pdp11

BASIC–PLUS–2

RSX–11M/IAS

User's Guide

Order No. AA-0157B-TC

digital

September 1978

This manual describes the use of the BASIC-PLUS-2 Compiler on the RSX-11M and IAS operating systems. The description includes compiler commands, linkage of object modules to produce an executable task, RMS Record I/O, and error messages.

BASIC-PLUS-2

RSX-11M/IAS

User's Guide

Order No. AA-0157B-TC

SUPERSESSION/UPDATE INFORMATION:	This manual supersedes the manual of the same name with the order number AA-0157A-TC.	
OPERATING SYSTEM AND VERSION:	RSX-11M	V3.1 and IAS V2.0
SOFTWARE VERSION:	PDP-11 BASIC-PLUS-2	V1.5

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

The following are trademarks of Digital Equipment Corporation:

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

2/79

Contents

Appendix B Run-Time Error Codes and Messages

Appendix C Compile Time Error Messages

Appendix D ASCII Codes and Data Representation

Index

Tables

Figures

Preface

The *BASIC-PLUS-2 RSX-11M/IAS User's Guide* describes the features and use of the BASIC-PLUS-2 Compiler on PDP-11 operating systems.

Chapter 1 describes the BASIC-PLUS-2 command format, debugging aids, and the creation of source programs.

Chapter 2 contains information on RMS (Record Management Services) file handling and Record I/O.

Chapter 3 explains the interface between the BASIC-PLUS-2 Compiler and operating systems that use the MCR command language. It also describes the procedures used to create an executable task from a BASIC-PLUS-2 source program on these systems.

Chapter 4 explains the interface between the BASIC-PLUS-2 Compiler and operating systems that use the DCL command language. It also describes the procedures used to create an executable task from a BASIC-PLUS-2 source program on these systems.

The manual also contains appendixes that describe compatibility issues and the Translator, the BASIC-PLUS-2 vocabulary, error messages and recovery procedures, and data and character representations.

Intended Audience

This manual is not a tutorial. You should be familiar with your operating system and the BASIC-PLUS-2 language before reading this user's guide. Information on the BASIC-PLUS-2 language can be found in the *PDP-11 BASIC-PLUS-2 Language Reference Manual*. Information on system documentation can be found in the Documentation Directory appropriate to your system. In addition, specific sections of this manual refer to other documents that provide information on the subject under discussion.

Documentation Conventions

Throughout this manual, symbols and other notation conventions are used to represent keyboard characters, textual information, and otherwise ease the exposition of material. The symbols and conventions used are explained below:

RET	The RET symbol represents a carriage return/line feed combination.
^	The circumflex represents a control character. For example, ^C indicates a CTRL/C. In some cases, a circumflex is also used to indicate exponentiation.
RED	Color-highlighted information in examples is typed by the user.
"print" and "type"	As these words are used in the text, the system prints and the user types.
BASIC	The term BASIC is used as a generic term for BASIC-PLUS-2. Where this may cause confusion, the practice is discontinued and the proper term is used.
UPPER CASE lower case	In examples of format, information that you type as shown appears in upper-case letters. Lower-case indicates that the information is user dependent.
{braces}	Braces indicate that, of several elements shown, one is chosen.
[brackets]	Brackets indicate user options.
MCR	Monitor Console Routine; the command language used on the RSX-11M operating system.
DCL	Digital Command Language; the command language used on the IAS operating system.

Chapter 1
BASIC–PLUS–2

The BASIC–PLUS–2 Language Processor is composed of a Compiler and an Object–Time System/Library. The OTS/Library is discussed in Section 1.2.9. This chapter contains information on the BASIC–PLUS–2 Compiler. It describes the syntax and use of BASIC commands, editing of programs, debugging aids, and the creation of source files.

1.1 BASIC–PLUS–2 Compiler

The BASIC–PLUS–2 Compiler produces an object module from your source program. As you enter the source program, the compiler checks each program line for syntax errors and returns an appropriate message when an error is found. You can then correct the program (if necessary) and compile it. Program compilation results in an object module that is linked and executed at the operating system command level. The creation and compilation of a source program is detailed in the rest of this chapter. Chapters 3 and 4 discuss the procedures used to link and run a BASIC–PLUS–2 task.

To invoke the BASIC–PLUS–2 Compiler, type a system-dependent command as described in Chapters 3 and 4. If access to BASIC–PLUS–2 is successful, BASIC prints an identification line (see Section 1.2.1) followed by a prompt. This prompt indicates that the compiler is prepared to accept input. Note that the system manager can optionally change the text of this prompt during the installation of BASIC–PLUS–2; Basic2 is the default prompt and is used throughout this manual.

1.2 Commands

Input to the compiler can be a BASIC command or a source program line. BASIC source programs are described in Section 1.4. This section and the subsections that follow describe the BASIC commands.

You use commands to perform various functions outside the context of programs. That is, commands require no line numbers and you type them directly to BASIC, along with any required arguments. Table 1–1 lists the BASIC commands with brief explanations of their use. Succeeding sections describe each command in detail. The commands listed in Table 1–1 can be used individually or combined in a user-created indirect command file. The command file allows you to execute a series of BASIC commands by means of a single command file specification.

Table 1–1: BASIC–PLUS–2 Commands

Command	Function
APPEND	Merges the current source program with a previously saved program.
BUILD	Creates a command file and an overlay description file from your source program. These files are used to specify input to the task builder program.
COMPILE	Translates a BASIC source program into an object module with a default file type of .OBJ.
LOCK/*sw*	Causes the switches you specify (*sw*) to be used as the default for succeeding COMPILE commands. A LOCK command with no arguments disables the specified switches and returns to the BASIC default switch settings.
DELETE	Erases a specified line or lines from a BASIC source program.
EXIT	Clears memory, closes all files, and returns you to operating system command level.
IDENTIFY	Causes BASIC–PLUS–2 to print an identification header.
LIBRARY	Allows you to specify a BASIC–PLUS–2 resident library.
LIST	Prints a copy of the current program or its specified lines.
NEW	Clears memory for the creation of a new program.
OLD	Clears memory and loads a specified existing program into memory.
RENAME	Changes the name of the current program in memory.
REPLACE	Stores the current program on the system default device and directory or a specified device.
SAVE	Copies and preserves a source program on the system default device and directory or a specified device.
SCALE	Controls the scale factor for double-precision (4–word floating-point) format.
UNSAVE	Deletes a specified file.

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

Command/Switch	Default	Function
BUILD/DUMP	/NODUMP	Instructs the Run-Time System to produce a binary dump of memory contents in the event of an abnormal exit from a user program.
BUILD/EXTEND:*n*	/NOEXTEND	Increases program storage by a minimum of *n* words.
BUILD/MAP	Installation Option	Causes the Task Builder to create a memory allocation map file with a default extension of .MAP.
BUILD/IND	/NOIND	Links in the code necessary to use RMS-11 Indexed file organization.
BUILD/REL	/NOREL	Links in the code necessary to use RMS-11 Relative file organization.
BUILD/SEQ	/NOSEQ	Links in the code necessary to use RMS-11 Sequential file organization.
COMPILE/DEBUG	/NODEB	Translates a BASIC source program and enables the debugging aid.
COMPILE/DOUBLE	Installation Option	Translates a BASIC program and enables the double-precision (4-word floating-point) math package.
COMPILE/MACRO	/NOMAC	Translates a BASIC source program into a MACRO source file with a default file extension of .MAC.
COMPILE/LINE	Installation Option	Translates a BASIC source program that uses internal line headers for error processing.

BASIC command specifications can be abbreviated to a minimum of three letters. For example, the COMPILE/DEBUG command can be abbreviated to COM/DEB. Note that if the abbreviation NH is used with the LIST command, NH must be appended to the command abbreviation, i.e., LISTNH. The specific abbreviations for each command are given in the appropriate subsection that follows.

1.2.1 IDENTIFY Command

The IDENTIFY command (IDE) prints a BASIC-PLUS-2 header. The header consists of the BASIC-PLUS-2 name and version number. IDENTIFY eliminates confusion as to which BASIC is currently in effect. That is, an identifying header is printed in response to this command only when the BASIC-PLUS-2 Compiler is present.

Consider the following example:

```
IDENTIFY (RET)
BASIC-PLUS-2     V01-50
```

```
Basic2
EXIT  RET
>
```

In this example, the current availability of BASIC–PLUS–2 is confirmed as a result of typing the IDENTIFY command. After you type EXIT (see Section 1.2.15), BASIC–PLUS–2 is replaced by the operating system command level. An IDENTIFY command to the operating system produces an error because the command is not part of that system's command set. Note that the same identification header is also printed when you first access BASIC–PLUS–2.

1.2.2 NEW Command

The NEW command reserves space for building programs by creating a temporary file. When you type NEW, any name and source code currently in the compiler's buffer or in a temporary file are deleted. After you type the command, BASIC prompts for the new program's name, as follows:

```
NEW  RET
NEW FILE NAME--
```

In response to this prompt, type any 1– to 6–character alphanumeric name.

You can also answer the NEW FILE NAME prompt with a carriage return, in which case BASIC supplies the name NONAME by default.

You may avoid the prompt altogether by typing the desired name after typing NEW. For example, if you type:

```
NEW PROG1  RET
```

BASIC assigns the name PROG1 to the program you create.

In all cases, NEW establishes space for the creation of source files, so the default file type is .B2S. If you specify any other file type in the NEW command, it is ignored.

1.2.3 LIST Command

The LIST command (LIS) prints a copy of the program that is currently in memory. This copy is printed on the terminal output device. It shows the program as it appears in memory with line numbers properly sequenced.

If you type:

```
LIST  RET
```

the entire program is printed, along with a header that contains the program name, the current time and date, and system information. To suppress this header material and print a copy of the program alone, type:

```
LISTNH  RET
```

where NH specifies no header.

You can also specify the printing of specific program lines, instead of the whole program, by means of the line number specification shown in the DELETE command (see Section 1.2.5). For example:

```
LIST 30, 70  RET
```

prints a copy of lines 30 and 70, with a header.

```
LISTNH 30-70  RET
```

prints a copy of lines 30 through 70, without the header.

1.2.4 APPEND Command

The APPEND command (APP) merges the contents of an existing BASIC source program with a program currently in memory (i.e., at compiler command level). To use APPEND, type:

```
APPEND  RET
```

to which the BASIC Compiler prompts:

```
APPEND FILE NAME --
```

In response, type the name of a previously created BASIC source program that you wish to merge with the current program. The compiler opens the specified program as secondary input and reads it into memory. The contents of the source program are then merged with, or appended to, the current program, depending on the order of line numbers. If both programs contain identical line numbers, the current program line is replaced by the appended program line.

To suppress the APPEND FILE NAME prompt, type:

```
APPEND filespec  RET
```

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

If both programs you wish to merge are saved on a system device, one of them must be brought into memory before the APPEND command is given. You bring a saved program into memory with an OLD command (see Section 1.2.10).

If you do not specify a file name in the APPEND command prompt but type only a carriage return, the compiler searches for a source program called NONAME.B2S. If no file of that name is found (either specified or NONAME), the following error message is printed:

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

The APPEND command does not change the name of the program currently in memory.

The following example illustrates the use of the APPEND command. You have built and saved two programs named AP1 and AP2. These programs appear as follows:

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

AP1

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

AP2

If you use an OLD command to bring the program AP1 into memory and then issue an APPEND command for AP2, the result appears as follows:

```
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
```

Note that the APPEND command does not change the name of the current program. Also, line 40 of the program in memory is replaced by line 40 of the appended program while the unique line 35 is merged sequentially.

1.2.5 DELETE Command

The DELETE command (DEL) removes a specified line or lines from the program currently in memory (i.e., at compiler command level).

To delete a program line, type the command followed by the desired line number. To delete a series of lines, specify the line numbers, separated by commas. To delete a consecutive group of lines, type the first and last line number of the group, separated by a hyphen.

For example:

```
DELETE 50  RET
```

removes line 50 from the program.

```
DELETE 50, 80  RET
```

removes lines 50 and 80 from the program.

```
DELETE 50-80 RET
```

removes lines 50 through 80 from the program.

```
DELETE 50, 60, 90-110 RET
```

removes lines 50, 60, and 90 through 110 from the program.

If you do not specify a line in the DELETE command, no lines are removed and an error message (`%Illegal DELETE command`) is returned. If you specify a range of lines and one of the specified lines does not exist, all of the lines within that range are removed. For example, if you type DELETE 50–80, all of the lines equal to, or greater than, 50 and equal to, or less than, 80 are erased. Line numbers must be specified in ascending order; if you type an illegal line specification such as DELETE 80–50, the command is ignored and an error message (`?Bad line number pair`) is returned.

1.2.6 SAVE Command

The SAVE command (SAV) preserves a completed source program by transferring it from memory into a file. For example, if you have a program in memory and type:

```
SAVE RET
```

the line numbers of the program are sequenced, and the program is stored on the system default device in the default directory as source code under the current program name with a .B2S file type. If you wish to specify a particular storage device, directory, file type, version number, or program name, type:

```
SAVE dev:[n,m]filespec
```

where *filespec* is a file specification that contains the desired name and device. If you have built an unnamed program, a SAVE command with no specification stores the program as NONAME.B2S. Note that BASIC–PLUS–2 permits a maximum 6–character name for programs in memory. However, you can specify up to a 9–character file name in the SAVE command. When you access a saved file that has a 6– to 9–character name with an OLD command (see Section 1.2.10), BASIC truncates the name in memory to six characters.

If you attempt to save a program that has the same file specification as one already saved, the system ignores the command and prints an error message:

```
?File exists - RENAME/REPLACE
```

This error prevents an inadvertent deletion of an existing program. For an explanation of RENAME and REPLACE see Sections 1.2.11 and 1.2.12.

1.2.7 COMPILE Command

The COMPILE command (COM) can only translate a program that is currently in memory into object code. This command can be used in conjunction with one or more of the following optional switches: /DEBUG, /DOUBLE, /MACRO, and /LINE. Note that any switch may be turned off by using a NO prefix, as in /NOLINE or /NOMACRO. A LOCK command is also available that allows you to specify default switch settings. Table 1–2 lists the COMPILE command switches and their default values.

When used alone, the COMPILE command translates the program into a linkable object module and stores it in the system default device and directory. The default file type, .OBJ, is appended to the program name. The program is not executed; it is only compiled and saved. Programs compiled as object modules must be linked by the BUILD command into a task image before they can be executed. You construct a task image by using the BUILD command together with appropriate switches and then using the task builder to generate the executable task image.

If the program is currently in memory (i.e., at compiler command level) and you type:

```
COMPILE  RET
```

the current program is compiled and saved. An alternative use of this command is to type:

```
COMPILE filespec
```

where *filespec* is a file specification. This command compiles the current program (which has been previously brought into memory by an OLD command) under the specified name, which can include a directory, device, or version number, and appends .OBJ to the name (if no other file type is specified). To compile a source program that is not in memory, you must first bring it into memory by means of an OLD command (see Section 1.2.10) and then type COMPILE.

The COMPILE/DEBUG command (COM/DEB) translates the program into object code and enables the use of the BASIC–PLUS–2 debugging aid. The debugging aid is described in Section 1.3.1. Note that the program must be compiled with the /DEBUG switch and linked by means of the task builder before the BASIC debugging aid can be used. Note that, because the debugging aid is module-oriented, it does not allow the use of the debugger in a module compiled without the /DEBUG switch.

The COMPILE/DOUBLE command (COM/DOU) translates the program into object code and indicates that double-precision format (4–word) is used for all floating-point operations. Note that an executable task cannot contain both single- and double-precision format. That is, all modules in the task must be the same format; mixed format causes a run-time error. Your system manager selected either single- or double-precision at the system default when BASIC-PLUS-2 was installed.

The COMPILE/MACRO command (COM/MAC) translates the program and saves it only as a MACRO source file with a .MAC default file type. This file can be listed to examine the compiler-generated code. It is generally used for diagnostic purposes.

The COMPILE/NOLINE command (COM/NOLIN) translates the program and reduces the memory requirements of the output program. The /NOLINE switch is an installation option that disables program line headers in memory and reduces program requirements by the following amounts:

- 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

The /NOLINE switch cannot be used when the compiled program references an ERL function, makes use of the debugging aid, or contains a RESUME statement without a line number specification. When the /NOLINE switch is enabled, the ERL value is set to 0. Note that a RESUME statement without a line number specification overrides the /NOLINE switch and causes a diagnostic error message:

```
%RESUME overrides /NOLINE
```

Also, a reference to the ERL function overrides the /NOLINE switch and causes a diagnostic error message:

```
%ERL overrides /NOLINE
```

In most cases, the switches described above can be combined in the COMPILE command. For example:

```
COMPILE/DEBUG/DOUBLE/NOLINE
```

You can use the LOCK command to facilitate multiple program compilations. That is, you can specify any legal combination of compiler switches to the LOCK command, and these become the defaults for successive COMPILE commands. This procedure avoids your having to respecify switches for each compilation. The specified switches are disabled by a LOCK command with no arguments. Note that a COMPILE command with no arguments creates an object file by default.

Consider the following example:

```
LOCK/NOLINE  RET
Basic2
OLD PROG1  RET
Basic2
```

```
COMPILE  RET
Basic2
OLD PROG2  RET
Basic2
COMPILE  RET
Basic2
LOCK  RET
Basic2
OLD PROG3  RET
Basic2
COMPILE/MACRO  RET
Basic2
```

In this example, three programs are brought into memory by OLD commands (see Section 1.2.10). The initial LOCK command sets the /NOLINE compiler switch as the default. When you compile PROG1 and PROG2, they become object modules with /NOLINE enabled. Finally, the LOCK command with no arguments disables all defaults (except those specified at installation) and PROG3 is compiled as a MACRO file with no switches other than installation option defaults in force. The result of these three compilations is as follows:

PROG1.OBJ (NOLINE enabled)
PROG2.OBJ (NOLINE enabled)
PROG3.MAC (no switches enabled)

1.2.8 BUILD Command

The BUILD command (BUI) accepts the names of one or more object modules as input and creates an indirect command file with the default file type .CMD. This file contains all of the task builder command input required to create an executable task image file with a default file type of .TSK and an optional memory allocation map with a default file type of .MAP. In addition to the command file, the BUILD command generates an overlay description language file (file type .ODL). You can edit the contents of this file to use overlaid program segments. The procedure used to input the command file to the task builder and to edit the BUILD command ODL file is described in Chapters 3 and 4.

An object module is created from a BASIC source program by the COMPILE command (see Section 1.2.7). You create object modules and link them by the task builder for the following reasons:

1. To create an executable task — In order to create a task from your object modules that is executable at the operating system level, you must use the task builder to process the modules and link the required BASIC-PLUS-2 library.

2. To produce an optional memory allocation map — The map is a file that contains descriptions of program code, storage allocation, and global symbol definitions.
3. To link subprograms — User subprograms must be separately compiled as object modules and selectively linked with your program to create a single executable file.
4. To access RMS required code — I/O operations on virtual, sequential, relative, or indexed files (see Chapter 2) require access to RMS library modules. To link this code with modules that use these operations, you must use the task builder.

The BUILD command generates all of the command input required by the task builder system program. This input includes a task and map file output specification, the object module names, and the required BASIC-PLUS-2 library (see Section 1.2.9). Because the BUILD command automatically creates an indirect command file that contains all of this information, task builder input can consist entirely of the indirect command file name. That is, the task builder can link the object modules and output an executable task image file and an optional map file from a single command file specification. If you wish to link your program with special task builder options, you must modify the BUILD command output as described in the *RSX-11M Task Builder Reference Manual* and in Chapters 3 and 4 of this manual.

To use the BUILD command, type:

```
BUILD main,sub1,sub2,,,,/sw
```

where *main* represents the name of a program that was previously compiled as an object module. This file name becomes the name of the indirect command file with the .CMD default file type appended to it. *Sub1, sub2, etc.*, represent the names of one or more optional subprograms, separated by commas, that have been separately compiled as object modules. Note that BUILD command arguments are file specifications that can contain device, directory, and version number specifications. You may specify up to eight modules in the command line but they must all fit on a single line (i.e., the command line cannot be continued). If any of the modules contain an OPEN statement, you must append the appropriate switch(es) to the end of the command line. The switches and their use are as follows:

/VIR — includes in the command file the RMS code required for virtual array or block I/O operations. Note that this switch is used when the program contains only virtual file operations.

/SEQ — includes in the command file the RMS code required for sequential file operations.

/REL — includes in the command file the RMS code required for relative file operations.

/IND — includes in the command file the RMS code required for indexed file operations.

/DUMP — instructs the Run-Time System to produce a binary dump of memory contents at the time of an abnormal exit if your system has the Post Mortem Dump (PMD) program installed.

/MAP — produces a memory allocation map of the resulting program.

/EXTEND:*n* — increases the program storage space by a minumim of *n* words (where *n* is rounded up to the next multiple of 32) and aligns the resulting extended program on a 1K boundary.

You can use any combination of the above switches on the command line, depending on the content of the modules. That is, if any module in the command line creates or opens a virtual, sequential, relative, or indexed file, the appropriate switch(es) must be appended. Because the code required for virtual file operations is a subset of /SEQ, /REL, and /IND, the virtual file switch may be omitted when using any other RMS switch. For information on RMS file operations, refer to Chapter 2. You may use the /DUMP, /MAP, and /EXTEND switches on the same line as any combination of the other switches.

Consider the following:

```
BUILD MAIN,SUB1,SUB2/REL  RET
```

This command line results in an indirect command file (MAIN.CMD) and an overlay description file (MAIN.ODL). These files contain the object modules MAIN.OBJ, SUB1.OBJ, and SUB2.OBJ, as well as the BASIC-PLUS-2 library specifications. In addition, the /REL switch generates instructions that cause the code required for RMS relative file operations to be associated with the task. To produce a linked task and map file, you must invoke the system task builder and specify the command file as input. To specify MAIN.CMD for example, type:

```
>@MAIN  RET
```

in response to your system's task builder prompt. Note that a BUILD command file cannot be combined with any other task builder input.

Following successful task creation, your user directory contains an executable task image file (MAIN.TSK) composed of the linked modules you specified as input. Your directory may also contain a memory allocation map (MAIN.MAP). The file name for both the task image and map is the name of the first module appearing as input in the BUILD command line. The actual linking operation is handled by the task builder. For more information on the task builder, refer to Chapters 3 and 4 of this manual and to the *RSX-11M Task Builder Reference Manual*. If you use the BUILD/DUMP command, you instruct the system to produce a dump of memory contents at the time of an abnormal exit during execution.

1.2.9 LIBRARY Command

You can optionally link to a user-created library, depending on the needs of your program. You use the LIBRARY command (LIB) to indicate BASIC2 or a user library. The command includes that library in the command file that is generated by the BUILD command (see Section 1.2.8).

When linking to user libraries, you should be aware that BASIC–PLUS–2 is supplied with a shareable resident library called BASIC2, which contains many of the run-time support routines required for an executable task image file. The reference to this shared library was specified by your system manager during installation and is included in your task builder command file.

