Using the CHAIN Statement

Program 1

```
10 PRINT "PROGRAM 1 IS WORKING"
20 OPEN "DATA1.DAT" FOR OUTPUT AS FILE #1%
30 FOR I% = 1% TO 100%
40 PRINT #1%, I% * 2%
50 NEXT I%
60 CLOSE #1%
70 CHAIN "PROG2"
80 END
```

Program 2

```
10 PRINT "PROGRAM 2 IS WORKING"
20 OPEN "DATA1" FOR INPUT AS FILE #1%
30 FOR I% = 1% TO 100%
40 INPUT #1%, I%
50 T% = T% +I%
60 NEXT I%
70 PRINT "THE TOTAL IS ";T%
80 CLOSE #1%
90 END
```

Compile and task-build these programs separately. When you run them, they return the following data:

```
PROGRAM 1 IS RUNNING
PROGRAM 2 IS RUNNING
THE TOTAL IS 10100
```

NOTE

The procedure and format for the CHAIN statement vary, depending on your operating system. See Chapters 6 through 10 of this manual for more information. DIGITAL recommends that you use subprogramming rather than chaining whenever you segment your programs.

Chapter 5
BASIC-PLUS-2 Utilities

The BASIC-PLUS-2 utilities are:

Translator	helps convert BASIC-PLUS programs to BASIC-PLUS-2.
Resequencer	renumbers program lines according to a specified increment.
Cross Reference Program	creates a cross reference file of: (1) line numbers, variables, user-defined functions, COMMONs, and MAPs, and (2) where they are referenced in the program.

The utilities are installation options. See your system manager if you cannot invoke them.

5.1 Translator

The translator utility converts BASIC-PLUS programs, written in either EXTEND or NOEXTEND mode, to BASIC-PLUS-2. In addition, it detects and issues warnings about potential incompatibilities between the two languages.

NOTE

The translator provides only a migration aid for converting programs from BASIC-PLUS to BASIC-PLUS-2. Using the translator can create inefficient BASIC-PLUS-2 programs. Therefore, you should recode your BASIC-PLUS programs into BASIC-PLUS-2.

5.1.1 Using the Translator

5.1.1.1 Calling the Translator — The translator utility program is in the system library account. To access it, type:

```
RUN $TRANS
```

and the translator prints an identification header. The translator then prints:

```
INPUT FILE?
```

To answer, type the name and extension of the BASIC-PLUS program you want to convert, in the format:

dev:[account-number]filename.extension.

The default extension is .BAS. The specified input program must be error free. If the program contains immediate mode statements, the translator issues a message that it has removed them. Syntax errors create unpredictable translator output.

You can enter only one program to the INPUT FILE prompt. Following this prompt, the translator prints:

```
OUTPUT FILE?
```

which lets you assign a different name and extension to the converted program. The default is the input file name with an extension of .B2S. The output file also uses default device and account specifications.

Following the OUTPUT FILE? prompt, the translator prints:

```
EXTEND MODE<NO>?
```

If your BASIC-PLUS program is written in EXTEND mode, respond to this prompt with YES. This converts any syntactic differences between BASIC-PLUS EXTEND and BASIC-PLUS-2. The translator shifts to the connect mode if it encounters an EXTEND or NOEXTEND statement in the input program.

NOTE

Typing /HELP in response to a prompt displays a brief description of that prompt (and those that follow) on your terminal.

5.1.1.2 Specifying Variable Names — If you select NOEXTEND, the translator requests the BASIC-PLUS variable name(s) you want to change. For example:

```
EXTEND MODE<NO>?

OLD NAME?
```

If you do not want to change any variable names, type a carriage return. Otherwise, type the existing BASIC-PLUS variable name. The translator then prompts for the new name (a BASIC-PLUS-2 name of up to 30 characters). You cannot include function, array subscript, string, or integer designations in the new name--for example, FN, (, $, or %. Consider the following:

```
OLD NAME? FNT1%
NEW NAME? TEMP1
```

When you have typed all the names you want to convert, type a carriage return.

The translator changes the variable names, but not the variable type or any subscripts. For example:

```
OLD NAME? A(
NEW NAME? AARRAY
```

changes A(in the BASIC-PLUS program to AARRAY(in the BASIC-PLUS-2 program. You can also build a command file containing the variable names you want to convert by specifying:

OLD NAME? @filespec[.CMD]

You can prepare this file with an editor. Place the old variable name on the line before the new name. For example:

```
A1%(
NEW.ARRAY
FNW$
NEW.FUNCTION
```

5.1.1.3 Translator Sample Run — This is a sample BASIC-PLUS program translation:

```
105       I% = 2%
110       A%(I%),I% = 10%
120       B% = 5%
130       A%,B%,C%,A%(I%) = 1%
140       A%(C%),A%(C% + 1) = 0%
210       INPUT A% B% C%
220       PRINT A%,B%,C%
230       P R I N T "KEY WORD WITH IMBEDDED SPACES AND TABS"
305       B% = 4%
308       PRINT " BRACKETS IN A LITERAL [] ARE LEFT ALONE"
310       X% = 5%*[3% - B%] + 2%
312       A%[2%] = 0%
315       PRINT X%
410       OPEN "DATA" FOR OUTPUT AS FILE #1, CLUSTER SIZE 4%
420       OPEN "DATA2" FOR OUTPUT AS FILE #2%, RECORD SIZE 512
502       DIM A(60)
505       A = 5 + 4 5 6 8
506       A(5 6) = 10 !5   6
510       PRINT 5 6 7
520       PRINT 5 6
53        0 PRINT 8 8 !
535       PRINT " 5 4 3 2 "
```

```
540     PRINT 45
610     PRINT 5 E 4
620     PRINT 3E - 4
630     PRINT 5E 4
640     PRINT 5 E4
705     OPEN "DATA" FOR OUTPUT AS FILE #2%
710     FIELD #2%, 10% AS A$
720     A$ = SYS(B$)
730     A% = PEEK(B%)
810     CHANGE A% TO A$
815     A%(5%) = 5%
817     A$ = "LUCKY"
820     CHANGE A$ TO A%
830     CHANGE B TO B$
835     B$ = "FROG"
837     B(2) = 10
840     CHANGE B$ TO B
910     MAT PRINT A%
915     A%(B%) = 0%
1005    OPEN "DATA" FOR OUTPUT AS FILE #1%
1010    PRINT #1,A$
1020    PRINT #1,
1030    PRINT USING "###,###" 100000
1040    PRINT #1,USING "###,###" 100000
1105    A% = 3%
1110    PRINT 4 + A%/2
1215    END
```

The following dialogue converts this program to BASIC-PLUS-2. Variable names are extended.

```
>

RUN $TRANS

PDP-11 BP2 TRANSLATOR V1.6 BL-01.60

INPUT FILE? TRXY15.BAS

OUTPUT FILE? TRXY15.B2S

EXTEND MODE<NO>?

OLD NAME? A%
NEW NAME? ASCALER
OLD NAME? B%
NEW NAME? BSCALER
OLD NAME? A%(
NEW NAME? ARRAYA
OLD NAME? B(
NEW NAME? ARRAYB
OLD NAME?
%Multiple assignments at line 110
%Multiple assignments at line 130
%Multiple assignments at line 410
%Possible system incompatibility: CLUSTERSIZE at line 40
%Possible system incompatibility: SYS at line 720
%Possible system incompatibility: PEEK at line 730
%Ambiguous division at line 1110
INPUT FILE? <ret>

>
```

If you list the converted program (TRXY15.B2S), it appears as:

```
105     I% = 2%
110     ARRAYA%(I%),I% = 10%
120     BSCALER% = 5%
130     ASCALER%,BSCALER%,C%,ARRAYA%(I%) = 1%
140     ARRAYA%(C%),ARRAYA%(C% + 1) = 0%
210     INPUT ASCALER% ,BSCALER% ,C%
220     PRINT ASCALER%,BSCALER%,C%
230     PRINT "KEY WORD WITH EMBEDDED SPACES AND TABS"
305     BSCALER% = 4%
308     PRINT "BRACKETS IN A LITERAL [] ARE LEFT ALONE"
310     X% = 5%*(3% - BSCALER%) + 2%
312     ASCALER%(2%) = 0%
315     PRINT X%
410     OPEN "DATA" FOR OUTPUT AS FILE #1, CLUSTERSIZE 4%
420     OPEN "DATA2" FOR OUTPUT AS FILE #2%, RECORDSIZE 512
502     DIM A(60)
505     A = 5 + 4568
506     A(56) = 10 !5  6
510     PRINT 567
520     PRINT 56
530     PRINT 88 !
535     PRINT " 5 4 3 2 "
540     PRINT 45
610     PRINT 5E4
620     PRINT 3E - 4
630     PRINT 5E4
640     PRINT 5E4
705     OPEN "DATA" FOR OUTPUT AS FILE #2%
710     FIELD #2%, 10% AS A$
720     A$ = SYS(B$)
730     ASCALER% = PEEK(BSCALER%)
810     CHANGE ARRAYA% TO A$
815     ARRAYA%(5%) = 5%
817     A$ = "LUCKY"
820     CHANGE A$ TO ARRAYA%
830     CHANGE ARRAYB TO B$
835     B$ = "FROG"
837     ARRAYB(2) = 10
840     CHANGE B$ TO ARRAYB
910     MAT PRINT ARRAYA%
915     ARRAYA%(BSCALER%) = 0%
1005    OPEN "DATA" FOR OUTPUT AS FILE #1%
1010    PRINT #1,A$
1020    PRINT #1,
1030    PRINT USING "###,###", 100000
1040    PRINT #1 USING "###,###", 100000
1105    ASCALER% = 3%
1110    PRINT 4 + ASCALER%/2
1215    END
```

This program did not translate with complete accuracy. As in any program translation, you should check the code and a sample run. This program requires the following corrections:

1. Examine lines 110, 130, and 140 for multiple assignment problems. Separate line 110 into two statements.

2. Remove the statements using CLUSTERSIZE, SYS, or PEEK if they are not valid keywords on your system.

3. Eliminate the ambiguous constant division at line 1110 by adding a percent sign (%) after the 2.

5.1.2 Translation Of BASIC Program Elements

This section describes: (1) program elements that translate to BASIC-PLUS-2, and (2) program elements that are fully compatible without translation.

5.1.2.1 Program Elements Translated To BASIC-PLUS-2 — The translator changes BASIC-PLUS syntax to BASIC-PLUS-2 syntax in these areas:

Continuation Lines

BASIC-PLUS continues program lines with a line feed or an ampersand (&) followed by a line terminator. BASIC-PLUS-2 continuations can be made only with an ampersand (&) followed by a line terminator.

In NOEXTEND mode, BASIC-PLUS does not recognize the ampersand as a continuation character. You can include it in comment fields with no effect. Since BASIC-PLUS-2 recognizes all ampersands followed by a line terminator as continuation characters, the translator encloses in quotation marks ampersands that appear in comment fields.

DEF Statements

BASIC-PLUS and BASIC-PLUS-2 user-defined functions differ in the way they pass arguments. However, BASIC-PLUS-2 supports the BASIC-PLUS method. The translator adds an asterisk to BASIC-PLUS DEF statements (DEF*) to indicate BASIC-PLUS argument passing.

PRINT Synonym

BASIC-PLUS accepts an ampersand as a synonym for PRINT. The translator changes all BASIC-PLUS ampersands to PRINT statements unless they are in string literals or are continuation characters.

CHAIN Statements

BASIC-PLUS CHAIN statements do not have the keyword LINE before line numbers. The translator therefore inserts this word.

Statement Separators

BASIC-PLUS separates multiple statements with a colon or a backslash. BASIC-PLUS-2 allows only a backslash. The translator converts all BASIC-PLUS statement separators to backslashes.

Embedded Spaces and Tabs

BASIC-PLUS ignores embedded spaces and tabs in unquoted string literals. BASIC-PLUS-2 considers them significant. The translator removes embedded spaces and tabs from numeric constants, line numbers, keywords with optional spaces in BASIC-PLUS (for example, CLUSTER SIZE), data statements, and exponential format numbers.

Functions

The translator changes:

- The BASIC-PLUS POS function to CCPOS
- The BASIC-PLUS CVT$$ function to EDIT$

Long Variable Names

BASIC-PLUS does not permit "long format" variable names. BASIC-PLUS-2 permits variable names of up to 30 characters. The translator provides you with prompts to change variable names. You can also create a command file containing BASIC-PLUS and BASIC-PLUS-2 names and change the names from the file.

Spaces

Because BASIC-PLUS-2 allows long variable names, it requires spaces or tabs between keywords, variable names, and literals to avoid ambiguity. The translator inserts spaces between these BASIC-PLUS elements.

Comment Separators

BASIC-PLUS comments begin with an exclamation point and end with a line terminator. BASIC-PLUS-2 comments begin and end with an exclamation point, which permits you to write comments at any point in a program line. The translator removes all but the first exclamation point from BASIC-PLUS lines and replaces them with asterisks to preserve spacing. If a BASIC-PLUS program uses the BASIC-PLUS-2 method of inserting comments, the translator will replace the second exclamation point with an asterisk. When you run the program, any program statements after the comment field will be ignored, and your program can return erroneous results. This does not apply to string literals.

You cannot continue BASIC-PLUS-2 comments; the translator inserts exclamation points to convert BASIC-PLUS continued comments.

Unterminated String Literals

BASIC-PLUS delimits string literals with a line terminator. BASIC-PLUS-2 requires matching single or double quotation marks on both sides of the string. The translator adds the correct delimiter to BASIC-PLUS string literals.

Multiple Assignment Statements

BASIC-PLUS evaluates multiple assignment statements from right to left. BASIC-PLUS-2 evaluates them from left to right. The translator issues a warning when multiple assignment statements contain array references, since the reverse order of evaluation may affect program execution.

Ambiguous Constants

BASIC-PLUS treats ambiguous constants (for example, 100 as opposed to 100% or 100.) as integers if an integer value appears to the left of the

constant in the expression. If such an integer were not present, BASIC-PLUS would treat the ambiguous constant as a floating-point number.

BASIC-PLUS-2 always treats ambiguous constants as floating-point numbers.

The translator prints a warning message if it encounters an ambiguous constant division (for example, A%/100).

Semicolons

In certain cases, BASIC-PLUS allows an implied semicolon in PRINT and INPUT statements. BASIC-PLUS-2 requires a comma or semicolon between items in a PRINT statement list and assumes a comma by default in INPUT statements. For example:

```
10 PRINT "SN" X
```

is legal in BASIC-PLUS, but returns a syntax error in BASIC-PLUS-2.

The translator inserts required semicolons in BASIC-PLUS PRINT and INPUT statements.

Line numbers

The translator removes percent signs from line numbers in BASIC-PLUS programs. For example:

```
100% PRINT
```

converts to:

```
100 PRINT
```

However, the percent sign remains in line number references (for example, GOTO 100%), but this does not affect execution.

Numeric Constants

BASIC-PLUS allows blanks and tabs in a numeric constant; BASIC-PLUS-2 does not. For example:

```
100 PRINT 5 32
```

in a BASIC-PLUS program outputs the constant 532. The same line in a BASIC-PLUS-2 program returns a syntax error. The translator compresses blanks and tabs in numeric constants.

RND Function Arguments

BASIC-PLUS permits an argument with the RND function. BASIC-PLUS-2 does not. The translator strips RND function arguments.

Null Arguments in Functions

BASIC-PLUS allows null arguments in user-defined functions; BASIC-PLUS-2 does not. The translator removes null arguments in functions. For example:

```
10 Y = FNA()
```

converts to:

```
10 Y = FNA
```

Files

BASIC-PLUS accepts BLOCK and RECORD clauses in record I/O statements. In BASIC-PLUS-2 these keywords are synonymous, and the translator converts BLOCK to RECORD.

In BASIC-PLUS, the comma is optional after the channel number. In BASIC-PLUS-2 they are required except for the PRINT # USING statement. The translator inserts commas where required.

5.1.2.2 Program Elements Not Requiring Translation — BASIC-PLUS-2 supports the following BASIC-PLUS features only on RSTS/E systems.

1. OPEN statements
2. Foreign buffers
3. Most SYS function calls
4. PEEK function calls

5.1.3 Translator Limitations

These sections describe translator limitations, including: (1) incomplete translations, (2) unresolved problems, and (3) incompatible BASIC-PLUS statements.

5.1.3.1 Incomplete Translations — The translator cannot completely translate: (1) continued statements longer than 512 characters, or (2) long and complex program lines.

- Continued statements cannot exceed 512 characters. After the first 512 characters, the translator issues an error message and goes to the next statement.
- Long and complicated lines exhaust available space in the translator's internal tables; the translator issues an error message and does not finish translating the line.

5.1.3.2 Unresolved Problems In Translation — These translation problems are unresolved:

1. Transfer into a FOR loop.
2. Array subscript checking. BASIC-PLUS checks the total, while BASIC-PLUS-2 checks each subscript. The following example is acceptable in BASIC-PLUS, but returns a '?Subscript out of range' error in BASIC-PLUS-2:

```
10    DIM A%(40,10)
20    PRINT A%(41,0)
```

3. SYS function calls. Certain SYS calls return an undefined result. In BASIC-PLUS, the value is the input string, while BASIC-PLUS-2 returns the null string as the function value.
4. PRINT USING statements. The formatting allowed by BASIC-PLUS-2 is more complex than that allowed by BASIC-PLUS. Therefore, BASIC-PLUS-2 accepts, as formatting directives, certain constructs that BASIC-PLUS treats as literals.
5. FIELD and CVT functions. BASIC-PLUS-2 supports these functions on all systems. However, BASIC-PLUS-2 CVT functions reverse the order of the bytes when moving them to or from a string. Thus, you can mix MAP and MOVE statements, but you cannot use FIELD and CVT functions on a file if you also plan to use MAP or MOVE.
6. Exiting from a function via the wrong FNEND statement. You can exit from a function in BASIC-PLUS by executing any FNEND statement. In BASIC-PLUS-2, however, you must exit via the FNEND statement associated with the DEF* statement that begins the function.
7. String length declarations. BASIC-PLUS permits you to declare lengths for any string or string array. BASIC-PLUS-2 does not permit string length declarations outside of virtual arrays, COMMONs, or MAPs. You must remove invalid length declarations before running the program.
8. LSET and RSET statements. Because BASIC-PLUS and BASIC-PLUS-2 handle string assignments differently, the string variables assigned in an LSET or RSET can produce unexpected results. The translator flags LSET and RSET for your review.
9. String concatenation with the null string. In BASIC-PLUS, the statement C$ = C$ + " " manipulates the string header. In BASIC-PLUS-2, this manipulation is optimized out. Recode the statement so that the null string is replaced with an unused string variable.

5.1.3.3 Incompatible BASIC-PLUS Statements — BASIC-PLUS-2 does not support:

1. PRINT # USING with RECORD clauses
2. GET USING statements

They cannot translate into valid BASIC-PLUS-2 statements.

5.1.3.4 System Incompatibilities — The keywords CLUSTERSIZE, LOCK, MSGMAP, PEEK, POKE, SYS, CHAIN (with a line number), MODE, TST, TSTEND and WRKMAP are system-specific in BASIC-PLUS-2. During translation, you receive the warning message "%Possible system incompatibility" to indicate that the program might not run on some operating systems. If you plan to run programs on a system that does not support these words, remove them.

5.1.4 Translator Error Messages

The translator prints a warning message when it detects an error or potential translation problem. The messages are:

%Ambiguous constant division at line n

WARNING- The program contains an ambiguous constant as a divisor (for example, A%/100) that can cause unexpected results when the program is run.

%Bad file specification - Please use form [#,#]NAME.EXT

WARNING- The format of the input or output file is not correct. The translator then reprompts for the input or output file.

?Can't find file or account

FATAL- The translator cannot find your input file.

%Entry already exists

WARNING- You entered a duplicate variable name in response to the NEW NAME? prompt. After printing this message, the translator uses the first name you specified.

%Illegal switch

WARNING - A slash (/) is not a valid response except when specified with the /HELP request.

%Immediate mode statement removed

WARNING- Because BASIC-PLUS-2 does not support immediate mode, the translator removes immediate mode statements from BASIC-PLUS programs. It then prints the warning message with a list of the statements it has removed.

%Internal Space Exhausted at line n

WARNING - Long and complicated lines exhaust available space in the translator's internal tables. The current line is incompletely translated.

%Invalid variable name

WARNING - The translator prints this message during the variable name substitution dialogue if:

1. A response to the NEW NAME? prompt ends in %, ., $, or (

2. You enter an illegal variable name

3. You enter a variable name that is too long

The translator then repeats the OLD NAME? prompt.

%LSET or RSET used at line n

WARNING - Because BASIC-PLUS and BASIC-PLUS-2 handle string assignments differently, the string variables assigned in an LSET or RSET can produce unexpected results.

%Multiple assignments at line n

WARNING - This message points out an array with a variable subscript (for example, A%(I%)) in an assignment list. It does not mean that an error will occur when you run the program; rather, an error might occur because the the order of assignments differ in BASIC-PLUS and BASIC-PLUS-2.

%Possible system incompatibility

WARNING - Your program contains one of these system-specific words: CLUSTERSIZE, LOCK, MSGMAP, PEEK, POKE, SYS, CHAIN (with a line number specification), MODE, TST, TSTEND or WRKMAP. If these words are invalid on your operating system, remove them.