The BASIC2 shareable library is 8K words long and contains the following run-time routines:

1. Math routines, which include library functions and arithmetic routines.
2. Routines to handle dynamic allocation of string storage and I/O buffers.
3. Routines to handle input/output operations.
4. Error handling routines to process errors in arithmetic, I/O, and system operations.

To use the LIBRARY command, type:

```
LIBRARY  RET
```

In response to the command, BASIC prompts for the name of the desired shareable resident library. For example:

```
LIBRARY  RET
Name[BASIC2]--  RET
Account[LB:[1,1]]--  RET
```

This example causes BASIC2 to be used in all succeeding BUILD commands. If the LIBRARY command is successful, the BASIC2 prompt is printed. If you follow this procedure with a BUILD command, the generated command file contains the BASIC2 library as well as any specified object modules (see Section 1.2.8). BASIC2 remains the BUILD command default library until you replace it by means of another LIBRARY command or exit from BASIC–PLUS–2.

1.2.10 OLD Command

The OLD command allows you to bring into memory a previously created and saved source program. When you type:

```
OLD  RET
```

BASIC replies:

```
OLD FILE NAME--
```

In answer to the prompt for a name, type the name of the program you wish to access. This command causes the highest version of the specified file, with a .B2S file type, to be read into memory and become the current program. The program is now ready for processing (i.e., editing, compiling, etc.).

If you type only a carriage return in response to the prompt, BASIC searches for a source program called NONAME.B2S. You can avoid the OLD FILE NAME prompt by specifying the desired program with the OLD command, as follows:

```
OLD filespec
```

where *filespec* is a file specification. If you specify a file specification that does not exist, or if you do not specify a program and NONAME.B2S cannot be found, BASIC returns an error message:

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

When you type the OLD command, any source code currently in memory is lost. Also, when BASIC reads in the specified file, it uses the first six characters as the program name and performs a minimal check on the contents.

1.2.11 RENAME Command

The RENAME command (REN) changes the name of the program currently in memory. For example, if you have a program in memory named PROG1 and you type:

```
RENAME PROG2
```

the name PROG1 is erased from memory and replaced with the name PROG2. If you type SAVE (see Section 1.2.6), the program is stored with the name PROG2.

If you bring a saved program named PROG1 into memory with an OLD command and type:

```
RENAME PROG2
```

the program is named PROG2 in memory but retains the name PROG1 on the disk.

1.2.12 REPLACE Command

The REPLACE command (REP) updates a program on the system default device or a specified device with one in memory. For example, if a program named FILE needs modification, bring it into memory with an OLD command, make the desired changes, then type:

```
REPLACE (RET)
```

This procedure updates the contents of the original program named FILE with the contents of the newly edited program.

You can also specify a new name, directory, device, or version number for the edited program in the REPLACE command. For example:

```
REPLACE [50,20]FILE1.B2S;2
```

where FILE is the name of the program currently in memory, retains the old version of FILE but also saves the edited version under the name FILE1.

The REPLACE command stores the program even if there is no program of the same name on disk. That is, if the program named FILE is currently in memory and there are no other programs with that name, REPLACE still writes the program onto the default device and directory.

1.2.13 SCALE Command

The SCALE command (SCA) implements and controls the scaled arithmetic features of BASIC-PLUS-2. You use SCALE to overcome accumulated round-off and truncation in fractional computations performed when double precision (4-word floating-point) format is enabled. SCALE allows you to maintain the decimal accuracy of fractional computations to a given number of places determined by the scale factor.

To specify a scale factor, type:

```
SCALE int
```

where *int* is a decimal integer in the range of 0 to 6 that represents the scale factor. The command causes the specified scale factor to be used for succeeding compilations. The scale factor remains in effect until you exit from BASIC-PLUS-2 or specify a new SCALE factor. Note that a SCALE command with no factor specification causes BASIC to print the current scale factor.

1.2.14 SHOW Command

The SHOW command allows you to display the current switch values on your terminal. To use the SHOW command, type:

```
SHOW RET
```

following a BASIC prompt. BASIC then prints the following lines on your terminal with the appropriate values reported:

```
Library is LB:[1,1]BASIC
Task extend size = 0
Scale factor = 0
Switch settings:
          NO:MAP
          NO:DUMP
             :CHAIN
             :LINE
      Output:OBJ
   Precision:Single
          NO:DEBUG
   File ORGS:Terminal I/O only
```

The above example shows the default settings for BASIC–PLUS–2. This includes the default values of options that can be determined at installation time. Possible alternate values may consist of the following:

1. Library may be BASIC2 or user-supplied.
2. Task extend can be any integer.
3. Scale factor can be any integer from 0 to 6.
4. Switch settings can be on (:LINE) or off (NO:MAP).
5. Output can be OBJ or MAC.
6. Precision can be single or double.
7. File ORGs can be any combination of Virtual, Relative, Indexed, Sequential, or Terminal-Format.

1.2.15 UNSAVE Command

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

```
UNS  RET
```

the file associated with the source program currently in memory is deleted from your directory on the default device. If you type:

```
UNSAVE filespec
```

the specified file, *filespec*, is deleted from the default device or specified device whether or not it is currently in memory. This command is useful for erasing unwanted files from the default device or other specified devices or directories. Note that you may use UNSAVE to produce a hard copy listing of the currently OLDed file by specifying the device as LP:.

The UNSAVE command causes BASIC to search for and delete a specified source program. If the program is not found, BASIC prints an error message:

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

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

```
UNSAVE DK1:[26,12]FILE.TSK
```

1.2.16 EXIT Command

The EXIT command (EXI) terminates access to BASIC–PLUS–2 and returns you to the operating system command level. This command is the only means of leaving BASIC–PLUS–2 that ensures proper closing of files and the immediate return of control to the operating system.

1.3 Editing BASIC–PLUS–2 Programs

There are a number of ways you can correct BASIC–PLUS–2 source programs. These editing methods include deleting incorrect characters and retyping entire program lines. However, programs must be in memory before edits can be made. That is, you edit a new program as it is entered, or a saved program after it is brought into memory by an OLD command. You cannot edit a task or an object module.

As you create new programs, you can erase misspelled words or incorrect characters with the DEL (Delete) key and type corrections at the terminal. (Note that DEL is labeled the RUBOUT key on some terminals.) This must be done before you enter the line into memory with a carriage return. For example, to correct a misspelled PRINT statement:

```
10 PRAND
```

erase the incorrect characters with the DEL key and retype as follows:

```
10 PRAND\DNA\INT
```

Press the DEL key once for each character you wish to delete (these characters usually print inside slashes on the terminal); then type the correct character(s) on the same line. Note that the DEL key erases characters one at a time from right to left beginning with the last character typed. You can then type a carriage return to enter the corrected line into memory.

To delete an entire line that has not been entered into memory (i.e., you have not yet typed a carriage return), use CTRL/U. That is, you press the CONTROL key and the U key simultaneously.

As you enter source lines into memory, the BASIC Compiler performs a syntax check. If BASIC detects an incorrect line, it prints the appropriate error message following input (see Appendix C). However, BASIC saves source program lines even with errors. To edit an incorrect line that has been entered into the program currently in memory, retype a corrected version of the line.

By typing the same line number followed by corrected text, you delete the old, incorrect line from memory and automatically replace it with the new one. Consider the following example:

```
10 LAD A=7\B=9\C=SRQ(144)  RET
?Syntax error
```

This incorrect line was entered into memory by the carriage return and an error message was printed. If you type:

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

the previous line 10 is erased from memory and replaced with the corrected version.

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

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

can be deleted from the source program by typing:

```
10  RET
```

Also, you can use the DELETE command to perform the same function (see Section 1.2.5).

1.3.1 Debugging

To help you locate any errors that may exist in your program, BASIC provides a set of interactive debugging commands. These commands allow you to check program operation and make corrections. The commands are BREAK, UNBREAK, STEP, TRACE, UNTRACE, PRINT, LET, CONTINUE, ERR, ERN$, ERL, STATUS, and RECOUNT. Their use is permitted only on programs or subprograms that are compiled with the /DEBUG switch (see Section 1.2.7) and linked by means of the task builder. After you have debugged the program and edited the source file to execute correctly, you can recompile the program without the /DEBUG switch to disable these commands. Note that the /DEBUG switch causes an increase in program memory requirements, therefore, recompiling the program without /DEBUG acts to conserve memory.

Note that when a program is composed of several subprograms, you do not have to compile each subprogram with the /DEBUG switch. To debug a single subprogram, the switch need only be enabled with that module.

When you run a program, execution stops the first time a module is entered that has the /DEBUG switch enabled. After execution halts, the debugging aid prints an identifying message:

```
DEBUG: prog name
```

where *prog name* is the name of the program or subprogram that was compiled with the /DEBUG switch. The debugging aid also prints a prompt (#) after the message as follows:

```
DEBUG: prog name
#
```

The prompt allows you to enter debugging aid commands. The debugging commands allow you varying degrees of control over program execution as explained in the following sections. If you enter a carriage return in response to a debugger prompt, a STEP 1 is performed (see section 1.3.1.2). This enables you to "single step through" a program by typing only carriage returns. To reinitiate program execution and cause the specified command action, type the CONTINUE (CON) command as follows:

```
DEBUG: prog name   RET
# BREAK 10   RET
# CON   RET
```

In this instance, the CON command reinitiates program execution as specified by the BREAK command, i.e., the program runs until line number 10 is executed. Note that the STEP command causes immediate execution of the first encountered statement and does not require the CONTINUE command.

Following the successful execution of a debugging command, a message is printed that identifies your current position in the program or subprogram. This message has the form:

```
command AT LINE n [,name]
```

command is the last executed debugging command, i.e., BREAK, STEP, TRACE, etc.

n is your current line number position in the program or subprogram.

name is the name of the currently executing subprogram. Note that this name does not appear if you are currently executing the main program.

After this message is printed, the # prompt is reissued.

To terminate the debugging process, type EXIT (see Section 1.2.16). This command terminates the debugger and returns you to operating system command level.

1.3.1.1 BREAK and UNBREAK Commands — You type the BREAK command in response to a debugging aid prompt as follows:

```
# BREAK arg
```

where *arg* is a command argument that causes a halt at a specified point in a program or subprogram compiled with the /DEBUG switch. The halts that

are set by a BREAK command argument are called breakpoints and their specification takes one of the following forms:

BREAK	a command with no argument sets a breakpoint at each program line number. Execution halts at each line number and the # prompt is reissued.
BREAK *n*	where *n* is a line number. Execution halts and the debugging prompt is issued whenever that line number is encountered.
BREAK *n*;	where *n* is a line number. The semicolon specifies that line number *n* is a breakpoint only in the currently executing program or subprogram.
BREAK *n*;*name*	where *n* is a line number. The semicolon followed by a module name (*name*) specifies that line number *n* is a breakpoint only in the named program or subprogram.

You can specify a maximum of 10 breakpoints as arguments in the BREAK command. When more than one argument is specified, they must be separated by a comma. For example:

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

This example causes execution to halt at the following points:

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

If you specify more than 10 breakpoints, the excess are ignored and an error message is printed:

```
?No room
```

To disable the breakpoints, use the UNBREAK command. This command has the same general format as BREAK, that is:

UNBREAK	a command with no arguments disables all breakpoints.
UNBREAK *n*	disables the breakpoint set at line number *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 module.

Note that, as in the BREAK command, you can specify a maximum of 10 breakpoints separated by commas in the UNBREAK command.

In addition to line number breakpoints, the BREAK command also allows you to specify a halt on CALL statements, user-defined functions, and loops. The BREAK arguments for these halts are CALL, DEF, and LOOP respectively, and they set breakpoints as follows:

```
BREAK ON {CALL}
         {DEF }
         {LOOP}
```

CALL causes a halt in execution each time a CALL statement is executed to a subprogram that is compiled with the /DEBUG switch. The break occurs immediately before the execution of the subprogram's first statement.

DEF causes a halt in execution each time the program executes a user-defined function. The break occurs immediately before the execution of the function, not at the declaration of the DEF statement.

LOOP causes a halt in execution each time a FOR, WHILE, or UNTIL statement or modifier is encountered. Breaks occur after the loop is initialized, immediately before execution of the loop body, and after exit from the loop. For example, if you have a FOR loop that is executed 10 times, you get 13 breaks.

Note that the BREAK ON command allows you to specify only one argument and this command can be combined with other breaks. For example:

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

This example causes execution to halt at the following points:

1. Line 45 whenever it is encountered in a /DEBUG enabled module, 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 module.

1.3.1.2 STEP Command — The STEP command causes program execution to proceed on a statement-by-statement basis. You type the command in response to the debugger prompt as follows:

```
# STEP n
```

STEP a command with no arguments causes execution of the next statement in the current program or subprogram.

n specifies the number of statements to be executed.

As with other debugging commands, the STEP command has effect only on programs or subprograms that are compiled with the /DEBUG switch. There-

fore, the statement executed by the STEP command is the first statement encountered in a /DEBUG enabled module. Note that typing a carriage return is equivalent to typing STEP 1.

The optional argument, *n*, must be a positive integer in the range of 1 to 32767.

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

The PRINT command has the form:

```
# PRINT var
```

where *var* is the name of the variable whose content you wish to examine. When this command is executed, the current content of the variable is printed. Note that you can specify only one variable as an argument in the PRINT commands.

The LET command has the form:

```
# LET var=value
```

where *var* is the name of the variable whose content you wish to change. The maximum length of a LET is 72 characters. The PRINT and LET debugging commands allow constants or variables as arguments, however, they do not allow expressions. The following are examples of legal LET commands:

```
# LET A$(I,J%)= B$(Q%,Z)   RET
# LET I% = A(I%,I%)   RET
```

1.3.1.4 TRACE and UNTRACE Commands — The TRACE command allows you to track the execution of a program or subprogram that is compiled with the /DEBUG switch. You can examine the path of execution by means of line numbers. You type the command in response to the debugger prompt as follows:

```
# TRACE
```

TRACE — prints the line-number and module-name of each line as it is executed. This command does not accept an argument.

You must enter a CONTINUE command to initialize the TRACE. It is advisable that you enter a break at some point in your program; otherwise, it will TRACE to the end of the program and EXIT. To disable the TRACE command, type UNTRACE in response to the # prompt.

1.3.1.5 ERR Command — The ERR command allows you to display the error number of the last trapped error. Type the command in response to the debugger prompt as shown:

```
# ERR
```

ERR is the command, without arguments, that 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.

1.3.1.6 ERL Command — The ERL command allows you to display the line number of the last trapped error. Type the command in response to the debugger prompt as shown:

```
# ERL
```

ERL is the command, without arguments, that displays the line number of the last error in the format:

ERL = *nn*

where *nn* is the line number containing the error.

1.3.1.7 ERN$ Command — The ERN$ command allows you to display the name of the module that contains the last trapped error. Note that ERN$ does not return a value unless an error has occurred. Type the command in response to the debugger prompt as shown:

```
# ERN$
```

ERN$ is the command name, without arguments, that returns the name of module containing the last trapped error, in the format:

ERN$ = *mod nam*

where *mod nam* is the six character module name.

1.3.1.8 RECOUNT Command — The RECOUNT (REC) command allows you to display the number of characters that are provided for the preceding input operation. Type the command in response to the debugger prompt as shown:

```
# RECOUNT
```

RECOUNT is the command, without arguments, that returns 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.

1.3.1.9 STATUS Command — The STATUS (STA) command allows you to display the status word containing characteristics of the last OPENed file. Type the command in response to the debugger prompt as shown:

```
# STATUS
```

STATUS is the command, without arguments, that returns a word containing the last opened file's characteristics in the format:

STATUS = *nn*

where *nn* is an additive form of the following:

1 - record-oriented device

2 - carriage-control device

4 - terminal

8 - multiple-directory device (disk)

16 - single-directory device

32 - sequential- and block-oriented device (magnetic tape)

1.4 BASIC–PLUS–2 Programs

A BASIC–PLUS–2 source program is composed of numbered lines that contain BASIC language elements as follows:

```
line-number <tab> text   RET
```

where the symbol RET represents the RETURN key that generates a carriage return/line feed terminator. In addition to a carriage return/line feed combination, BASIC–PLUS–2 accepts an escape (ESC key) as a line terminator.

A BASIC–PLUS–2 line number must be a positive number in the range of 1 to 32767. If you type a line number that is outside the legal range, the number is ignored and BASIC prints an error message:

```
?Illegal line number
```

A line number with no text is considered to be a line deletion (see Section 1.3). Text with no line number (except for legal commands and continuation lines) is ignored and BASIC prints an error message:

```
?What?
```

The BASIC Compiler checks each source program line for correct syntax, returns a message for errors, and saves the line even if errors are found. The lines are saved in ascending numeric order and are executed in the same order.

BASIC–PLUS–2 programs do not require an END statement.

1.4.1 Source Lines

BASIC source lines can contain multiple statements on a single line. However, you must separate multiple statements with a backslash (\). For example:

```
10      LET A=5\B=7\C=9
```

BASIC source lines can also be continued over more than one line. You signify continuation by typing the character "&" (ampersand) and a line terminator. The following is a valid continued line:

```
10      LET A=5\B=7    &
\       C=A+B
```

Because the ampersand signifies a continued line to the compiler, you cannot use this character as the last non-blank character of a non-continued line.

You can place comments in BASIC source lines by using an exclamation point separator (!). Comments in a line are printed when the program is listed, but are ignored when the program executes. You can place a comment at any point on the line as long as it is separated from any other element of the line by the exclamation point separator (!).

Consider the following:

```
5       !THIS IS A LEGAL COMMENT
10      LET A=10 !SO IS THIS! \!LET B=5
20      LET A=10 \B=5 !AND THIS
```

Note that a comment separator cannot take the place of a statement separator. That is, backslashes are always required on multi-statement lines. Also, comments cannot be continued with an ampersand; each program line must begin comments with an exclamation point. You can, however, include the comment in a REM statement which, as with any statement, can be continued.

BASIC accepts any character in text as long as it is part of the ASCII character set. A table of the ASCII characters appears in Appendix D. Null characters are ignored as meaningless; however, non-printing characters (space, tab, etc.) are accepted in literal string constants. A warning message is issued for non-printing characters that appear outside of string literals. Also, the compiler treats lower-case alphabetics in line text as upper case, but lower-case alphabetics in literal strings remain lower case.

BASIC accepts integers in the range of -32767 to +32767. The value of subscript variables is in the range of 0 to +32767. Single precision (2-word) floating-point values are rounded down to seven digits of accuracy and lie in the range of .29 x 10^-38 to .17 x 10^39. Double precision (4-word) floating-point values are rounded down to 17 digits of accuracy and lie in the range of .29 x 10^-38 to .17 x 10^39. For more information on data representation, see Appendix D.

1.4.2 Subprograms

BASIC-PLUS-2 allows you to write subprograms and insert them into OTS or user libraries. These subprograms can be written in BASIC-PLUS-2 or in MACRO assembly language. This section describes the subprogram calling conventions and linkage. It also describes the creation of an assembly language subprogram; for information on writing BASIC-PLUS-2 subprograms, refer to the *PDP-11 BASIC-PLUS-2 Language Reference Manual.*

MACRO subprograms used with BASIC-PLUS-2 are subject to the following restrictions:

1. MACRO subprograms cannot call BASIC-PLUS-2 subprograms.
2. Virtual arrays cannot be passed to MACRO subprograms.
3. MACRO subprograms that use RMS or FCS I/O cannot employ any LUN or event flag used by the BASIC-PLUS-2 program, nor execute any operation that alters the RMS dynamic space pool (i.e., $OPEN, $CONNECT, $CLOSE, or $DISCONNECT).
4. MACRO subprograms cannot create or dynamically alter strings.

Note that if the MACRO subprogram requires a string, you must use BASIC-PLUS-2 to create the string and define its size before the MACRO subprogram uses it.

BASIC-PLUS-2 subprogram calls are subject to the following restrictions:

1. BASIC-PLUS-2 can call a BASIC-PLUS-2 subprogram with a CALL statement. BASIC-PLUS-2 can call a MACRO subprogram with either a CALL or CALL BY REF statement.
2. BASIC-PLUS-2 can call a MACRO subprogram that is also callable from FORTRAN with a CALL BY REF statement.
3. BASIC-PLUS-2 cannot call a FORTRAN subprogram.
4. BASIC-PLUS-2 cannot be called by a MACRO or FORTRAN subprogram.
5. The maximum allowable number of arguments in a BASIC-PLUS-2 subprogram is eight.
6. BASIC-PLUS-2 can call system directives that are also callable from FORTRAN.

1.4.2.1 Subprogram Linkage — BASIC-PLUS-2 programs call MACRO subprograms with the following instruction:

```
JSR    PC,routine
```

where JSR is a Jump to Sub instruction and PC is the Program Counter.

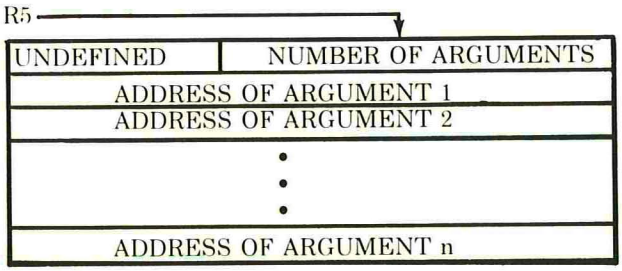
The instruction used to return control from the subprogram to the calling program is:

```
RTS    PC
```

where RTS is the Return from Sub instruction.

Arguments are passed from BASIC-PLUS-2 programs to MACRO subprograms in the form of an argument list. When the MACRO subprogram starts, register 5 (R5) contains the address of an argument list as shown in Figure 1-1.

Figure 1-1: Argument List Format



1.4.2.2 Subprogram Register Usage — A MACRO subprogram that is called by a BASIC-PLUS-2 program does not need to preserve any registers. However, register 6 (SP) must point to the same location on entry to, and exit from, the subprogram. That is, each "push" onto the stack must be matched by a "pop" from the stack before the subprogram returns control to the BASIC-PLUS-2 program.

1.4.2.3 Subprogram Calls — Arguments can be passed to a MACRO subprogram by means of either a CALL or CALL BY REF statement. These statements are used to pass integer, real (single precision), and double (double precision) values, strings and arrays. The methods used to pass integer, real, and double value arguments are the same for CALL and CALL BY REF. However, these two statements differ in their method for passing string and array arguments. Refer to Appendix D for a description of data formats.

In terms of the content of the argument list in R5, the passing mechanism is as follows:

Integer — The R5 argument list contains the address of the integer value.

Real — The R5 argument list contains the address of the high-order word for the single precision value.

Double — The R5 argument list contains the address of the high-order word for the double precision value.

String — When CALL is used, the R5 argument list contains the address of a 2-word string header. The first word is the address of the first byte in the string. The second word is the length of the string in bytes.

When CALL BY REF is used, the R5 argument list contains the address of the first byte in the string; the string length is not available.

Array When CALL is used, the R5 argument list contains the address of the second word in the array header. The array header contains subscript information and the address of the first byte of the array.

When CALL BY REF is used, the R5 argument list contains the address of the first element in the array; the array header is not available. Note that if an element of an array is specififed, the value of that element is stored in a temporary variable. The address of the temporary variable is passed when the statement is executed.

Consider Figures 1–2 and 1–3. These figures are examples of two MACRO subprograms and illustrate the methods used to pass arguments to subprograms. Figure 1–2 is an example of the use of the CALL statement. Figure 1–3 is an example of the use of the CALL BY REF statement.

Figure 1–2: CALL Statement

```
        .TITLE INSRT
;
;               CALL INSRT(A$,B$,C%)
;
; INPUTS:       ARG1 = ADDRESS OF A$ STRING HEADER
;               ARG2 = ADDRESS OF B$ STRING HEADER
;               ARG3 = ADDRESS OF C%
;
; OUTPUTS:      C% = 0 IF OPERATION WAS SUCCESSFUL
;                  = -1 IF OPERATION FAILED
;                  = UNCHANGED FROM CALL IF WRONG NUMBER OF ARGUMENTS
                   PASSED
;
; EFFECTS:      THIS SUBPROGRAM OVERWRITES THE SUBSTRING B$ INTO THE
;               STRING A$ BEGINNING AT CHARACTER POSITION C%.
;               RETURNS 0 IN C% IF THE OPERATION WAS SUCCESSFUL.
;               RETURNS -1 IN C% IF THE OPERATION FAILED.
;
INSRT::
        CMPB    #3.,@R5         ; NUMBER OF ARGUMENTS
        BNE     ERREX
        MOV     2(R5),R0        ; R0 = ADDRESS OF A$ STRING HEADER
        MOV     4(R5),R1        ; R1 = ADDRESS OF B$ STRING HEADER
        MOV     @6(R5),R2       ; R2 = C%
        BLE     ERREX           ; BR TO ERROR IF C% <= 0
        ADD     2(R1),R2        ; R2 = C% PLUS LENGTH OF B$
        CMP     R2,2(R0)        ; WILL B$ FIT INTO A$ ?
        BGT     ERREX           ; BR TO ERROR IF B$ WON'T FIT INTO A$
        MOV     @R0,R0          ; R0 = ADDRESS OF A$
        MOV     @6(R5),R2       ; R2 = C%
        DEC     R2              ; R2 = C% MINUS ONE
        ADD     R2,R0           ; R0 = ADDRESS IF A$ PLUS C%
        MOV     2(R1),R2        ; R2 = LENGTH OF B$
        BEQ     ERREX           ; BR TO ERROR IF LENGTH OF B$ = 0
        MOV     @R1,R1          ; R1 = ADDRESS OF B$
1$:     MOVB    (R1)+,(R0)+     ; INSERT A CHARACTER INTO A$ FROM B$
        SOB     R2,1$
        CLR     @6(R5)          ; SET C% TO 0 (OPERATION SUCCESSFUL)
        RETURN

ERREX:  MOV     #-1,@6(R5)      ; SET C% TO -1 (OPERATION FAILED)
        RETURN
        .END
```

Figure 1-3: CALL BY REF Statement

```
        .TITLE INSRT
;
;               CALL INSRT BY REF(A$,LEN(A$),B$,LEN(B$),C%)
;
; 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 WAS SUCCESSFUL
;                  = -1 IF OPERATION FAILED
;                  = UNCHANGED FROM CALL IF WRONG NUMBER OF ARGUMENTS
                   PASSED
;
; EFFECTS:      THIS SUBPROGRAM OVERWRITES THE SUBSTRING B$ INTO THE
;               STRING A$ BEGINNING AT CHARACTER POSITION C%.
;               RETURNS 0 IN C% IF THE OPERATION WAS SUCCESSFUL.
;               RETURNS -1 IN C% IF THE OPERATION FAILED.
;
INSRT::
        CMPB    #5.,R5          ; NUMBER OF ARGUMENTS
        BNE     ERREX
        MOV     @12(R5),R2      ; R2 = C%
        BLE     ERREX           ; BR TO ERROR IF C% <= 0
        ADD     @10(R5),R2      ; R2 = C% PLUS LENGTH OF B$
        CMP     R2,@4(R5)       ; WILL B$ FIT INTO A$ ?
        BGT     ERREX           ; BR TO ERROR IF B$ WON'T FIT INTO A$
        MOV     2(R5),R0        ; R0 = ADDRESS OF A$
        MOV     @12(R5),R2      ; R2 = C%
        DEC     R2              ; R2 = C% MINUS ONE
        ADD     R2,R0           ; R0 = ADDRESS IF A$ PLUS C%
        MOV     @10(R5),R2      ; R2 = LENGTH OF B$
        BEQ     ERREX           ; BR TO ERROR IF LENGTH OF B$ = 0
        MOV     6(R5),R1        ; R1 = ADDRESS OF B$
1$:     MOVB    (R1)+,(R0)+     ; INSERT A CHARACTER INTO A$ FROM B$
        SOB     R2,1$
        CLR     @12(R5)         ; SET C% TO 0 (OPERATION SUCCESSFUL)
        RETURN

ERREX:  MOV     #-1,@12(R5)     ; SET C% TO -1 (OPERATION FAILED)
        RETURN
        .END
```

1.4.3 BASIC–PLUS–2 Sample Program

The following example summarizes the building of BASIC source programs.

```
IDENTIFY  RET
BASIC-PLUS-2  V01.50
Basic2

NEW  RET
NEW FILE NAME--SORT02  RET

Basic2

10  DIM SORT(100)                          !MAX NUMBER OF ELEMENTS RET
20  INPUT "NUMBER OF ENTRIES"; CNT%        !GET NUMBER OF ELEMENTS          &RET
\   IF CNT% <2% OR CNT%> 100%              !CHECK CORRECT NUMBER            &RET
        THEN PRINT "LIMITS - 2 TO 100"     !WRONG - INFORM USER             &RET
\           GO TO 20                       !TRY AGAIN                       &RET
        ELSE INPUT SORT(I%) FOR I%=1% TO CNT% RET
30  REM                                                                     &RET
            B U B B L E   S O R T                                           &RET
                                                                            &RET
    CHECK EACH PAIR OF ELEMENTS                                             &RET
                                                                            &RET
    IF IN WRONG ORDER, SWITCH THEM                                          &RET
    SORT.FLG IS SET TO FALSE (0) WHEN A SWITCH IS MADE                      &RET
    PASS OVER THE ENTIRE LIST UNTIL NO SWITCH IS MADE                       &RET
                                                                            &RET
31  SORT.FLG%=1%                           !SET TO TRUE INITIALLY           &RET
\   WHILE SORT.FLG%<>0%                    !LOOP UNTIL SORT.FLG IS FALSE    &RET
\       SORT.FLG%=0%                       !SET TO FALSE BEFORE PASS        &RET
\       FOR I%=Q% TO CNT%-1%               !LOOP THROUGH ENTIRE LIST        &RET
\           IF SORT (I%)<=SORT(I%+1%)      !CHECK A PAIR                    &RET
                THEN SORT.FLG%=-1%         !IF WRONG-FORCE ANOTHER PASS     &RET
\                   T=SORT(I%)             !SWAP ELEMENTS                   &RET
\                   SORT(I%)=SORT(I%+1%)                                    &RET
\                   SORT(I%+1%)=T                                           &RET
40      NEXT I% RET
50  NEXT RET
60  PRINT SORT(I%), FOR I%=1% TO CNT%      !PRINT ELEMENTS IN ORDER RET
32767  END RET

SAVE  RET

Basic2

COMPILE  RET

Basic2

BUILD  RET

Basic2

EXIT  RET
> TKB @SORT02  RET
```

```
> RUN SORT02  RET

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

The program shown above accepts up to 100 numbers as input, sorts them by size, and prints them in descending order. The procedure used to enter, compile, build, and run the program is detailed below. The explanations are keyed to the commands.

Command	Explanation
`IDENTIFY`	The IDENTIFY command (see Section 1.2.1) prints a BASIC-PLUS-2 header.
`NEW` `NEW FILE NAME--SORT02`	The NEW command (see Section 1.2.2) clears a space in the temporary buffer for creation of the source program. When you type NEW, any source code in the buffer is lost. When you type SORT02 in reply to the prompt (NEW FILE NAME--), you assign the name SORT02 to your program.
`Basic2`	Basic2 is printed by BASIC to indicate that the compiler is prepared to accept input. It also indicates that the previous command (NEW) has been successfully executed.
`SAVE`	SAVE (see Section 1.2.6) copies and preserves the program on the system default device. The program now resides on the system as a source program (file type .B2S) named SORT02.
`COMPILE`	The COMPILE command (see Section 1.2.7) translates the program into an object module. The default file type, .OBJ, is appended to the program name.
`BUILD`	The BUILD command (see Section 1.2.8) creates a command file composed of the specified object module as well as the command input required by the task builder.

EXIT	The EXIT command (see Section 1.2.16) terminates access to BASIC-PLUS-2 and returns you to operating system command level. To create an executable task, invoke the system task builder and specify @SORT02 on the command line.
TKB	The TKB command invokes the Task Builder system program. Specifying @SORT02 provides the Task Builder with the indirect command file it uses to create an executable task.
RUN	The RUN command causes the program to be executed. As part of the execution, you are prompted for "Number of entries" and by a "?" for each number you enter.

Chapter 2
Files

You can perform efficient input/output operations on large amounts of related data by collecting that data into files. Record Management Services (RMS) can increase this efficiency by allowing you to organize a file into manageable units of data called records. For example, a company may wish to document an inventory of its capital equipment. A file that contains data on all equipment is created for this purpose. This data is organized into individually accessible records, each of which describes a particular item.

BASIC–PLUS–2 allows you to create block I/O or record I/O files. RMS is the vehicle for creating and accessing record files. This chapter describes block I/O, the use of RMS, the file organizations available under RMS, and the operations allowed on each type of organization.

For additional information on the BASIC–PLUS–2 syntax used to create and manipulate files, refer to the *PDP–11 BASIC–PLUS–2 Language Reference Manual*.

NOTE:

This chapter is a minimal discussion of file handling using BASIC–PLUS–2, designed to enable you to get an application running. If you want more information, refer to appropriate documentation.

2.1 OPEN Statements

The manner in which data are stored in a file is determined by the organization that you specify in the OPEN statement. The organization, in turn, determines the operations and access methods that you can use on the file.

BASIC allows you to choose one of four types of organizations when creating files; virtual, sequential, relative, or indexed. When you create a file, the organization must be the first file attribute specified in the OPEN statement as follows:

OPEN *filename* [{ FOR OUTPUT / FOR INPUT }] AS FILE [#]*num-exp*

,[ORGANIZATION] { VIRTUAL / UNDEFINED / SEQUENTIAL / RELATIVE / INDEXED }

[,*attributes*]

filename
is a file specification.

FOR OUTPUT
indicates the creation of a new file.

FOR INPUT
indicates accessing an existing file.

AS FILE *#num-exp*
associates the file with a channel number in the range of 1 to 12.

,ORGANIZATION
is an optional keyword preceded by a comma and followed by a required keyword that represents one of the five types of organization.

,attributes
are file characteristics that you define in the OPEN statement. Attributes differ for each file organization and their specification is described in the appropriate section.

The organization you specify when the file is created is permanently assigned to the file. When any existing file is opened for processing, you must respecify the organization. An organization specification that does not match the initial file assignment results in an error (i.e., `?File attributes not matched`).

If you fail to include the appropriate BUILD command switches (/SEQ, /IND, /REL, and/or /VIR), you will receive an error message at file open time:

```
?Illegal operation at line n
```

The organization you choose depends on the access methods and operations that you wish to perform on the file. A comparison of these organizations may be helpful in making this choice.

Virtual files can contain either user-defined blocks or one or more virtual arrays. This file organization permits block I/O operations, but it does not allow record operations. Virtual files are allowed only on random-access devices.

Sequential files contain records that are stored in series. You cannot access one record without successfully accessing all preceding records. Sequential files are allowed on disk, ANSI-formatted magnetic tape, or unit record devices such as line printers and terminals. If you do not specify a file organization in an OPEN statement, the default organization is terminal-format which is a subset of sequential files.

Relative files contain records that are stored in numbered locations of a fixed size. You can access a record sequentially or by number. Relative files are allowed only on disk media.

Indexed files contain records that are associated with individual key values within each record. You can access a record sequentially or by reference to a key. Indexed files are allowed only on disk media on a system with RMS–11K installed.

Files opened as undefined must first exist with a defined organization. The undefined organization allows you to open a file with READ access only to ascertain a file's organization so that you can subsequently re-open the file using the correct attributes.

2.2 Special Case Files

2.2.1 Virtual Files

The virtual file organization specifies a block-structured file. Input and output operations are performed by means of RMS block I/O (see Section 2.5.1). Virtual files can contain data organized as elements in an array which is fully compatible with BASIC–PLUS virtual arrays. When the file contains virtual arrays, it must be dimensioned with a DIM # statement. This statement is described in the *PDP–11 BASIC–PLUS–2 Language Reference Manual.*

If your program accesses virtual files, you must use the BUILD/VIR command to include the required supporting code.

The OPEN statement used to specify a virtual file allows you to assign the following attributes:

```
,[ORGANIZATION] VIRTUAL
[,ACCESS {READ  }  ]
         {MODIFY}
         {WRITE }
[,ALLOW  {NONE  }  ]
         {READ  }
         {MODIFY}
         {WRITE }
```

[,MAP <*map-name*>]

[,FILESIZE <*num-exp*>]

[,CONTIGUOUS]

[,RECORDSIZE <*num-exp*>]

[,TEMPORARY]

,ORGANIZATION VIRTUAL
specifies the creation or access of a virtual file and allows the use of block I/O. The ORGANIZATION keyword is optional.

,ACCESS
specifies the operations that you will perform on the file. MODIFY is the default. Refer to Section 2.3.4.

,ALLOW
specifies the operations that you will permit other programs to perform on the file. READ is the default. Refer to Section 2.3.4.

,MAP
references a MAP statement and can be used to define record size (see Section 2.7). Note that MAP must not be used with a file that contains arrays.

,FILESIZE
preallocates space for a file whose length is defined in terms of a number of disk blocks. The default is pack dependent.

,CONTIGUOUS
specifies that the contents of the file are contiguous on disk devices. The default is non-contiguous.

,RECORDSIZE
defines the maximum size of data blocks in the file. The default size is 512 bytes; the maximum is 65535 bytes. Refer to Section 2.7.

,TEMPORARY
creates a temporary file that is deleted when you close the file. The default is non-temporary.

When you specify a RECORDSIZE that exceeds the default minimum of 512 bytes for a file that contains virtual arrays, the specification should be a multiple of 518. During I/O operations on virtual arrays, blocks are read in by the program as required. If sufficient record size is not available to contain the accessed blocks, space is obtained by writing the first block that was read.

Note that the virtual organization allows block I/O file operations, but it disallows RMS record operations.

You can specify file attributes in the OPEN statement in any order. Consider the following example:

```
130 OPEN "VATST4.TMP" FOR OUTPUT AS FILE #2  &
    ,ORGANIZATION VIRTUAL,ACCESS MODIFY      &
    ,ALLOW NONE
```

This OPEN statement creates a new file named VATST4.TMP. The file is assigned to channel 2 and is defined as a virtual file. The OPEN statement also sets the ACCESS status to MODIFY and the ALLOW status to NONE. Note that ALLOW NONE is the equivalent of ALLOW READ (see Section 2.2.4).

2.2.2 Organization Undefined

The undefined organization lets you open a file for input only. You do not have to know all of the file's attributes beforehand. The use of the ORGANIZATION UNDEFINED statement is recommended for advanced programmers only. It allows you to write general purpose programs that access files whose attributes are not known in advance. You can also use this OPEN statement to discern the attributes of a file so that you can subsequently re-OPEN it by specifying the correct file descriptors. Note that you cannot create a file with ORGANIZATION UNDEFINED.

The OPEN statement used to specify an undefined file access allows you to specify the following attributes:

OPEN *filename* FOR INPUT AS FILE [#]*num-exp*

,[ORGANIZATION] UNDEFINED

[,ACCESS READ]

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

[,MAP <*map-name*>]

[,RECORDSIZE <*num-exp*>]

,ORGANIZATION UNDEFINED
specifies the access of the file specified in the OPEN statement for input only. The ORGANIZATION keyword is optional.

,ACCESS
specifies the operations that the current user will perform on the file. READ is the only permissable access method.

,ALLOW
specifies the operations that the current user will permit other programs to perform on the file. READ is the default.

,MAP
references a MAP statement and can be used to define record size. Note that MAP cannot be used with a file that contains arrays.

,RECORDSIZE
defines the maximum size of data blocks in the file. The default size is 512 bytes. Refer to Section 2.7.

You can specify file attributes in the OPEN statement in any order. Consider the following example:

```
130  OPEN "VATST5.TMP" FOR INPUT AS FILE #3              &
     ,ORGANIZATION UNDEFINED, ACCESS READ, ALLOW NONE    &
     ,MAP MAP3
```

Please note that when you use ORGANIZATION UNDEFINED, you must include all possible organizational types accessed by the program, i.e., BUILD/IND/REL/SEQ.

2.2.2.1 FSP$ Function — The function FSP$ returns the file organization data for an opened file. This function is intended for use with files OPENed as ORGANIZATION UNDEFINED. The syntax of the FSP$ function is as follows:

```
X$=FSP$(channel-number)
```

Consider the following example:

```
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$ returns the following values:

- A%(0) returns file characteristics in the form:

  High byte is the RMS Organization (ORG) field
  Low byte is the RMS record format (RFM) field
- A%(1) returns the RMS maximum record (MRS) field.
- A%(2) and A%(3) return the RMS allocation quantity (ALQ) field.
- A%(4) and A%(5) return the RMS bucketsize (BKS) field for disk files or the RMS blocksize (BLS) field for magnetic tape files.
- A%(6) returns the number of keys.
- A%(7) returns the RMS maximum record number (MRN) if the file is a relative file.
- A%(8) and A%(9) return the current block/record number.

Refer to the *IAS/RSX-11M RMS-11 MACRO Programmer's Reference Manual* for detailed descriptions of the RMS fields returned by FSP$.

2.3 Introduction to RMS

Record Management Services (RMS) is a set of library routines. These routines effect the transmission of data between files and BASIC programs. Files are composed of records that act as the storage media for a related collection of data.

RMS ensures that every record written into a file can be subsequently retrieved and passed to a program. You determine the size and content of data in the record, the organization of records in the file, and the method used to access the records. You make these determinations by means of statements written in the BASIC language, either through the attributes you specify for new files in the OPEN statement or through the operations you perform on existing files.

To maintain an efficient relationship between RMS and the programs you write, you must have a general understanding of RMS files. This chapter describes the components of RMS files. The chapter is divided into five parts, as follows:

1. File organization - RMS files contain records that are organized in one of three fashions: sequential, relative, or indexed. You select one of these organizations and assign it to a file by means of the ORGANIZATION clause in the OPEN statement.
2. Record access - Record access represents the methods you can use to store and retrieve records. RMS provides two access methods: sequential and random. The organization of the file and the syntax of the individual record operation determine which of these is used.
3. Record format - RMS files can contain fixed-length, variable-length or ASCII stream-format records.
4. Data structure - Data items are maintained in records, which are contained in storage structures called blocks and buckets. RMS provides you with a means of controlling the size of these structures.
5. Record mapping - Mapping provides you with a means of directing the assignment of data in the record. It also allows you to identify certain data elements as access keys for records in indexed files.

Table 2-1 illustrates the record access methods and operation types allowed on each file organization.

Table 2-1: Comparison of File Organizations

	File Organizations		
Access and Operations	**Sequential**	**Relative**	**Indexed**
Sequential access	Y	Y	Y
Random access	N	Y (by rec no.)	Y (by key)
Record replacement	Y	Y	Y
Record insertion	Y (at end of file only)	Y	Y
Record deletion	N	Y	Y

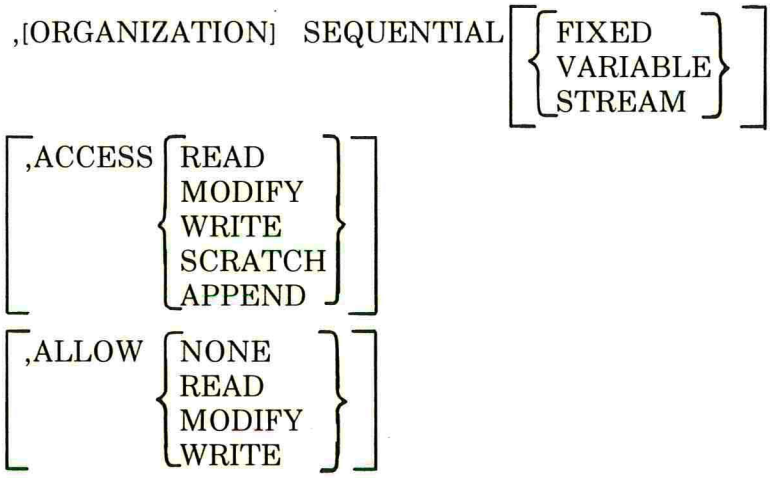
The following subsections describe each file organization in detail.

2.3.1 Sequential Files

The sequential file organization is the default and specifies a file that can contain records of varying lengths and can be stored on disk, ANSI-formatted magnetic tape, or a unit record device.

If your program accesses sequential files, you must use the BUILD/SEQ command to include the required supporting code.

The OPEN statement format used to create and access a sequential file allows you to specify the following attributes:



[,MAP *<map-name>*]

[,RECORDSIZE *<num-exp>*]

[,NOSPAN]

[,SPAN]

[,FILESIZE *<num-exp>*]

[,BLOCKSIZE *<num-exp>*]

[,CONTIGUOUS]

[,NOREWIND]

[,TEMPORARY]

,ORGANIZATION SEQUENTIAL
specifies the creation or access of a sequential file. The ORGANIZATION keyword is optional.

FIXED
VARIABLE
STREAM
one of these three attributes is used to specify the format of records within the file. FIXED indicates fixed-length records. VARIABLE is the default and indicates variable-length records. STREAM indicates ASCII-stream

records and is only permitted on disk files. Files that perform terminal input and output operations must be opened with ORGANIZATION SEQUENTIAL STREAM. Records for these terminal-format files cannot exceed 132 characters and include a carriage-return/line-feed combination as a line terminator.

,ACCESS
specifies the operations that you will perform on the file. MODIFY is the default. Refer to Section 2.3.4.

,ALLOW
specifies the operations that you will permit other programs to perform on the file. READ is the default. Note that you cannot specify an ALLOW attribute if the ACCESS designation is SCRATCH. Refer to Section 2.3.4.

,MAP
references a MAP statement that can be used to define record size. Refer to Section 2.7.

,RECORDSIZE
defines the maximum size of records within the file. Note that you must specify record size with either a MAP or RECORDSIZE specification in the OPEN statement. The largest record size permitted is 65535 bytes. Refer to Section 2.7.