%PRINT USING used at line n

WARNING - BASIC-PLUS-2 permits more complex formatting strings than does BASIC-PLUS. Literals in BASIC-PLUS formatting strings can be interpreted as formatting characters.

%Statement too long at line n

WARNING - The translator cannot completely translate a single continued statement longer than 512 characters.

%Static String declaration at line n

WARNING - BASIC-PLUS permits declared lengths for any string or string array. BASIC-PLUS-2 does not permit such declarations outside virtual arrays, COMMONs, or MAPs. You must remove invalid string declarations before running the program.

5.2 Resequencer

The resequence utility renumbers program lines (and references to those lines) throughout a specified program. You can reorder up to 2500 program lines at one time. You can divide these lines in up to ten segments and specify a different resequencing scheme for each segment. In this way, you can resequence large programs piece by piece until your program has the numbering order you want.

5.2.1 Invoking the Resequence Utility

The command:

```
RUN $B2RESQ
```

invokes the Resequence Utility. If it does not work, see your system manager for the way to invoke this on your system.

5.2.2 Running the Resequencer

The resequencer prompts you for information about renumbering. You can respond to these prompts individually, or specify a indirect command file instead. The following sections describe these procedures.

5.2.2.1 Resequence Utility Dialogue — The resequence utility prompts for a file name and line number specification(s) as follows:

- ENTER BASIC-PLUS-2 Program To Resequence?

  Enter the name of the file you want to renumber. The default extension is .B2S.

- ENTER OUTPUT FILE?

  asks for the name of the renumbered file. Respond with a new output file name. The default extension is .B2S.

- Number of Program Segments to be Resequenced?

  asks for the number of program segments you want resequenced. You can respond by typing:

  - H — prints a help file list of resequence utility commands.
  - A carriage return (RET) — resequences the entire program, starting at line 10 and incrementing by 10.
  - An indirect command file name — specifies resequencing instructions. See Section 5.2.2.2. If you specify an indirect command file, you do not receive the next four prompts.
  - A number — specifies how many segments you want to resequence. You then receive the next four prompts for each program segment.

- SEGMENT n OLD BEGINNING LINE NUMBER?

  asks for the number where resequencing begins. The default is line 1.

- OLD END LINE NUMBER?

  asks for the line number that ends the segment you are resequencing. The default is line 32767.

- NEW BEGINNING LINE NUMBER?

  asks for the segment's new starting line number. The default is line 10.

- NEW INCREMENT THIS SEGMENT?

  asks for the increment between line numbers. The default is 10 (for example: 10, 20, 30, and so forth).

After the last prompt, the resequence utility renumbers the program according to the segment definitions. It then updates line number references in control statements (for example, GOTO and THEN) to reflect the new order.

5.2.2.2 Command File Input to Resequencer Dialogue — Instead of responding to dialogue prompts, you can include your answers in an indirect command file. You specify this file as input to the NUMBER OF PROGRAM SEGMENTS TO BE RESEQUENCED prompt. The format for resequence commands is:

```
command:[,command[:...command]]
```

where:

command	is one of four resequence commands summarized in Table 5–1. You can continue commands on the next line with the ampersand (&) continuation character.
colon (:)	is a command separator. You end each command except the last with a colon.
comma (,)	is a segment separator. A segment is a unique group of program lines.

Table 5–1: Resequence Commands

Command	Meaning
O m-n	Resequence the segment defined as line numbers m through n. The default for m is 1; the default for n is 32767.
N m	Begin resequencing the segment at line m. The default for m is line 10.
I d	Increment line numbers by a value of d. The default is 10.

For example:

```
01-100:N10:I1,0150-200:N50, &
01000-10000:N1000:I50
```

resequences the program with these changes:

Command	Change
O1-100:N10:I1	resequences old line numbers 1 through 100 (O1-100:), starting at line 10 (N10:) and incrementing by 1 (I1).
O150-200:N50	resequences old line numbers 150 through 200 (O150-200:), starting at line 50 (N50). Because the command line specified no increment value, the default is 10.
O1000-10000:N1000:I50	resequences old line numbers 1000 through 10000 (O1000-10000:), starting at line 1000 (N:1000) and incrementing by 50 (I50).

After creating the file, type:

@filespec

in response to the NUMBER OF PROGRAM SEGMENTS TO BE RESEQUENCED? prompt. The resequence utility then renumbers your program. If the filespec contains no extension, the default is .CMD.

You will not receive the rest of the resequence dialogue.

5.2.3 Error Messages

The Resequence Utility prints an error message when it detects a resequencing error. The messages are:

%Bad program format. A line number was expected and not found

The input file contains a non-continued line that does not start with a line number.

?File not found

The Resequence Utility cannot find your input file.

?Input line numbers are out of strictly ascending order

The line numbers in the file are not in ascending order. You can re-order program lines by:

- Invoking the BASIC-PLUS-2 compiler.
- Bringing the program into memory with the OLD command.
- Issuing the REPLACE command to save the program.

The compiler re-orders the line numbers and returns the corrected program to your directory. You can then resequence the corrected program.

?Invalid segment parameters

You specified an end line number lower than the starting line number.

%Line # not found, resequencing continuing

The input program contains a transfer to a non-existent line number. This line number reference remains unchanged in the output file.

?Proposed resequencing out of integer bounds

The line numbers of a segment will exceed 32767 when resequenced.

?Proposed resequencing overlaps

The new program segments will overlap each other when resequenced.

?Resequenced segment encompasses unresequenced line

A segment being resequenced will overlap a line you did not specify for resequencing.

?Segment descriptors overlap

You entered a line number in more than one segment.

?Specification file error - expecting more command data

Your command file is not correctly formatted.

?Specification file not found

The Resequence Utility cannot find your indirect command file.

?Syntax error, number too large for integer

The command line contained an integer outside the valid range (1%–32767%).

?Syntax error

The command line input contained an error.

?Two segments have identical new beginning value

Two program segments cannot start with the same line number.

5.3 Cross Reference Program

The cross-reference utility (B2XREF) creates an index of keywords, functions, and variables in a BASIC-PLUS-2 program. Your input program must be error free and in ascending line-number order.

5.3.1 Invoking B2XREF

The command:

```
RUN $B2XREF
```

accesses the B2XREF Utility. In response, B2XREF prints an identification header. If the prompt does not work, see your system manager.

5.3.2 Running B2XREF

B2XREF asks for file information with the "B2X" prompt. To respond, use the format:

outfile = infile / switches

For example:

```
TESTPG = TESTPG/FUN/SOU/OFF
```

You must specify an input file name. There is no default. If you specify no extension, B2XREF adds the default extension .B2S to the input file name.

The default output file name is the input file name with the extension .CRF. The available B2XREF switches are:

/APP[ENDABLE-LIST] prints a variable list you can append to a BASIC program. This switch suppresses references to line numbers, library functions, and keywords. The output file's extension becomes .APP, and its width is 80 characters.

/BAS[IC-PLUS] specifies a BASIC-PLUS input file. The default extension becomes .BAS.

/FUN[CTIONS] includes a reference for all library functions.

/HEL[P] prints a help file of the B2XREF command line and switches. If you specify /HELP, BASIC ignores the current command line.

/ISO[LATE]:arg prints line numbers, variables, user-defined functions, and subprogram calls referenced in the line numbers specified by the argument.

The argument can be a single line number or a pair of line numbers. A dash separates multiple line numbers. Up to 20 multiple arguments are valid for each /ISOLATE:arg switch. A comma separates multiple arguments. For example:

```
/ISOLATE:100-400,650-690,1240-1580
```

You can specify additional /ISOLATE:arg switches if needed.

/KEY[WORDS] includes a reference to all BASIC keywords.

/MIC[ROFICHE] prints the input file name as the first six characters on each page. This file name can then become part of a microfiche index.

/MORE

prompts for additional input. /MORE must be the last switch on the current command line.

/NOF[LAGS]

suppresses printing of the #, @, and P flags in the output file. The pound sign (#) indicates that a function, array, or subprogram is defined at the specified program line. The at sign (@) indicates a new value assigned to a variable. P indicates a variable used as a parameter in a subprogram call.

/OFF[SET]

prints references in the format:

n:o

where:

- n is the program line number.
- o is the number of the statement on the line.

For example:

```
80:3
```

refers to line 80, statement 3.

/REP[ETITION]

prints references in the format:

n(r)

where:

- n is the program line number.
- r is the occurrence of the reference on the line.

For example:

```
350(2)
```

indicates that the language element occurred twice on line 350.

/SOU[RCE]

includes a source listing with the output file. The default extension for the source file is .LST.

/SUP[RESS-SPACING]

eliminates the blank line between variable names in the output file.

/WID[E]

specifies a width of 132 characters in the output file.

/WID[TH]:n

specifies a width of n characters. The minimum width is 72. The default for a terminal is 80; for other devices, 132.

5.3.3 B2XREF Output

If you specify no switches with the input file name, the output is a list of line numbers only. Depending on the switches you select, the output appears in the following order:

- Variables

  The at sign (@) indicates that the variable was assigned a value at that line number. A "P" indicates that the variable is a parameter in a subprogram CALL. The value of that variable can change in the subprogram.

- User-defined Functions
- Subprograms
- COMMON and MAP Areas

  The names of COMMON and MAP areas are listed first, followed by the variables contained in them.

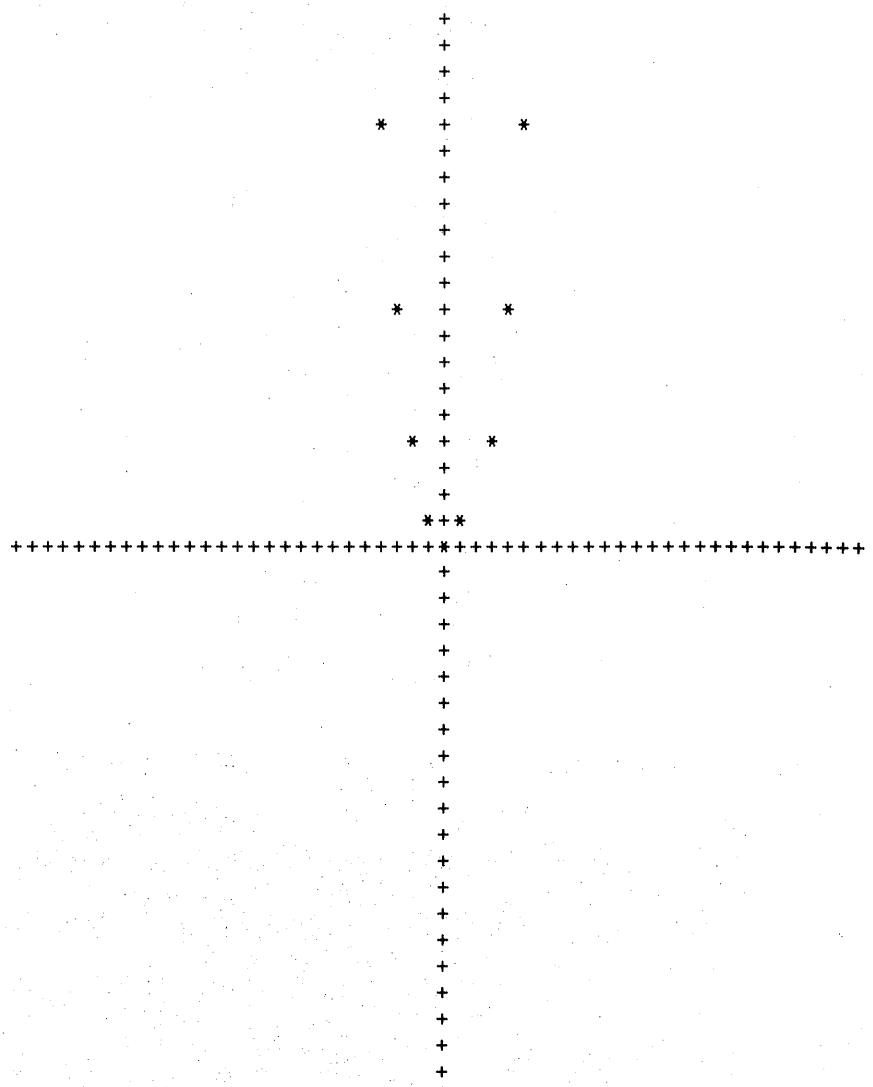
- Keywords
- Library Functions
- Variables not Assigned Values
- Distinct Identifiers and Number of Identifier References
- Number of Work Entries and Disk Blocks Required

5.3.4 B2XREF Sample Run

The following program illustrates the B2XREF program output.

```
RUN $B2XREF

PDP-11 BP2 CROSS REFERENCER V1.6 BL - 01.60

B2X> TESTRF = TESTRF/FUN/SOU/OFF

B2X> ^Z

PIP TESTRF.CRF/LI

Cross-Reference Listing of TESTRF on 24-Jan-80 at 04:32 PM

5         REM THIS IS A SAMPLE PROGRAM TO DEMONSTRATE B2XREF
10        DEF FNA(X,Y) = Y**2
20        FOR I = 20 TO -20 STEP -1
30             FOR J = -20 TO 20
40                  IF I = FNA (I,J) &
                         THEN PRINT '*'; &
                         ELSE IF I = 0 OR J = 0 &
                              THEN PRINT '+'; &
                              ELSE PRINT ' ';
70             NEXT J
80             PRINT
90        NEXT I
100       PRINT
110       END
```

```
Cross-Reference Listing of TESTRF on 24-JAN-80 at 04:32 PM
Variables
I       @20     40          40          40:3        90

J       @30     40          40:3        70

X       10

Y       10      10

User-defined functions

FNA( )          #10         40

Please check the use of the following:

X       Y

There were 5 identifiers listed, and 14 references to them. 8
work-file entries (2 disk blocks) were used.

RUN TESTPG
```



Chapter 6
BASIC–PLUS–2 on RSX–11M

This chapter describes system-specific usage of BASIC–PLUS–2 on RSX–11M. This includes:

- Invoking the compiler.
- Using the CHAIN, NAME AS, and SLEEP statements.

6.1 Invoking the Compiler

To invoke the BASIC–PLUS–2 compiler, type:

```
RUN $BASIC2
```

Depending on how your system was installed, you can also invoke the compiler with the concise command "BP2". See your system manager for the concise command form used on your system.

After you invoke the compiler, BASIC–PLUS–2 prints an identifying line and awaits your input. You can then create BASIC source programs and object modules (see Chapter 1).

6.2 BASIC–PLUS–2 Statements

This section describes the implementation differences for the CHAIN, NAME AS and SLEEP statements.

6.2.1 CHAIN Statement

The BASIC–PLUS–2 CHAIN statement enables chaining at specified points in the program. RSX–11M requires that you chain at the first line of the program. Use the syntax:

CHAIN "task name"

You can use only six characters for the task name.

Chaining uses the RQST$ system directive. This directive requires:

- Task image format. You can chain to an executable task image only.
- MCR installation. Tasks must be installed with the MCR INSTALL command.
- A unique copy of the task. You cannot have more than one running copy of the task when you chain to it.

6.2.2 NAME AS Statement

The NAME AS statement enables you to rename an existing file. It has the format:

```
NAME string 1 AS string 2
```

where:

string 1	is the file specification of the old file.
string 2	is the new file specification.

On RSX-11M:

- You must have write access to the directory of the target file.
- The specified files must reside on the same physical device.
- The PSECT $$FSR2 must be in the root segment of the task. You can: (1) rework the ODL file to force $$FSR2 into the root, or (2) include NAME AS in the root.

The NAME AS statement renames the file without changing its contents. For open files, the new name takes effect when the file is closed.

6.2.3 SLEEP Statement

The SLEEP statement suspends program execution for a specified time. The format is:

```
SLEEP num-exp%
```

where:

num-exp%	is the number of seconds that execution is suspended.

By enabling CTRL/C trapping (CTRLC function), you can awaken the job before the time period expires.

Chapter 7
BASIC–PLUS–2 on IAS

This chapter describes system-specific usage of BASIC–PLUS–2 on the IAS operating system. These system-specific uses include:

- Invoking the compiler.
- Using the CHAIN, NAME AS, and SLEEP statements.
- Restrictions on use.

7.1 Invoking the Compiler

To invoke the BASIC–PLUS–2 compiler, type:

```
RUN LB:[11,1]BASIC2
```

Depending on how your system was installed, you can also invoke the compiler with the concise command "BP2". See your system manager for the concise command form used on your system.

BASIC–PLUS–2 prints an identifying line and awaits your input. You can then create BASIC source programs and object modules (see Chapter 1).

7.2 BASIC–PLUS–2 Statements

This section describes the implementation differences for the CHAIN, NAME AS, and SLEEP statements.

7.2.1 CHAIN Statement

BASIC–PLUS–2 implements the CHAIN statement as an RQST$ system directive. Your system manager must provide the privileges for RQST$.

7.2.2 NAME AS Statement

The NAME AS statement enables you to rename an existing file. It has the format:

NAME string 1 AS string 2

where:

string 1 is the file specification of the old file.

string 2 is the new file specification.

On IAS:

- You must have write access to the directory of the target file.
- The specified files must reside on the same physical device.
- The PSECT $$FSR2 must be in the root segment of the task. You can: (1) rework the ODL file to force $$FSR2 into the root, or (2) include NAME AS in the root.

The NAME AS statement renames the file without changing its contents. For open files, the new name takes effect when the file is closed.

7.2.3 SLEEP Statement

The SLEEP statement suspends program execution for a specified time. The format is:

SLEEP num-exp%

where:

num-exp% is the number of seconds that execution is suspended.

7.3 Restrictions

The following sections describe system specific restrictions for BASIC-PLUS-2 running on IAS.

7.3.1 CTRL/C Trapping

BASIC-PLUS-2 does not recognize CTRL/C trapping on IAS systems.

7.3.2 IAS Batch Stream

In batch streams, BASIC requires a terminal device (TI:). In IAS batch, however, TI: is invalid, and BASIC cannot assign it.

To compile programs from batch, use an indirect command file. For example:

```
$JOB MYACCT EXAMPLE 20
$BP2 @IND.CMD
$@TSTPRG
$RUN TSTPRG
$EOJ
```

The indirect command file, IND.CMD, contains the BASIC commands that OLD, COMPILE, and BUILD the program.

Batch streams also prevent your using terminal control functions such as RCTRLO and CTRLC. BASIC disables these functions when you run in IAS batch. Your program then behaves identically in IAS batch and from the terminal.

7.3.3 Post Mortem Dumps

The post mortem dump of memory contents is not available on IAS systems. Therefore, you cannot use the /DUMP switch when compiling.

Chapter 8
BASIC–PLUS–2 on VMS (Compatibility Mode)

This chapter describes system-specific usage of BASIC–PLUS–2 on the VAX/VMS operating system. This includes:

- Invoking the compiler.
- Using the CHAIN, NAME AS, SLEEP, and KILL statements.
- Compiler commands.
- Restrictions on use.

8.1 Invoking the Compiler

To invoke the BASIC–PLUS–2 compiler, type:

```
BASIC
```

or

```
BASIC/RSX
```

After you invoke the compiler, BASIC–PLUS–2 prints an identifying line and awaits your input. You can then create BASIC source programs and object modules (see Chapter 1).

8.2 BASIC–PLUS–2 Statements

This section describes the implementation differences for the CHAIN, NAME AS, SLEEP, and KILL statements.

8.2.1 CHAIN Statement

BASIC-PLUS-2 implements the CHAIN statement with the RQST$ system directive. This directive is invalid on VMS-Compatibility Mode. Therefore, VMS-Compatibility Mode does not support chaining.

8.2.2 NAME AS Statement

The NAME AS statement enables you to rename an existing file. It has the format:

```
NAME string 1 AS string 2
```

where:

string 1 is the file specification of the old file.

string 2 is the new file specification.

On VMS-Compatibility Mode:

- You must have write access to the directory of the target file.
- The specified files must reside on the same physical device.
- The PSECT $$FSR2 must be in the root segment of the task. You can: (1) rework the ODL file to force $$FSR2 into the root, or (2) include NAME AS in the root.

The NAME AS statement renames the file without changing its contents. For open files, the new name takes effect when the file is closed.

8.2.3 SLEEP Statement

The SLEEP statement suspends program execution for a specified time. The format is:

```
SLEEP num-exp%
```

where:

num-exp% is the number of seconds that execution is suspended.

By enabling CTRL/C trapping (CTRLC function), you can awaken the job before the time period expires.

8.2.4 KILL Statement

The KILL statement is invalid on an open file. You must close the file before deleting it.

8.3 Compiler Commands

When performing batch I/O, you can use the LOCK/ECHO and LOCK/NOECHO commands to enable and disable the display of characters in your BASIC command file in the batch stream. If you do not want the BASIC commands displayed, type LOCK/NOECHO.

Specifying LOCK/ECHO for your terminal displays all compiler commands twice: once for the terminal echo, and a second time when the compiler reprints the command line.

8.4 Restrictions

The following sections describe system specific restrictions for BASIC-PLUS-2 running on VMS-Compatibility Mode.

8.4.1 Invalid Compiler Commands

VMS-Compatibility Mode does not support disk resident libraries. Therefore, you cannot use the BRLRES, LIBRARY, or RMSRES commands.

8.4.2 File Sharing

RMS-11 running in VMS-Compatibility Mode does not let you share files for writing. Therefore, two users cannot write at the same time, nor can one write while the other reads.

Chapter 9
BASIC–PLUS–2 on RSX–11M PLUS

This chapter describes system-specific usage of BASIC–PLUS–2 on RSX–11M PLUS. This includes:

- Invoking the compiler.
- Using the NAME AS and SLEEP statements.
- Restrictions on use.