,NOSPAN
,SPAN
SPAN is the default and allows records to cross block boundaries. Refer to Section 2.6.1.

,FILESIZE
preallocates space for a file whose length is defined in terms of a number of disk blocks. The default is determined by the extend option of the MOUNT command.

,BLOCKSIZE
specifies the number of records contained in a block on magnetic tape. The default is 512 bytes long. Refer to Section 2.6.1.

,CONTIGUOUS
specifies that the contents of the file are contiguous on disk devices. The default is logically continguous.

,NOREWIND
overrides the default rewind action on magnetic tape. The default is to rewind to the beginning of the tape on OPEN or CLOSE operations; NOREWIND causes the pointer to remain at the end of the last accessed tape position.

,TEMPORARY
creates a temporary file that is deleted when you close the file. Default is non-temporary.

Note that you can specify file attributes in any order. Consider the following example:

```
10  OPEN "RMSEQ1.FIX" FOR OUTPUT AS FILE #3      &
    ,ORGANIZATION SEQUENTIAL VARIABLE, ACCESS    &
    MODIFY, MAP MAP1, NOSPAN
```

This OPEN statement creates a new file named RMSEQ1.FIX and assigns it to channel 3. The organization is sequential, the record format is defined as variable, and the ACCESS attribute is set to MODIFY (the ALLOW attribute defaults to READ).

The OPEN statement also contains a map attribute that references a MAP statement named MAP1. The MAP statement, which must appear in the same program, defines the content of records in the file (see Section 2.7). The NOSPAN attribute overrides the SPAN default and prohibits records from crossing block boundaries (see Section 2.6).

A sequentially organized file maintains a strict relationship among the records on the file. The file is structured such that the location of any particular record is fixed in relationship to the preceding and succeeding records. The serial arrangement of the records is determined by the order in which they are written and is permanent.

Because of this serial order, access to any record in the file begins with the next record and continues with each succeeding one until the desired record is reached. For example, to read the 12th record in the file, the BASIC program first must open the file, then successfully read records 1 through 11, and finally read 12. After reading record 12, the program can read all succeeding records (in serial order) but it cannot read a preceding record without returning to the beginning of the file.

Sequential files allow the following operations:

GET (read)
PUT (write)
UPDATE
FIND
SCRATCH
RESTORE

Sequential organization imposes the following restrictions on these file operations:

1. GET and FIND operations can be performed only in sequential order.
2. PUT operations can be performed only at the end of the file. Note that to open an existing sequential file on disk and add records at the end of the file, you must specify ACCESS APPEND in the OPEN statement.
3. UPDATE operations are only allowed on sequential files that reside on disk media. Also, UPDATE requires that the target and updated records be the same length and that the target record be successfully located by a GET or FIND before the UPDATE is made.

4. SCRATCH operations erase the contents of the file beginning at the program's current record position up to the end of the file. The current record position becomes the end of the file. Exclusive file access is required for SCRATCH operations. If you want to erase an entire line, you must precede the SCRATCH operation with a RESTORE operation followed by a GET or FIND operation.
5. RESTORE operations set the program at the beginning of the file but do not erase the file's content.

2.3.2 Relative Files

When you specify relative file organization, RMS builds a file in which records are assigned to numbered positions. Access to these records is based on the numbered position that they occupy in the file.

If your program accesses relative files, you must use the BUILD/REL command to include the required supporting code.

The OPEN statement used to create or access a relative file allows you to specify the following attributes:

,[ORGANIZATION] RELATIVE [{FIXED / VARIABLE}]

[,ACCESS {READ / MODIFY / WRITE}]

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

[,MAP <*map-name*>]

[,RECORDSIZE <*num-exp*>]

[,CONTIGUOUS]

[,BUCKETSIZE <*num-exp*>]

[,FILESIZE <*num-exp*>]

[,TEMPORARY]

[,BUFFER <*num-exp*>]

[,CONNECT <*num-exp*>]

,ORGANIZATION RELATIVE
specifies the creation or access of a relative file. The ORGANIZATION keyword is optional.

FIXED
VARIABLE
specifies the format of records within the file. FIXED indicates fixed-length records. VARIABLE is the default and indicates variable-length records. Refer to Section 2.5.

,ACCESS
specifies the operations that you will perform on the file. MODIFY is the default. Refer to Section 2.3.4.

,ALLOW
specifies the operations that you will permit other programs to perform on the file. READ is the default. Refer to Section 2.2.4.

,MAP
references a MAP statement and can be used to define record size. Refer to Section 2.6.

,RECORDSIZE
defines the maximum size of records in the file. Note that you must specify a record size with either the MAP or RECORDSIZE attribute in the OPEN statement. Refer to Section 2.6.

,CONTIGUOUS
specifies that the contents of the file are contiguous on disk devices. The default is non-contiguous.

,BUCKETSIZE
specifies the size of a bucket in terms of the number of records. The default is 32. Refer to Section 2.6.2.

,FILESIZE
allocates space for a file whose length is defined in terms of a number of disk blocks. The default is pack dependent.

,TEMPORARY
creates a temporary file that is deleted when you close the file. The default is non-temporary.

,BUFFER
specifies the number of buckets maintained in memory. The default is 1.

,CONNECT
performs multi-stream connect to the base file that is open on the channel specified. Refer to the *RMS-11 User's Guide*.

Consider the following example:

```
10 OPEN "RMSIVX.FIX" FOR OUTPUT AS FILE #3  &
   ,ORGANIZATION RELATIVE FIXED, ACCESS     &
   MODIFY, ALLOW NONE, MAP MAP1
```

This OPEN statement creates a new file named RMSIVX.FIX and assigns it to channel 3. The organization is relative, the record format is fixed, the

ACCESS attribute is set to MODIFY, and ALLOW is NONE. Note that a NONE specification in the ALLOW attribute is equivalent to READ (see Section 2.3.4). The OPEN statement also contains a map attribute that references a MAP statement named MAP1. The MAP statement, which must appear in the same program, defines the content of records in the file (see Section 2.7). Because the file contains fixed-length records, the MAP attribute defines the size of each record in the file.

RMS structures a relative file into a series of record positions. All positions are the same size and each can contain a single record. RMS considers the first record position in the file to be number one and sequentially numbers each succeeding position. When you write or read records on the file, you can designate a number for the desired record. This number represents the record's position relative to the beginning of the file. The record/position number is unique in the file and can therefore be used to specify location (in a PUT operation) or a record (in a GET operation). For example, record #1 occupies file position #1, record #2 occupies position #2, etc. A record number is not required for sequential GET, FIND, and PUT operations.

Unlike sequential files, relative files are allowed only on disk devices. However, relative files do have advantages over sequential files.

First, though both organizations arrange records in serial order, BASIC programs can access relative file records by means of a known position number. This allows you to access records randomly (i.e., GET #2, RECORD 5%; GET #2, RECORD 20%; GET #2, RECORD 13%, etc.) in addition to proceeding in strict serial order.

Second, each relative file record position does not have to contain a record. Each position contains the same amount of space but this space can be empty. Also, empty record positions can appear anywhere in the file. Note that sequential GET and FIND operations that do not specify a record number locate the next occupied position and bypass empty positions.

BASIC allows the following operations on relative files:

GET (read)
PUT (write)
UPDATE
DELETE
FIND
RESTORE

The relative file organization imposes the following restrictions on record operations:

1. GET or PUT operations can use a specified number to select a record or position in the file. This selection method is similar to BASIC's use of a subscript to select an item from an array. Record/position numbers allow you to perform GET and PUT operations in random order and, therefore, access a record at any point in the file. In addition, new records can be inserted into the empty positions of existing files. Note that a PUT operation can be performed only on an empty position or at the end of the file.

2. FIND operations can also use a specified number to locate a record or position in the file. UPDATE and DELETE operations require a previously successful GET or FIND.
3. DELETE and UPDATE operations do not allow a record number specification. Because a successful GET or FIND must be done before a record is erased (DELETE) or replaced (UPDATE), the record number is already known. Note that this also restricts DELETE and UPDATE operations to existing records.
4. RESTORE operations set the program at the beginning of the file without disturbing the data. Note that a SCRATCH operation is not allowed on relative files.

2.3.3 Indexed Files

Indexed files require the presence of RMS–11K on the system.

If your program accesses indexed files, you must use the BUILD/IND command to include the required supporting code.

The OPEN statement used to create or access an indexed file allows you to specify the following attributes:

```
,[ORGANIZATION] INDEXED [{FIXED   }]
                        {VARIABLE}

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

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

[,MAP <map-name>]

[,RECORDSIZE <num-exp>]

[,CONTIGUOUS]

[,BUCKETSIZE <num-exp>]

[,FILESIZE <num-exp>]

[,CONNECT <num-exp>]

[,BUFFER <num-exp>]

[,TEMPORARY]

,PRIMARY[KEY]<name>[DUPLICATES]

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

,ORGANIZATION INDEXED
specifies the creation or access of an indexed file. The ORGANIZATION keyword is optional.

FIXED
VARIABLE
one of these two attributes is used to specify the format of records within the file. FIXED indicates fixed-length records. VARIABLE is the default and indicates variable-length records. Refer to Section 2.5.

,ACCESS
specifies the operations you will perform on the file. MODIFY is the default. Refer to Section 2.3.4.

,ALLOW
specifies the operations that you will permit other programs to perform on the file. READ is the default. Refer to Section 2.3.4.

,MAP
references a MAP statement and must be used to define record key positions. Refer to Section 2.7.

,RECORDSIZE
defines the maximum size of records in the file. Note that you must specify a record size with either a MAP or RECORDSIZE specification in the OPEN statement. Refer to Section 2.7. The largest record size allowed is 65565 bytes.

,CONTIGUOUS
specifies that the contents of the file are contiguous on disk devices. The default is logically contiguous.

,BUCKETSIZE
specifies the size of a bucket in terms of the number of records. Refer to Section 2.6.2. The default is 1 record.

,FILESIZE
preallocates space for a file whose length is defined in terms of a number of disk blocks. The default is determined by the extend option of the MOUNT command.

,CONNECT
performs multi-stream connect to the base file that is open on the channel specified. Refer to the RMS documentation.

,BUFFER
specifies the number of buckets maintained in memory. The default is 1.

,TEMPORARY
creates a temporary file that is deleted when you close the file. The default is non-temporary.

,PRIMARY
defines the primary key for a particular record. This attribute is required. Refer to Section 2.2.3.1.

,ALTERNATE
allows you to define up to 254 alternate keys. This attribute is optional. Refer to Section 2.2.3.1.

NODUPLICATES
DUPLICATES
specifies the use of a duplicate key in the file. NODUPLICATES is the default. Refer to Section 2.2.3.1.

NOCHANGES
CHANGES
specifies the use of a key field change in the file. NOCHANGES is the default. Refer to Section 2.2.3.1.

Consider the following example:

```
5   MAP (MAP1) NAME$=30%,ID%,HRWAGE,FILL,FILL%,FILL$
10  OPEN "RMSIXV.VAR" FOR OUTPUT AS FILE #3          &
    ,ORGANIZATION INDEXED VARIABLE, ACCESS           &
    MODIFY, ALLOW NONE, MAP MAP1                     &
    ,PRIMARY NAME$
```

This OPEN statement (line 10) creates a new file named RMSIXV.VAR and assigns it to channel 3. The organization is indexed, the record format is variable, the ACCESS attribute is set to MODIFY, and ALLOW is NONE. Note that a NONE specification in the ALLOW attribute is equivalent to READ (see Section 2.2.4).

The OPEN statement also contains a map attribute that references a MAP statement named MAP1. The MAP statement (line 5) defines the content of records in the file (see Section 2.6). Because this is an indexed file, the MAP statement is also used to define the size and location of key fields in the record. The PRIMARY attribute associates the primary index key with NAME$, which is defined in the MAP statement on line 5.

The location of records in an indexed file, unlike the record location in sequential or relative files, is completely under the control of RMS. You control sequential and relative record location at input by performing an end-of-file PUT operation (for sequential) or by specifying a position number (for relative). The placement of indexed file records, however, is governed by the presence of keys in the record. RMS uses these keys to determine record location, a process that is transparent to you.

A key is a data field that exists in every record. A data field is one of the many discrete pieces of information that compose records. For example, an individual employee record in a company personnel file is usually composed of data fields such as the employee's name, address, social security number, and department. You can designate one or more of these data fields as a key for accessing the whole record.

The position and length of each key data field in a record is identical for each record in the file; only the content can differ. For example, all employee records in a personnel file reserve the same amount of space at the same position for the employee name data field; only the name itself will differ for each record. When you create an indexed file, you designate the length and position of the data fields RMS will use as keys. Once a specific data field has been selected as an RMS key, your BASIC program can use the key to access the record.

Indexed files require that at least one key, called the primary key, be associated with every record. When you create the file, you use a MAP statement to define the primary key in terms of its position and length in the record. To access the record, you provide the BASIC program with a key number of 1 (meaning the primary key) and a key value. RMS locates the record with that value in its primary key field.

In addition to a primary key specification for each record in an indexed file, you can optionally define up to 254 alternate keys for a record. Alternate keys represent secondary data fields and are defined in the same manner as a primary key. Your program can also use these alternate keys to identify and retrieve records. Alternate keys are numbered (first alternate, second alternate, etc.) according to their order of appearance in the OPEN statement.

As with relative files, indexed files are allowed only on disk devices. The operations allowed on indexed files are:

GET (read)
PUT (write)
UPDATE
DELETE
FIND
RESTORE

GET, FIND, and RESTORE operations can require a key of reference specification. That is, when records contain alternate or primary keys, you must indicate to RMS which key field to search. UPDATE, PUT, and DELETE operations do not require a key of reference specification.

GET operations can be performed randomly or sequentially. When you perform a series of sequential GET, FIND, or RESTORE operations, a key number specification is required for the initial operation and it remains in effect until changed by another explicit specification.

2.3.3.1 Primary and Alternate Key Record Access — Access to records in an indexed file is based on key specifications that appear in your program. That is, each record in the file contains one or more data fields that RMS recognizes as keys.

RMS allows you to have duplicate primary and alternate keys if you specify DUPLICATES in the OPEN statement. That is, more than one record is allowed to contain the same value in the data field that composes the key. Such records are said to have the same record identifier. For example, a

personnel file may contain many records that have the same value in the field defined as "Department." If you do not specify DUPLICATES in the OPEN statement, RMS rejects any attempt to write a record that contains key data field values already present in another record of the same file.

RMS also allows you to change alternate key values during update if you specify CHANGES in the OPEN statement. That is, you are allowed to read a record from the file, modify a particular alternate key data field within the record, then write the record back to the file. If you do not specify CHANGES in the OPEN statement, RMS rejects any attempt to write a record containing a modified key value. Note that primary keys are not allowed to change.

Note that you cannot specify CHANGES without also specifying DUPLICATES.

To randomly access records in an indexed file, you must specify the key of reference. That is, you must specify the desired key name that refers to defined values in a MAP statement. A record operation key specification has the following format:

```
GET #channel no., KEY #num-exp{GT|GE|EQ}str-exp
```

where *#num-exp* is a number that specifies the key of reference (0 is the primary key, 1 is the first alternate, etc.). The *str-exp* is a quoted character string or string variable that represents the content of the data field. GET and FIND operations allow you to specify an exact key, approximate key, or generic key. To specify an exact or approximate key, you use EQ for exact key, GT for an approximate key that is greater than the string expression, or GE for an approximate key that is the same or greater than the string expression.

An exact key specification requires that you specify the complete key field identifier in the program statement as follows:

```
GET #channel no., ,KEY #num-exp EQ str-exp
```

An approximate key specification allows you to access a record based on a specified relationship. That is, you can specify a search for a record that is equal to (EQ), or greater than or equal to (GE), or greater than (GT) the record key. For example, the format:

```
FIND #channel no., ,KEY #num-exp GE str-exp
```

causes RMS to search for a record whose key value is equal to that specified by the string expression. If RMS determines that the specified record key does not exist in the table, it searches for the next highest value in that key index table.

A generic search accesses a record based on an initial portion of the record's key field. This search is automatically initiated when you specify a key data field (str-exp) that contains fewer characters than are defined for that key in the file. A generic search causes RMS to return the first record whose key value begins with the specified characters.

When you specify search keys you must pay strict attention to the length and justification of the string fields. BASIC–PLUS–2, by default, left justifies and optionally either blank pads or truncates these fields. Common or Map fields have a specified, fixed length. Input fields are left-justified but not padded.

When you institute a key search, RMS searches for a match based on the number of characters you specify. However, if you use the map field, BASIC–PLUS–2 pads the key you specify with blanks so that it matches the length of the field specified in the map.

To illustrate generic key access, assume that you have a personnel file. Each record in this file contains a data field composed of a 9–character social security number. These numbers have been defined in terms of record position and length in a MAP statement and have been assigned to the variable SSN$. This definition takes place before any record operation. Also, in the OPEN statement, you have defined SSN$ as the primary key.

Consider the following GET statement:

```
GET #1%, KEY #0% EQ "013"
```

- #1% is a channel number that identifies the file.
- #0% is the key of reference. Because 0% is the primary key, the key index SSN$ is searched.
- "013" is a string expression that represents the first three characters of the data field associated with SSN$.

This GET statement causes RMS to search the key index represented by SSN$. RMS returns the first record in that index with a data field of 013 at the defined position and length.

2.3.4 File Sharing

With the exception of sequential files on non-disk devices, all files are capable of being shared by any number of programs. Sequential files on non-disk devices can be read or written only by a single program. Sequential files on disk devices can be shared by multiple readers, but allow only a single program. Relative and indexed files can be shared by multiple programs.

While the organization of the file determines the sharing capability, the type of sharing that actually occurs at run time is determined by the specifications you make in the OPEN statement.

The ALLOW attribute in the OPEN statement is used to specify the types of operations that you will permit other programs to perform on the file while you have it open. With the ALLOW attribute, you can control the sharing of the file. The specifications you can make in the ALLOW attribute, and the operations they permit other users to perform, are as follows:

READ allows GET and FIND operations on the records in the file.

WRITE allows PUT operations on the records in the file.

MODIFY allows GET, FIND, PUT, and UPDATE operations on records in sequential, relative, and indexed files; additionally, it allows DELETE operations on records in relative and indexed files.

NONE is the equivalent of READ.

The ACCESS attribute in the OPEN statement is used to specify the record operations that you will perform on the file. The specifications you can make in the ACCESS attribute, and the operations they refer to, are as follows:

READ specifies GET and FIND operations on the records in the file.

WRITE specifies PUT operations on the records in the file.

MODIFY specifies GET, FIND, PUT, and UPDATE operations on records in sequential, relative, and indexed files; it also specifies DELETE operations on records in relative and indexed files.

SCRATCH specifies GET, FIND, PUT, UPDATE, and SCRATCH operations on records in sequential files that reside on disk.

APPEND specifies PUT operations at the end of a sequential file that resides on disk.

Operations on the virtual file organization should not be shared. If another program attempts to modify a block that is already open, the block is changed in the second program's buffer but not on the disk. When the second program closes the file or attempts another block operation, the data from the first program is overwritten and lost.

Note that FIND and GET operations on relative and indexed files cause the bucket that contains the accessed record to be inaccessible to other programs. This process is called locking and it ensures that the modifications that you make to a record are not interfered with by another program. The lock remains in effect until you specify a PUT, DELETE, UPDATE, or another GET or FIND operation. Note that if the second GET or FIND operation accesses the same bucket, the lock is reenabled. (For information on buckets, refer to Section 2.5.)

You can explicitly unlock the bucket that was locked by your program by specifying an UNLOCK statement. For example:

```
70   UNLOCK #1%
```

causes the records contained in the file on channel 1% to remain accessible to other programs.

If another program attempts an operation on a locked bucket, the operation fails and an error message is printed:

```
?Record/bucket locked
```

Note that a lock is made on a bucket and not on the individual record. Therefore, more than one record can be locked at the same time.

2.3.5 RMS Memory Allocation

The use of RMS-structured files in a BASIC-PLUS-2 program causes the compiler to allocate space in memory to the needs of that program. Static space for code is initially allocated when a file organization is specified in the BUILD command (see Section 1.2.8). Additional space is allocated at run time for each channel that the program opens.

The space that is initially allocated when a file organization is specified is system dependent. However, the static space that is dynamically allocated to each open file in the program is determined by the algorithms contained in Table 2-2. Note that this space is deallocated when the file is closed.

Table 2-2: Allocation Algorithms

File Type	Allocated Space
Sequential files	736 bytes + the record length
Relative files	224 bytes + the bucket size (in bytes) + the record length
Indexed files	264 bytes + 2 * the bucket size (in bytes) + the number of keys * 104 + 2 * the maximum key size + the record length

You can reduce your program overhead for task extension by pre-determining the number of bytes the program needs for simultaneously opened files and using that number in the BUILD/EXTEND:*n* command.

2.4 Record Access Methods

The methods that you use to store or retrieve records in a file are determined by the file's organization. The organization of a file is fixed at the time you create it, but, depending on the access allowed, a specified access method can change each time the file is opened for program execution. In some cases, you can vary the access to records during program execution.

RMS allows you two types of record access: sequential and random. If you use sequential access, records are accessed in serial order as established by the file

organization. If you use random access, record operations can take place at any point in the file.

Table 2–3 shows the relationship between file organization and record access.

Table 2–3: Access Methods

	Access Methods	
File Organization	**Sequential**	**Random**
Sequential	yes	no
Relative	yes	yes
Indexed	yes	yes

The following subsections discuss each type of record access.

2.4.1 Sequential Access

All RMS file organizations allow you to access records sequentially. Sequential record access is employed when you issue a series of requests for the next record. RMS interprets these sequential operations within the context of the file organization. That is, record operations are performed in terms of a predecessor-successor record relationship. RMS assumes that for each successfully accessed record (except the last) there is a succeeding record somewhere in the file.

The sequential file organization allows only sequential access. In these files, the predecessor-successor relationship is physical (i.e., each record, except the last, is physically adjacent to the next record). A record in a sequential file can be processed only after each preceding record has been successfully accessed. Similarly, once a record is processed, the program must be repositioned to the beginning of the file before preceding records can be accessed. A RESTORE operation, or reopening the file, positions the program at the beginning of the file.

In terms of operations, a PUT requires that the program be positioned at the end of the file (i.e., immediately following the last record). An existing sequential file on disk can be opened at the end-of-file position if you specify ACCESS/APPEND in the OPEN statement. A FIND operation moves the program to the next sequential record position. Therefore, a series of FIND operations can be used to locate the end of the file (i.e., an unsuccessful FIND indicates end-of-file).

UPDATE operations on sequential files require a successful GET or FIND operation to move the program to the desired record before the UPDATE is specified. A GET causes the program to locate the next record and perform the GET operation. A succeeding GET or FIND operation moves the program to the next record.

The relative file organization allows sequential access as established by the contents of record positions. Relative files allow empty record positions that can be caused by a record deletion or by a program that purposely leaves the positions empty. RMS maintains the predecessor-successor relationship through its ability to recognize empty or occupied record positions.

Sequential PUT operations on relative files are used when you create a new file or append records to an existing file because RMS requires that new records be written in empty positions. That is, a sequential PUT operation causes RMS to place a record in a location whose position number is one higher than the previous operation. If the position is occupied, the operation fails. A GET or FIND operation causes the program to locate the next existing record in position number order. In addition, the GET operation reads the located record. The program remains at this location until another operation is specified. DELETE and UPDATE operations require that a FIND operation position the program at the desired location.

The indexed file organization also supports sequential access. In indexed files, the predecessor-successor relationship exists among the entries in the index. RMS sequentially accesses records on behalf of the program by moving through a specified index table in serial fashion. The records are retrieved in the same order that key values appear in the table.

PUT operations on indexed files write the record and place its key value in the appropriate index. On GET operations, the pointer for the specified key of reference locates the first record associated with that index and makes it available to the program. The next GET updates the pointer to the record whose key appears next in that index and accesses the record. FIND operations perform in the same manner but without reading the record. UPDATE and DELETE operations require a prior, successful GET or FIND.

2.4.2 Random Access

Random access allows the BASIC program, rather than file organization, to control the order of record access. The program identifies each record of interest in each operation requested of RMS. This procedure allows you to access records in any order at any point in the file.

Random access is not permitted on sequential files because of the strict predecessor-successor relationship maintained among records. Relative and indexed files do allow random access.

Programs employ random access on relative files through the specification of a particular record number. RMS interprets the number as representing a record position in the file. If the operation is a GET or FIND and no record exists in the specified location, RMS returns an error (`%Record not found`). If the operation is a PUT and a record already exists in the specified location, RMS also returns an error (`%Record already exists`).

Note that DELETE and UPDATE operations do not allow record identity specifications. A prior GET or FIND is required. Also, random access imposes no restriction on the order of operations. For example, you can specify a series of GET operations on a relative file in any order (record number 3, record number 9, record number 5, etc.).

Programs initiate random access on indexed files by means of a key specification. You specify a number and key value in a manner determined by the desired operation. For all operations, the specified key value indicates the

contents of a record data field and the number identifies the index that RMS uses to locate that record.

On GET or FIND operations, a specification indicating the content of the desired key field is required. RMS searches the key index table indicated by the specification, finds the desired key value (if present), reads the record pointed to by the index, and passes the record to the program.

PUT operations do not allow an explicit key specification because RMS uses the record's data to interpret the new record in terms of content, position, and length of key data fields.

Indexed files allow you to specify key values in three ways: exact key, approximate key, and generic key. You specify an exact key by including the entire content of the desired field in the operation. You specify an approximate key in your program by indicating that the desired record's key field can be equal to, or greater than, the specified key. You specify a generic key in the program by indicating an initial portion of a key field. These three methods are described in Section 2.2.3.1.

Consider the following example:

```
5     ON ERROR GO TO 19000
10    MAP (PDATA) NAME$=30%,ID$=6%,JOBDES$=20
20    OPEN "PFILE.DAT" FOR OUTPUT AS FILE #1%           &
          ,ORGANIZATION INDEXED FIXED,ACCESS MODIFY     &
          ,ALLOW NONE,MAP PDATA                         &
          ,PRIMARY NAME$,ALTERNATE ID$
30    INPUT "NAME";NAME$
40    IF NAME$=" " THEN 50 ELSE                         &
\             INPUT "ID      ";ID$                      &
\             INPUT "JOBDES ";JOBDES$                   &
\             PUT #1%                                   &
\             GO TO 30
50    CLOSE #1%
60    OPEN "PFILE.DAT" FOR INPUT AS FILE #1%            &
          ,ORGANIZATION INDEXED FIXED,ACCESS READ     ; &
          ,ALLOW NONE,MAP PDATA                         &
          ,PRIMARY NAME$,ALTERNATE ID$
70    GET #1%                                           &
\     PRINT NAME$;ID$;JOBDES$
80    INPUT "ID ";IDENT$
90    IF IDENT$=" " THEN 210 ELSE                       &
            GET #1%,KEY #1% EQ IDENT$                   &
\           PRINT ID$;NAME$;JOBDES$                     &
\           GO TO 80
19000   PRINT "ERROR";ERR,"AT LINE";ERL
32767   CLOSE #1% \END
```

This program creates an indexed file, accepts record data from the terminal, and closes the file. The file is then reopened and its records are accessed with sequential and random GET operations. The program is composed of the following lines:

Lines 5 and 19000	are an error handling routine.
Line 10	is a MAP statement that defines a primary and two alternate keys in terms of their size and location in the record.

Line 20	is an OPEN statement that creates an indexed file, identifies the primary and alternate keys, and references the MAP statement that defines those keys.
Lines 30 and 40	accept record data from you by means of an INPUT statement. The PUT statement writes the data to the file and the MAP statement variables format the data in the record.
Line 50	closes the file.
Line 60	reopens the file. Note that the file attributes are respecified in the OPEN statement.
Line 70	is a GET statement that accesses the first record (sequential access) and prints it.
Line 80	is an INPUT statement that requests an alternate key.
Line 90	is a GET statement that accesses a record based on the alternate key you specify in response to line 80. This is a random operation. Line 90 also prints the record.
Line 32767	closes the file.

The capability to shift from random to sequential access (or vice versa) is only allowed on relative and indexed file organizations. Sequential file organization does not support random access. There is no restriction on the number of shifts that can be made while processing a file.

As an example, consider a program that randomly accesses a file and then dynamically shifts to sequential access. RMS considers the currently accessed record (by the random operation) as the predecessor record when the shift is made to sequential access.

Relative and indexed file organizations impose their own restrictions on the sequence of operations. For example, a GET operation always shifts the program to the specified record. If you follow a series of sequential GET operations with a random PUT, the program remains at the location of the last GET. A sequential GET after the random PUT will resume at the point of the previous GET operation.

2.5 Record Format

RMS is indifferent to the logical content of records, but it does require that you specify the record format. Record format determines the manner in which RMS stores records in the file. The format is specified when the file is created and is permanently assigned to each record read into that file.

BASIC allows you to specify one of two formats. These are:

FIXED	the file contains records of equal and fixed length.
VARIABLE	the file may contain records of different lengths.

The file organization determines which of the formats you can select.

The record format must be specified when the file is created. You specify record format in the BASIC program as part of the organization clause, as follows:

```
OPEN filename  [FOR OUTPUT] AS FILE [#]num-exp
,[ORGANIZATION] {SEQUENTIAL} [{FIXED   }]
                {RELATIVE  } [{VARIABLE}]
                {INDEXED   }
```

Variable format is the default for sequential, indexed, and relative organizations and record length is indicated by a count field appended to each record.

The following subsections discuss each record format in detail.

2.5.1 Fixed-Length Records

Fixed length describes a file condition in which records are of equal and nonvarying length. Under fixed-length format, each record in a file occupies an identical amount of space.

You specify the length of records in the BASIC program when the file is created. The length, in bytes, can be explicitly stated in the RECORDSIZE clause or implicitly defined by a map reference in the MAP clause. RMS stores and maintains the record length specification in the file description header. When a program requests a record from the file, the desired record is passed to the program within the length restrictions defined for that file.

Fixed-length format is optional for sequential, relative, and indexed files. Relative files, however, store records in fixed-length positions, regardless of the format specification. That is, RMS stores relative file records in locations that are each equal to the maximum record size specified when the file was created. This condition is true whether the format is fixed or variable. For example, when you create a relative file, a record position space is allocated that is equal to the largest record described for that file. RMS stores the size in the file header. A program request for a relative file record is performed within the specified amount of space.

2.5.2 Variable-Length Records

Variable-length describes a file condition in which the length of each record is allowed to differ. Variable format is the default for sequential, relative, and indexed file organizations.

When variable-length format is used, you must specify the length of the file's longest record in the RECORDSIZE clause or with a map reference in the MAP clause.

Because record retrieval operations require a record size, RMS prefixes a count field to each record as it is written to the file. The count field identifies individual record size in bytes to RMS but is transparent to the BASIC program. When a program requests a record, RMS releases a record whose length is that specified by the count field.

There are two types of count fields, depending on the device you use to contain the file. Records in files residing on disk devices contain a 1-word (2-byte) binary count field that precedes the data portion of the record. This count field is aligned on a word boundary. The length indicated by the count field does not include the count field itself.

Records in files residing on ANSI magnetic tape (sequential files only) contain a 4-character decimal count field that precedes the data portion of the record. The size indicated by the field includes the field itself. In the context of ANSI tapes, this record format is known as D format.

Relative files are an exception in that variable format is allowed but record position length is fixed. The length of each record position is defined by the size of the largest record. A count field prefixes each record, but these records need not fill an entire record position.

When you create relative or indexed files with variable format, you must define the record size as a non-zero specification that represents the size of the largest record. Note that a record is never allowed to exceed the RMS maximum of 16,383 bytes.

2.6 Data Structure

Data structure is a term that describes the storage of a file on a particular medium. When you create a file, RMS uses certain data storage structures to allocate and maintain the records that compose that file. These structures are blocks and buckets.

A block is a physical storage structure that can contain a partial record, one full record, or more than one record. The size of a block on disk devices is fixed at 512 bytes. The size of a block on magnetic tape can be defined in your program. Because sequential is the only file organization allowed on magnetic tape, the size of a block is a consideration only when creating sequential files on magnetic tape. This consideration is discussed in Section 2.5.1.

A bucket is a logical data structure that is composed of blocks. Buckets are used for relative and indexed files on disk devices and RMS allows you to establish the size of a bucket in terms of an integral number of blocks. Buckets are described in Section 2.5.2.

2.6.1 Blocks

The records that your program writes to a file are contained on blocks. The size of these records determines whether a block contains a partial record, one full record, or more than one record. RMS considers each block within a file as a contiguous array of data. When you write a record that is larger than one block, RMS allocates successive blocks sufficient to contain the entire record. The procedure whereby records cross block boundaries is called spanning.

The length of a block on disk devices is fixed at 512 bytes. This size is set by the hardware and cannot be altered. The length of a block on magnetic tape is defined as the length of data that the program writes between two inter-record

gaps. With ANSI-formatted tapes, you can specify this size in the BLOCKSIZE clause as a positive integer that represents the number of records. The range of this integer is from a minimum of 18 bytes to a maximum determined by program buffer requirements.

The BLOCKSIZE clause appears in the OPEN statement that is used to create sequential files on magnetic tape. The BLOCKSIZE specification defines block length in terms of the number of records and permanently assigns it to the file. Consider the following:

OPEN *filename* [FOR OUTPUT] AS FILE [#]*num-exp*
,[ORGANIZATION] SEQUENTIAL
,RECORDSIZE *num-exp*
,BLOCKSIZE *num-exp*

RECORDSIZE
defines the size of the largest record in the file.

BLOCKSIZE
defines the size of a block in number of records. The default for disk devices is 512 bytes.

2.6.2 Buckets

A bucket is a logical storage structure that RMS uses to build and maintain files on disk devices. A bucket is composed of an integral number of blocks in the range of 1 to 31. Bucket size is defined in terms of the number of records it contains and this number can be defaulted to one record or specified in your program.

Because relative and indexed files are allowed only on disk media, the length of a block for these files is set at 512 bytes. This size cannot be altered in your program. A bucket, however, is a logical structure and its size can be tailored to program requirements.

Unlike blocks, a bucket cannot contain a partial record. That is, RMS does not allow records to span bucket boundaries. Therefore, when you specify a bucket size in your program, you must consider the size of the largest record in the file. If a default bucket size is used, BASIC makes this consideration automatically.

In addition to your file's records, buckets contain internal information that is maintained and understood only by RMS.

There are two methods you can use to establish the number of blocks in a bucket. The first is to use the BASIC default. The second method involves a specification of the number of records you desire in each bucket. BASIC calculates a default based on the number of records you specify. These two variations on default sizes are discussed in Section 2.5.2.1.

2.6.2.1 Bucket Size — The default bucket size assigned to relative and indexed files is designed to make the bucket size as small as possible. The

default size minimizes memory buffer space requirements but also decreases the speed of I/O operations.

A default bucket size is selected by BASIC on the basis of information that you provide when the file is created. If you do not define the BUCKETSIZE clause in the OPEN statement, BASIC assumes that there is only one record in the bucket, calculates a size, and assigns the required number of blocks. If you define BUCKETSIZE and specify the number of records (when more than one is desired in each bucket), BASIC uses a different formula to arrive at the necessary number of blocks. BASIC also considers file organization and record format when determining default bucket size. These considerations are shown in the following formulas and tables. Note that record size can alternately be defined by a map reference.

The BASIC syntax used to create a file in which BASIC completely controls bucket size is as follows:

```
OPEN filename [FOR OUTPUT] AS FILE [#]num-exp
,[ORGANIZATION] {RELATIVE}[{FIXED   }]
                {INDEXED }[{VARIABLE}]
,RECORDSIZE num-exp
```

The BASIC syntax used to create a file in which you state the number of records desired in the bucket is as follows:

```
OPEN filename [FOR OUTPUT] AS FILE [#]num-exp
,[ORGANIZATION] {RELATIVE}[{FIXED   }]
                {INDEXED }[{VARIABLE}]
,RECORDSIZE num-exp
,BUCKETSIZE num-exp
```

where the BUCKETSIZE specification is the number of records expressed as a positive integer.

The default bucket size for relative files is derived from the following formulas:

- Fixed-length records with no BUCKETSIZE specification,

  $Bnum=(1+Rlen)/512$

- Fixed-length records with BUCKETSIZE specified,

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

- Variable-length records with no BUCKETSIZE specification,

  $Bnum=(3+Rmax)/512$

- Variable-length records with BUCKETSIZE specified,

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

B*num* is the number of blocks per bucket in a range of 1 to 31 blocks. The bucket size is rounded up to the next highest integer, where required.

R*len* is the length in bytes of the file's fixed-length records as defined in the RECORDSIZE clause.

R*max* is the length in bytes of the largest variable-length record in the file as defined in the RECORDSIZE clause.

R*num* is the number of records that you desire in each bucket as defined in the BUCKETSIZE clause.

1 represents the existence byte that RMS uses to determine the presence or absence of records in the file.

3 represents the existence byte plus two bytes that indicate the count field.

Table 2–4 shows a partial list of the default bucket sizes selected by BASIC when the number of records is undefined (i.e., the bucket contains only one record).

Table 2–4: Relative File Default Bucket Size

Bnum	*Rlen*	*Rmax*
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

The default bucket size for indexed files is derived from the following formulas:

- Fixed-length records with no BUCKETSIZE specification,

  B*num*=(22+R*len*)/512

- Fixed-length records with BUCKETSIZE specified,

  B*num*=((7+R*len*) * R*num*)+15/512

- Variable-length records with no BUCKETSIZE specification,

  B*num*=(24+R*max*)/512

- Variable-length records with BUCKETSIZE specified,

  B*num*=((9+R*max*) * R*num*)+15/512

B*num* is the number of blocks per bucket in a range of 1 to 31 blocks. The bucket size is rounded up to the next highest integer, where required.

R*len* is the length in bytes of the file's fixed-length records as defined in the RECORDSIZE clause.

R*max* is the length in bytes of the largest variable-length record in the file as defined in the RECORDSIZE clause.

R*num* is the number of records you desire 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. (Note that when BUCKETSIZE is defined, 7 bytes are allotted to each record in the bucket and 15 bytes to the bucket as a whole.)

24 is a 15-byte RMS bucket overhead plus 9 bytes for the variable-format record header length. (Note that when BUCKETSIZE is defined, 9 bytes are allotted to each record in the bucket and 15 bytes to the bucket as a whole.)

Table 2-5 shows a partial list of the default bucket sizes selected by BASIC when the number of records is undefined (i.e., the bucket contains only one record).

Table 2-5: Indexed File Default Bucket Size

B*num*	R*len*	R*max*
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

When you specify a bucket size for files in your program, you should keep in mind the space versus speed considerations involved. That is, a large bucket size increases the speed of file processing but also increases the memory space required for buffer allocation. Likewise, a small bucket size minimizes buffer requirements and also decreases the speed of operations. For example, a large bucket size contains a greater amount of the file in each bucket. When an I/O operation accesses a bucket, this greater amount of file is made available for processing. However, a like amount of buffer space is required to contain the file.

2.7 Record Mapping

NOTE:

Because a RECORDSIZE specification overrides a MAP, it is possible to define a record size that is larger than the MAP and cause a record operation to overwrite mapped areas. A fatal error results if you specify a RECORDSIZE that is larger than a previously defined MAP statement for the same file.

When you initiate a record operation, such as a PUT or UPDATE, the record appears to move directly to your program from the file or to the file from your program. RMS transports these records from or to blocks or buckets, depending on the organization of the file (see Section 2.5).

RMS, however, does not directly transfer records between programs and files. Transparent to you, RMS reads or writes records into internal memory areas called buffers. Buffers, therefore, are an intermediate step between files and programs. The unit of transfer between the file and the buffer is the storage structure (i.e., a block or bucket). The unit of transfer between the program and the buffer is a record.

During record operations, RMS controls the content of buffers. However, the program determines the allocation of buffer space and the content of the records in those buffers through record mapping.

The buffer is a data storage location whose size and content can be described in an optional MAP statement. The MAP statement acts as a template for the placement of data in a record. That is, it generates a PSECT of the same name with a length equal to the sum of the MAP elements. The MAP clause in the OPEN statement references the MAP statement and associates it with a particular file.

The MAP statement appears in your program as follows:

```
MAP (map-name) element-list
```

The MAP name is enclosed by parentheses and represents the buffer name. It cannot be a BASIC-PLUS-2 reserved word. It provides RMS and the program with a vehicle for associating record operations with a buffer in the OPEN statement. The element list is composed of variables that represent the data. The list also defines how that data is to be placed in the record.

More than one MAP statement can exist with the same name. If this is the case, the variables in the element-list must be contained in the same position in each map. In addition, MAP statements can appear before or after the OPEN statement.

Because the MAP statement defines the data content of the record, it also acts to define the position and length of indexed file keys. Both the primary and alternate KEY clauses in an indexed file OPEN statement refer to ele-

ments in a MAP statement when key values are specified. Note that once a key field has been defined, by means of a KEY specification and a map reference, it is not allowed to change.

The MAP clause that associates a defining MAP statement with a particular file appears in the OPEN statement as follows:

OPEN *filename* [FOR OUTPUT] AS FILE [#]*num-exp*
,[ORGANIZATION] { SEQUENTIAL } [{ FIXED }]
{ RELATIVE } [{ VARIABLE }]
{ INDEXED }
,MAP *map-name*

The *map-name* in the MAP clause is associated with the file while the file is open.

If you use the MAP clause, the allocated buffer space is the MAP. However, if you use RECORDSIZE to define the length of records, buffer space is allocated from the program's dynamic free space. Consider the following example:

```
10    PRINT "SEQUENTIAL MAP TEST WITH FIXED LENGTH RECORDS"
20    OPEN "RMSSEQ.FIX" FOR OUTPUT AS FILE #1%          &
         ,ORGANIZATION SEQUENTIAL FIXED,ACCESS          &
         MODIFY,MAP MAP1
30    MAP (MAP1) NAME$=30%,IDNUM%,JOBCLASS$=9%
40    INPUT "NAME       ";NAME$                         &
\     IF NAME$= "END" THEN 100
50    INPUT "ID NUMBER";IDNUM%                          &
\     INPUT "JOB CLASS";JOBCLASS$
60    PUT #1% \GO TO 40
100   CLOSE #1% \END
```

This program creates a sequential file with fixed-length records. The maximum record size is 41 bytes and the length of the record's content is defined in a map reference. The map reference is contained in line 20. Line 30 contains the defining MAP statement referred to in line 20.

Because the MAP statement defines the length of data in the record, it should be used in the OPEN statement to define the size of records. In addition, a map reference and a RECORDSIZE specification should not appear in the same OPEN statement. Note that when both a map reference and a RECORDSIZE specification are used, the RECORDSIZE specification takes precedence.

Chapter 3
BASIC–PLUS–2 on RSX–11M

This chapter describes the interface between the BASIC–PLUS–2 compiler and operating systems that use the MCR command language. The description includes compiler invocation, linkage of object modules to produce an executable task, and task execution. The operating system specific information in this chapter is a summary only. You are expected to be familiar with the operating system and with the information found in the documentation that is specific to your application.

3.1 Compiler Invocation on RSX–11M

To invoke the BASIC–PLUS–2 Compiler on systems with an RSX command interface, type the following command in response to your system prompt:

```
> RUN $BASIC2 RET
```

If compiler invocation is successful, BASIC–PLUS–2 prints an identifying line (see Section 1.2.1). With the compiler invoked, you can create a BASIC source program and object modules as described in Chapter 1. Note that an option in the installation procedures allows the system manager to change the BASIC–PLUS–2 invocation command, as follows:

```
> BP2 RET
```

3.2 Task Builder Usage on RSX–11M

The Task Builder is a system program that is used to process one or more object modules into a single, executable file in task image format. Refer to the *RSX–11M Task Builder Reference Manual* for information about using the Task Builder program.

An object module is a user program that has been compiled with the BASIC command COMPILE (see Section 1.2.7). Programs created as object modules have the .OBJ file type appended to the file name by default. They can be executed only after being processed by the task builder.

The task builder accepts object code as input, resolves any switches or options you have specified in the command line, and outputs code in executable task image format.

The BASIC–PLUS–2 compiler generates both Overlay Description Language (ODL) files and indirect Task Builder command files that are based upon the BUILD command. These files are sufficient for single segment tasks. You must reconstruct the ODL file if you overlay the user segment, as described in the *RSX–11M Task Builder Reference Manual.*

The BASIC Compiler BUILD command (see Section 1.2.8) offers you a simplified procedure for specifying task builder input. The BUILD command accepts object module names in its command line and produces a command file. This file contains all of the required task builder command input. For example:

```
BUILD MOD1, MOD2, MOD3/MAP  RET
```

generates a command file named MOD1.CMD. When this file is typed in response to the task builder prompt:

```
TKB> @MOD1  RET
```

the task builder generates a task image file (MOD1.TSK) and a map (MOD1.MAP). Note that if you desire task builder options you must edit the command file generated by the BUILD command.

3.2.1 Task Builder Options

The options are specified as input to the task builder and define the characteristics of the task image. The options take the form of a keyword followed by an equal sign and an argument. The argument assigned to the option is dependent on the desired characteristic. This section summarizes the options that are most useful to BASIC programmers. For a complete description of task builder options, refer to the Task Builder Reference Manual appropriate to your system.

Table 3-1 lists the option keywords and their meanings.

Table 3-1: Task Builder Options

Keyword	Meaning
ASG	Declares device assignments to logical units
EXTTSK	Extends the amount of memory allocated to a task
LIBR	Associates task with shareable library
UNITS	Declares the maximum number of logical units

The ASG option assigns a specified physical device to one or more logical units.

The ASG option has the form:

```
ASG = device name:unit 1:,,,unit n
```

device name is a 2-character alphabetic device name followed by an optional 1- or 2-digit device unit number.

unit is a decimal integer that indicates the logical unit number.

The default is ASG = SY0:1:2:3:4,TI:5,CL:6

Note that there is a direct correspondence between BASIC-PLUS-2 channel numbers and operating system logical unit numbers (LUNs). BASIC-PLUS-2 requires LUN 13 for the user terminal and LUN 14 for a work unit, therefore, you must specify 14 units and assign unit 13 to your terminal. When the UNITS option and ASG are part of the same input specification, UNITS must precede ASG.

The UNITS option specifies the number of logical units used by the task and reserves sufficient space for the number of specified units in the task's header. The number of logical units assigned by default is 6 and the maximum number that can be used in a BASIC-PLUS-2 task is 14.

The UNITS option has the form:

```
UNITS = max-units
```

where *max-units* is a decimal integer in the range of 0 to 14.

The EXTTSK option extends the amount of memory that is initially allocated to a task. The option causes additional memory allocation when the task is loaded.

The EXTTSK option has the form:

```
EXTTSK = length
```

where *length* is a decimal number that specifies in words the increase in task memory allocation. Note that the task itself attempts to expand as required. If you attempt to extend memory allocation beyond the system partition size or the resident library maximum allocation (i.e., 16K words), a fatal error is returned at run time (`?Not enough available memory`).

The EXTTSK option can be used to preallocate space for string manipulation and I/O buffers. BASIC–PLUS–2 normally uses the minimum required space. Therefore, the use of EXTTSK can provide additional space and cause some increase in the speed of program execution by decreasing the number of task extends. Refer to Table 2–2 for the formulas that determine initial space allocation estimates.

The LIBR option associates the task with a specific resident shareable library in memory. BASIC–PLUS–2 programs can use the optional BASIC2 library or a user-created library. If you use LIBR to access an optional library, the task builder includes the symbol definition file of the specified library in the input file. For example, if you specify BASIC2, the task image is associated with the BASIC2 resident library and BASIC2.STB (located in account [1,1]) is included in the input file.

The LIBR option has the form:

```
LIBR = library:RO
```

where *library* is a specified resident shareable library and RO is read-only access.

Note that if you wish to specify BASIC2 or a user-created library in the BUILD command output, you must use the LIBRARY command (see Section 1.2.9). You must edit the Task Builder indirect command file if you include multiple shared libraries.

3.3 Task Execution on RSX–11M Systems

The task builder outputs executable code that can be invoked and executed at operating system level. The sequence of events leading up to task execution is as follows:

1. Creating one or more object modules by means of the BASIC command COMPILE.
2. Specifying the object modules, along with any desired switches and options, as input to the task builder, or using BUILD to create a command file that contains task builder command input.
3. Obtaining task builder output of executable code (task image) and a map file if desired.
4. Issuing the appropriate system command to execute the created task.

As examples of the procedures you might use to build an executable task, consider the following series of commands.

Input consists of two object modules (MYPRG1 and MYPRG2), and a BASIC2 library.