9.1 Invoking the Compiler

To invoke the BASIC–PLUS–2 compiler, type:

```
RUN $BASIC2
```

Depending on how your system was installed, you can also invoke the compiler with the concise command "BP2". See your system manager for the concise command form used on your system.

After you invoke the compiler, BASIC–PLUS–2 prints an identifying line and awaits your input. You can then create BASIC source programs and object modules (see Chapter 1).

9.2 BASIC–PLUS–2 Statements

This section describes the implementation differences for the NAME AS and SLEEP statements.

9.2.1 NAME AS Statement

The NAME AS statement enables you to rename an existing file. It has the format:

NAME string 1 AS string 2

where:

string 1 is the file specification of the old file.

string 2 is the new file specification.

On RSX-11M PLUS:

- You must have write access to the directory of the target file.
- The specified files must reside on the same physical device.
- The PSECT $$FSR2 must be in the root segment of the task. You can: (1) rework the ODL file to force $$FSR2 into the root, or (2) include NAME AS in the root.

The NAME AS statement renames the file without changing its contents. For open files, the new name takes effect when the file is closed.

9.2.2 SLEEP Statement

The SLEEP statement suspends program execution for a specified time. The format is:

SLEEP num-exp%

where:

num-exp% is the number of seconds that execution is suspended.

By enabling CTRL/C trapping (CTRLC function), you can awaken the job before the time period expires.

9.3 Restrictions

Under RSX-11M PLUS, the BASIC library (BASIC2) cannot be in supervisor mode. All BASIC libraries run out of user mode.

Chapter 10
BASIC–PLUS–2 on TRAX

This chapter describes the use of BASIC–PLUS–2 on the TRAX operating system. This includes:

- Separate application and support environments
- Invoking the compiler
- Using the CHAIN, NAME AS, and SLEEP statements
- Restrictions on use of BASIC–PLUS–2 in the TRAX support environment

10.1 TRAX Environments

Unlike RSX–11M, IAS, or VMS, TRAX has two programming environments: application and support. These environments offer different resources and restrictions.

10.1.1 Application Environment

TRAX application environment programs perform common business processing functions, such as data base inquiry and update, input validation, and mathematical calculation. Programming for the application environment, however, requires a different approach than conventional programming. You write the code for each transaction as a series of short, independent subroutines, called transaction step tasks (TST). Each TST performs only part of the transaction.

The TRAX application environment offers these special features for BASIC–PLUS–2 TSTs:

- TST and TSTEND statements to define transaction step tasks
- MSGMAP and WRKMAP statements to describe the exchange message and transaction workspace data

- LOCK, UNLOCK, and FREE statements to enable and disable record-locking
- Library routines such as PRCEED and GETRAN, available through CALL BY REF statements

The application environment imposes these restrictions on BASIC–PLUS–2 TSTs:

- They cannot use terminal I/O statements such as PRINT and INPUT.
- They cannot access virtual files.
- They cannot perform magnetic tape operations.

See the *TRAX Application Programmer's Guide* for detailed information on the programming techniques, resources, and restrictions for BASIC–PLUS–2 TSTs.

10.1.2 Support Environment

The TRAX support environment is primarily used for system management, batch processing, and program development. You can also run programs in this environment.

In general, BASIC–PLUS–2 programs written for the support environment should conform to the rules and syntax explained in Chapters 1 through 5. The remainder of this chapter explains additions and exceptions specific to the TRAX support environment.

10.2 Invoking the Compiler

To invoke the BASIC–PLUS–2 compiler from a TRAX support environment terminal, type:

```
> RUN $BP2
```

or

```
> BASIC
```

BASIC–PLUS–2 prints an identifying line and returns the prompt "BASIC2." You can then create BASIC source programs and object modules (see Chapter 1).

10.3 BASIC–PLUS–2 Statements

This section describes how to use the CHAIN, NAME AS, and SLEEP statements in TRAX support environment programs.

NOTE

Do not use these BASIC–PLUS–2 statements in TRAX application environment TSTs.

10.3.1 CHAIN Statement

The CHAIN statement transfers control from the current program to the first line of another program. The format is:

```
CHAIN "task"
```

where:

task is the name of the program to which you wish to chain. The name must be a string expression of one to six alphanumeric characters.

BASIC implements the CHAIN statement with an RQST$ system directive. This directive requires:

- Task image format. You can chain to an executable task only.
- A unique copy of the task. You cannot have more than one running copy of the task you chain to.
- An installed task. You must install the task to which you will chain before running the chaining program. Consult your system manager for information on the privileges this requires.

10.3.2 NAME AS Statement

The NAME AS statement lets you rename an existing file. It does not change the file's contents. The format is:

```
NAME "string1" AS "string2"
```

where:

string1 is the file specification of the old file

string2 is the new file specification

TRAX requires that you have write access to the directory of the file you are renaming. In addition, the old and new files must reside on the same physical device. For open files, the new name takes effect only when you close the file.

Either the program segment containing the NAME AS statement or the $$FSR2 routine from the BASIC–PLUS–2 library must be in the non-overlayable root segment of the task. See Chapter 4 for more information on program segmentation.

10.3.3 SLEEP Statement

The SLEEP statement suspends program execution for a specified length of time. The format is:

SLEEP num-exp%

where:

num-exp% is the number of seconds that execution is suspended.

By enabling CTRL/C trapping (CTRLC function), you can awaken the job before the time period ends.

10.4 Restrictions

The following sections describe restrictions specific to BASIC–PLUS–2 programs written for the TRAX support environment.

10.4.1 Compiler Commands

TRAX does not supply the RMS and BASIC–PLUS–2 memory resident libraries for support environment programs. Therefore, TRAX does not support the RMSRES, LIBRARY, and BRLRES compiler commands.

10.4.2 Task-Building

The TKB command is not available on TRAX. To link your program, use either:

```
> LINK @filespec
```

or

```
> LINK/BASIC filespec
```

where:

filespec is the indirect command file created by the BUILD command or text editor.

See the *TRAX Linker Reference Manual* and the *TRAX Support Environment User's Guide* for more information.

10.4.3 TRANSLATOR Utility

The BASIC–PLUS–2 Translator Utility is not available on TRAX.

Appendix A
BASIC–PLUS–2 Language Elements

This appendix summarizes the program elements, commands, statements, operators, and functions supported by BASIC–PLUS–2. More information on language elements is available in the *BASIC–PLUS–2 Language Reference Manual.*

A.1 Program Elements

BASIC–PLUS–2 programs contain:

1. Arrays

   An array is an ordered arrangement of elements (subscripted variables) in one or two dimensions. You specify an array with a floating-point, integer, or string variable followed by integer subscripts in the range zero to 32767. Enclose subscripts in parentheses. Non-integer subscripts are truncated to an integer value. A single subscript indicates a 1–dimensional array, or list; two subscripts, separated by commas, indicate a 2–dimensional array, or matrix.

   You should explicitly initialize all variables in virtual arrays at the start of your program.

2. Backslash Statement Separators

   The backslash statement separator (\) separates statements in a multi-statement line.

3. Characters

   BASIC–PLUS–2 accepts the full ASCII character set (see Appendix D). It ignores null characters and accepts non-printing, non-control characters in string literals, but generates warning messages otherwise. BASIC changes all lower-case letters to upper case, except those in string literals.

4. Comments

   Comments begin with an exclamation point (!) and end with an exclamation point or a line terminator. You can insert comments before, in, or between all statements except the DATA statement. Comments have no effect on execution speed or program size.

5. Constants

   BASIC-PLUS-2 accepts three types of constants: floating-point, integer, and string. Floating-point constants are decimal numbers in the range 1E-38 to 1E38. Integer constants are also decimals in the range -32767 to +32767, but they end with a percent sign. String constants are alphanumeric characters delimited by matched pairs of single or double quotation marks. Quoted strings can contain from zero to 255 characters.

6. Continued Lines

   Program lines continue to the next line if they end with an ampersand (&) followed by a line terminator.

7. Expressions

   Expressions contain constants, variables, or functions separated by operators.

8. Functions

   Functions perform a series of numeric or string operations on the arguments you specify and return the result to your program. Functions are multi-character names followed by optional parentheses. The parentheses contain one-to-eight function arguments separated by commas. A null argument is not allowed. User-defined functions follow this general format; however, the function name begins with FN followed by 1-to-30 letters, digits, or periods.

9. Line Length

   A physical line can contain up to 256 characters, but continuations can logically extend the line. The line length is restricted only by the maximum program size.

10. Line Numbers

    All program lines except continuation lines need line numbers. BASIC-PLUS-2 line numbers are positive integers in the range 1 to 32767. Numbers outside this range, fractional line numbers, line numbers with embedded spaces or line numbers with percent signs generate errors. Leading zeroes are ignored.

11. Line Terminators

    A carriage return/line feed or an escape key (ESC) ends a program line.

12. Operators

    BASIC-PLUS-2 accepts arithmetic, relational, and logical operators. See Tables A-1 through A-3 at the end of this appendix.

13. Variables

BASIC-PLUS-2 accepts three types of variables: floating-point, integer, and string.

- Floating-point variables contain a single letter, followed by up to 29 optional letters, digits, and periods.
- Integer variables also contain a single letter, followed by up to 29 optional letters, digits, and periods, followed by a percent sign. If a percent sign is not specified, the variable is considered floating-point.
- String variables contain a single letter, followed by up to 29 optional letters, digits, and periods, followed by a dollar sign.
- You can use any alphanumeric combination except keywords for a variable name. Using keywords generates compilation errors. Variables are initialized to zero or a null string at the start of program execution.

  Variable names cannot start with FN unless defining or calling a function.

A.2 Commands

Commands allow you to perform operations on your program. They do not need line numbers. You can type them directly to BASIC along with any valid arguments.

The following is a short description of the BASIC commands, including their format and use.

Command	Use
APPEND filespec	merges a previously saved source program (filespec) with the program in memory.
BRLRES filespec	allows you to specify the BASICS shared library or a user-created shared library to be linked to your program during task building. User names need full file specifications.
BUILD filespec/sw	generates a command file from specified object modules. This file contains all of the task builder command input needed to create a task image and memory allocation map.
COMPILE filespec/sw	converts the current program. You can add switches to this command to specify the form of the output. If you specify a file name, the program is compiled under that name.
DELETE line number(s)	erases specified lines from the current program.
DSKLIB filespec	allows you to select BASIC2, BP2COM, or a user-created disk library to be linked to your program during task building.
EXIT	ends access to the BASIC-PLUS-2 Compiler and returns you to the default run-time system.
HISEG	allows you to select one of two BASIC-PLUS-2 run-time systems. RSTS/E only.
IDENTIFY	prints a header that identifies the BASIC-PLUS-2 Compiler.

Command	Use
LIBR	associates the task with a resident shareable library. IAS, RSX, TRAX, and VAX-C only.
LIST[NH]line number(s)	prints a copy of all (or part) of the current program.
LOCK/sw	sets BUILD and COMPILE switch specifications as defaults.
NEW filename	clears the user area of memory for the creation of a program. If you specify a file name, the new program is assigned that name.
ODLRMS filespec	allows you to select an RMS overlay description language (ODL) file when you build the program.
OLD filename	brings a program from disk into memory.
RENAME filename	changes the name of the program in memory to the specified name.
REPLACE filespec	saves the current program by overriding any file with that name.
RMSRES filespec	allows you to specify the RMS resident library that will be linked to your program to supply code for RMS file and record operations.
RUN	executes a specified program.
SAVE filespec	stores the current program (as source code) under the current name unless another name is specified.
SCALE val	sets the scale factor to a specified integer value or prints the current value if none is specified. The range of val is zero to 6.
SEQUENCE	enters program line numbers beginning at a number you specify. You can also specify the increments between line numbers. The default starting point is 100 and the default increment is 10.
SHOW	prints the current compiler defaults on the terminal.
UNSAVE filespec	deletes a specified file.

A.3 Statements

Statements are used in program lines. They allow you to assign values, input data, transfer program control, and so forth. Each statement in the following section includes formatting rules, a sample program line, and an explanation of use.

CALL

CALL name [(actual arguments)]

```
200 CALL SUB1 (A,B)
```

The CALL statement transfers control to a specified subprogram, transfers parameters, and maintains the status of the calling program. Parameters contained in the argument list must agree in type and number with parameters in the corresponding SUB statement.

CALL BY REF

CALL name BY REF (variable or array)

```
150 CALL SUB2 BY REF (ARRAY1(,),B%)
```

CALL BY REF allows you to transfer variables and entire arrays to a subprogram. These values are used in computations, and the results are returned to the main program.

CHAIN

CHAIN string [LINE line number]

```
15   CHAIN "SEE" LINE 70
```

The CHAIN statement transfers control to a specified program. If no line number is indicated, execution starts at the beginning of the program.

Chaining to a specific line number is a RSTS/E only feature.

CHANGE

CHANGE list TO string variable
CHANGE string expression TO list

```
25   CHANGE A TO A$
```

The CHANGE statement converts a list of integers into a character string, and vice versa; it truncates real numbers. The length of the string depends on the value found in element zero of the list.

CLOSE

CLOSE [#] expression(s)

```
150  CLOSE #6%, 8%
```

The CLOSE statement writes data from active buffers to your terminal or to the file. It then ends I/O to the device.

COMMON

COMMON (name) element(s)

```
60   COMMON (RESERV)A%,B$
```

The COMMON statement defines a named, shared area of storage. This area stores values that may be read or changed by any other program module.

DATA

DATA constant(s)

```
50   DATA 4.3, "ABC", 18, 7.9, 'XYZ'
```

The DATA statement allows you to supply data to your program without waiting for input prompts. An accompanying READ statement instructs the program when to access this data. A DATA statement must be the only statement on a line. If you specify more than one data item, you must separate them with commas.

DEF (single-line)

DEF FNa [(b1,b2,b3,...b8)] = expression

where:

a	is 1 to 30 letters, digits, or periods.
(b1,b2,b3,...b8)	are function arguments that can be integer, floating point, or string variables. A function can have from zero to eight arguments. If a function has no arguments, the parentheses must also be omitted.

expression	can contain any dummy variable. The expression is evaluated every time the function is used.

```
10 DEF FNX (A,B) = A * B
```

Make sure the expression is the same data type (string or numeric) as the function name. If the expression is floating point and the function name is integer, or vice-versa, the expression will be changed to the type specified by the function name.

DEF (multi-line)

DEF FNa [(b1,b2,b3,...b8)]

where:

a	represents 1 to 30 letters, digits, or periods followed by an optional percent sign (%) for integers or dollar sign ($) for string function values.
(b1,b2,b3,...b8)	can be zero to eight dummy arguments. If no arguments are used, the parentheses must also be omitted.

```
10 DEF FNX% (A,B)
20 IF A > B THEN C = 3.4 ELSE C = 0
30 REM C IS ASSIGNED A VALUE OF 3.4 IF A > B
40 FNX% = A * B + C
50 FNEND
```

Single and multi-line DEF statements have similar formats. However, multi-line DEFs do not set the function equal to an expression on the first line. Instead, the function is set equal later in the definition (in this example, at line 40). If the function is not set equal to an expression, the function value is zero or a null string. All multi-line DEFs end with FNEND.

DEF*

DEF* FNa [(b1,b2,b3,...b8)]

```
40 DEF* FNC (X,Y,Z)
```

DEF* indicates the BASIC-PLUS argument passing method. You can use it in place of any DEF, making all BASIC-PLUS-2 functions acceptable to BASIC-PLUS.

DELETE

DELETE #num-exp%

```
60 DELETE #5%
```

The DELETE operation erases a record from a relative or indexed file.

DIM[ENSION]

DIM array name (subscripted variable(s))

```
30 DIM B(2,3)
```

The DIM statement reserves storage for arrays. The size of the reserved storage depends on the subscripts. One subscript dimensions a list; two subscripts dimension a matrix.

DIM #

DIM #num-exp%, array name (variable(s)) [=integer]

```
50 DIM #2%, A(10,15), B(50)
```

The DIM # statement: (1) declares a virtual array, (2) specifies how many dimensions the array has, and (3) specifies the maximum value of each subscript.

Line 50 allocates space for two arrays on the file associated with logical number 2. Because no integer value was specified, the default string storage length is 16 bytes. You cannot use DIM as part of a conditional expression.

END

END

```
100 END
```

The END statement stops program execution and closes all files. END must be the last statement in a program module.

FIELD

FIELD #num-exp%, expression AS string variable
[,expression AS string variable...]

```
75 FIELD #2%, 10% AS A$, 20% AS B$, 3% AS F$
```

The FIELD statement dynamically associates string names with all or part of an I/O buffer. FIELD statements do not move data; they permit direct access to sections of the I/O buffer by means of string variables. FIELD is an executable statement, not a compiler directive.

FIND

FIND #num-exp1% [,RECORD num-exp2]

or

FIND #num-exp1%, KEY #num-exp3%

$$\left\{\begin{matrix}\text{GT}\\\text{GE}\\\text{EQ}\end{matrix}\right\}\text{string exp}$$

```
40 FIND #7%
```

```
40 FIND #7%, RECORD 25
```

```
40 FIND #7%, KEY#2% GE "JONES"
```

FIND locates a record in the specified file. For sequential FINDs, BASIC starts at the beginning of the file and locates successive records. Relative files permit random FINDs with the specification of a record number. Indexed files permit random FINDs with the specification of a key value.

FNEND

FNEND

```
40 FNEND
```

The FNEND statement causes an exit from a multi-line DEF and signals the function's logical and physical end.

FNEXIT

FNEXIT

```
70 FNEXIT
```

The FNEXIT statement permits early exit from a multi-line DEF.

FOR

FOR variable = num-exp1% TO num-exp2% [STEP num-exp3%]

```
25 FOR I = 1 TO 5 STEP 2
```

The FOR statement starts and controls a loop. You must use a simple numeric variable after the FOR, and the same variable must be used in the accompanying NEXT statement. The first numeric expression is the initial loop value; the second is the terminating loop value. The optional STEP expression is the loop increment; +1 is the default. Do not transfer into the middle of a loop.

FOR (conditional)

FOR variable = num-exp1%[STEP num-exp2%]

$$\left\{\begin{matrix}\text{WHILE}\\\text{UNTIL}\end{matrix}\right\}\text{condition}$$

```
80 FOR I = 1 UNTIL I > 10

80 FOR I = 1 WHILE I < = 25
```

A conditional FOR loop ends when the WHILE clause is false or the UNTIL clause is true.

FREE

FREE #num-exp%

```
40 FREE #1%
```

The FREE statement unlocks all records in a file. TRAX only.

GET

GET #num-exp1% [,RECORD num-exp2]

or

GET #num-exp1%, KEY #num-exp3% $\left\{\begin{matrix}\text{GE}\\\text{GT}\\\text{EQ}\end{matrix}\right\}$ string exp

```
50 GET #5%

50 GET #5%, RECORD 8%

50 GET #5%, KEY #3% EQ "HT1-544"
```

GET reads records in the specified file. For sequential GETs, BASIC starts at the beginning of the file and reads successive records. Relative files permit random GETs with the specification of a record number. Indexed files permit random GETs with the specification of a key value.

GOSUB (GO SUB)

GOSUB line number

```
25 GOSUB 120
```

The GOSUB statement transfers control to a subroutine that begins at the specified line number.

GOTO (GO TO)

GOTO line number

```
40 GOTO 85
```

The GOTO statement unconditionally transfers control to a specified line number.

IF

IF cond-exp THEN $\left\{\begin{matrix}\text{statement}\\\text{line number}\end{matrix}\right\}$ $\left[\text{ELSE } \left\{\begin{matrix}\text{statement}\\\text{line number}\end{matrix}\right\}\right]$

or

IF cond-exp GOTO line number

```
25 IF A = 0 THEN PRINT "A EQUALS 0"

25 IF A = 0 THEN PRINT "A EQUALS 0" ELSE 330
```

The IF statement allows branches in a program. It can also execute most statements, with the exception of: DIM, REM, DATA, END, DEF, FNEND, MAP and SUB.

IFEND

IFEND #num-exp%

THEN { line number / restricted statement }

or

IFEND #num-exp% GOTO line number

```
120 IFEND #3% THEN PRINT "DATA ENTRY COMPLETE"
```

The IFEND statement executes a restricted statement or transfers control to a specified line number at the end of the file. DECsystem-20 Only

IFMORE

IFMORE #num-exp%

THEN { line number / restricted statement }

or

IFMORE #num-exp% GOTO line number

```
40 IFMORE #1% THEN CLOSE #1%
```

```
40 IFMORE #1% GOTO 200
```

IFMORE executes the restricted statement or transfers control to the specified line number if there are more records in the file. DECsystem-20 Only

IMAGE

: unquoted string

or

IMAGE unquoted string

```
10: ###.##
20 PRINT USING 10, 234.9783
```

```
70 IMAGE ++ 'LLL ++ 'CCCCCC
80 PRINT USING 20, A$,B$
```

IMAGE formats numeric and string data for output. It is non-executable without PRINT USING. DECsystem-20 Only

INPUT

INPUT ["string constant" { , / ; }] variable(s)

```
25 INPUT A,B,C%
```

The INPUT statement prompts for program data with a question mark and an optional string expression you provide.

INPUT #

INPUT #num-exp%, variable

```
25 INPUT #6%, A
```

The INPUT # statement reads a value from a terminal format file and assigns it to a program variable.

INPUT LINE

INPUT LINE ["string constant",] string variable

```
15 INPUT LINE A$
```

The INPUT LINE statement permits you to input a character string to a specified variable. The line terminator is included. The optional string constant prints as part of the prompt for data.

INPUT LINE #

INPUT LINE #num-exp%, string variable

```
10 INPUT LINE #4% , A$
```

The INPUT LINE # statement reads a string value from a terminal format file and assigns that value, including the line terminator, to a program variable. BASIC does not check the syntax of file contents.

KILL

KILL string expression

```
10 KILL "SALARY.DAT"
```

The KILL statement deletes the specified file. KILL does not take effect until all users have completed I/O and the file is closed.

LET

[LET] variable(s) = expression

```
10 LET A = 65
```

The LET statement assigns constants and expressions to variables. The keyword LET is optional.

LINPUT

LINPUT ["string constant",] string variable

```
40 LINPUT NEXT.LINE$
```

The LINPUT statement permits you to input a character string to a specified variable. The line terminator is not included as part of the string. The optional string constant is printed as part of the prompt for data.

LINPUT #

LINPUT #num-exp%, string variable

```
80 LINPUT #1% , EXAMPLE$
```

The LINPUT # statement reads a string value from a terminal format file and assigns that value to a string variable. The line terminator is not included with the string. BASIC does not check line syntax.

LOCK

GET #num-exp% [,RECORD num-exp] [,LOCK]

or

GET #num-exp1% [,KEY #num-exp2%] $\left\{\begin{matrix}\text{GT}\\\text{GE}\\\text{EQ}\end{matrix}\right\}$ string-exp] [,LOCK]

```
390 GET #4% , RECORD 77% , LOCK

460 GET #2% , KEY #0% GT "BLACKWELL" , LOCK
```

TRAX only.

LOCK makes the record you GET unavailable to other users. If you do not specify LOCK, the record you GET remains unlocked. TRAX only.

LSET

LSET string variable(s) = string expression

```
10 LSET A$,B$ = X$ + Y$
```

The LSET statement assigns string expressions to string variables. The data is left-justified, and the length of the target string is not changed.

MAP

MAP (name) element(s)

```
10 MAP (BUFF1) A%, B$, C, D$ = 25%
```

The MAP statement defines the data fields in the record buffer and associates them with program variables. A GET moves data from the file to the buffer so you can access MAP statement variables. A PUT writes the buffer to the file. You cannot specify a MAP as part of a conditional expression.

MARGIN

MARGIN [#num-exp%] { , / : } num-exp%

```
90 MARGIN #2%, 80%
```

MARGIN modifies margin settings for terminal format files. The default is the user's current terminal width. DECsystem-20 Only

MARGIN ALL

MARGIN ALL { , / : } num-exp%

```
30 MARGIN ALL, 75%
```

MARGIN ALL specifies the same margin width for all currently opened terminal format files. DECsystem-20 Only

MAT INPUT

MAT INPUT array(s)

```
50 MAT INPUT A
```

The MAT INPUT statement assigns data you input to the elements of a specified array. Elements are stored in row order as they are typed.

MAT INPUT #

MAT INPUT #num-exp%, array name

```
100 MAT INPUT #2%, ARN
```

The MAT INPUT # statement reads values from a terminal format file and assigns them to a specified array. The elements are stored in the destination array in row order.

MAT LINPUT

MAT LINPUT string array name

```
300 MAT LINPUT VECTOR,NAME$
```

The MAT LINPUT statement assigns data you input to elements of a string array.

MAT LINPUT #

MAT LINPUT #num-exp% string array name

```
90 MAT LINPUT #1%, A$
```

The MAT LINPUT statement reads string data from a terminal format file and assigns it to elements of a string array. MAT LINPUT # does not include the string's line terminator as part of the array element.

MAT PRINT

MAT PRINT array(s)

```
120 MAT PRINT A;
```

The MAT PRINT statement prints all elements of a specified array.

MAT PRINT #

MAT PRINT #num-exp%, array name

```
60 MAT PRINT #5%, TESTAR
```

MAT PRINT # prints the contents of an array to a terminal format file.

MAT READ

MAT READ array(s)

```
50 MAT READ B,C
```

The MAT READ statement reads data statement values into elements of a 1- or 2-dimensional array.

MATRIX ASSIGNMENT

MAT array name = array name

```
15 MAT A = B
```

In matrix assignment, the MAT statement sets each entry of Array A equal to the corresponding entry of Array B. A is redimensioned to the size of B.

MOVE

MOVE {FROM / TO} #num-exp%, I/O list

```
15 MOVE TO #5, A$, B, C(), FILL%
```

The MOVE statement moves data in a record to or from the variables you specify in the I/O list.

MSGMAP in TSTs

MSGMAP variable list

The MSGMAP statement is used in TSTs to describe the exchange message area. See the TRAX documentation for rules on formatting the list of variables. TRAX only.

NAME AS

NAME string1 AS string2

```
15 NAME "MONEY" AS "ACCNTS"
```

NAME AS renames a file without changing its contents. IF the file is open, NAME AS assigns the new name when the file is closed.

NEXT

NEXT variable

```
15 NEXT I%
```

The NEXT statement ends a FOR, WHILE, or UNTIL loop. The variable must correspond to the variable in the accompanying FOR statement.

NODATA

NODATA [#num-exp%], line number

```
50 NODATA , 110
```

NODATA transfers control to the specified line number if all DATA for a program or subprogram is exhausted. If a channel number is specified (num-exp%), the statement is the same as:

IFEND #num-exp% THEN line number

DECsystem-20 Only

ON ERROR

ON ERROR GOTO line number

or

ON ERROR GO BACK

```
25 ON ERROR GOTO 50
```

```
25 ON ERROR GO BACK
```

The ON ERROR GOTO statement transfers program control to a specified line that contains an error-handling routine. ON ERROR GOTO should be the first line in the program.

The ON ERROR GO BACK statement allows a subprogram containing an error to return to the calling program for error handling.

ON GO SUB
(ON GOSUB)

ON num-exp% GOSUB line number(s)

```
50 ON A + B GOSUB 80 , 95 , 100
```

The ON GOSUB statement conditionally transfers program control to subroutines or to entry points in subroutines.

ON GOTO

```
ON num-exp% {GOTO } line number(s)
            {THEN }
```

```
20 ON J% GOTO 85, 90, 95, 100
```

The ON GOTO statement transfers program control to a location that depends on the value of num-exp%.

ON THEN

See ON GOTO

OPEN

```
OPEN filespec-exp {[FOR OUTPUT]} AS FILE [#] num-exp%
                  {[FOR INPUT] }
[,ORGANIZATION] { SEQUENTIAL } { [FIXED]    }
                { RELATIVE   } { [VARIABLE] }
                { INDEXED    } { [STREAM]   }
                { UNDEFINED  } {            }
                { VIRTUAL    } {            }

{[,ACCESS  READ]   }
{          WRITE]  }
{          MODIFY] }
{          SCRATCH]}
{          APPEND] }

{[,ALLOW   NONE]   }
{          READ]   }
{          WRITE]  }
{          MODIFY] }

[,MAP mapname ]
[,MODE]
[,RECORDSIZE num-exp]
[,BLOCKSIZE num-exp%]
[,FILESIZE num-exp%]
[,SPAN]
[,NOSPAN]
[,CONTIGUOUS]
[,TEMPORARY]
[,BUCKETSIZE num-exp%]
[,CONNECT]
[,NOREWIND]
[,WINDOWSIZE num-exp%]
[,CLUSTERSIZE num-exp]
[,BUFFERSIZE num-exp%]
,PRIMARY [KEY]name  [DUPLICATES]
                    [NODUPLICATES]

[,ALTERNATE [KEY]name  [DUPLICATES [CHANGES]]
    [NODUPLICATES [NOCHANGES]]
```

```
10 OPEN "FIL4.DAT" FOR INPUT AS FILE #4%
```

The OPEN statement enables you to create a new file or access an existing file, and specify that file's attributes. CLUSTERSIZE is available only on RSTS/E. WINDOWSIZE is available on all other systems.

PAGE

PAGE [#num-exp%] { , | : } num-exp%

```
15 PAGE #8%, 58%
```

PAGE sets a page size in lines for terminal format files. This page size (num-exp%) remains in effect until:

- The page size is set again by the PAGE statement
- The file is set to nopage mode by the NOPAGE statement
- Execution ends
- The file is closed

When program execution ends, the terminal returns to its previous default page setting. DECsystem-20 Only

NOPAGE

NOPAGE #num-exp%

```
60 NOPAGE #8%
```

NOPAGE stops automatic paging of the specified terminal format file. DECsystem-20 Only

PRINT

PRINT [expression(s)]

```
30 PRINT A + B
```

PRINT outputs specified data to the terminal. The expression list can be expressions, variables, or quoted strings separated by a comma or a semicolon. Commas cause output to terminal print zones; semicolons suppress spacing between elements.

PRINT #

PRINT #num-exp%, list

```
60 PRINT #6%, A$
```

The PRINT # statement writes data to a specified terminal format file.

PRINT USING

PRINT USING format string, list

```
10 PRINT USING "****.##", A,B,C
```

The PRINT USING statement formats data for output. See the *BASIC-PLUS-2 Language Reference Manual.*

PRINT # USING

PRINT #num-exp% USING format string, list

```
90 PRINT #4% USING "# # # # #", 4, 2, 8, 23,
```

PRINT # USING writes records to terminal format files in a format you specify.

PUT

PUT #num-exp1% [,RECORD num-exp2] [,COUNT num-exp3]

```
15 PUT #7%

15 PUT #7%, RECORD 15%

15 PUT #7%, RECORD 20%, COUNT 80%
```

The PUT statement writes a record from the record buffer to a specified file. Sequential files allow PUT operations only at the end of the file. The RECORD clause is used for random PUTs to relative or block I/O files. The COUNT clause redefines the size of the record.

RANDOM[IZE]

RANDOMIZE

```
10 RANDOMIZE
```

The RANDOMIZE statement changes the starting point of the RND function to a new and unpredictable location.

READ

READ variable(s)

```
75 READ A,B%,C$,D(5)
```

The READ statement directs BASIC to input values from a DATA statement.

REM[ARK]

REM comment

```
30 REM THIS IS A COMMENT
```

The REM statement documents a program with user-written comments. It has no effect on program execution.

RESTORE [#]

RESTORE #num-exp% [,KEY num-exp%]

```
30 RESTORE #3%

80 RESTORE #3%, KEY 8%
```

The RESTORE # statement resets the specified file to its first record. The RESTORE # statement with the KEY clause resets an indexed file to the beginning of the specified key. RESTORE without a file expression restores the data in a DATA statement.

RESUME

RESUME [line number]

```
50 RESUME 35
```

The RESUME statement is the last statement in an error-handling routine. It shifts control from that routine to a specified line number in the program. If no line number is specified, control shifts back to the point of error generation.

RETURN

RETURN

```
60 RETURN
```

The RETURN statement is the last statement in a subroutine. It shifts control to the statement following the last executed GOSUB statement.

RSET

RSET string variable(s) = string expression

```
10 RSET A$,B$ = X$ + Y$
```

The RSET statement assigns new values to string variables. The new data is right-justified and the target string's length is not changed.

SCRATCH

SCRATCH #num-exp%

```
25 SCRATCH #6%
```

The SCRATCH statement deletes a sequential file from the current record to the end-of-file. You can use SCRATCH only if the file was opened with ACCESS SCRATCH.

SLEEP

SLEEP num-exp%

```
10 SLEEP 30%
```

The SLEEP statement causes a temporary halt in execution. The length of delay (in seconds) depends on the value of the expression.

STOP

STOP

```
110 STOP
```

The STOP statement halts program execution and prints a message indicating the location of the halt. STOP does not close opened files.

SUB

SUB name [(formal argument(s))]

```
40 SUB TEST (A,B%)
```

The SUB statement starts a subprogram and defines the type and number of subprogram parameters.

SUBEND

SUBEND

```
25 SUBEND
```

The SUBEND statement ends a subprogram and returns control to the calling program.

SUBEXIT

SUBEXIT

```
899 SUBEXIT
```

The SUBEXIT permits early exit from a subprogram. It is equivalent to GO TO a SUBEND statement.

TST, TSTEND

TST TSTEP(ARG1$, ARG2$)
TSTEND

```
40 TST TSTEP(EXPA$,EXPB$)

70 TSTEND
```

The TST statement is the first statement in a TST module. The module ends with TSTEND. See the TRAX documentation for more information on TST and TSTEND. TRAX only.

UNLESS

statement UNLESS condition

```
15 PRINT A UNLESS A = 0
```

A statement with an UNLESS modifier will execute only if the condition is false. The UNLESS modifier simplifies the negation of a logical condition.

UNLOCK

UNLOCK #num-exp%

```
30 UNLOCK #1%
```

The UNLOCK statement unlocks all buckets in a stream designated by a channel number.

UNTIL

UNTIL conditional exp

```
50 FOR I = 1 UNTIL I = A \ B * 6

60 PRINT I

70 NEXT
```

The UNTIL modifier permits execution of a loop until the condition is true. An accompanying NEXT statement is required.

UPDATE

UPDATE #num-exp% [,COUNT num-exp%]

```
50 UPDATE #1%
```

The UPDATE statement replaces a record in the file. For sequential files, the new record must be the same size as the old one.

When the file permits duplicate primary keys, the new record must be the same length as the old one. When the program does not permit duplicate primary keys, the new record:

- Can be no longer than the maximum record size
- Must include at least the primary key field. If the new record omits one of the old record's alternate key fields, the OPEN statement must specify CHANGES for that key field.

WAIT

WAIT num-exp%

```
50 WAIT 15%
```

The WAIT statement specifies the maximum number of seconds allowed for input before BASIC generates an error. A zero or null value disables the WAIT.

WHILE

WHILE conditional exp

```
70 FOR I% = 20% WHILE I% < 125%
80 PRINT I%
90 NEXT
```

The WHILE modifier sets up a loop that executes until the condition is false. WHILE must have an accompanying NEXT statement.

WRKMAP

```
100 WRKMAP list
```

WRKMAP describes the work area of a TST. See the TRAX documentation for more information. TRAX only.

A.4 Functions

This section describes the numeric and string functions available in BASIC.

Function	Usage
ABORT (N%)	causes an exit to the editor if N% = zero. If N% = 1, the program exits to an editor and the current working buffer is scratched. In either case, no READY prompt is printed on the terminal. DECsystem-20 Only
ABS(X) ABS%(X%)	returns the absolute value of X for real numbers (ABS(X)) or integers (ABS%(X%)).
ACCESS$(filespec)	returns access privilege information for a specified file. DECsystem-20 Only
ASCII(X$)	returns the decimal ASCII value of the first character of a specified string.
ATN(X)	returns the arctangent of X in radians.
BUFSIZ(N%)	returns an integer value, which is the size of the buffer in bytes. If the file channel is closed, BUFSIZ equals zero. The BUFSIZ format is: BUFSIZ(N%) where: N% equals the channel number.
CCPOS(N%)	returns the current position on the output line for the given channel number. The CCPOS format is: CCPOS(N%) where: N% is the I/O channel number. It can range from zero to 12. CCPOS(0%) prints the character position of the current output line.
CHR$(X%)	returns the character equivalent of the ASCII value X%.
CLK$	returns the time of day in the form HH:MM:SS. The hours are based on a 24–hour clock. DECsystem–20 Only
COMP%(X$,Y$)	compares two numeric strings and returns: 1 if X$ > Y$ 0 if X$ = Y$ -1 if X$ < Y$
CON	sets the elements of an array to a value of one.
COS(X)	returns the cosine of X in radians.
COUNT	specifies the number of bytes written in a PUT or UPDATE operation. The default is the maximum record size (MRS). The COUNT clause must equal the recordsize for fixed length records, and be less than or equal to the recordsize for variable length records.

Function	Usage
CTRLC	enables CTRL/C trapping.
CVT$$(string,B%)	manipulates a character string and generates a new one. CVT$$ does not change the internal format of the data. See EDIT$. RSTS/E Only
CVT$%(X$)	maps the first two characters of a string into an integer. If the string has fewer than two characters, BASIC pads it with nulls. RSTS/E compatible
CVT$F(X$)	maps the first four characters of a string into a floating point number. If the math package is double precision, the first eight characters are mapped. If the string has fewer than the required number of characters, BASIC pads it with nulls. RSTS/E compatible
CVT%$(X%)	maps an integer into a 2-character string. RSTS/E compatible
CVTF$(X)	maps a floating-point number into a four or eight character string, depending on the system's math package. RSTS/E compatible
DATE$(0%)	returns the current date.
DATE$(X%)	returns a calendar date according to the formula: (Day of year) + [(years since 1970) * 1000]
DET	returns the determinant of a matrix.
DIF$(X$,Y$)	subtracts string Y$ from string X$ and returns the difference.
ECHO(N%)	enables terminal echo of characters sent to the system from your terminal.
EDIT$(string,N%)	formats the string using the flag N%. The values for N% are shown in Table 5-1 (EDIT$ Conversions) in the *BASIC-PLUS-2 Language Reference Manual.*
ERL	returns the line number at which an error occurred.
ERN$	returns the name of the subprogram in which an error occurred.
ERR	returns the number of the run-time error your program generated.
ERT$(N%)	returns the text error message associated with a given value of N%. N% equals the error code for the current error (see Appendix C).
EXP(X)	returns the value of: e^X where: e = 2.71828, the base of natural logarithms.
FILL FILL$ FILL%	masks parts of a record or buffer to hold space for future use, or to skip over data not used in a routine. FILL items are acceptable in MAP, COMMON and MOVE TO statements.
FIX(X)	returns the value of X truncated to an integer.
FORMAT$(A,B$)	returns a numeric variable formatted according to the contents of the associated string. See PRINT USING for formatting rules.

Function	Usage
FNAME$ (filespec)	returns the file specification assigned to your channel. If you specify FNAME$ with a channel number not assigned to an opened file, BASIC returns an error message. DECsystem-20 Only
FSP$(N%)	returns a string that describes the file opened on a given channel. N% is the channel number.
FSS$(A$,B%)	performs a filename string scan on A$, starting at position B%.
IDN	sets up an identity matrix: all elements are zero except for those on the (I,I) diagonal, which are set to one.
INSTR(Z%,X$,Y$)	returns the position of substring Y$ in the main string X$, starting at position Z%.
INT(X)	returns the integral part of X. INT(X) returns the same value as FIX(X) for equal values of X, but INT(X) does not change X.
INV	generates a matrix that is the inverse of another. For example, MAT N = INV(M) makes Matrix N the inverse of Matrix M. Matrix M must be a square matrix.
LEFT$(X$,Y%) LEFT(X$,Y%)	returns a substring of X$, beginning at the leftmost position, for a total length of Y% characters.
LEN(X$)	returns the number of characters in X$.
LINO(N%)	returns line number N% to RESEQUENCE for full use of the ERL function. DECsystem-20 Only
LOG(X)	returns the natural logarithm of X. If: $e^X = Y$ then: $\ln Y = \log(e)\ Y = X$
LOG10(X)	returns the common logarithm of X. Common logarithms, unlike natural logarithms, have a base of 10, not 2.71828.
MAGTAPE(X%,Y%,Z%)	provides flexibility in non-file structured processing by permitting the program to control magtape functions such as rewind or tape density. In this function: X% is the function code (1 to 9). Y% is the integer parameter. Z% is the channel number on which the selected magnetic tape is open.
MAR%(#num-exp%) MAR(#num-exp%)	returns the margin width currently associated with the file-expression. DECsystem-20 Only
MID$(string,n1%,n2%) MID(X$,Y%,Z%)	returns a substring n2% characters long, starting at position n1% of the string.
MOD%(A, B)	returns the remainder of A/B (A mod B) in integer form. DECsystem-20 Only
MOD(A, B)	returns the remainder of A/B (A mod B) in real number form. DECsystem-20 Only
NOECHO(N%)	disables terminal echo.