```
OLD MYPRG1  RET

Basic2

COM  RET

Basic2

OLD MYPRG2  RET

Basic2

COM  RET

Basic2

BUILD MYPRG1, MYPRG2/IND  RET

Basic2

EXIT  RET

>TKB @MYPRG1  RET

>
```

In this command series, BUILD is used to create a command file (MYPRG1.CMD) composed of a previously compiled object module. The command file contains all of the libraries and options required as input to the task builder as well as the BASIC switch (/IND) required to enable the use of RMS indexed I/O. The command file is used as input to the task builder prompt and the result is an optional map file and an executable task. The use of an RMS switch (/VIR, /SEQ, /REL, or /IND) causes the BUILD command to change the generated .ODL file as required for RMS I/O. These changes are made automatically when the appropriate switch is appended to the BUILD command. Consider the following example of MYPRG1.ODL:

```
     .ROOT BIROT4-USER,RMS
USER:,FCTR MYPRG1-MYPRG2-LIBR
LIBR:,FCTR [1,1]BASIC2/LB
RMS: ,FCTR BIO047
@SY:[1,1]     BASIC4
     .END
```

3.4 BASIC–PLUS–2/RSX–11M Notes

The adaptation of BASIC–PLUS–2 to different operating system environments causes differences in the implementation of certain BASIC features. The following sections describe those areas of difference that apply to operating systems with the MCR command interface.

3.4.1 CHAIN Statement

The BASIC–PLUS–2 CHAIN statement allows a line number specification that permits you to initiate chaining at a specified point in the program. However, the RSX–11M operating system requires that a chain begin at the first line of the program. Thus, the BASIC–PLUS–2 syntax:

```
CHAIN file-exp[LINE num-exp]
```

is not permitted on RSX–11M. For RSX systems, the syntax is as follows:

```
CHAIN "task name"
```

where *task name* is the name of a previously installed task. Also, BASIC–PLUS–2 on RSX–11M accepts only the first six characters of the task name in the statement line.

3.4.2 NAME AS Statement

The BASIC–PLUS–2 NAME AS statement permits you to rename an existing file. The statement has the following format:

```
NAME string 1 AS string 2
```

where *string 1* is the file specification of the target file and *string 2* is the new file specification.

On RSX–11M systems, the NAME AS statement is subject to the following restrictions:

1. You must have write access to the directory of the target file.
2. The files specified in the statement line must reside on the same physical device and have the same User Identification Code (UIC).
3. The PSECT $$FSR2 must be in the root segment of the task. This can be done by reworking the ODL file to force $$FSR2 into the root, or by including the "NAME AS" in the root. If it is necessary for other subroutines to rename files, they should call the one in the root segment.

The NAME AS statement does not alter the contents of a file. It renames the first specified file to that of the second file without changing the version number. Since the NAME AS statement alters the file name, you must include a file organization switch in the BUILD command. Also, if the target of the statement is an open file, the new name does not take effect until the file is closed.

3.4.3 SLEEP Statement

The BASIC–PLUS–2 SLEEP statement suspends program execution for a specified amount of time. The statement has the format:

```
SLEEP num-exp
```

where *num-exp* is the number of seconds that execution is suspended.

On RSX–11M systems, you may enable CONTROL C trapping for the job prior to issuing the SLEEP statement. Then, if you wish to prematurely reactivate a job, the CTRL/C system function can be utilized.

Chapter 4
BASIC–PLUS–2 on IAS

This chapter describes the interface between the BASIC–PLUS–2 compiler and IAS operating systems that use the DCL command language. The description includes compiler invocation, linkage of object modules to produce an executable task, and task execution. The operating system specific information in this chapter is a summary only. You are expected to be familiar with the operating system and with the information found in the documentation that is specific to your application.

4.1 Compiler Invocation on IAS

To invoke the BASIC–PLUS–2 Compiler on systems with a DCL command interface, type the following command in response to your system prompt:

```
BASIC2 (RET)
```

If compiler invocation is successful, BASIC–PLUS–2 prints an identifying line (see Section 1.2.1). With the compiler invoked, you can create a BASIC source program and object modules as described in Chapter 1.

4.2 Task Builder Invocation on IAS

The task builder is a system program that is used to process one or more object modules into a single, executable file in task image format.

An object module is a user program that has been compiled with the BASIC command COMPILE (see Section 1.2.7). Programs created as object modules have the .OBJ file type appended to the file name by default. They can be executed only after you process them by means of the task builder. The task builder accepts the object code, resolves any references to BASIC library modules, and outputs code in executable task image format.

To invoke the task builder on systems with a DCL command interface, type the following command:

```
PDS>@input
```

where *input* can be a command file that was generated with the BUILD command (see Section 1.2.8) or a LINK command line. These two specifications are discussed in Section 4.2.1.

4.2.1 Link Command Line Input

In the LINK command line you can specify qualifiers and files in the following format:

```
LINK [/qualifiers] filespec 1 [,filespec 2,,,]
```

qualifiers are one or more specifications that modify task builder output as described in Section 4.2.2.

filespec are one or more object modules with a file specification and an .OBJ file type. Each specified object module is separated by commas.

After you type the command line, the task builder builds the task image, outputs a task image file and a map (if these files are requested), and resolves any specified qualifiers.

The BASIC Compiler command BUILD (see Section 1.2.8) offers you a simplified procedure for specifying task builder input. The BUILD command accepts object module names in the command line and produces a command file. This file contains all of the required task builder input. For example:

```
BUILD MOD1, MOD2, MOD3
```

generates a command file named MOD1.CMD. To invoke the task builder and input this file, type the following command:

```
PDS>@MOD1    RET
```

This command line results in a task image file (MOD1.TSK) and a storage map file (MOD1.MAP). Note that you cannot use the unmodified output of the BUILD command when you desire to qualify task builder output or specify options (see Section 4.2.2). In these cases, you must specify the complete LINK command line or use an editor to modify the BUILD command file.

The files that are processed by the task builder are assigned file types by default. Table 4–1 lists these file types and the applicable file.

Table 4–1: IAS Default File Types

File Type	File
.TSK	Task image file
.MAP	Memory Allocation map
.OBJ	Input object module
.OLB	Library file
.ODL	Overlay description file
.CMD	Command file

Input to the task builder consists of one or more object modules, any required libraries, optional qualifiers, and options.

The object modules can be input as file specifications or the file names alone. When you type the complete file specification, the task builder assigns any specified UIC number, device, and file type to the task image. If you specify the file names alone, the system defaults are used.

The qualifiers and options also have default settings. In most cases, you can override these by specifying the desired setting in the command line. The qualifiers and options, their defaults, and functions, are summarized in Section 4.2.2. For additional information on these specifications, refer to the Task Builder Reference Manual appropriate to your system.

4.2.2 Qualifiers

The specification of a qualifier follows the LINK command and is preceded by a slash, as follows:

```
LINK/qualifier
```

No specification is required when the qualifier is the default, however, you can precede the qualifier with NO to negate its effect. For example:

`/qualifier`	specifies action
`/NOqualifier`	negates the action

This section summarizes the qualifiers that you can specify to the task builder. The section describes the action caused by the qualifier, the file it applies to, and its default. The qualifiers described here are those that would be most useful to the BASIC programmer; for information on the full set of qualifiers refer to the Task Builder Reference Manual appropriate to your system.

The /TASK qualifier causes the task builder to generate a task image file and has the following format:

```
/[NO]TASK[:filespec]
```

/TASK is the default. If you specify /NOTASK, the task builder does not construct an executable task image file. The task builder does, however, check the input for errors and print appropriate diagnostic error messages. *Filespec* represents a file specification and allows you to assign a name to the generated task image file. If you omit *filespec*, the task builder assigns the name of the leftmost input file as the task name. The task image file type is .TSK by default.

The /MAP qualifier causes the task builder to generate a memory allocation map and has the following format:

```
/[NO]MAP[:filespec]
```

/NOMAP is the default. *Filespec* represents a file specification and allows you to assign a name to the map file with a .MAP default file type. If you do not specify a file name and a map is requested, the task builder assigns the name of the leftmost input file.

The /SYMBOLS qualifier causes the task builder to generate a symbol table file and has the following format: /[NO]SYMBOLS[:filespec] /NOSYMBOLS is the default. *Filespec* represents a file specification and allows you to assign a file name to the symbol table file with an .STB default file type. If you do not specify a file name and an .STB file is requested, the task builder assigns the name of the leftmost input file.

The /OPTIONS qualifier causes the task builder to solicit task options. The format of this qualifier is as follows:

```
/[NO]OPTIONS
```

/NOOPTIONS is the default. The options and their effect on the linked task are described in Section 4.2.3.

The /OVERLAY qualifier causes the task builder to create a task image based on a defined overlay structure. The format of this qualifier is as follows:

```
/[NO]OVERLAY:filespec
```

/NOOVERLAY is the default. If you specify /OVERLAY, the overlay structure must be defined in a specified .ODL file (filespec). Because input files are described in the overlay structure and are included in the .ODL file, input file specifications (.OBJ files) cannot be included in the LINK command line. Refer to Section 3.3 for a description of overlays and the creation of .ODL files.

The /LIBRARY qualifier is appended to input files that contain object module libraries and has the following format:

```
LINK[/qualifiers] filespec1/LIBRARY
```

where *filespec1* is an input object module with an .OLB default file type that references a BASIC library. /NOLIBRARY is the default. Refer to Section 1.2.9 for a description of BASIC libraries.

Consider the following example:

```
LINK/MAP MOD1, MOD2, MOD3, BASIC2/LIBRARY
```

This command line links the object modules MOD1, MOD2, and MOD3 into an executable task named MOD1.TSK. BASIC2 is identified as a library by

the /LIBRARY qualifier. The /MAP qualifier causes the task builder to generate a memory allocation map file named MOD1.MAP.

4.2.3 Link Options

When you specify the /OPTIONS qualifier in the LINK command line, the task builder expects one or more option specifications to appear on the following line. You specify options in the form of a keyword followed by an equal sign and an argument. The argument assigned to the option is dependent on the desired task characteristic. This section summarizes the options that are most useful to BASIC programmers. For a complete description of options, refer to the Task Builder Reference Manual appropriate to your system.

Table 4–2 lists the option keywords and their meanings.

Table 4–2: LINK Options

Keyword	Meaning
NASG	Declares device assignments to logical units.
EXTTSK	Extends the amount of memory allocated to a task.
UNITS	Declares the maximum number of units.

The ASG option assigns a specified physical device to one or more logical units.

The ASG option has the form:

```
ASG = device name:unit 1:...unit n
```

device name is a 2-character alphabetic device name followed by an optional 1- or 2-digit device unit number.

unit is a decimal integer that indicates the logical unit number.

The default is ASG=SY0:1:2:3:4:,TI:5,CL:6

Note that there is a direct correspondence between BASIC-PLUS-2 channel numbers and operating system logical unit numbers (LUNs). BASIC-PLUS-2 requires LUN 13 for the user terminal and LUN 14 for a work unit, therefore, you must specify 14 units and assign unit 13 to your terminal. If your program requires the use of units (channels) five and six, you must override the ASG default with an explicit ASG specification.

Also, when the UNITS option and ASG are both given as options, UNITS must precede ASG.

The UNITS option specifies the number of logical units used by the task and reserves sufficient space for the number of specified units. The number of

logical units assigned by default is 4 and the maximum number that you can specify in the option is 14.

The UNITS option has the form:

```
UNITS = max-units
```

where *max-units* is a decimal integer in the range of 0 to 14.

The EXTTSK option extends the amount of memory that is initially allocated to a task. The option causes additional memory allocation when the task is loaded. The EXTTSK option has the form:

```
EXTTSK = length
```

where *length* is a decimal number that specifies in words the increase in task memory allocation. Note that the task itself attempts to expand as required. If you attempt to extend memory allocation beyond the system partition size or the resident library maximum allocation (16K words), a fatal error is returned at run time (`?Not enough available memory`).

The EXTTSK option can be used to pre-allocate space for string manipulation and I/O buffers. BASIC–PLUS–2 normally uses the minimum required space. Therefore, the use of EXTTSK can provide additional space and cause some increase in the speed of program execution.

4.3 Task Execution on IAS

The task builder outputs executable code that can be invoked and executed at operating system level. The sequence of events leading up to task execution is as follows:

1. Creating one or more object modules by means of the BASIC command COMPILE.
2. Specifying the object modules, along with any desired qualifiers and options, as input to the task builder, or using the BUILD command to create a command file that contains all of the required task builder input.
3. Obtaining task builder output of executable code (task image) and a map file if desired.
4. Issuing the appropriate system command to execute the created task.

As examples of the procedures you might use to build an executable task, consider the following series of commands:

```
LINK/MAP MYPROG1, MYPROG2,[1,1]BASIC2/LIBRARY
```

This command causes the output of a task image (MYPROG1.TSK) and a memory allocation map file (MYPROG1.MAP). Input consists of a MAP qualifier, two object modules (MYPROG1 and MYPROG2), and a

BASIC-PLUS-2 library (BASIC2). The library specification contains the directory under which the library is stored, the library file specification, and the LIBRARY qualifier.

```
OLD NONAME  RET

Basic2

COM  RET

Basic2

BUILD /IND  RET

Basic2

EXIT  RET

PDS>@NONAME  RET
```

In this command series, BUILD is used to create a command file (NONAME.CMD) composed of a previously compiled object module. The command file contains all of the libraries and options that are required input to the task builder as well as the BASIC switch (/IND) that enables the use of RMS indexed I/O. The command file is used as input to the task builder and results in a map file and an executable task (NONAME.MAP and NONAME.TSK).

The use of an RMS switch (/VIR, /SEQ, /REL, or /IND) causes the BUILD command to change the generated .ODL file as required for RMS I/O. These changes are made automatically when the appropriate switch is appended to the BUILD command. Consider the following example of NONAME.ODL:

```
      ,ROOT BIROT4-USER,RMS
USER:,FCTR NONAME-LIBR
LIBR:,FCTR [1,1]BASIC2/LB
RMS: ,FCTR BIO047
@SY:[1,1]  BASIC4
      ,END
```

4.4 IAS Restrictions

The use of BASIC-PLUS-2 on IAS operating systems is subject to the following restrictions:

1. You cannot use Control-C trapping in BASIC-PLUS-2 running on IAS V3.0. If you type CTRL/C, the system does not take any action. This does allow programs that utilize Control-C trapping on other operating systems to run on IAS.
2. If you use the BASIC-PLUS-2 CHAIN statement, IAS implements it as an RQST system directive. Therefore, in order to use the CHAIN statement in BASIC-PLUS-2 programs, contact your System Manager to obtain the privileges and instructions necessary for RQST$ macro and real-time usage.

Appendix A
BASIC–PLUS–2 Language Elements

This appendix summarizes the BASIC–PLUS–2 commands, statements, operators, and functions that are supported on PDP–11 operating systems. If you desire more information on the language elements, refer to the *PDP–11 BASIC–PLUS–2 Language Reference Manual.*

The documentation conventions used in examples of usage are as follows:

KEYWORD	Words in upper case indicate BASIC–PLUS–2 vocabulary that you type as shown.
data	Words in *lower case* indicate variable information that you supply.
[ ]	Square brackets indicate optional information.
{ }	Braces indicate that, of several elements, one must be selected.

A.1 Line and Data Format

BASIC–PLUS–2 program lines are composed of the following elements:

1. Line Numbers

   Program lines require line numbers except where the line is a continuation. BASIC–PLUS–2 line numbers are positive numbers in the range of 1 to 32767. A number outside of this range generates an error. A fractional line number or a line number with a percent sign appended to it generates an error during compilation. Leading zeroes have no effect; leading spaces are allowed.

2. Comments

   Comments begin with an exclamation point (!) and end with another exclamation point or a line terminator. You can insert comments before, within, or between statements and, in these cases, the comments are delimited on both sides by exclamation points. Comments are listed with the program but have no effect on execution speed or size.

3. Statement Separator

   You must separate each statement on a multi-statement line with a backslash statement separator (\).

4. Continuation

   Program lines are continued to the next line when you type an ampersand (&) followed by a line terminator. Note that this usage disallows the appearance of a non-continuation ampersand as the last character of a line (except for those in string literals).

5. Line Length

   BASIC-PLUS-2 places no restriction on the length of a logical program line. A physical line is limited to 255 characters, but you can use continuations to logically extend the line.

6. Line Terminator

   You can terminate program lines with a carriage return/line feed combination or an escape (ESC key).

BASIC-PLUS-2 program lines can contain the following elements:

1. Character Set

   BASIC-PLUS-2 accepts the full ASCII character set as described in Appendix D. Null characters are ignored; non-printing, non-control characters are accepted in string literals but are ignored outside of strings and generate warnings. BASIC converts all lower-case alphabetics to upper case (except for those in string literals).

2. Operators

   BASIC-PLUS-2 accepts arithmetic, relational, and logical operators. Tables A-1 through A-3 illustrate these operators and their use.

3. Constants

   BASIC-PLUS-2 accepts three types of constants: floating-point, string, and integer. Floating-point constants are numbers in the range of $10E-38<n<10E38$ (where n is the constant). Integer constants are also decimal numbers in the range of -32767 to +32767 but are terminated with a percent sign. String constants are alphanumeric characters delimited by single or double quotation marks. The quotation marks must be a matched set and must appear on both sides of the constant. Quoted strings can contain from 0 to 255 characters.

4. Variables

   BASIC-PLUS-2 accepts three types of variables: floating-point, string, and integer. Floating-point variables consist of a single letter followed by up to 29 optional letters, digits, and periods. Integer variables also consist of a single letter optionally followed by up to 29 letters, digits, and periods and terminated by a percent sign. If a percent sign is not specified, the variable is considered floating-point. String variables consist of a single letter optionally followed by up to 29 letters, digits, and periods and terminated by a dollar sign.

You can use any alphanumeric combination for a variable name with the exception of keywords (that is, words that are part of the BASIC language). Keywords are reserved and their use as variable names will produce an error during compilation (see Section A.5).

You designate an array by specifying a floating-point, integer, or string variable followed by subscripts in parentheses. Subscripts are in the range of 0 to 32767 and a maximum of two can be specified. One subscript indicates a 1-dimensional array; two subscripts separated by commas indicate a 2-dimensional array. Subscripts can be integers or expressions, but non-integer subscripts are truncated to an integer value.

Variables are initialized to 0 or a null string at the start of program execution. However, it is recommended that you explicitly initialize all program variables as desired. Note that variables in COMMON, MAP statements, and virtual arrays are not zeroed.

5. Expressions

   An expression can consist of constants, variables, or functions separated by an operator.

6. Functions

   Functions are listed in Section A.4. The general format of a function is a multi-character name 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 except that the function name begins with FN followed by 1 to 30 letters, digits, or periods. A percent sign or dollar sign terminator is also allowed for integer and string functions, respectively.

A.2 Commands

Commands allow you to perform certain operations on the program and do not require a line number. You type them directly to BASIC along with any legal arguments.

The following is a brief description of the BASIC commands, their format, and use. For a more detailed explanation, refer to Chapter 1 of this manual.

Command Format	Use
APPEND *filespec*	Merges a previously saved source program (*filespec*) with one in memory.
BUILD *filespec/sw*	Produces a command file from specified object modules and uses certain switches. This file contains all of the task builder command input required to create a task image.
COMPILE *filespec/sw*	Translates the current program. This command can be combined with certain switches. If a file name is specified, the program is compiled under that name.

DELETE *line number(s)*	Erases specified lines from the current program.
EXIT	Terminates access to the BASIC–PLUS–2 Compiler and returns you to you private default run-time system.
IDENTIFY	Prints a header that identifies the BASIC–PLUS–2 Compiler.
LIBRARY	Allows you to specify a BASIC–PLUS–2 resident library.
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 your directory area for the creation of a program. If you specify a file name, the new program is assigned that name.
OLD *filename*	Brings a program from disk into memory.
RENAME *filename*	Changes the name of the current program to the specified name.
REPLACE *filespec*	Saves the current program by overriding any file with that name.
SAVE *filespec*	Stores the current program as source code. The program is saved under the current name unless another is specified.
SCALE *val*	Sets the scale factor to a designated value or prints the current value if none is specified. The range of *val* is 0 to 6.
SHOW	Prints the current switch values on the terminal.
UNSAVE *filespec*	Deletes a specified file.

A.3 Statements

CALL

CALL *name* [(*actual arguments*)]