Function	Usage
NUL$	sets the value of all elements in a string array to null string. NUL$ does not set row zero or column zero. DECsystem-20 Only
NUM	contains the number of rows input to a matrix. For a one dimensional array (list) NUM contains the number of elements entered.
NUM2	contains the number of elements entered in the last row of a matrix.
NUM$(N)	returns the value of N as it is printed by a PRINT statement. For example, NUM$(1.000) = (space)1(space) and NUM$(-1.000) = -1(space).
NUM1$(N)	returns N as PRINT would write it, but without spaces or E format.
ONECHR(X%)	enters single-character input mode on channel X%.
PEEK	allows a privileged user to check any word location in the monitor part of memory. The user program can check words in small or large buffers in the resident portion of the file processor, and in the low memory and tables of memory. The function does not allow a user program to check the contents of another user's program. RSTS/E only.
PI	returns a constant value: 3.14159.
PLACE$(X$,N%)	returns X$ with precision according to N%.
POS(X$,Y$,Z%)	returns the position of substring Y$ in that portion of the main string X$ that extends from position Z% to the end of the main string. See also INSTR
PPS%(X)	returns the page count on channel X. DECsystem-20 Only
PROD$(X$,Y$,N%)	returns the product of X$ and Y$, with precision depending on N%.
QUO$(X$,Y$,N%)	divides X$ by Y$ and returns the quotient, with precision depending on N%.
RAD$(X%)	converts the integer X% to its RADIX-50 equivalent.
RCTRLC	disables CTRL/C trapping.
RCTRLO(N%)	cancels the effect of typing CTRL/O on channel N%. See your *System User's Guide* for a description of the effect of CTRL/O on your system.
RECOUNT	contains the number of characters transferred by the latest input operation.
RIGHT$(X$,Y%) RIGHT(X$,Y%)	returns a substring of X$ that extends from the Yth character to the end of the string.
RND	returns a random real number between zero and 1.
SEG$(X$,Y%,Z%)	returns the substring of X$ that extends from the Yth character to the Zth character. See MID$

Function	Usage
SGN(X)	returns the following values: 1 if X is positive 0 if X is zero -1 if X is negative
SIN(X)	returns the sine of X in radians.
SPACE$(X%)	generates and returns a string X spaces long.
SPEC%(W%,X,Y%,Z%)	performs special operations on peripheral devices. For more information, see the *RSTS/E Programming Manual*. RSTS/E only.
SQR(X) SQRT(X)	returns the square root of the absolute value of X.
STATUS	returns a 16-bit variable that contains information about the last channel on which your program executed an OPEN statement. Your program can test each bit to determine the status of the channel.
STR$(N)	returns the value of N as it is printed by a PRINT statement, but without the leading and trailing blanks. See also NUM$(N)
STRING$(X%,Y%)	creates and returns a string X% characters long that represents the ASCII value of Y%. See also ASCII
SUM$(X$,Y$)	returns the sum of X$ and Y$.
SWAP%(X%)	reverses the bytes in an integer word. The low byte takes the high byte position, and vice versa.
SYS(Y%)	allows system function calls in your program to: (1) perform special I/O functions, (2) establish special characteristics for a job, (3) set terminal characteristics, and (4) cause the monitor to execute special operations. RSTS/E only.
TAB(X%)	moves the print head to the Xth position.
TAN(X)	returns the tangent of X in radians.
TIME$(X%)	returns the time as X minutes before midnight.
TIME$(0%)	returns the present time in a system-defined format.
TIME(0%)	returns the clock time in seconds since midnight.
TRM$(A$)	trims the trailing blanks from a string.
TRN	creates a new array that is the "transpose" of the original. If Matrix A has m rows and n columns, MAT C = TRN(A) will generate an array with n rows and m columns.
TYP(file-exp%, X$)	determines if the file specified by file-exp% is the same type as indicated by X$. TYP returns: +1 if file-exp% is the same as X$. 0 if the file types are not the same. -1 if the file type is invalid or the specified file does not exist.

Function	Usage
	The file type X$ can be: SEQUENTIAL RELATIVE INDEXED TERMINAL VIRTUAL DECsystem-20 Only
TYPE$(file-exp%)	identifies the current file type as one of the following: SEQUENTIAL RELATIVE INDEXED TERMINAL VIRTUAL TYPE$ returns a null string if the file type is not known. DECsystem-20 Only
USEAGE$(file-exp%)	indicates the file's usage by returning one of the following: INPUT OUTPUT I/O APPEND USEAGE$ returns a null string if the file's usage is not known. DECsystem-20 Only
USR$	returns the user I.D. for the current job. DECsystem-20 Only
VAL(X$) VAL%(X$)	computes the numeric value of the numeric string X$; X$ must be acceptable numeric input.
VPS%(X%)	returns the vertical position on channel X%. DECsystem-20 Only
XLATE(A$,B$)	translates one string to another using a translation table, B$.
ZER	initializes all elements of an array to zero. This condition is true of all arrays when first created, except those in a virtual array, MAP, or COMMON area.

Table A-1: Arithmetic Operators

Operator	Use	Meaning
^ or **	5^2 or 5**2	exponentiation
*	A * B	multiplication
/	A /B	division
+	A + B	addition, unary plus, string concatenation
-	A - B	subtraction, unary minus

Table A-2: Logical Operators

Operator	Use	Meaning
NOT	NOT A	logical negative of A.
AND	A AND B	logical product of A and B.
OR	A OR B	logical sum of A and B.
XOR	A XOR B	logical exclusive OR of A and B.
EQV	A EQV B	logical equivalence between A and B.
IMP	A IMP B	logical implication of A and B.

Table A-3: Relational Operators

Operator	Use	Meaning
=	A = B	A is equal to B.
<	A < B	A is less than B.
>	A > B	A is greater than B.
<= or =<	A < = B	A is less than or equal to B.
>= or =>	A > = B	A is greater than or equal to B.
<> or ><	A <> B	A is not equal to B.
==	A==B	A is approximately equal to B.

Note that A is approximately equal to B (A==B) if the difference between A and B is less than 10^(-6). If A$ and B$ are strings, the relation (==) is true if the contents of A$ and B$ are the same in length and composition.

Appendix B
Compile–Time Error Messages

BASIC–PLUS–2 diagnoses compile–time errors and indicates the program line that generated the error. The error message format is:

<message> at line X statement n

where:

<message>	is the text of the message.
at line X	is the location of the error.
statement n	is the statement in line X that contains the error.

Error messages contain either a percent sign (%) or a question mark (?) prefix. A percent sign is a warning; compilation can continue, but the result is not predictable. A question mark indicates a fatal error; compilation can continue, but the compiler will produce no task or object module.

The following is an alphabetized list of compilation error messages:

?Arguments don't match

FATAL – The function call arguments differ in quantity or type from those defined for the function. Check the function definition. Change the arguments or definition to conform.

?Arguments don't match in x() at line n

FATAL – The argument in a user–defined function call does not match the type (string or numeric) or number of the dummy argument defined in the DEF statement. In this message, x is the user–defined function name and n is the line number of the call. Check the program to make sure that function arguments agree with those defined in the DEF statement.

% CALL/SUB forces OBJ output

WARNING - You cannot produce a task image file from programs that contain CALL or SUB statements. You must produce object modules (COM/OBJ) and task build them. The compiler automatically generates an object module when it encounters CALLS or SUBS in a program. RSTS/E Only

?COM/MAP cannot have modifer

FATAL - You cannot add the modifiers FOR, IF, UNLESS, UNTIL, or WHILE to a COMMON or MAP statement.

?COM/MAP without list

FATAL - The program contains a COMMON or MAP statement without an accompanying variable list.

% COMMON and MAP with same name (x)

WARNING - There is a potential problem in the redefinition of a COMMON or MAP. Determine if the COMMON and MAP should overlay each other.

% Compile time variable .x redefined

WARNING - The variable .x appears in more than one .DEFINE.

% /DEB forces OBJ output

WARNING - You cannot produce a task image file from programs containing /DEBUG switches. You must produce an object module and task build the program. RSTS/E Only

?DEF with no contents

FATAL - A DEF statement is immediately followed by FNEND. BASIC expects one or more program lines to accompany the DEF.

?DEF without name

FATAL - A DEF statement has no function name. You must supply one.

% Division by zero

WARNING - You should not divide by the constant zero.

?END statement without a program

FATAL - The END statement has no accompanying program.

?END/SUBEND not last statement

FATAL - END or SUBEND must be the last statement in a program module.

% ERL overrides /NOLINE

WARNING - A program compiled with the /NOLINE switch contains an error handler that references the ERL function. The /NOLINE switch is nullified.

?ERROR n at line m in x, compiling line p

FATAL - This message indicates a severe compiler error.

n represents the value of the ERR variable.

m is the line number where the error originated in the compiler.

x is the name of the compiler module that contains the error.

p is the currently compiling program line number in the user modules.

This error causes the loss of your program, an exit from BASIC, and a return to the operating system command level. Submit a Software Performance Report to DIGITAL and include all relevant output.

?Expression too complex at line n

FATAL - You have written an expression that is too complex to compile. Rewrite the expression as two or more assignment statements and recompile.

?FNEND cannot have modifier

FATAL - You cannot add the modifiers FOR, IF, UNLESS, UNTIL, or WHILE to the FNEND statement.

?FNEND without DEF

FATAL - A FNEND statement has no preceding DEF statement. Define the function before inserting a FNEND statement in the program.

?FNEXIT while not in DEF

FATAL - A FNEXIT statement has no preceding DEF statement. Define the function before inserting a FNEXIT.

?Loops or conditional expressions nested too deep

FATAL - Internal space is exhausted for loops and conditional expressions.

?Illegal argument passing in CALL

FATAL - You are incorrectly passing an argument in a CALL statement; for example, a string array in a CALL BY REF. Check all elements for proper format.

?Illegal Assignment List

FATAL - You cannot place a non-variable on the left-hand side of the equal sign (=).

% Illegal character

WARNING - Your program contains illegal or incorrect characters. Examine the program line for correct usage of the BASIC-PLUS-2 character set.

?Illegal clause in I/O statement

FATAL - You cannot:

- Include a RECORD clause with other than FIND, GET, or PUT
- Include a COUNT clause with other than PUT or UPDATE
- Include a KEY clause with other than GET or FIND
- Include a LOCK clause with other than GET, FIND, PUT, or UPDATE

?Illegal clause in OPEN

FATAL - You have specified illegal attributes for the file type being opened. Substitute valid attributes.

?Illegal COM/MAP/SUB name

FATAL - A MAP, COMMON, or subroutine name exceeds six characters or contains a percent sign. Correct the program line.

?Illegal dummy argument

FATAL - Either:

- The same variable appears more than once in a SUB statement argument list. Assign unique variables.

  or

- A DEF statement argument is used as a parameter in a SUB statement. Select a new SUB statement parameter.

?Illegal FIELD variable

FATAL - You cannot include a FIELD variable in a COMMON, MAP, or virtual array.

?Illegal file number

FATAL - You must include a pound sign (#), number, percent sign (%) and comma(,) in your file number. Check for these elements.

?Illegal FILL Specification

FATAL - You cannot specify a length in a FILL or FILL% specification (for example, FILL% = 10%). Each FILL or FILL% specification allocates a fixed amount of space. Specify additional FILL or FILL% fields to allocate more space.

?Illegal FN redefinition

FATAL - A function can be defined only once in a program. Use a different function name for each function definition.

?Illegal KEY specification

FATAL - You cannot:

- Specify a string array element as a key in an indexed file OPEN statement

  or

- Specify CHANGES for the primary key

?Illegal loop nesting

FATAL - The program contains nested loops that overlap. Examine the program logic to make sure that all nested loops start and end correctly.

?Illegal MAP statement

FATAL - You have not named a MAP. Omitting the name is allowed for COMMONs only.

% Illegal matrix operation

WARNING - You have attempted a matrix division. The operation is treated as a MAT multiply, and the program continues.

?Illegal mode mixing

FATAL - You cannot mix string and numeric operands. Use a function to convert the data types.

% Illegal number

WARNING - You cannot:

- Specify an integer or real number outside the legal range. Legal integers are in the range -32767 to +32767. Legal floating-point numbers are in the range 1E-38 to 1E38.
- Specify a real exponent for a number in E format.

?Illegal READ statement

FATAL - You cannot use a channel number with a READ statement. Use proper syntax to access the file type.

?Illegal redefinition of COM or MAP variable x in (y)

FATAL - You have defined the variable x in: (1) more than one COMMON, or (2) more than once in a COMMON or a MAP.

?Illegal relative operator

FATAL - You have specified an invalid relative operator; for example, "<<" or ">>".

?Illegal reserved word <word>

FATAL - You have assigned a reserved word as a variable name. You must rename this variable.

?Illegal string operator

FATAL - The program contains an incorrect string operator; for example, A$=B$-C$.

?Illegal subscript

FATAL - An array reference contains a subscript of an incorrect data type.

% Inconsistent function usage in x() at line n

WARNING - You have called the function with a floating-point argument, although the corresponding dummy argument in the function definition is integer. The floating-point argument is truncated to an integer value, and the compilation continues.

?Inconsistent subscript usage

FATAL - You have referenced an array with an incorrect number of subscripts. Specify single or double subscripts as required.

?Input with no arguments

FATAL - Your input statement has specified no variable list after the channel number.

?Logical operation on non-integer quantity

FATAL - The program contains an incorrect data type in a logical operation (for example, A%=B where B must be an integer). Use consistent data types in logical operations.

% Loop will not execute

WARNING - The program contains a FOR/NEXT loop that is not executable; for example, FOR I = 1 TO 0. The program compiles correctly, but ignores the loop.

% MAP <map-name> not defined

WARNING - You have referenced a nonexistent MAP in the OPEN statement. You must define every MAP with the MAP statement.

% MAT INV forces OBJ output

WARNING - You cannot produce a task image file from programs that contain a MAT INV statement. You must produce an object module and task build the program. RSTS Only

% Matrix dimension error

WARNING - You cannot:

- Perform a MAT IDN, MAT TRN, MAT INV on a one-dimensional array (list)

  or

- Perform any MAT operation on arrays of different subscripts.

?Missing FNEND

FATAL - A multi-line DEF statement has no terminating FNEND.

?Missing NEXT

FATAL - A FOR, WHILE, or UNTIL loop has no accompanying NEXT statement.

?Missing SUBEND

FATAL - A subprogram has no corresponding SUBEND statement.

?Misspelled keyword

FATAL - A keyword in the program requires correct spelling.

?Multiply allocated variable

FATAL - You cannot define a variable in more than one COMMON, MAP, DIM, or any combination of these. Define a variable once.

?Nested FOR loops with same index

FATAL - Two or more FOR/NEXT loops cannot have the same index, as in:

```
10 FOR I = 1 TO 10

20 FOR I = 1 TO 5

30 NEXT I

40 NEXT I
```

In nested loops, each index must be unique.

?Nesting too deep at line x

FATAL – The compiler's internal storage is exhausted because there are too many nested FOR/NEXT, WHILE/NEXT, UNTIL/NEXT, or IF–THEN–ELSE constructions in the program.

?NEXT without FOR

FATAL - A NEXT statement has no preceding FOR statement.

?NEXT without WHILE/UNTIL

FATAL - A NEXT statement has no preceding WHILE or UNTIL statement.

?Numeric array has size in MOVE

FATAL - You cannot specify a string length for a numeric array in a MOVE statement.

```
50 MOVE FROM  1%, A()=3
```

?Program data space too big

FATAL - You cannot compile a program with data definitions that exceed the allowable memory space. Recompile the program as two or more object modules.

?Program too big to compile

FATAL - You cannot compile a source program that generates a program larger than the machine allows. Recompile the program as two or more object modules.

% RESUME overrides /NOLINE

WARNING - A program compiled with the /NOLINE switch contains a RESUME statement. The /NOLINE switch is nullified.

?Stack error in x, compiling line n

FATAL - This message indicates a severe compiler error. This error causes the loss of your program, an exit from BASIC, and a return to the operating system command level. Submit a Software Performance Report to DIGITAL and include all relevant output.

?SUB cannot have modifier

FATAL - You cannot add a FOR, IF, UNLESS, UNTIL, or WHILE modifier to the SUB statement.

?SUB with no contents

FATAL - There is no intervening text between SUB and SUBEND.

?SUB without name

FATAL - You have not named a SUB program.

?SUBEND cannot have modifier

FATAL - You cannot add a FOR, IF, UNLESS, UNTIL, or WHILE modifier to the SUBEND statement.

?SUBEND without SUB

FATAL - You have a SUBEND statement without a preceding SUB.

? SUBEXIT while not in SUB

FATAL - A SUBEXIT statement has no prior SUB statement.

?Syntax error

FATAL - A program line contains illegal syntax or illegal format. Correct the line to conform to BASIC-PLUS-2 syntax requirements.

?Too few arguments

FATAL - A function call contains fewer arguments than are defined for that function.

?Too many arguments

FATAL - A function call contains more arguments than are defined for that function.

?TSK OUTPUT not possible

FATAL - You cannot produce a task image (COM/TSK) when one of the following is present in the program module:

- A subprogram
- A CALL statement that references an external subprogram
- RMS I/O operations
- The /DEBUG option, when DEBUG is not in the BASIC2 HISEG
- A matrix inversion statement

Instead of a task image, you must produce object modules (COM/OBJ) and task build them. RSTS/E Only

% Unaligned COM or MAP variable x in (y)

WARNING - A string, composed of an odd number of characters and preceding a numeric variable, has caused a COMMON or MAP variable to fall on an odd address. The compiler aligns the variable to the next highest word boundary and continues compiling.

% Undefined compile time variable .x

WARNING - The variable .x has not been defined by a .DEFINE.

?Undefined function x() called at line n

FATAL - You have not defined the function x(). Make sure that user-defined functions are defined with a DEF statement.

% Undefined line number n

WARNING - A control statement directs the program to a nonexistent line (represented by n). The compiler assumes that the next highest line number is the control destination.

?Unmapped variable x in key clause at line n

FATAL - Your MAP statement does not define a KEY included in the OPEN statement.

?Unsupported feature in TST environment

FATAL - INPUT and PRINT are not allowed in TST mode. TRAX Only

?Unterminated string

FATAL - You have mixed single and double quotation marks in delimiting a string. For example: "ABC' and 'ABC" are both invalid; a correctly terminated string would be: "ABC" or 'ABC'.

?Variable or function name too long

FATAL - You cannot:

- Specify a variable that exceeds 30 characters (excluding a percent or dollar sign)

  or

- Specify a function name that exceeds 30 characters (excluding FN and a percent or dollar sign)

?Virtual array space exceeded

FATAL - You have created an array larger than the allowable area. Reduce the array dimensions.

Appendix C
Run–Time Error Messages

BASIC returns run–time error messages when executing a program. If you compile your programs with the /DEBUG switch, BASIC also generates error messages from debugging routines. Error messages are either: (1) warnings or (2) fatal. Warning error messages contain a percent sign (%) prefix; they indicate that program execution can continue, but the results will be unpredictable. Fatal error messages contain a question mark (?) prefix; they indicate that program execution has been aborted. You can recover from most fatal errors by writing an error–handling routine.

Section 1 describes common run–time errors; section 2 describes debugging errors.

C.1 Common Run–Time Errors

1 ?BAD DIRECTORY FOR DEVICE

The device directory does not exist or is unreadable.

2 ?ILLEGAL FILE NAME

You cannot specify a file name that contains embedded blanks or unacceptable characters.

3 ?ACCOUNT OR DEVICE IN USE

The specified operation cannot be performed because the file device has already been opened by another user.

4 ?NO ROOM FOR USER ON DEVICE

No user storage space exists on the specified device.

5 ?CAN'T FIND FILE OR ACCOUNT

The specified file or current user account numbers are not on the device.

6 ?NOT A VALID DEVICE

The device is illegal or non-existent.

7 ?I/O CHANNEL ALREADY OPEN

The specified I/O channel is already open for input or output.

8 ?DEVICE NOT AVAILABLE

The requested device is in use.

9 ?I/O CHANNEL NOT OPEN

You cannot perform I/O unless your program has opened a channel.

10 ?PROTECTION VIOLATION

You are not allowed to perform the requested operation on the specified file.

11 ?END OF FILE ON DEVICE

You cannot perform input beyond the end of a data file.

12 ?FATAL SYSTEM I/O FAILURE

An I/O error has occurred at the system level. The last operation will not be completed.

13 ?USER DATA ERROR ON DEVICE

One or more characters may have transmitted incorrectly because of a parity error, bad punch combination on a card, or similar error.

14 ?DEVICE HUNG OR WRITE LOCKED

A hardware device cannot function properly; check tape drives, line printers, card punches, and similar devices.

15 ?KEYBOARD WAIT EXHAUSTED

No input was received during the execution of a WAIT statement.

16 ?NAME OR ACCOUNT NOW EXISTS

You cannot store a program or insert an account code if duplicate names or codes already exist in the system.

17 ?TOO MANY OPEN FILES ON UNIT

Only one open DECtape output file is permitted per DECtape drive. Only one open file per magtape drive is permitted.

18 ?ILLEGAL SYS() USAGE

Illegal use of the SYS system function. RSTS/E only

19 ?DISK BLOCK IS INTERLOCKED

The requested disk block segment is already in use (locked).

20 ?PACK IDS DON'T MATCH

You have specified an incorrect identification code for the disk pack. RSTS/E only

21 ?DISK PACK IS NOT MOUNTED

No disk pack is mounted on the specified disk drive. RSTS/E only

22 ?DISK PACK IS LOCKED OUT

The specified disk pack is mounted, but is temporarily disabled. RSTS/E only

23 ?ILLEGAL CLUSTER SIZE

The specified cluster size is unacceptable. The cluster size must be a power of 2. For a file cluster, the size must be equal to or greater than the pack cluster size, and must not be greater than 256. For a pack cluster, the size must be equal to or greater than the device cluster size, and must not be greater than 16. The device cluster size is fixed by type. RSTS/E only

24 ?DISK PACK IS PRIVATE

You do not have access to the specified disk pack. RSTS/E only

25 %DISK PACK NEEDS 'CLEANING'

A non-fatal disk mounting error has occurred; use the CLEAN operation in UTILTY. RSTS/E only

26 ?FATAL DISK PACK MOUNT ERROR

Your disk cannot be successfully mounted. RSTS/E only

27 ?I/O TO DETACHED KEYBOARD

You cannot perform I/O to a hung-up dataset or to a detached console keyboard. RSTS/E only

28 ?PROGRAMMABLE ^C TRAP

You entered a CTRL/C which invoked the ON ERROR GOTO error handler.

29 ?CORRUPTED FILE STRUCTURE

(1) A fatal error in a CLEAN operation has occurred (RSTS/E only), or (2) RMS has detected an invalid file structure on disk.

30 ?DEVICE NOT FILE-STRUCTURED

You cannot access a non-disk device that is not file-structured. This error occurs, for example, when you try to gain a directory listing for a non-directory device.

31 ?ILLEGAL BYTE COUNT FOR I/O

The buffer size specified in the RECORDSIZE option of the OPEN statement does not match the I/O you attempted.

32 ?NO BUFFER SPACE AVAILABLE

No buffer is available for file access. Possible causes are: (1) the receiving program has exceeded the pending message limit, or (2) the sending program has attempted to send a message and no small buffer is available for the operation. RSTS/E only

33 ?ODD ADDRESS TRAP

You cannot address: (1) nonexistent memory, or (2)an odd address using the PEEK function. Submit an SPR if this message appears for any other reason, and include all relevant output.

34 ?RESERVED INSTRUCTION TRAP

If floating point hardware is not available, you cannot execute an illegal, reserved, or FPP instruction. If you have floating point hardware, submit an SPR and include all relevant information.

35 ?MEMORY MANAGEMENT VIOLATION

You cannot specify an illegal Monitor address when using the PEEK function. Submit an SPR if this message appears for any other reason, and include all relevant information.

36 ?SP STACK OVERFLOW

You cannot extend the program stack beyond its legal size. If you generate this error, submit an SPR and include all relevant information.

37 ?DISK ERROR DURING SWAP

The system has swapped your job into or out of memory. The contents of your job area are lost, but the job remains logged into the system and is reinitialized to run the NONAME program. Report such occurrences to the system manager. RSTS/E only

38 ?MEMORY PARITY FAILURE

The memory occupied by your job has a parity error. Contact your System Manager.

39 ?MAGTAPE SELECT ERROR

The specified magtape drive is off-line. RSTS/E only

40 ?MAGTAPE RECORD LENGTH ERROR

A magtape record was longer than the buffer designated to handle it. RSTS/E only

41 ?NON-RES RUN-TIME SYSTEM

The run-time system is not resident in memory. RSTS/E only

42 ?VIRTUAL BUFFER TOO LARGE

Virtual memory buffers must be at least 512 bytes long.

43 ?VIRTUAL ARRAY NOT ON DISK

You cannot reference a virtual array on a non-disk device.

44 ?MATRIX OR ARRAY TOO BIG

Your array is too large for memory.

45 ?VIRTUAL ARRAY NOT YET OPEN

You cannot use a virtual array before you open its corresponding disk file.

46 ?ILLEGAL I/O CHANNEL

You have specified an I/O channel outside the legal range.

47 ?LINE TOO LONG

You cannot input a line longer than the record buffer. To expand the buffer, specify a larger value for RECORDSIZE in the OPEN statement.

48 %FLOATING POINT ERROR

Floating point overflow or underflow has occurred. If your program does not transfer to an error handling routine, BASIC returns (1) a zero as the floating-point value for underflow, and (2) the system's maximum positive number for overflow.

49 %ARGUMENT TOO LARGE IN EXP

A value in your program is outside of the legal range.

50 %DATA FORMAT ERROR

You have specified the wrong data type in an INPUT or READ statement.

51 %INTEGER ERROR

You cannot use a number as an integer when it is outside the allowable integer range. If your program does not transfer to an error handling routine, a zero is returned as the integer value.

52 ?ILLEGAL NUMBER

Your input is improperly formed. For example, "1..2" is an improperly formed number.

53 %ILLEGAL ARGUMENT IN LOG

You cannot pass a negative or zero argument to a log function.

54 %IMAGINARY SQUARE ROOTS

You cannot take the square root of a number less than zero. If your program does not transfer to an error handling routine, the value returned is the square root of the absolute value of the argument.

55 ?SUBSCRIPT OUT OF RANGE

You cannot reference an array element beyond its DIMensioned limits.

56 ?CAN'T INVERT MATRIX

You cannot invert a singular matrix.

57 ?OUT OF DATA

A READ requested additional data from an exhausted DATA list.

58 ?ON STATEMENT OUT OF RANGE

The index value in an ON GOTO or ON GOSUB statement is less than 1 or greater than the number of line numbers in the list.

59 ?NOT ENOUGH DATA IN RECORD

An INPUT statement did not find enough data in one line to satisfy all the specified variables.

60 ?INTEGER OVERFLOW, FOR LOOP

Your FOR loop has exceeded its index limit.

61 %DIVISION BY 0

You cannot divide a quantity by zero. If your program does not transfer to an error handling routine, zero is returned as the result.

62 ?NO RUN-TIME SYSTEM

The run-time system you requested is not part of the operating system. RSTS/E only

63 ?FIELD OVERFLOWS BUFFER

You cannot use FIELD to access more space than exists in the specified buffer.

64 ?NOT A RANDOM ACCESS DEVICE

You cannot use random access on the specified device.

65 ?ILLEGAL MAGTAPE() USAGE

Your MAGTAPE function arguments are not properly formatted.

66 ?MISSING SPECIAL FEATURE

Your program employs an unavailable system feature. RSTS/E only

67 ?ILLEGAL SWITCH USAGE

The switch operation or specification is illegal. RSTS/E only

68 UNUSED

69 UNUSED

70 UNUSED

71 ?STATEMENT NOT FOUND

You cannot CHAIN into a program at a nonexistent line number. RSTS/E only

72 ?RETURN WITHOUT GOSUB

Your program contains a RETURN statement before a GOSUB.

73 ?FNEND WITHOUT FUNCTION CALL

Your program contains a FNEND statement before a function call.

74 ?UNDEFINED FUNCTION CALLED

Your program has called a function that has not been defined. BASIC-PLUS only

75 ?ILLEGAL SYMBOL

Your program contains an unrecognizable character: for example, a line consisting of a # character. BASIC-PLUS only

76 ?ILLEGAL VERB

A verb in your statement is either misspelled or otherwise incorrect. BASIC-PLUS only

77 ?ILLEGAL EXPRESSION

Your program contains double operators, missing operators, mismatched parentheses, or some similar error. BASIC-PLUS only

78 ?ILLEGAL MODE MIXING

You cannot mix string and numeric operations.

79 ?ILLEGAL IF STATEMENT

Your IF statement is incorrectly formatted. BASIC-PLUS only

80 ?ILLEGAL CONDITIONAL CLAUSE

Your conditional expression is incorrectly formatted. BASIC-PLUS only

81 ?ILLEGAL FUNCTION NAME

You have used an illegal name to define a function. BASIC-PLUS only

82 ?ILLEGAL DUMMY VARIABLE

One of the dummy variables is not a legal variable name. BASIC-PLUS only

83 ?ILLEGAL FN REDEFINITION

You cannot redefine a user function. BASIC-PLUS only

84 ?ILLEGAL LINE NUMBER(S)

You have made a line number reference outside the legal range of 1 to 32767. BASIC-PLUS only

85 ?MODIFIER ERROR

You have: (1) used one of the statement modifiers (FOR, WHILE, UNTIL, IF, or UNLESS) incorrectly, or (2) placed an OPEN statement specifier, such as RECORDSIZE, out of the correct order. BASIC-PLUS only

86 ?CAN'T COMPILE STATEMENT

The statement cannot be compiled. Check its syntax. BASIC-PLUS only

87 ?EXPRESSION TOO COMPLICATED

Parentheses are nested too deeply for the given expression. BASIC-PLUS only

88 ?ARGUMENTS DON'T MATCH

The arguments in a function call do not match the arguments defined for the function, either in number or in type.

89 ?TOO MANY ARGUMENTS

A user-defined function can have a maximum of eight arguments.

90 %INCONSISTENT FUNCTION USAGE

You cannot reference a function with a different number of arguments than specified in its definition. BASIC-PLUS only

91 ?ILLEGAL DEF NESTING

The range of one function definition cannot cross the range of another. BASIC-PLUS only

92 ?FOR WITHOUT NEXT

A FOR statement was encountered without a corresponding NEXT statement to terminate the loop. BASIC-PLUS only

93 ?NEXT WITHOUT FOR

A NEXT statement was encountered without a previous FOR statement. BASIC-PLUS only

94 ?DEF WITHOUT FNEND

One of your function definitions requires an accompanying FNEND statement. BASIC-PLUS only

95 ?FNEND WITHOUT DEF

Your program contains a FNEND statement before a DEF statement.

96 ?LITERAL STRING NEEDED

You used a variable name where a numeric or character string was necessary. BASIC-PLUS only

97 ?TOO FEW ARGUMENTS

You cannot call a function with fewer arguments than were defined for the function.

98 ?SYNTAX ERROR

A statement is incorrectly formatted. BASIC-PLUS only

99 ?STRING IS NEEDED

You used a number or variable name where a character string was required. BASIC-PLUS only

100 ?NUMBER IS NEEDED

You used a character string or variable where a number was required. BASIC-PLUS only

101 ?DATA TYPE ERROR

You used a floating point, integer, or character string format variable or constant where some other data type was required.

102 ?1 OR 2 DIMENSIONS ONLY

You cannot assign more than two dimensions to a matrix.

103 ?PROGRAM LOST-SORRY

A fatal system error caused your program to be lost. This error can indicate hardware problems or the use of an improperly compiled program. Consult your system manager.

104 ?RESUME AND NO ERROR

Your program has encountered a RESUME statement without having transferred into an error handling routine.

105 ?REDIMENSIONED ARRAY

Your program has implicitly redimensioned an array. BASIC-PLUS only

106 %INCONSISTENT SUBSCRIPT USE

You have specified the wrong number of subscripts.

107 ?ON STATEMENT NEEDS GOTO

A statement beginning with ON does not contain a GOTO or GOSUB clause. BASIC-PLUS only

108 ?END OF STATEMENT NOT SEEN

Your statement contains too many elements to be processed correctly. BASIC-PLUS only

109 ?WHAT?

Your command or immediate mode statement could not be processed. Check for illegal verbs or improper formats.

110 ?BAD LINE NUMBER PAIR

You have incorrectly formatted line numbers specified in a LIST or DELETE command. BASIC-PLUS only

111 ?NOT ENOUGH AVAILABLE MEMORY

Your program exceeds the job's allowable memory size. Your program must be privileged, or your system manager must increase the job's memory size. BASIC-PLUS only

112 ?EXECUTE ONLY FILE

You cannot add, delete, or list a statement in a compiled program. BASIC-PLUS only

113 ?PLEASE USE THE RUN COMMAND

You cannot transfer control (as in a GOTO, GOSUB, or IF-GOTO statement) while in immediate mode.

114 ?CAN'T CONTINUE

You have stopped or ended your program. Execution cannot be resumed.

115 ?FILE EXISTS-RENAME/REPLACE

You cannot SAVE a file that already exists. Type REPLACE to save the file under the same name, or RENAME the file before saving it.

116 ?PRINT-USING FORMAT ERROR

You made an error in formatting the PRINT-USING string used to specify the output format of a PRINT-USING statement.

117 ?MATRIX OR ARRAY WITHOUT DIM

You referenced a matrix or array element outside the range of an implicitly defined array.

118 ?BAD NUMBER IN PRINT USING

You cannot use a format specified in the PRINT-USING string to print one or more values. BASIC-PLUS only

119 ?ILLEGAL IN IMMEDIATE MODE

You have issued an immediate mode statement that is executable only in a program.

120 ?PRINT-USING BUFFER OVERFLOW

You cannot specify a format that contains a field too large to be manipulated by the PRINT-USING statement.

121 ?ILLEGAL STATEMENT

You cannot execute a statement with unresolved compilation errors. BASIC-PLUS only

122 ?ILLEGAL FIELD VARIABLE

The FIELD variable specified is unacceptable; for example, a COM/MAP string or a parameter in a SUB.

123 STOP

A STOP statement was executed. Continue program execution by typing CONT and a carriage-return.

124 ?MATRIX DIMENSION ERROR

You have: (1) assigned more than two dimensions to a matrix, or (2) made a syntax error in a DIM statement.

125 ?WRONG MATH PACKAGE

Your main program was compiled with floating point precision different from that of one of your subprograms.

126 ?MAXIMUM MEMORY EXCEEDED

Your program has insufficient string and I/O buffer space because: (1) its allowable memory size has been exceeded, or (2) the system's maximum memory capacity has been reached.

127 ?SCALE FACTOR INTERLOCK

You cannot execute a subprogram with a scale factor that does not match that of the main program.

128 ?TAPE RECORDS NOT ANSI

The records in the magtape you accessed are neither ANSI D nor ANSI F format.

129 ?TAPE BOT DETECTED

You cannot perform a rewind or backspace operation on a tape already at the beginning of the file.

130 ?KEY NOT CHANGEABLE

Your UPDATE operation has tried to change the value of a key field that does not have the CHANGES attribute specified in the OPEN statement.

131 ?NO CURRENT RECORD

A previous GET or FIND is missing or was unsuccessful. The current DELETE or UPDATE therefore fails.

132 ?RECORD HAS BEEN DELETED

A record previously located by its Record File Address (RFA) has been deleted.

133 ?ILLEGAL USAGE FOR DEVICE

The requested operation cannot be performed because:

- The device specification contains illegal syntax.
- The specified device does not exist on your system.

- The specified device is inappropriate for the requested operation (for example, magtape for an indexed file).

134 ?DUPLICATE KEY DETECTED

You cannot duplicate key fields for indexed file PUT operations if duplicate key values were not permitted when the file was created.

135 ?ILLEGAL USAGE

You either: (1) opened a file of undeclared organization, or (2) did not specify the record operation in the ACCESS clause.

136 ?ILLEGAL OR ILLOGICAL ACCESS

The requested access is impossible because:

- The attempted record operation and the ACCESS clause in the OPEN statement are incompatible.
- The ACCESS clause is inconsistent with the file organization.
- READ or APPEND was specified when the file was created. Change the ACCESS clause.

137 ?ILLEGAL KEY ATTRIBUTES

An illegal combination of key characteristics has occurred. Check the OPEN statement for either:

NODUPLICATES and CHANGES

CHANGES without DUPLICATES

138 ?FILE IS LOCKED

The file has been locked by another user, or by the system in a program that does not allow shared access.

139 ?INVALID FILE OPTIONS

You have selected invalid file options in the OPEN statement.

140 ?INDEX NOT INITIALIZED

You cannot GET or FIND in an empty indexed file.

141 ?ILLEGAL OPERATION

The requested operation is illegal because:

- An OPEN statement specifies a file organization that was not included in the BUILD.
- You have task built the program with the wrong RMS file support.
- DELETE cannot be performed on a sequential file.
- UPDATE cannot be performed on a magtape file.

- Block I/O cannot be performed on an RMS file. (Block I/O requires virtual organization.)
- RMS I/O cannot be performed on a block I/O file. (RMS I/O requires sequential, relative, or indexed organization.)

142 ?ILLEGAL RECORD ON FILE

The count field record in the file is invalid.

143 ?BAD RECORD IDENTIFIER

The requested operation cannot be performed because:

- Random access operations cannot be performed with a zero or negative record number specification.
- A GET or FIND on an indexed file cannot contain a null key value.

144 ?INVALID KEY OF REFERENCE

You cannot perform a GET, FIND, or RESTORE with an invalid key of reference value.

145 ?KEY SIZE TOO LARGE

The key length on a GET or FIND is either zero or larger than the key length defined for the target record.

146 ?TAPE NOT ANSI LABELLED

BASIC supports only ANSI-labelled magtape for file structured access.

147 ?RECORD NUMBER EXCEEDS MAXIMUM

Either the maximum record number at file creation is negative, or the specified record number exceeds the maximum specified for this file.

148 ?BAD RECORDSIZE VALUE ON OPEN

The value in the RECORDSIZE clause in the OPEN statement is zero.

149 ?NOT AT END OF FILE

You attempted a PUT operation (1) on a sequential file before the last record, or (2) without opening the file for WRITE access.

150 ?NO PRIMARY KEY SPECIFIED

You cannot create an indexed file without a primary key.

151 ?KEY FIELD BEYOND END OF RECORD

The position given for the key field exceeds the maximum size of the record.

152 ?ILLOGICAL RECORD ACCESSING

You cannot perform the specified operation on the file type. For example, a random access on a sequential file.

153 ?RECORD ALREADY EXISTS

An attempted random access PUT on a relative file has encountered a pre-existing record.

154 ?RECORD/BUCKET LOCKED

Another program has locked the target bucket.

155 ?RECORD NOT FOUND

A random access GET or FIND was attempted on a deleted or nonexistent record.

156 ?SIZE OF RECORD INVALID

The COUNT specification is invalid because:

- COUNT equals zero.
- COUNT exceeds the maximum size of the record.
- COUNT conflicts with the actual size of the current record during a sequential file UPDATE on disk.
- COUNT does not equal the maximum record size for fixed format records.

157 ?RECORD ON FILE TOO BIG

The record accessed is larger than the input buffer.

158 ?PRIMARY KEY OUT OF SEQUENCE

You cannot PUT a record with a key value lower than the previous record when performing sequential access on an indexed file.

159 ?KEY LARGER THAN RECORD

The key specification exceeds the maximum record size.

160 ?FILE ATTRIBUTES NOT MATCHED

The following attributes in the OPEN statement do not match the corresponding attributes of the target file:

ORGANIZATION

BUCKETSIZE

BLOCKSIZE

RECORDSIZE

KEY

record format

161 ?MOVE OVERFLOWS BUFFER

The combined length of the elements in the MOVE statement I/O list exceeds the RECORDSIZE defined for the file. (This error occurs when you attempt to MOVE data to/from the record buffer.)

162 ?CANNOT OPEN FILE

Your file cannot be opened. Check the STATUS variable for system error codes.

163 ?NO FILE NAME

Your file cannot be opened. Check spelling, syntax, and so forth. DECsystem-20 only

164 ?TERMINAL FORMAT FILE REQUIRED

PRINT and INPUT statements require a terminal format file.

165 ?CANNOT POSITION TO EOF

The operating system could not find the end of a sequential file opened with ACCESS APPEND. The file could be corrupted.

166 ?NEGATIVE FILL OR STRING LENGTH

You cannot use FILL elements with a value less than zero in a MOVE statement I/O list.

167 ?ILLEGAL RECORD FORMAT

The record format is illegal because:

- The record given is illegal for the file's organization.
- The record given is illegal for the operating system on which this file resides.
- There are embedded carriage control characters in variable length records.

168 ?ILLEGAL ALLOW CLAUSE

The value specified for the ALLOW clause is illegal for the type of file organization or for the operating system on which the file resides.

169 UNUSED

170 ?INDEX NOT FULLY OPTIMIZED

Your record was successfully written, but the index was not optimized. This will slow record access.

171 ?RRV NOT FULLY UPDATED

RMS wrote your record successfully, but did not update one or more Record Retrieval Vectors. Therefore, you cannot retrieve any records associated with those vectors. Delete the records and reinsert them.

172 ?RECORD LOCK FAILED

You have read a locked record; the RLK bit in the ROP field has failed. (TRAX only)

173 ?INVALID RFA FIELD

During a FIND or GET by RFA, an invalid record's file address was contained in the RAB. Please submit an SPR and include relevant output.

174 ?FILE EXPIRATION DATE NOT YET REACHED

You cannot write to a file before its expiration date. (VAX/VMS only)

175 ?NODE NAME ERROR

You have included a node name in your file specification that: (1) is in error (it is part of a network and exists on another system), or (2) is nonexistent. (VAX/VMS only)

176-178 UNUSED

179 UNEXPIRED FILEDATE

180-229 UNUSED

230 ?NO FIELDS IN IMAGE STRING

Your image string does not contain proper symbols. DECsystem-20 only

231 ?ILLEGAL STRING IMAGE

DECsystem-20 only

232 ?NULL IMAGE

You have not included an image field in your string image line. For example, the image string " " generates this error. DECsystem-20 only

233 ?ILLEGAL NUMERIC IMAGE

DECsystem-20 only

234 ?NUMERIC IMAGE FOR STRING

You cannot PRINT a string USING a numeric image. DECsystem-20 only

235 ?STRING IMAGE FOR NUMERIC

You cannot PRINT a numeric character USING a string image. DECsystem-20 only

236 ?TIME LIMIT EXCEEDED

You have exceeded the time limit set for your job by the operating system. DECsystem-20 only

237 ?1ST ARG TO SEG$ > 2ND

You cannot specify a starting point in a string search that is greater than the end point. DECsystem-20 only

238 ?ARRAYS MUST BE SAME DIMENSION

You cannot perform matrix addition or subtraction on arrays of different dimensions.

239 ?ARRAYS MUST BE SQUARE

You cannot perform a matrix inversion (MAT INV) on an array that is not square.

240 ?CAN'T CHANGE ARRAY DIMENSIONS

You cannot redimension a one dimensional array to two dimensions. DECsystem-20 only

241 ?FLOATING OVERFLOW

You have exceeded the upper range of the system's math capability. This is a fatal error. DECsystem-20 only

242 ?FLOATING UNDERFLOW

You have exceeded the lower range of the system's math capability. This is a fatal error. DECsystem-20 only

243 ?CHAIN TO NONEXISTENT LINE NO.

The line number in the CHAIN statement does not exist. If the program was compiled with the /NOLINE switch, recompile it without that switch. DECsystem-20 only

244 ?EXPONENTIATION ERROR

The attempted exponentiation is illegal. The result is a value of zero and the program continues. DECsystem-20 only

245 ?ILLEGAL EXIT FROM DEF*

You cannot exit from a multi-line DEF* function directly to an END or SUBEND statement.