```
200  CALL SUB1 (A,B)
```

The CALL statement transfers control to a specified subprogram, transfers parameters, and saves the state of the calling program. Parameters contained in the argument list must agree in type and number with the corresponding SUB statement.

CHAIN

CHAIN *string*

```
15  CHAIN "SEE"
```

The CHAIN statement passes control to a specified, previously installed program.

CHANGE

CHANGE *list* TO *string-variable*

or

CHANGE *string-expression* TO *list*

```
25  CHANGE A% TO A$
```

The CHANGE statement converts a list of integers (real numbers are truncated) into a string of characters and vice versa. The length of the string is determined by the value found in element 0 of the list.

CLOSE

CLOSE [#] *expression(s)*

```
150  CLOSE #6,8
```

The CLOSE statement terminates I/O to a device and writes all active buffers. The number sign is optional.

COMMON

COM [(*name*)] *list*

```
50  COM (TEST) A,B,C
```

The COM, or COMMON statement allows you to establish a named storage area that can be shared by 2 or more subprograms. The common area name must be 1 to 6 characters.

DATA

DATA *constant(s)*

```
50  DATA 4.3, "ABC", 18, 42
```

The DATA statement allows you to provide a pool of information that is accessible to the program by means of a READ statement. A DATA statement must be the only statement on the line and, when you specify more than one item, you must separate them with commas. DATA statements cannot have comments.

DEF

DEF (*single-line*)

DEF FN*a* [(*b1,b2,b3,...b8*)]=*expression*

```
10  DEF FNX (A,B)=A*B
```

The DEF statement establishes a user-defined function. The function name can be any legal variable name and must begin with FN. The variable type

determines the function type. The optional arguments represent dummy parameters and cannot contain array elements. The function definition can refer to any of the dummy parameters or to other program variables but the definition cannot be recursive. Single-line user-defined functions are local to the main program or subprogram in which they are contained.

DEF (*multi-line*)

DEF FN*a* [(*b1,b2,b3,...b8*)]

```
10  DEF FNX (A,B)
```

The multi-line DEF establishes user-defined functions and allows you to include other statements in the body of the function. The function name can be any variable name preceded by FN. Any statement can appear in a function except SUB, SUBEND, RETURN or another DEF. The DATA and DIM statements are not local to the function definition. A GOTO, GOSUB, ONGOTO, or ONGOSUB transfer outside the function is not allowed. The function definition must end with an FNEND statement.

DELETE

DELETE # *<num-exp>*

```
60  DELETE #5%
```

The DELETE operation is used on relative and indexed files only. The operation erases an existing record from the file.

DIMENSION

DIM *subscripted variable(s)*

```
30  DIM B(2,3)
```

The DIM statement reserves storage for arrays. The size of the reserved storage is determined by the subscripts, (constant). A maximum of 2 subscripts is permitted and, when 2 are used, must be separated by a comma.

DIM

DIM # *expression, array* $\left\{\begin{matrix}(\$)\\(\%)\end{matrix}\right\}$[=*integer*]

```
50  DIM #2, A(10,15), B(50)
```

This statement allocates space for the specified arrays on the file associated with the logical number 2. Storage is allocated at the beginning of the file such that the rightmost subscript varies the fastest. The default string storage length is 16 bytes and the space is preallocated.

END

END

```
100  END
```

The END statement terminates program execution and closes all files. END must be the last statement in the program.

FIND

FIND # <*num-exp*> [,RECORD <*num-exp*>]

[,KEY #<*num-exp*> { GT / GE / EQ } *string exp*]

```
50  FIND #7%, RECORD 25
```

The FIND operation causes a RECORD search in the specified file. For sequential files, the FIND starts at the beginning of the file and locates each successor record for each FIND operation. Relative files allow the specification of a record number. Indexed files allow the specification of a key or a sequential search through the key table. The RECORD and KEY specifications are restricted to relative and indexed files, respectively.

FNEND

FNEND

```
40  FNEND
```

The FNEND statement causes an exit from a user-defined function and signals the function's logical and physical end.

FNEXIT

FNEXIT

```
70  FNEXIT
```

The FNEXIT statement is equivalent to a GOTO, where the destination is the FNEND statement for the current multi-line DEF. FNEXIT is legal only inside 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 initiates and controls a loop. A simple numeric variable must be used after the FOR, and the same variable must appear in the required NEXT statement. The first numeric expression is the initial loop value; the second expression is the terminating loop value. The optional STEP expression is the loop increment; +1 is the default. Transfer into an uninitialized loop is illegal.

FOR (conditional)

FOR *variable=<num-exp1>*[STEP *<num-exp2>*] [WHILE *conditional* / UNTIL *exp*]

```
80  FOR I=1 UNTIL I>10
```

The conditional FOR statement duplicates the previous FOR statement except that loop termination is determined by a false expression in the WHILE clause or a true expression in the UNTIL clause.

GET

GET # *<num-exp>* [,KEY # *<num-exp>* {GE / GT / EQ} *string-exp*]
[,RECORD *<num-exp>*]

```
50  GET #5%
```

The GET operation reads a record from a specified file into a buffer. On sequential files, GET operations are performed on succeeding records starting at the beginning of the file. Relative and Block I/O files allow the specification of a record number, and indexed files allow the specification of a key name.

GOSUB

GOSUB *line number*

```
25  GOSUB 120
```

The GOSUB statement transfers control to a subroutine that begins at a specified line number.

GOTO

GOTO *line number*

```
40  GOTO 85
```

The GOTO statement unconditionally transfers control to a specified line number.

IF

IF *<conditional-exp>* THEN *<statements>* ELSE *<statements>*

```
25  IF A=0 THEN PRINT "A EQUALS 0"
```

The various forms of the IF statement allow branches in the program. The IF statement can also cause execution of statements except the following:

DIM, REM, DATA, END, DEF, FNEND and SUB.

INPUT

INPUT ["*string constant*",] *variable(s)*

```
25  INPUT A,B,C%
```

The INPUT statement allows you to type in data to the program from the terminal. The program requests data by printing your optional string constant and a question mark on the terminal and then waiting for you to respond.

INPUT

INPUT # *expression, variable(s)*

```
25  INPUT #6%, A,B,C
```

The INPUT # statement acts very much as the INPUT statement. However, the INPUT # statement requests data from a terminal-format file rather than from you.

INPUT LINE and LINPUT

INPUT LINE ["*string constant*",] *string variable(s)*

LINPUT ["*string constant*",] *string variable(s)*

```
15  INPUT LINE A$, B$
```

The INPUT LINE statement allows a character string to be input to a specified variable. The line terminator is included in the string with INPUT LINE but discarded with LINPUT. The optional string constant is printed before a question mark prompt for data.

INPUT LINE # and LINPUT

INPUT LINE # *expression, string variable(s)*

LINPUT # *expression, string variable(s)*

```
10  INPUT LINE # 4%, A$, B$
```

The INPUT LINE # and LINPUT # statements read strings without editing from a terminal-format file.

KILL

KILL *string expression*

```
10  KILL "SALARY.DAT"
```

The KILL statement deletes a file from storage.

LET

LET *variable(s)=expression*

```
10  LET A=65
```

The LET statement assigns constants and expressions to variables. The keyword LET is optional.

LINPUT

See INPUT LINE

LINPUT #

See INPUT LINE #

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 is not changed.

MAP

MAP (*name*) *element(s)*

```
10  MAP (BUFF1) A%, B$, C
```

The MAP statement associates a named buffer with a file. Specified data in the element list is moved from the file to the buffer on a GET and from the buffer to the file on a PUT.

MAT INPUT

MAT INPUT *array(s)*

```
50  MAT INPUT A
```

The MAT INPUT statement allows element values to be entered in an array. Input is read from the terminal. Elements are stored in row order as they are typed.

MAT PRINT

MAT PRINT *array(s)*

```
120  MAT PRINT A;
```

The MAT PRINT statement outputs each element of a specified array.

MAT READ

MAT READ *array(s)*

```
50  MAT READ B,C
```

The MAT READ statement reads the values into elements of a 1- or 2-dimensional array from a DATA statement.

MOVE

MOVE $\left\{\begin{matrix}\text{FROM}\\\text{TO}\end{matrix}\right\}$ *file 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.

NAME AS

NAME *string1* AS *string2*

```
15  NAME "MONEY" AS "ACCNTS"
```

The NAME AS statement renames a file without changing the contents of the file.

NEXT

NEXT *variable*

```
15  NEXT I
```

The NEXT statement terminates a FOR, WHILE, or UNTIL loop. The variable must correspond with the variable in the initial FOR statement. Nested loops cannot cross each other.

ONERROR {GO TO / GO BACK}

ONERROR GOTO *line number*

ONERROR GO BACK

```
25  ONERROR GOTO 50
30  ON ERROR GO BACK
```

The ONERROR GOTO statement allows the program to transfer control to an error-handling routine. The ONERROR GO BACK statement allows a subprogram containing an error to return to the program that called it for error handling.

ON GOSUB

ON *<num-exp>* GOSUB *line number(s)*

```
50  ON A+B GOSUB 80, 95, 100
```

The ON GOSUB statement is used to conditionally transfer control to one of several subroutines or to one of several entry points into one or more subroutines.

ON GOTO

ON *<num-exp>* GOTO *line number(s)*

```
20  ON J% GOTO 85, 90, 95, 100
```

The ON GOTO statement allows the program to transfer control to one of several different places in the program depending on the value of *num-exp*.

OPEN

OPEN *filename exp* [FOR INPUT | FOR OUTPUT] AS FILE [#] *expression*

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

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

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

[,MAP <*mapname*>]
[,RECORDSIZE <*num-exp*>]
[,BLOCKSIZE <*num-exp*>]
[,FILESIZE <*num-exp*>]
[,SPAN] [,NOSPAN]
[,CONTIGUOUS]
[,TEMPORARY]
[,BUCKETSIZE <*num-exp*>]
[,CONNECT <*num-exp*>]
[,CLUSTERSIZE <*num-exp*>]
[,BUFFERSIZE <*num-exp*>]
,PRIMARY [KEY] <*name*> [DUPLICATES]
[NODUPLICATES NOCHANGES]
[,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.

PRINT

PRINT [*expression(s)*]

```
30  PRINT A+B
```

The PRINT statement causes the data you specify to be output on 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 supress spacing between elememts.

PRINT

PRINT # *expression, list*

```
65  PRINT # G%, A, B+C
```

The PRINT # statement writes data into the specified terminal-format file.

PRINT USING

PRINT [# *expression*] USING *string, list*

```
10  PRINT USING "****.##", A,B,C
```

The PRINT USING statement causes output to be printed in a specified format. The optional expression indicates the channel number of the file in which to print the list.

PUT

PUT # <*num-exp*> [,KEY # *exp* { GT / EQ / GE } *str-exp* / ,RECORD <*num-exp*> / ,COUNT <*num-exp*>]

```
25  PUT #7%, RECORD 15%
```

The PUT statement writes a record from a buffer to a specified file. The RECORD clause is used for relative or Block I/O files. Sequential files allow PUT operations only at the end of the file. The COUNT clause can redefine the size of the record.

RANDOMIZE

RANDOMIZE

```
10  RANDOMIZE
```

The RANDOMIZE statement changes the starting point of the RND function to a new unpredictable location.

READ

READ *variable(s)*

```
75  READ A,B%,C$, D(5)
```

The READ statement directs BASIC to read from a list of values built into a data block by a DATA statement.

REM

REM *comment*

```
30  REM this is a comment
```

The REM statement contains user written comments and has no effect on program execution.

RESTORE [#]

RESTORE [[# <*num-exp*>] [,KEY#<*num-exp*>]]

```
30  RESTORE #3
```

The RESTORE # statement with the KEY clause resets an indexed file to the beginning of the key specified. The RESTORE # statement without the KEY clause resets the specified file to the first record in the file. 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 subroutine. If no line number is specified, control is shifted back to the point of error generation. If a line number is specified, control is shifted to that line.

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 length is unchanged.

SCRATCH

SCRATCH # *file-exp*

```
25  SCRATCH #6
```

The SCRATCH statement allows you to truncate a sequential file. SCRATCH can only be used if the file was OPENed with ACCESS SCRATCH.

STOP

STOP

```
110  STOP
```

The STOP statement causes a halt in program execution. Files are not closed and a message indicating the location of the halt is printed.

SUB

SUB *name* [(*dummy argument(s)*)]

```
40  SUB TEST (A,B%)
```

THE SUB statement marks the beginning of a subprogram and defines the type and number of subprogram parameters.

SUBEND

SUBEND

```
25  SUBEND
```

The SUBEND statement marks the end of the subprogram and returns control to the calling program. It must appear at the end of all subprograms.

SUBEXIT

SUBEXIT

```
899  SUBEXIT
```

The SUBEXIT statement is equivalent to a GOTO, where the destination is the SUBEND statement in the current subprogram. SUBEXIT is legal only in a subprogram.

UNTIL

UNTIL <*conditional-exp*>

```
50  UNTIL I=0
```

The UNTIL statement sets up a loop that must have a corresponding NEXT statement. The loop executes until the expression is true.

UPDATE

UPDATE # <*num-exp*>[,COUNT *exp*]

```
50  UPDATE #1
```

The UPDATE statement changes an existing record in the file. On sequential files the new record size as defined in the MAP or COUNT clause, must be the same as the record it replaces. An UPDATE must be preceded by a successful GET or FIND.

WHILE

WHILE <*conditional-exp*>