246 ?ERROR TRAP NEEDS RESUME

The error handler has run off the end of: (1) a program unit, or (2) a DEF or DEF* where the error occurred. You must include a RESUME statement before the END, SUBEND, or FNEND statement.

247 ?ILLEGAL RESUME TO SUBROUTINE

You cannot use RESUME without a line number if the current module name does not match the error module name.

248 ?ILLEGAL RETURN FROM SUBROUTINE

Your program contains an external subroutine RETURN statement before a CALL.

249 ?ARGUMENT OUT OF BOUNDS

250 ?NOT IMPLEMENTED

You have referenced a language element that does not exist in your version of BASIC-PLUS-2.

251 ?RECURSIVE SUBROUTINE CALL

Your program contains a subroutine that attempts to call itself. This is illegal. Correct the flow of control in the program. (Not applicable to VAX/VMS)

252 ?FILE ACP FAILURE

The Operating System's file handler reported an error to RMS. The corresponding value is in STATUS.

253 ?DIRECTIVE ERROR

An executive directive reported an error. The corresponding value is in STATUS.

254 UNUSED

255 UNUSED

C.2 Debugging Procedures and Error Messages

BASIC provides interactive debugging commands to help you locate run-time errors in your program. These commands allow you to check program operation and make corrections. This section describes: (1) debugging procedures and (2) common debugging error messages. This section does not apply to VAX. For more information, see Chapter 1.

C.2.1 Debugging Procedures

The debugging commands are:

BREAK	CONTINUE	I/O BUFFER
UNBREAK	ERL	STRING
STEP	ERR	FREE
TRACE	ERN	CORE
UNTRACE	RECOUNT	
PRINT	STATUS	
LET	EXIT	

You can use these commands only on programs or subprograms that have been compiled with the /DEBUG switch.

When you run a program, execution stops the first time you enter a module that was compiled with the /DEBUG switch. After execution stops, the debugging aid prints an identifying message and prompt:

DEBUG: module name

where:

module name — is the name of the program or subprogram compiled with the /DEBUG switch.

— signals you to enter debugging aid commands.

Then, to continue the program and execute the command, type the CONTINUE (CON) command:

```
DEBUG: module name
#BREAK 10 RET
#CON RET
```

In this example, the CON command resumes program execution until line 10.

Following the successful execution of a debugging command, a message identifies your current position in the program or subprogram:

command AT LINE n [,name]

where:

command	is the last executed debugging command (for example, BREAK or STEP) that stops execution.
n	is your current line number in the program or subprogram.
name	is the name of the currently executing subprogram. This name is not displayed if you are executing the main program.

After this message the debugger gives the # prompt.

C.2.2 Error Messages

This section lists debugger error messages and their possible causes.

What ?

The debugger does not understand your command. Check spelling, syntax, and so forth.

% Stop at line n in subprogram x

A STOP statement in your program halted debugging. Type CONTINUE to proceed.

% Illegal Syntax in LET

Your LET statement formatting is incorrect. The format for LET is:

LET variable=value

where:

variable	is the name of the variable whose content you want to change. You can specify one constant or a variable as an argument. You cannot specify expressions.

% Illegal Syntax in PRINT

Your PRINT statement formatting is incorrect. The format for the PRINT command is:

PRINT n

where:

n	is the variable whose contents you PRINT. You can specify one constant or variable as an argument. You cannot specify an expression.

% ON ERROR entry in debugging

A CTRL/C trap or program error has started the debugger.

% Can't Continue or STEP

Your program encountered an error it cannot handle. Execution cannot continue.

% Data Error in LET or PRINT

The debugger encountered a data conversion error while processing a LET or PRINT statement.

% Bad line spec in (UN)BREAK

You have: (1) specified a non-existent line number, (2) used incorrect syntax in specifying a program or subprogram, or (3) used incorrect syntax when listing multiple arguments.

% No room

You specified too many breakpoints for a BREAK or UNBREAK command. The maximum is 8.

Appendix D
ASCII Codes and Data Representation

D.1 ASCII Character Codes

Table D-1: ASCII Codes

Decimal Code	7-Bit Octal Code	Character	Remarks
0	000	nul	Null, tape feed, shift, ^P
1	001	SOH	Start of heading, start of message, ^A
2	002	STX	Start of text, end of address, ^B
3	003	ETX	End of text, end of message, ^C
4	004	EOT	End of transmission, shuts off TWX machine, ^D
5	005	ENQ	Enquiry, WRU, ^E
6	006	ACK	Acknowledge, RU, ^F
7	007	BEL	Bell, ^G
8	010	BS	Backspace, format effector, ^H
9	011	HT	Horizontal tab, ^I
10	012	LF	Line feed, ^J
11	013	VT	Vertical tab, ^K
12	014	FF	Form feed, page, ^L
13	015	CR	Carriage return, ^M
14	016	SO	Shift out, ^N
15	017	SI	Shift in, ^O
16	020	DLE	Data link escape, ^P
17	021	DC1	Device control 1, ^Q
18	022	DC2	Device control 2, ^R
19	023	DC3	Device control 3, ^S
20	024	DC4	Device control 4, ^T

(continued on next page)

Table D-1: ASCII Codes (Cont.)

Decimal Code	7-Bit Octal Code	Character	Remarks
21	025	NAK	Negative acknowledge, ERR, ^U
22	026	SYN	Synchronous idle, ^V
23	027	ETB	End-of-transmission block, logical end of medium, ^W
24	030	CAN	Cancel, ^X
25	031	EM	End of medium, ^Y
26	032	SUB	Substitute, ^Z
27	033	ESC	Escape, prefix, shift, ^K
28	034	FS	File separator, shift, ^L
29	035	GS	Group separator, shift, ^M
30	036	RS	Record separator, shift, ^N
31	037	US	Unit separator, shift, ^O
32	040	SP	Space
33	041	!	Exclamation point
34	042	"	Double quotation mark
35	043	#	Number sign
36	044	$	Dollar sign
37	045	%	Percent sign
38	046	&	Ampersand
39	047	'	Apostrophe
40	050	(	Left parenthesis
41	051	)	Right parenthesis
42	052	*	Asterisk
43	053	+	Plus sign
44	054	,	Comma
45	055	-	Minus sign, hyphen
46	056	.	Period, dot
47	057	/	Slash, statement separator
48	060	0	Zero
49	061	1	One
50	062	2	Two
51	063	3	Three
52	064	4	Four
53	065	5	Five
54	066	6	Six
55	067	7	Seven
56	070	8	Eight
57	071	9	Nine
58	072	:	Colon
59	073	;	Semicolon
60	074	<	Left angle bracket
61	075	=	Equal sign
62	076	>	Right angle bracket
63	077	?	Question mark
64	100	@	At sign
65	101	A	Upper-case A
66	102	B	Upper-case B
67	103	C	Upper-case C

(continued on next page)

Table D-1: ASCII Codes (Cont.)

Decimal Code	7-Bit Octal Code	Character	Remarks
68	104	D	Upper-case D
69	105	E	Upper-case E
70	106	F	Upper-case F
71	107	G	Upper-case G
72	110	H	Upper-case H
73	111	I	Upper-case I
74	112	J	Upper-case J
75	113	K	Upper-case K
76	114	L	Upper-case L
77	115	M	Upper-case M
78	116	N	Upper-case N
79	117	O	Upper-case O
80	120	P	Upper-case P
81	121	Q	Upper-case Q
82	122	R	Upper-case R
83	123	S	Upper-case S
84	124	T	Upper-case T
85	125	U	Upper-case U
86	126	V	Upper-case V
87	127	W	Upper-case W
88	130	X	Upper-case X
89	131	Y	Upper-case Y
90	132	Z	Upper-case Z
91	133	[	Left bracket, shift K
92	134	\	Backslash, shift L
93	135	]	Right bracket, shift M
94	136	^	Caret, circumflex
95	137	_	Underscore
96	140	`	Accent grave
97	141	a	Lower-case a
98	142	b	Lower-case b
99	143	c	Lower-case c
100	144	d	Lower-case d
101	145	e	Lower-case e
102	146	f	Lower-case f
103	147	g	Lower-case g
104	150	h	Lower-case h
105	151	i	Lower-case i
106	152	j	Lower-case j
107	153	k	Lower-case k
108	154	l	Lower-case l
109	155	m	Lower-case m
110	156	n	Lower-case n
111	157	o	Lower-case o
112	160	p	Lower-case p
113	161	q	Lower-case q
114	162	r	Lower-case r
115	163	s	Lower-case s

(continued on next page)

Table D-1: ASCII Codes (Cont.)

Decimal Code	7-Bit Octal Code	Character	Remarks
116	164	t	Lower-case t
117	165	u	Lower-case u
118	166	v	Lower-case v
119	167	w	Lower-case w
120	170	x	Lower-case x
121	171	y	Lower-case y
122	172	z	Lower-case z
123	173	{	Left brace
124	174	¦	Vertical line
125	175	}	Right brace
126	176	~	Tilde
127	177	DEL	Delete, rubout

NOTES

1. Teleprinters manufactured by Teletype Corporation, Skokie, Illinois, have used codes 175 (ALT) and 176 for ESC. Programs should avoid including 175 and 176 if you wish to use these codes for ESC on older teleprinters.

2. ASCII is a 7-bit character code with an optional parity bit (200) added for many devices. Programs normally use seven bits internally; the extra bit is either stripped or added so the program will operate with either parity or non-parity generating devices.

   ISO Recommendation R646 and CCITT Recommendation V.3 (International Alphabet No. 5) are identical to ASCII except that: (1) the number sign (043) is represented as instead of #, and (2) certain characters are reserved for national use.

D.2 Radix-50 Character Set

Many items in RSTS/E, such as filenames and extensions, are stored in Radix-50 format. This format allows 3 characters of data to be stored as a 2-byte integer (one 16-bit word).

Table D-2 lists the characters representable in Radix-50 format, together with their ASCII octal and Radix-50 octal equivalents.

Table D-2: Radix-50 Character Set

Character	ASCII Octal Equivalent	Radix-50 Octal Equivalent
space	40	0
A-Z	101-132	1-32
$	44	33
.	56	34
0-9	60-71	36-47

Radix-50 evaluates a character according to the format:

X = Y * 50^Z

where:

X is the value of the character.

Y is the Radix-50 octal equivalent of the character.

50 is a constant (in octal).

Z is the character's position in the string. The leftmost digit is assigned position #2, the middle character is assigned position #1, and the right-most character is assigned position zero.

To represent a 3-character string in Radix-50 format, the first character of a string (or a single character) is placed in the leftmost position of the Radix-50 word. For example, in the string "X2B," the character X (30 octal) is multiplied by 50^2 to give 113000 (octal). The character 2 (40 octal) is multiplied by 50^1 to give 002400. The character B (2 octal) is multiplied by 50^0 to give 000002. Adding the value of each character gives the full octal value of the Radix-50 word.

X = 30 * 50^2 = 113000

2 = 40 * 50^1 = 002400

B = 02 * 50^0 = 000002

TOTAL = 115402 (octal)

Note that addition is also carried out in octal.

Table D-3 simplifies this process by listing the value of each Radix-50 character for each position.

Table D-3: ASCII/Radix-50 Equivalents

First or Single Character	Second Character	Third Character
space 000000	space 000000	space 000000
A 003100	A 000050	A 000001
B 006200	B 000120	B 000002
C 011300	C 000170	C 000003
D 014400	D 000240	D 000004
E 017500	E 000310	E 000005
F 022600	F 000360	F 000006
G 025700	G 000430	G 000007
H 031000	H 000500	H 000010
I 034100	I 000550	I 000011
J 037200	J 000620	J 000012
K 042300	K 000670	K 000013
L 045400	L 000740	L 000014
M 050500	M 001010	M 000015
N 053600	N 001060	N 000016
O 056700	O 001130	O 000017
P 062000	P 001200	P 000020
Q 065100	Q 001250	Q 000021
R 070200	R 001320	R 000022
S 073300	S 001370	S 000023
T 076400	T 001440	T 000024
U 101500	U 001510	U 000025
V 104600	V 001560	V 000026
W 107700	W 001630	W 000027
X 113000	X 001700	X 000030
Y 116100	Y 001750	Y 000031
Z 121200	Z 002020	Z 000032
$ 124300	$ 002070	$ 000033
. 127400	. 002140	. 000034
0 135600	0 002260	0 000036
1 140700	1 002330	1 000037
2 144000	2 002400	2 000040
3 147100	3 002450	3 000041
4 152200	4 002520	4 000042

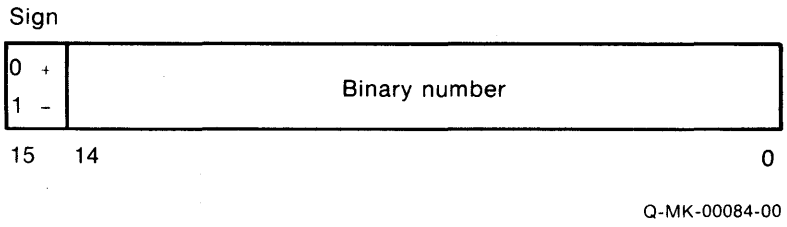
(continued on next page)

Table D-3: ASCII/Radix-50 Equivalents (Cont.)

First or Single Character	Second Character	Third Character
	space 000000	space 000000
5 155300	5 002570	5 000043
6 160400	6 002640	6 000044
7 163500	7 002710	7 000045
8 166600	8 002760	8 000046
9 171700	9 003030	9 000047

D.3 Integer Format

Figure D-1: Integer Format



Integers are stored in two's complement representation. For example:

+6 = 000006 (octal)

+22 = 000026 (octal)

-7 = 1777 (octal)

-1 = 177777 (octal)

Integer constants must be in the range -32767 TO +32767.

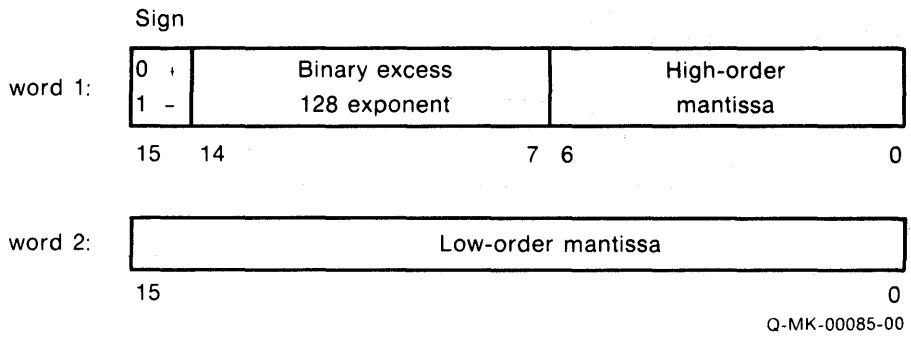
D.4 Floating-Point Formats

- The exponent for both 2-word and 4-word floating-point formats is stored in excess 128 (200 octal) notation. Binary exponents from -128 TO +127 are represented by the binary equivalents of zero through 255 (zero through 377 octal).
- Fractions are represented in sign-magnitude notation, with the binary Radix point to the left.
- Numbers are assumed to be normalized. The most significant bit is assumed to be 1 and is not stored. However, if the exponent is zero, the bit is also zero. The value zero is represented by two or four words of zeros. For example:

NUMBER	2-WORD FORMAT	4-WORD FORMAT
+1.0	40200 0	40200 0 0 0
-5	140640 0	140640 0 0 0

D.4.1 Real Format (2-Word Floating-Point)

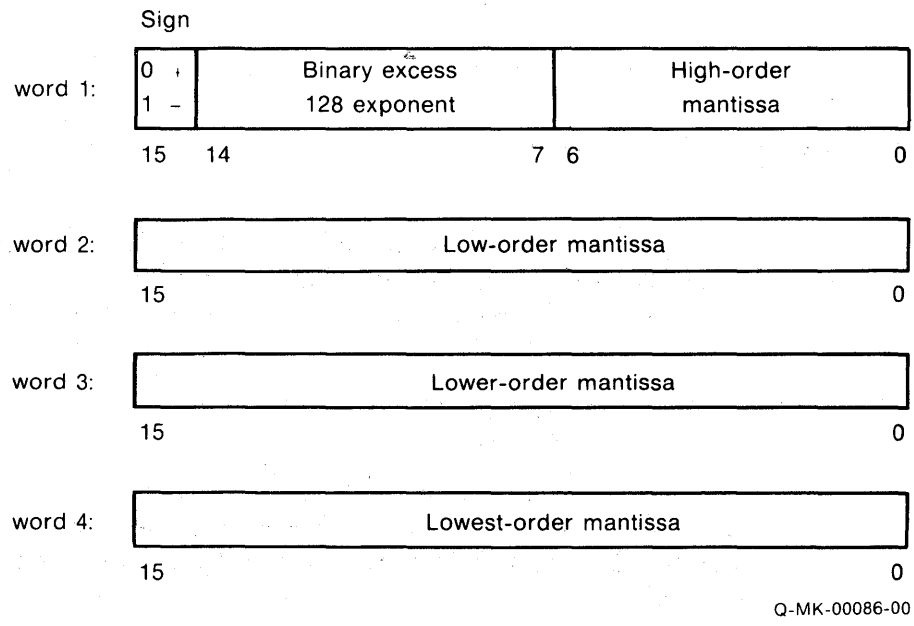
Figure D-2: Real Format (2-Word Floating Point)



Because the high-order bit of the mantissa is always 1, it is discarded. This gives an effective precision of 24 bits (7 digits of accuracy). The magnitude range is .29E-38 to .17E39.

D.4.2 Double-Precision Format (4-Word Floating-Point)

Figure D-3: Double Precision Format

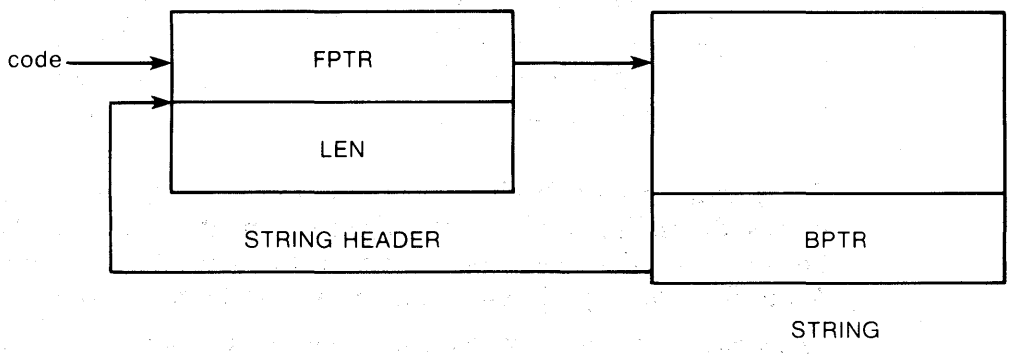


The effective precision is 56 bits (17 decimal digits of accuracy). The magnitude range is .29E-38 to .17E39.

D.5 String and Array Format

D.5.1 String Format

Figure D-4: Dynamic String Format

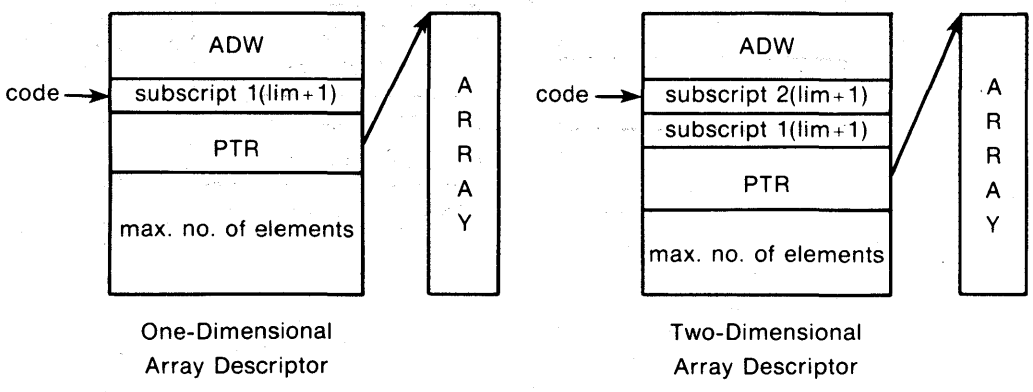


Q-MK-00087-00

Dynamic strings contain a 2-word string header. The first word is a forward pointer (FPTR) that points to the first byte of the string. The second word represents the length (LEN) of the string in bytes. Following the data in the string and aligned on the next higher word boundary is a word that points back to the free pointer. This word is internally specific and should not be accessed.

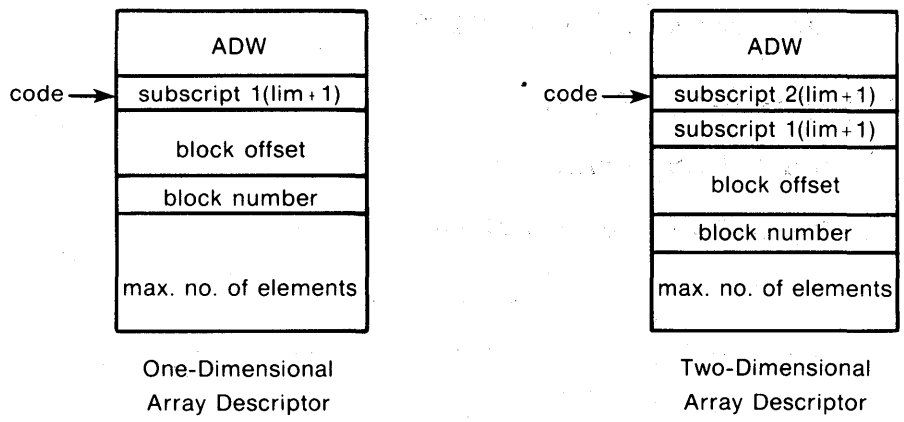
D.5.2 Array Format

Figure D-5: Format of Arrays in Memory



Q-MK-00088-00

Figure D-6: Format of Virtual Arrays



Every array in a file begins in a new block. An array can occupy one or more blocks, depending on the length of each element and the number of elements. For example, if Array A is the first array in the file, it begins in block #1. Array B could begin in block #2, Array C in block #5, and so forth.

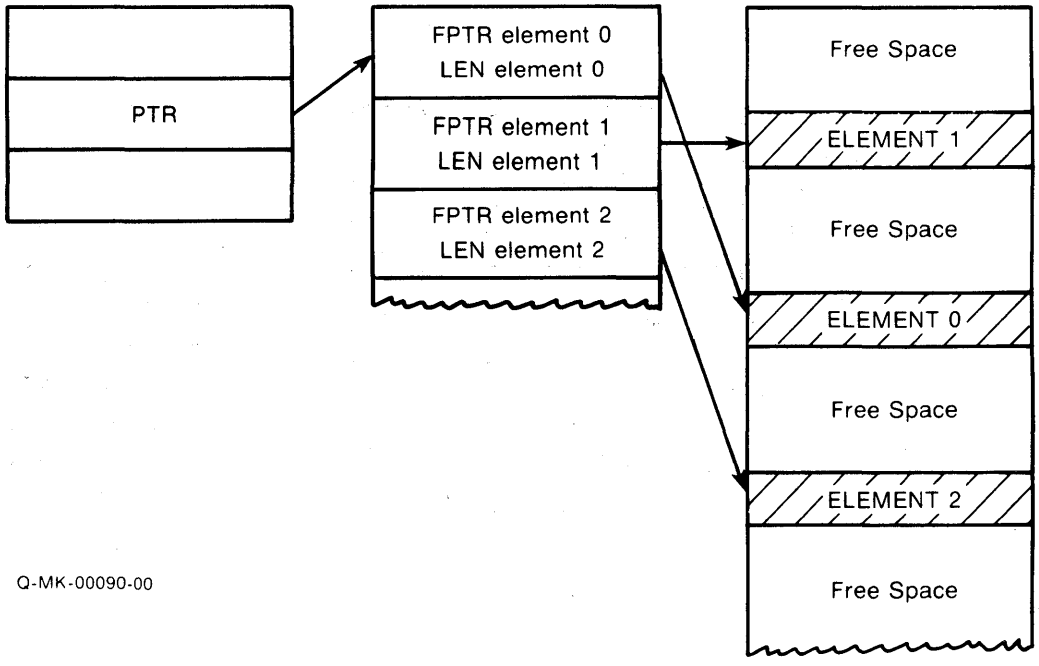
Array elements are positioned according to the number of bytes from the start of the array's block. This is known as "offset." For example, the array:

D(3,3) = 2

has 16 elements, and each element is two words long. When they are stored, the elements are read into the block in row order, with element (0,0) offset zero words. The next element, (0,1), would be offset two words; element (0,2) would be offset four words, and so forth. The last element, (3,3) would have an offset of 30.

With the exception of dynamic string arrays, the pointer (PTR) points to the array elements. For dynamic string arrays, PTR points to a list of string headers as follows:

Figure D-7: Dynamic String Array Pointers



D.5.3 Array Descriptor Word

The array descriptor word (ADW) is a 16 bit word used by the operating system to describe the characteristics of an array. The bits of the ADW are explained in Table D-4.

Table D-4: Array Descriptor Word

	Bits															
Array Type	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Numeric Memory	0	L	0	S	T		0	0	0	0	0	0	0	0	0	0
Numeric Virtual	0	0	1	S	T		0	0	Channel Number							
String Memory	1	0	0	S	0	0	0	0	0	0	0	0	0	0	0	0
String Common	1	1	0	S	Element				Length in bytes							
String Virtual	1	0	1	S	LOG_2 (Len)				Channel Number							

Code:

T - Data Type
S - Number of subscripts minus 1 (0 is one-dimensional, 1 is two-dimensional)
L - Location (memory or common)

Each array sets the bits of the ADW as follows:

Numeric memory - Bits 0 through 9 are set to 0. Bits 10 and 11 set the data type (for example, 00 for integer, 01 for floating point, 10 for double precision). Bit 12 sets the number of subscripts minus 1. Bit 13 is set to 0. Bit 14 is set to 0 if the array is in memory and 1 if the array is a COMMON. Bit 15 is set to 0.

Numeric virtual - Bits 0 through 7 represent the channel number. Bits 8 and 9 are set to 0. Bits 10 and 11 set the data type. Bit 12 sets the number of subscripts minus 1. Bit 13 is set to 1. Bits 14 and 15 are set to 0.

String memory - Bits 0 through 11 are set to 0. Bit 12 sets the number of subscripts minus 1. Bits 13 and 14 are set to 0. Bit 15 is set to 1.

String common - Bits 0 through 11 represent the element length in bytes. Bit 12 sets the number of subscripts minus 1. Bit 13 is set to 0. Bits 14 and 15 are set to 1.

String virtual - Bits 0 through 7 represent the channel number. Bits 8 through 11 represent LOG_2 (i.e., the string length). Bit 12 sets the number of subscripts minus 1. Bit 13 is set to 1. Bit 14 is set to 0. Bit 15 is set to 1.

The maximum number of elements is present in the array descriptor only when the array is redimensioned or when the array is used as a subroutine argument. The number of elements is stored as a double-precision integer.

Appendix E
Reserved Words in BASIC–PLUS–2

ABORT
ABS
ABS%
ACCESS
ACCESS%
ALL
ALIGNED
ALLOW
ALTERNATE
AND
APPEND
AS
ASCII
ATN
ATN2
BACK
BEL
BIN
BIN%
BIN$
BINARY
BIT
BLOCK
BLOCKSIZE
BROADCAST
BS
BUCKETSIZE
BUFFER
BUFFERSIZE
BUFSIZ
BY
CALL
CALLR
CCPOS
CHAIN
CHANGE
CHANGES
CHR$
CLK$
CLOSE
CLUSTERSIZE
COM
COMMON
COMP%
CON
CONNECT
CONTIGUOUS
COS
COT
COUNT
CR
CTRLC
CVT$$
CVT$%
CVT%$
CVTF$
DAT
DAT$
DATA
DATE
DEF
DEF*
DEFAULTNAME
DEL
DELETE
DELIMIT
DENSITY
DESC
DET
DEF$
DIM
DIMENSION
DOUBLE
DOUBLEBUF
DUPLICATES
ECHO
EDIT$
ELSE
END
EQ
EQV
ERL
ERN$
ERR
ERROR
ERT$
ESC
EXP
EXTEND
EXTENDSIZE
EXTERNAL
FF
FIELD

FILE
FILESIZE
FILL
FILL%
FILL$
FIND
FIX
FIXED
FNEND
FNEXIT
FOR
FORCEIN
FORMAT$
FREE
FROM
FSP$
FSS$
FUNCTION
GE
GET
GO
GOSUB
GOTO
GT
HANGUP
HEX
HEX%
HEX$
HT
IDN
IF
IFEND
IFMORE
IMAGE
IMP
INDEXED
INIMAGE
INPUT
INSTR
INT
INTEGER
INV
INVALID
KEY
KILL
LEFT
LEFT$
LEN
LET
LF
LINE
LINO
LINPUT
LOC
LOCK
LOF
LOG
LOG10
LONG
LSA
LSET
MAGTAPE
MAP
MAR
MAR%
MARGIN
MAT
MAX
MID
MID$
MIN
MOD
MOD%
MODE
MODIFY
MOVE
MSGMAP
NAME
NEXT
NOCHANGES
NODATA
NODUPLICATES
NOECHO
NOEXTEND
NOMARGIN
NONE
NOPAGE
NOQUOTE
NOREWIND
NOSPAN
NOT
NOTAPE
NUL$
NUM
NUM$
NUM1$
NUM2
OCT
OCT%
OCT$
ON
ONECHR
ONENDFILE
ONERROR
OPEN
OR
ORGANIZATION
OUTPUT
PAGE
PEEK
PI
PLACE$
POKE
POS
POS%
POS$
PRIMARY
PRINT
PROD$
PUT
QUO$
QUOTE
RAD%
RAD$
RAN
RANDOM
RANDOMIZE
RCTRLC
RCTRLO
READ
REAL
RECORD
RECORDSIZE
RECOUNT
REF
RELATIVE
REM
RESET
RESTORE
RESUME
RETURN
RIGHT
RIGHT$
RND
RSET
SCRATCH
SEG$
SEQUENTIAL
SGN
SHIFT

SI
SIN
SINGLE
SLEEP
SO
SP
SPACE$
SPAN
SPEC%
SQR
SQRT
STATUS
STEP
STOP
STR$
STREAM
STRING
STRING$
SUB
SUBEND
SUBEXIT
SUBPROGRAM
SUM$
SWAP%
SYS
TAB
TAN
TAPE
TASK
TEMPORARY
TERMINAL
THEN
TIM
TIME
TIME$
TO
TRM$
TRN
TST
TSTEND
TYP
TYPE
TYPE$
UNALIGNED
UNDEFINED
UNLESS
UNLOCK
UNTIL
UPDATE
USEAGE
USEAGE$
USEROPEN
USING
USR
USR$
VAL
VAL%
VALUE
VARIABLE
VFC
VIRTUAL
VPS%
VT
WAIT
WHILE
WINDOWSIZE
WITH
WORD
WRITE
WRKMAP
XLATE
XOR
ZER
.ABORT
.DEFINE
.ENDC
.INDENT
.IF
.IFDF
.IFF
.IFNDF

Appendix F
Program and Subprogram Coding Conventions

This appendix presents a recommended format for writing and documenting BASIC-PLUS-2 programs and subprograms. Section F.1 explains how you should organize and document the program. Section F.2 is a sample template summarizing these coding conventions. Both main programs and subprograms have the same organization and require similar documentation.

F.1 Program and Subprogram Organization and Documentation

In order of coding, the sections of your program should be:

- TITLE - Line 1

  Include:

  - A comment field with the program name
  - An ON ERROR GOTO statement to enable error handling
  - A comment field containing:
    * The program version number
    * The program edit level
    * The date of the most recent edit
    * The program's author
- COPYRIGHT - Line 11

  Include a comment field containing:

  - The word COPYRIGHT
  - A legal copyright symbol and date

- The copyright holder's name
- Statements of reservation of rights
- Caveats for use

- CALL FORMAT - Line 14

  Include a CALL format for subprograms only. The comment field should contain:

  - The words CALL FORMAT
  - An explanation of the format for calling the subprograms
  - The names of subprogram arguments
  - An explanation of each subprogram argument

- MODIFICATION HISTORY LOG - Line 20

  Include a comment field containing:

  - The words MODIFICATION HISTORY LOG
  - The module's version number
  - The date of the most recent change
  - Initials of the programmer who made the last change
  - Reasons for the change

- PROGRAM OR SUBPROGRAM DESCRIPTION - Line 100

  Include a comment field containing:

  - The words [SUB-]PROGRAM DESCRIPTION
  - A summary of the program's purpose
  - An explanation of the program's logic (optional)

- CHANNEL ASSIGNMENTS - Line 300

  Include a comment field containing:

  - The words CHANNEL ASSIGNMENT
  - A list of the file I/O channel numbers used by the program
  - A description of each channel number's purpose and use

- VARIABLES AND ARRAYS - Line 400

  Include a comment field containing:

  - The words VARIABLES AND ARRAYS USED
  - The name of each variable and array in the program
  - An explanation of what the data in each variable or array represents

- SUBROUTINES - Line 700

  Include a comment field containing:

  - The words SUBROUTINE USED
  - The name of each subroutine in the program module
  - The starting line number of the subroutine
  - A description of the subroutine's function

- FUNCTIONS - Line 750

  Include a comment field containing:

  - The words FUNCTIONS USED
  - The name of each function in the program module
  - The line number where each function is defined
  - A description of what the function does

- DATA DECLARATIONS - Line 800

  Include:

  - A comment field with the words COMMON/MAP DECLARATIONS
  - All COMMON statements
  - All MAP statements

- DIMENSION DECLARATIONS - Line 900

  Include:

  - A comment field with the words DIMENSION DECLARATIONS
  - Local dimension declarations
  - Virtual array dimension declarations

- MAIN PROGRAM - Line 1000

  Include:

  - A comment field with the words MAIN PROGRAM LOGIC or SUBPROGRAM LOGIC
  - The program or subprogram source code
  - A standard default error trap

- LOCAL SUBROUTINES - Line 10000

  Include:

  - A comment field with the words SUBROUTINES LOCAL TO THIS [SUB-]PROGRAM
  - Each subroutine referenced by main the program and described in the subroutines section

- LOCAL FUNCTIONS - 14000

  Include:

  - A comment field with the words FUNCTIONS LOCAL TO THIS [SUB-]PROGRAM
  - Each multi-line function and its definition as described in the function section

- STANDARD ERROR HANDLING - Line 19000

  Include:

  - A comment field with the words STANDARD ERROR HANDLING
  - Error handling routines appropriate for your program logic (error texts, error numbers, branching, and so forth)

- END OF PROCESSING - Line 32000

  Include a comment field with the words END OF PROCESSING to mark the end of the module's main logic

- END OF PROGRAM - Line 32767

  Include:

  - A comment field with the words END OF [SUB-]PROGRAM
  - An END statement, if the program is the main module
  - A SUBEND statement, if the program is a subprogram

F.2 Sample Program Coding Template

This section presents a format for coding BASIC programs. The recommended program order and documenting procedures clarify the program's history, purpose, and logical development. This organization should help the program to run with fewer errors.

```
1       !*********************************************************************&
        !                                                                     &
        !                                                                     &
        ! S U B - P R O G R A M     T I T L E                                 &
        !                                                                     &
        !                                                                     &
        !                                                                     &
        SUB XXXXXX  ! Start sub-program here                                  &
        \ ON ERROR GOTO 19000                                                 &
        !                                                                     &
        !                                                                     &
        !                                                                     &
        ! Version: VV                                                        &
        ! Edit level: EE                                                     &
        ! Edit date: DD-MMM-YY                                               &
        ! Author(s): AAA                                                     &
        !                                                                     &
        !*********************************************************************
```

```
11      !*****************************************************************&
        !                                                                 &
        !                                                                 &
        !   C O P Y R I G H T                                             &
        !                                                                 &
        !                                                                 &
        !                                                                 &
        !    (C) Copyright 1977, 1978, 1979                               &
        !    Digital Equipment Corporation, Maynard, Massachusetts        &
        !                                                                 &
        !    This software is furnished under a license for use only      &
        !    on a single computer system and may be copied only with      &
        !    the inclusion of the above copyright notice. This            &
        !    software, or any other copies thereof, may not be pro-       &
        !    vided or otherwise made available to any other person        &
        !    except for use on such system and to one who agrees to       &
        !    these license terms. Title to and ownership of the           &
        !    software shall at all times remain in DIGITAL.               &
        !                                                                 &
        !    The information in this software is subject to change        &
        !    without notice and should not be construed as a commit-      &
        !    ment by Digital Equipment Corporation.                       &
        !                                                                 &
        !    DIGITAL assumes no responsibility for the use or relia-      &
        !    bility of its software on equipment that is not sup-         &
        !    plied by DIGITAL.                                            &
        !                                                                 &
        !*****************************************************************
15      !*****************************************************************&
        !                                                                 &
        !                                                                 &
        !  C A L L    F O R M A T                                         &
        !                                                                 &
        !                                                                 &
        !                                                                 &
        ! CALL XXXXXX(<argument list, if present>)                        &
        !                                                                 &
        ! Arguments:                                                      &
        !                                                                 &
        ! Name   Description                                              &
        ! ------------------ ------------                                 &
        !                                                                 &
        !*****************************************************************
20      !$****************************************************************&
        !                                                                 &
        !                                                                 &
        ! M O D I F I C A T I O N   H I S T O R Y   L O G                 &
        !                                                                 &
        !                                                                 &
        !                                                                 &
        !    VER/ED DATE INITIAL REASON                                   &
        ! ------ ------ --- ---------------                                &
        !                                                                 &
        !                                                                 &
        !*****************************************************************
100     !$****************************************************************&
        !                                                                 &
        !                                                                 &
        ! S U B - P R O G R A M   D E S C R I P T I O N                   &
        !                                                                 &
        !                                                                 &
        !*****************************************************************
```

```
300     !$*********************************************************************&
        !                                                                      &
        !                                                                      &
        ! C H A N N E L     A S S I G N M E N T S                              &
        !                                                                      &
        !                                                                      &
        !                                                                      &
        ! CHANNEL   ASSIGNMENT                                                &
        ! --------- --------------------                                       &
        !                                                                      &
        !**********************************************************************
400     !$*********************************************************************&
        !                                                                      &
        !                                                                      &
        ! V A R I A B L E S   A N D     A R R A Y S    U S E D                  &
        !                                                                      &
        !                                                                      &
        !                                                                      &
        ! NAME   DESCRIPTION                                                  &
        ! ----   ---------------------------                                   &
        !                                                                      &
        !                                                                      &
        !**********************************************************************
700     !$*********************************************************************&
        !                                                                      &
        !                                                                      &
        !  S U B R O U T I N E S    U S E D                                    &
        !                                                                      &
        !                                                                      &
        !                                                                      &
        ! NAME/LINE   DESCRIPTION                                              &
        ! ----------- ------------------                                       &
        !                                                                      &
        !                                                                      &
        !**********************************************************************
750     !$*********************************************************************&
        !                                                                      &
        !                                                                      &
        !  F U N C T I O N S    U S E D                                        &
        !                                                                      &
        !                                                                      &
        !                                                                      &
        ! LINE  NAME  DESCRIPTION                                              &
        ! ----- ------------------------                                       &
        !                                                                      &
        !                                                                      &
        !**********************************************************************
800     !$*********************************************************************&
        !                                                                      &
        !                                                                      &
        ! C O M M O N       D E C L A R A T I O N S                            &
        !                                                                      &
        !                                                                      &
        !**********************************************************************
900     !$*********************************************************************&
        !                                                                      &
        !                                                                      &
        ! D I M E N S I O N     D E C L A R A T I O N S                        &
        !                                                                      &
        !                                                                      &
        !                                                                      &
        ! Lines 901-929 denote local dimension declarations.                   &
        ! Lines 930-949 denote library dimension declarations.                 &
        ! Lines 950-979 denote MAP statements.                                 &
        !                                                                      &
        !**********************************************************************
```

```
1000    !$*****************************************************************&
        !                                                                  &
        !                                                                  &
        ! M A I N    S U B - P R O G R A M    L O G I C                    &
        !                                                                  &
        !                                                                  &
        !*****************************************************************&
        !                                                                  &
        !                                                                  &
        ! Set up standard default error trap                               &
10000   !$*****************************************************************
        !                                                                  &
        !                                                                  &
        ! S U B R O U T I N E S    L O C A L    To                          &
        !                                                                  &
        !  T H I S    S U B - P R O G R A M                                &
        !                                                                  &
        !                                                                  &
        !*****************************************************************
15000   !$*****************************************************************&
        !                                                                  &
        !                                                                  &
        ! F U N C T I O N S     L O C A L                                    &
        !                                                                  &
        !  T H I S    S U B - P R O G R A M                                &
        !                                                                  &
        !                                                                  &
        !*****************************************************************
19000   !$*****************************************************************&
        !                                                                  &
        !                                                                  &
        ! S T A N D A R D     E R R O R    H A N D L I N G                  &
        !                                                                  &
        !                                                                  &
        !*****************************************************************
19900     PRINT                                                            &
        \ ERR.MESSAGE$ = ERT$(ERR)                                         &
        \ PRINT "??Error ";ERR.MESSAGE$                                    &
        \ PRINT "in ";ERN$;" at line ";ERL                                 &
        \ ERR.MESSAGE$ = ""                                                &
        \ ERR.IND% = -1%                                                   &
        \ RESUME 32767                                                     &
        !                                                                  &
        ! For all unaccounted for errors, print the line and               &
        ! error number, as supplied by the BASIC-PLUS-2                    &
        ! variables, ERR and ERL.                                          &
        ! Exit through the SUBEND statement.                               &
32000   !$*****************************************************************
        !                                                                  &
        !                                                                  &
        ! E N D    O F    P R O C E S S I N G                               &
        !                                                                  &
        !                                                                  &
        !*****************************************************************
32767   !$*****************************************************************&
        !                                                                  &
        !                                                                  &
        ! E N D    O F    S U B - P R O G R A M                             &
        !                                                                  &
        !                                                                  &
        !*****************************************************************&
        !                                                                  &
        \ SUBEND
```

Index

S

BASIC–PLUS–2
RSX/IAS/VMS
User's Guide
AA-0157C-TC
AD-0157C-T1, T2

Reader's Comments

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

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

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number. _____

_____

_____

_____

_____

_____

_____

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

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

—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

ATTN: Commercial Engineering Publications MK1-2/2H3
DIGITAL EQUIPMENT CORPORATION
CONTINENTAL BOULEVARD
MERRIMACK N.H. 03054

— — Do Not Tear - Fold Here and Tape —