```
75  WHILE A%<10%
```

The WHILE statement sets up a loop that must have a NEXT statement. The conditional expression is evaluated before each loop iteration. If the expression is true, BASIC executes the statements in the loop. If the expression is false, BASIC executes the statements following the NEXT statement.

A.4 Functions

Function	Usage
ABS(x)	returns the absolute value of x.
ASCII($x$$)	returns the decimal ASCII value of the first character of a specified string.
ATN(x)	returns the arctangent of x in radians.
BUFSIZ(n%)	In certain applications, it is important for a program to determine the buffer size of an open channel, particularly if the OPEN statement spcifies a logical device name. Your program can execute the integer function BUFSIZ to extract this information. The BUFSIZ function returns an integer value, which is the size of the buffer in bytes. If the channel is closed, BUFSIZ equals 0. The format of the BUFSIZ function is: BUFSIZ(N%) where N% equals the channel number.
CCPOS(n%)	Returns the current position on the output line for the given channel number. The format of the CCPOS function is: CCPOS(N%) where N% is the I/O channel number. N% may range from 0 to 14. CCPOS(0%) returns the position for your terminal.
CHR$($x$%)	returns the character equivalent of the ASCII value x%.
COMP%($x$$,$y$$)	returns the following: Value Returned — Relationship of $x$$ to $y$$ 1 — greater than 0 — equal to -1 — less than
COS(s)	returns the cosine of x.
CTRLC	CTRLC enables CTRL/C trapping.
CVT$$	See EDIT$.
DATE$(0%)	returns the current date in the form *dd-mmm-yy*.
DATE$($x$%)	returns the date in the form *dd-mmm-yy* according to the formula: x% = 1000 * (years since 1970) + (Julian day of the year)

DET	returns the determinant of a matrix.
DIF$(*x$,y$*)	subtracts *y$* from *x$* and returns the difference.
ECHO(*n%*)	ECHO enables terminal echo of characters sent to the system from your terminal.
EDIT$(*string,n%*)	converts the string to an integer.
ERL	returns the line number on which an error occurred.
ERN$	returns the name of the subprogram in which an error occurred.
ERR	returns the error code (see Appendix C).
ERT$(*n%*)	The ERT$ function 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.
FIX(*x*)	returns the value of *x* truncated to an integer.
FORMAT$(*B$,A*) (*A,B$*)	returns the numeric variable formatted according to the contents of the associated string. The formatting rules are the same as for PRINT USING.
FSP$(*N%*)	returns the string that describes the file that is open on a given channel. *N%* is the channel number.
FSS$(*A$,B%*)	performs a filename string scan on *A$* starting at position *B%*.
INSTR(*z%,x$,y$*)	returns the position of substring *y$* in the main string *x$* starting at position *z%*.
INT(*x*)	returns the integral part of *x*. (INT(*x*) returns the same value as FIX(*x*) for equal values of *x*, but INT(*x*) does not change *x*.)
LEFT$(*x$,y%*)	returns a substring of *x$* beginning at the leftmost position for a total length of *y%* characters.
LEN(*x$*)	returns the number of characters in *x$*.
LOG(*x*)	returns the natural logarithm of *x*. In the following formula where: $e^{\wedge}x = y$ then $\ln y = \log(e)\ y = x$
LOG10(*x*)	returns the common logarithm of *x*. Common logarithms differ from natural logarithms in that the base of common logarithms is 10, as opposed to 2.71828 for natural logarithms.

MID$(*string,n1%,n2%*)	returns a substring *n2%* characters long starting at position *n1%* of string.
NOECHO(*N%*)	disables terminal echo.
NUM	is the number of columns you enter in a matrix.
NUM2	is the number of elements in the last column of a matrix.
NUM$(*n%*)	returns *n%* as PRINT would write it.
NUM1$(*n%*)	returns *n%* as PRINT would write it, but without spaces or E format.
PI	returns a constant value: 3.14159.
PLACE$(*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.
PROD$(*x$,y$,n%*)	returns the product of *x$* and *y$* with precision according to *n%*.
QUO$(*x$,y$,n%*)	divides *x$* by *y$* and returns the quotient with precision according to *n%*.
RAD$(*x%*)	converts the integer *x%* to its RADIX-50 equivalent.
RCTRLC	disables CTRL/C trapping.
RCTRLO(*N%*)	cancels the effect of typing CTRL/O on channel *N%*. See your system of user's guide for a description of the effect of CTRL/O on your system.
RECOUNT	A GET or INPUT operation can transfer a variable number of bytes of data. This occurs when you re doing input from a device such as a terminal or mag-tape or from a file with variable records. The system variable RECOUNT allows you to determine how much data was actually read. RECOUNT contains the number of characters read after each input operation. RECOUNT is set by every input operation on any channel, including channel 0 (your terminal). For this reason, if you need to know the value of RECOUNT for testing, you should copy it immediately after you execute a GET statement. Note that if an error occurs during the GET operation, RECOUNT is not properly set.
RIGHT$(*x$,y%*)	returns a substring of *x$* that extends from the *yth* character to the end of the string.
RND	returns a real random number between 0 and 1.

SEG$(*x$,y%,z%*)	returns the substring of *x$* that extend from the *yth* character to the *zth* character (compare with RIGHT).
SGN(*x*)	returns: 1 if *x* is: positive 0 zero –1 negative
SIN(*x*)	returns the sine of *x* in radians.
SPACE$(*x*)	produces and returns a string of *x* spaces.
SQR(*x*)	returns the square root of *x*; also SQRT(*x*).
STATUS	The variable STATUS contains information about the last channel on which your program executed an OPEN statement. STATUS is a 16–bit word. Your proram can test each bit to determine the status of the channel. See your User's Guide to determine the interpretation of each bit.
STR$(*x*)	returns the value of an expression without the leading and trailing blanks (see also NUM$(*x*)).
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$* and *y$*.
TAB(*x*)	moves the print head to the *xth* position.
TAN(*x*)	returns the tangent of *x* in radians.
TIME$(*x%*)	returns the time *x* minutes before midnight.
TIME$(0%)	returns the present time.
TIME(0)	returns the clock time in seconds since midnight.
VAL(*x$*)	computes the numeric value of the numeric string *x$*; *x$* must be acceptable numeric input.
XLATE(*A$,B$*)	translates a string to another using a translation table, *B$*.

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	A is logically equivalent to 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.

A.5 Reserved Keywords

BASIC-PLUS-2 statements, function names, and record attribute specifications are reserved. That is, the language keywords cannot be used for variable names. Table A-4 lists all of the BASIC-PLUS-2 language elements that are reserved. If you attempt to use one of the listed words as the name of a variable, external subroutine, MAP, or COMMON area, an error is returned. You can, however, use a variation on the reserved keyword. For example, IF$, AND%, and DIM$ are allowed. Note that the use of a period in the second or third character position of the variable name permits a faster program compilation because the compiler is not required to perform a keyword table search.

Table A-4: Reserved Keyword List

ABORT*	EQ	MID	SIN
ABS	EQV	MID$	SLEEP
ABS%	ERL	MOD*	SO
ACCESS	ERN$	MOD%*	SP
ACCESS%*	ERR	MODE	SPACE$
ALL*	ERROR	MODIFY	SPAN
ALIGNED*	ERT$	MOVE	SQR

(Continued on next page)

Table A-4: Reserved Keyword List (Cont.)

ALLOW	ESC	MSGMAP***	SQRT*
ALTERNATE	EXP	NAME	STATUS
AND	EXTEND*	NEXT	STEP
APPEND	FF	NOCHANGES	STOP
AS	FIELD	NODATA*	STR$
ASCII	FILE	NODUPLICATES	STREAM
ATN	FILESIZE	NOECHO	STRING$
ATN2*	FILL	NONE	SUB
BACK	FILL$	NOPAGE*	SUBEND
BEL	FILL%	NOQUOTE*	SUBEXIT
BIN$*	FIND	NOREWIND	SUM$
BINARY*	FIX	NOSPAN	SWAP%
BIT*	FIXED	NOT	SYS**
BLOCK	FNEND	NOTAPE*	TAB
BLOCKSIZE	FNEXIT	NUL$	TAN
BROADCAST*	FOR	NUM	TAPE
BS	FORCEIN*	NUM$	TASK
BUCKETSIZE	FORMAT$	NUM1$	TEMPORARY
BUFFER	FREE***	NUM2	TERMINAL*
BUFFERSIZE*	FROM	OCT$*	THEN
BUFSIZ	FSP$	ON	TIM*
BY	FSS$	ONECHR	TIME
CALL	GE	ONENDFILE*	TIME$
CCPOS	GET	ONERROR	TO
CHAIN	GO	OPEN	TRM$
CHANGE	GOSUB	OR	TRN
CHANGES	GOTO	ORGANIZATION	TST***
CHR$	GT	OUTPUT	TSTEND***
CLK$*	HANGUP*	PAGE*	TYP*
CLOSE	HT	PEEK**	TYPE*
CLUSTERSIZE**	IDN	PI	TYPE$*
COM	IF	PLACE$	UNALIGNED*
COMMON	IFEND*	POS	UNDEFINED
COMP%	IFMORE*	POS%*	UNLESS
CON	IMAGE*	PPS%*	UNLOCK
CONNECT	IMP	PRIMARY	UNTIL
CONTIGUOUS	INDEXED	PRINT	UPDATE

(Continued on next page)

Table A-4: Reserved Keyword List (Cont.)

COS	INIMAGE*	PROD$	USEAGE*
COT*	INPUT	PUT	USEAGE$*
COUNT	INSTR	QUO$	USING
CR	INT	QUOTE*	USR*
CTRLC	INV	RAD$	USR$*
CVT$$	INVALID*	RANDOM	VAL
CVT$%	KEY	RANDOMIZE	VAL%
CVT$F	KILL	RCTRLC	VARIABLE
CVT%$	LEFT	RCTRLO	VIRTUAL
CVTF$	LEFT$	READ	VPS%*
DAT*	LEN	RECORD	VT
DAT$*	LET	RECORDSIZE	WAIT
DATA	LF	RECOUNT	WHILE
DATE$	LINE	REF	WINDOWSIZE****
DEF	LINO*	RELATIVE	WITH*
DEL*	LINPUT	REM	WRITE
DELETE	LOC*	RESET	WRKMAP***
DELIMIT*	LOCK***	RESTORE	XLATE
DENSITY*	LOF*	RESUME	XOR
DET	LOG	RETURN	ZER
DIF$	LOG10	RIGHT	.ABORT
DIM	LSA*	RIGHT$	.DEFINE
DIMENSION	LSET	RND	.ENDC*
DOUBLEBUF*	MAGTAPE	RSET	.IF*
DUPLICATES	MAP	SCRATCH	.IFDF*
ECHO	MAR*	SEG$	.IFF*
EDIT$	MAR%*	SEQUENTIAL	.IFNDF*
ELSE	MARGIN*	SGN	
END	MAT	SI	

* - Reserved word, included for compatibility with DECSystem 20
** - Supported on RSTS/E only
*** - Supported on TRAX-11 only
**** - Supported on non RSTS/E only

Appendix B
Run–Time Error Codes and Messages

Table B–1 contains values of ERR that appear most commonly on your system at execution time. RMS–specific errors have values of ERR ranging from 128 to the end of the table. Refer to the User's Guide for the meanings of error messages that do not appear in this table.

The question mark (?) or percent sign (%) that precedes each message printed indicates whether the error is fatal or a warning, respectively.

Table B–1: ERR Values, Error Messages, and Their Meanings

ERR	Message Printed	Meaning
1	?BAD DIRECTORY FOR DEVICE	The directory the device referenced is in an unreadable format.
2	?ILLEGAL FILE NAME	The filename specified is not acceptable. It contains embedded blanks or unacceptable characters.
3	?ACCOUNT OR DEVICE IN USE	The specified operation cannot be performed because the file has already been opened by someone else. This message has a general "file in use" meaning.
4	?NO ROOM FOR USER ON DEVICE	Storage space allowed for the current user on the device specified has been used or the device as a whole is too full to accept further data.
5	?CAN'T FIND FILE OR ACCOUNT	The file specified or current user account numbers were not found on the device specified. This message has a general "not here" meaning.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
6	?NOT A VALID DEVICE	Attempt to use an illegal or non-existent device.
7	?I/O CHANNEL ALREADY OPEN	An attempt was made to open one of the I/O channels which had already been opened by the program.
8	?DEVICE NOT AVAILABLE	The device requested is currently reserved by another user.
9	?I/O CHANNEL NOT OPEN	Attempt to perform I/O on one of the channels that has not been previously opened in the program.
10	?PROTECTION VIOLATION	The current user is not allowed to perform the requested operation on the specified file. Input may have been requested from an output-only device or vice versa. This message has a general "can't do that" meaning.
11	?END OF FILE ON DEVICE	Attempt to perform input beyond the end of a data file.
12	?FATAL SYSTEM I/O FAILURE	An I/O error has occurred on the system level. The user has no guarantee that the last operation has been performed.
13	?USER DATA ERROR ON DEVICE	One or more characters may have been transmitted incorrectly due to a parity error, bad punch combination on a card, or similar error.
14	?DEVICE HUNG OR WRITE LOCKED	Check hardware condition of device requested. Possible causes of this error include a line printer out of paper or high-speed reader being off-line.
15	?KEYBOARD WAIT EXHAUSTED	Time requested by WAIT statement has been exhausted with no input received from the specified keyboard.
16	?NAME OR ACCOUNT NOW EXISTS	An attempt was made to rename a file with the name of a file which already exists.
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.
28	?PROGRAMMABLE ^C TRAP	ON ERROR-GOTO subroutine was entered through a program trapped by means of Control/C.
29	?CORRUPTED FILE STRUCTURE	Fatal error in CLEAN operation.
30	?DEVICE NOT FILE-STRUCTURED	An attempt is made to access a device, other than a disk as a file-structured device. This error occurs, for example, when the user attempts to gain a directory listing of a non-directory device.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
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 attempted.
33	?UNIBUS TIMEOUT FATAL TRAP	This hardware error occurs when an attempt is made to address non-existent memory or an odd address using the PEEK function. An occurrence of this error in any other case is cause for an SPR.
35	?MEMORY MANAGEMENT VIOLATION	This hardware error occurs when an illegal Monitor address is specified using the PEEK function. An occurrence of this error in situations other than using PEEK is cause for an SPR.
42	?VIRTUAL BUFFER TOO LARGE	Virtual memory buffers must be at least 512 bytes long.
43	?VIRTUAL ARRAY NOT ON DISK	A non-disk device is open on the channel upon which the virtual array is referenced.
44	?MATRIX OR ARRAY TOO BIG	In-core array size is too large.
45	?VIRTUAL ARRAY NOT YET OPEN	An attempt was made to use a virtual array before opening the corresponding disk file.
46	?ILLEGAL I/O CHANNEL	Attempt was made to open a file on an I/O channel outside the range of legal channel.
47	?LINE TOO LONG	Attempt to input a line longer than the buffer.
48	%FLOATING POINT ERROR	Floating point overflow or underflow. If no transfer is made to an error handling routine, a 0 is returned as the floating-point value for underflow and the maximum positive number for overflow.
49	%ARGUMENT TOO LARGE IN EXP	Value is outside of legal range.
50	%DATA FORMAT ERROR	The input data is floating point, but the INPUT or READ statement specifies integer input.
51	%INTEGER ERROR	Attempt to use a number as an integer when that number is outside the allowable integer range. If no transfer is made to an error handling routine, a 0 is returned as the integer value.
52	%ILLEGAL NUMBER	Improperly formed input. For example, "1..2" is an improperly formed number.
53	%ILLEGAL ARGUMENT IN LOG	Negative or zero argument to log function. Value returned is the argument as passed to the function.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
54	%IMAGINARY SQUARE ROOTS	Attempt to take square root of a number less than 0. If no transfer is made to an error handling routine, the value returned is the square root of the absolute value of the argument.
55	?SUBSCRIPT OUT OF RANGE	Attempt to reference an array element larger than the maximum specified.
56	?CAN'T INVERT MATRIX	Attempt to 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	The integer index in a FOR loop attempted to go beyond implementation defined limits.
61	%DIVISION BY 0	Attempt by the user program to divide some quantity by 0. If no transfer is made to an error handling routine, the largest positive number is returned as the result.
62	?NO RUN-TIME SYSTEM	The run-time system referenced has not been added to the system list of run-time systems.
63	?FIELD OVERFLOWS BUFFER	Attempt to use FIELD to allocate more space than exists in the specified buffer.
64	?NOT A RANDOM ACCESS DEVICE	Random I/O was attempted on a non-random access device. Use another device, if possible.
65	?ILLEGAL MAGTAPE USAGE	Improper use of MAGTAPE function. See *RSX-11M Programming Manual* for details.
67	?ILLEGAL SWITCH USAGE	The switch operation or specification is illegal.
71	?STATEMENT NOT FOUND	An attempt was made to CHAIN into a program at a nonexistent line number.
72	?RETURN WITHOUT GOSUB	RETURN statement encountered in user program without a previous GOSUB statement having been executed.
73	?FNEND WITHOUT FUNCTION CALL	An FNEND statement was encountered in the user program without a previous DEF statement having been executed.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
88	?ARGUMENTS DON'T MATCH	Arguments in a function call do not match, in number or in type, the arguments defined for the function.
89	?TOO MANY ARGUMENTS	A user-defined function may have up to five arguments.
97	?TOO FEW ARGUMENTS	The function has been called with a number of arguments not equal to the number defined for the function.
103	?PROGRAM LOST - SORRY	A fatal system error has occurred which caused the user program to be lost. This error can indicate hardware problems or use of an improperly compiled program. Consult your system manager or the discussion of such errors in the *RSX-11M System Manager's Guide.*
104	?RESUME AND NO ERROR	A RESUME statement was encountered where no error had occurred to cause a transfer into an error handling routine via the ON ERROR GO TO statement.
105	?REDIMENSIONED ARRAY	Usage of an array or matrix within the user program has caused BASIC-PLUS-2 to redimension the array implicitly.
116	?PRINT-USING FORMAT ERROR	An error was made during the construction of the string used to supply the output format in a PRINT-USING statement.
125	?WRONG MATH PACKAGE	Subprogram was compiled with either the 2-word or 4-word math package and an attempt is made to run the subprogram with a main program having the opposite math package. Recompile the subprogram using the math package of the main program with which it will be run.
126	?MAXIMUM MEMORY EXCEEDED	Program attempts to expand itself beyond limits set by swap maximum or run-time system maximum. This can be caused by too many open I/O channels or too much string activity.
127	%SCALE FACTOR INTERLOCK	Subprogram was compiled with a different scale factor from main program with which it is running. Recompile the subprogram with the scale factor used by the main program.
128	?TAPE RECORDS NOT ANSI	A GET was attempted on variable length records from a file on magtape. The records must be in ANSI D format.
130	%KEY NOT CHANGEABLE	An UPDATE was attempted on an indexed file. The replacement record may not contain key fields that duplicate another record's key fields. Specify CHANGES in the OPEN for this key.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
131	%NO CURRENT RECORD	A previous GET or FIND is missing or was unsuccessful. This PUT or UPDATE therefore fails.
132	?RECORD HAS BEEN DELETED	A record previously located by its Records File Address (RFA) has been deleted.
133	?ILLEGAL USAGE FOR DEVICE	The requested operation cannot be performed because: 1. The device specification contains illegal syntax. 2. The specified device does not exist on this system. 3. The specified device is inappropriate for the requested operation (e.g., magtape for an indexed file).
134	%DUPLICATE KEY DETECTED	A PUT operation was attempted with one or more duplicate key fields in an indexed file where duplicate key values were not permitted at file creation time.
135	?ILLEGAL USAGE	An OPEN was attempted on a file of undeclared organization or the specfied record operation was not stated in the ACCESS clause.
136	?ILLEGAL OR ILLOGICAL ACCESS	The requested access is impossible because: 1. The attempted record operation and the ACCESS clause in the OPEN statement are incompatible. 2. The ACCESS clause is incorrect for the organization of this file. 3. READ or APPEND was specified at file creation time. Change the ACCESS clause.
137	?ILLEGAL KEY ATTRIBUTE	An illegal combination of key characteristics has occurred. Check the OPEN statement of this file for either of the following: NODUPLICATES and CHANGES CHANGES without DUPLICATES
138	%FILE IS LOCKED	This file has been locked by another user or by the system in a program that does not allow shared access.
140	?INDEX NOT INITIALIZED	A GET or FIND was attempted on an empty file.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
141	?ILLEGAL OPERATION	The requested operation is illegal because: 1. DELETE is impossible on a sequential file. 2. UPDATE is impossible on a magtape file. 3. Block I/O is impossible on an RMS file. (Block I/O requires VIRTUAL organization.) 4. RMS I/O is impossible on a block I/O structured 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: 1. Random access operations cannot be performed with a zero or negative record number specification. 2. A GET or FIND on an indexed file cannot contain a null key value.
144	%INVALID KEY OF REFERENCE	A GET, FIND, or RESTORE was attempted 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.
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	A sequential file must be at end of file before a PUT is attempted. (This error may also occur when an existing file is opened for WRITE access.)
150	?NO PRIMARY KEY SPECIFIED	An indexed file cannot be created 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.
153	%RECORD ALREADY EXISTS	An attempted random access PUT on a relative file has encountered a pre-existing record.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
154	%RECORD/BUCKET LOCKED	The target bucket has been locked by another program.
155	%RECORD NOT FOUND	A random access GET or FIND was attempted on a nonexistent record (either never written or previously deleted).
156	%SIZE OF RECORD INVALID	The COUNT specification is invalid because: 1. COUNT equals zero. 2. COUNT exceeds the maximum size of the record. 3. COUNT conflicts with the actual size of the current record during a sequential file UPDATE on disk. 4. 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	The key value of this record is less than the key value of the previous record in a sequential access PUT 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 record format number, position, and length of indexed file keys
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 an attempt is made to MOVE data to the output buffer.)
164	?TERMINAL FORMAT FILE REQUIRED	The 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 that was opened for APPEND access. The file may be corrupted.
166	%NEGATIVE FILL OR STRING LENGTH	The FILL elements in a MOVE statement I/O list is less than zero. Reformat the I/O list.

(continued on next page)

Table B-1: ERR Values, Error Messages, and Their Meanings (Cont.)

ERR	Message Printed	Meaning
167	?ILLEGAL RECORD FORMAT	The record given is illegal for the organization or operating system on which this file resides. The existence of embedded carriage control characters in variable length records can cause this error.
168	?ILLEGAL ALLOW CLAUSE	The value specified for the ALLOW clause is illegal for the type of organization or for the operating system on which the file resides.
170	%INDEX OPTIMIZATION ERROR	During a PUT or UPDATE operation on an indexed file, the record was successfully written. The record can later be retrieved, but RMS-11 was unable to optimize the structure of the index at the time the record was inserted. Several indirections will therefore occur on retrieval, slowing the process.
171	?RRV ERROR	During a PUT or UPDATE operation on an indexed file, the record was successfully written. RMS-11 was unable to update one or more Record Retrieval Vectors (RRVs) and the records associated with those RRVs cannot be retrieved later using alternate indices on RFA addressing mode. The best solution is to delete the records involved and reinsert them.
173	?INVALID RFA FIELD	The value specified as the RFA to an RMS routine is incorrect or beyond the bounds of the field.
238	?ARRAYS MUST BE SAME DIMENSION	A matrix addition or subtraction operation was attempted on arrays of different dimensions.
239	?ARRAYS MUST BE SQUARE	Matrix multiplication was attempted on non-square array(s).
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 without that switch.
244	%EXPONENTIATION ERROR	The attempted exponentiation is illegal. The result of this operation is set to zero and the program continues. Be sure that the size of the exponent is not too large or too small for your system.
250	?NOT IMPLEMENTED	Some enhancement to the kernel BASIC-PLUS-2 language is not implemented on this operating system.
251	?RECURSIVE SUBROUTINE CALL	A subroutine is attempting to call itself, either directly or indirectly. Examine the flow of control from the first call to the point at which this error occurs.

Appendix C
Compile Time Error Messages

BASIC-PLUS-2 prints a diagnostic message when it detects an error. These messages contain information on the type of error and, where possible, the program line that generated the error. The message indicates error location by including the phrase:

```
AT LINE xxx
```

following the error type. The value of *xxx* is the program line number where the error is located. Note that error location will not appear in the message if the program is compiled with the /NOLINE switch.

The BASIC-PLUS-2 compile-time error messages (see Section C.2) contain additional location information. That is, the error type is followed by a phrase that indicates the erroneous statement as well as the line number. These messages have the form:

```
message AT LINE xxx IN STATEMENT y
```

where *y* is a number that identifies a particular statement on line *xxx*. Note that the statement number appears during the initial error detection.

The printed error messages are preceded, in most cases, by either a question mark (?) or a percent sign (%). A question mark indicates a fatal error; compilation continues, but no output is produced. A percent sign indicates a warning message; execution can continue, but the result is unpredictable. If neither symbol is present, the message is for information only.

Section C.2 contains the BASIC-PLUS-2 error messages printed during compilation. Included with each error message is an explanation and a general recovery procedure. The explanation indicates the general reason for the error's occurrence and also shows the error's severity (i.e., fatal, warning, or information).

C.1 Traceback

BASIC-PLUS-2 provides a traceback mechanism that traces the path of program execution when an error occurs in a function or subroutine. Traceback takes effect only if error trapping is not enabled. When a fatal error occurs in a function or subroutine, the error message is printed. The message is followed by text that describes the execution path of the program beginning at the point of the error back to the initial call in the main routine. Note that Traceback does not describe a path across chained routines. The Traceback text describes the routine name that was called and the line number and routine name that initiated the call. If a routine is compiled with the /NOLINE switch, line number 0 is used in the text.

Consider the following example that lists three routines. When the routines are run and an error detected, the Traceback text is printed:

MAIN.B2S

```
100     PRINT "LINE 100"
200     GOSUB 1000
300     PRINT "LINE 300"
400     GO TO 32767
1000    A%=FNS%(3%)
1100    RETURN
1200    DEF FNS%(E%)
1210    CALL SUBR(E%)
1220    FNS%=E%+1%
1230    FNEND
32767   END
```

SUBR.B2S

```
1000    SUB SUBR(D%)
1100    PRINT "LINE 1100 IN SUBR"
1200    GOSUB 2000
1300    PRINT "LINE 1300 IN SUBR"
1400    GO TO 32767
2000    A=FNZ%
2200    RETURN
3000    DEF FNZ%
3100    CALL SUBR2(33%)
3200    FNEND
32767   SUBEND
```

SUBR2.B2S

```
10      SUB SUBR2(I%)
110     PRINT "LINE 110 IN SUBR2"
123     GOSUB 666
200     GO TO 32767
666     E=FNE
667     RETURN
668     GO TO 32767
2000    DEF FNE
2050    LET A%=0
```

```
2100    FNE=1/A% !NOTE THAT THIS LINE ALWAYS PRODUCES AN ERROR
2200    FNEND
32767   SUBEND
```

When these routines are executed, the output appears as follows:

```
LINE 100
LINE 1100 IN SUBR
LINE 110 IN SUBR2
%Division by 0 at line 2100 in "SUBR2 "
FUNCTION called at line 666 in "SUBR2 "
GOSUB called at line 123 in "SUBR2 "
"SUBR2 " called at line 0 in "SUBR "
FUNCTION called at line 0 in "SUBR "
GOSUB called at line 0 in "SUBR "
"SUBR " called at line 1210 in "MAIN "
FUNCTION called at line 1000 in "MAIN "
GOSUB called at line 200 in "MAIN "

Basic2
```

Note that the routine SUBR is compiled with the /NOLINE switch enabled.

C.2 Compile–Time Error Messages

The following alphabetized list describes the error messages that BASIC–PLUS–2 returns during compilation. The description includes the general cause of the error and the steps that you can take to recover from it. The severity of the error is also noted.

```
? 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 that you supplied in a user-defined function call does not match 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. The argument inconsistency can be in terms of type (i.e., string and numeric) or number of arguments. Examine the program and ensure that function arguments agree with those defined in the DEF statement.

```
% CALL/SUB forces OBJ output
```

WARNING – An attempt is made to compile a program that contains a CALL or SUB statement into a task image file. Programs that contain these statements must be compiled as object modules and linked by the Task Builder. The compiler automatically generates an object module when it encounters a CALL or SUB statement in the program.

```
% /DEB forces OBJ output
```

WARNING – An attempt is made to compile a program to task image format which contains the /DEBUG switch. The /DEBUG option requires the production of an object module on systems that do not support floating-point processors.

```
% ERL overrides /NOLINE
```

WARNING – An error routine that requires an ERL variable is contained in a program that is compiled with the /NOLINE switch. The /NOLINE switch is nullified.

```
?? Error n at line m in x, compiling line p
```

FATAL – In this message, *n* represents the value of the ERR variable, *m* is the line number where the error originated, *x* is the name of the module that contains the error, and *p* is the currently compiling program line. This error causes the loss of your program, an exit from BASIC, and a return to operating system command level (this degree of severity is indicated by the double question mark). It is a compiler error and should not occur. In the event that it does, use a Software Performance Report to report the error to DIGITAL and include a copy of the source program.

```
? Expression too complex at line n
```

FATAL – The compiler encounters an expression that is too complex to compile. In this message, *n* is the line number that contains the expression. Rewrite the expression as two or more assignment statements and retry the compilation. This error should never occur.

```
? FNEND without DEF
```

FATAL – The compiler encounters an FNEND statement without first encountering a DEF statement. Ensure that the desired function is defined before the FNEND statement in the program.

```
? Illegal character
```

FATAL – An attempt is made to compile a program line that contains illegal or incorrect characters. Examine the program line for correct usage of the BASIC–PLUS–2 character set.

```
? Illegal COM/MAP/SUB name
```

FATAL – A MAP, COMMON, or subprogram name exceeds six characters or contains illegal non-alphanumeric characters. Correct the program line.

? Illegal DELETE command

WARNING - An attempt is made to use the DELETE command with no line number argument. The DELETE command requires a specified line number. No lines are deleted from the program.

? Illegal dummy argument

FATAL - There are two occurrences that can cause this error.

1. The same variable appears more than once in a SUB statement argument list.
2. A DEF statement argument is also used as a parameter in a SUB statement.

? Illegal FILL specification

FATAL - An attempt is made to include a length statement in an integer or floating-point FILL or FILL% specification. That is, a MAP or MOVE statement that contains a FILL or FILL% specification allocates a specific amount of space. If you attempt to specify a length in the program (e.g., FILL%=10%), an error results. To allocate additional space, you must specify a FILL specification argument in the MAP or MOVE statement; for example, FILL%(5%). Note that the FILL$ specification does allow you to define a length in number of characters.

? Illegal FN redefinition

FATAL - An attempt is made to redefine a function. A function can be defined only once in a program. Use a different function name for each function definition.

? Illegal loop nesting

FATAL - The program contains nested loops that overlap each other. Examine the program logic and ensure that all nested loops are properly initialized and terminated.

? Illegal MAP statement

FATAL - The compiler encounters a MAP statement that does not contain a legal map name. Ensure that a 1- to 6-character name enclosed in parentheses is used to label the map.

? Illegal mode mixing

FATAL - An attempt is made to mix string and numeric operations. Ensure that the program does not contain incompatible data operations.

```
% Illegal number
```

WARNING - This error is caused by integer overflow or underflow or by floating-point overflow. Ensure that the specified numbers are within the legal range of +32767 to -32767 for integers and 1E38 to 1E-38 for floating-point.

```
? Illegal relative operator
```

FATAL - This message indicates a compiler error and should not occur. In the event that it does, use a Software Performance Report to report the error to DIGITAL and include all pertinent output.

```
? Illegal string operator
```

FATAL - An incorrect string operator is detected in the program. For example, A$=B$-C$ can cause this error. Examine the program for correct string operations.

```
? Illegal subscript
```

FATAL - A DIM statement or array reference contains a subscript in illegal format (e.g., DIM A(A$)). Use a subscript of the correct data type.

```
% Inconsistent function usage in (x) at line n
```

WARNING - A user-defined function that contains an integer dummy argument is supplied with a floating-point argument in the function call. In this message, *x* is the user-defined function name and *n* is the line number of the call. The floating-point argument is truncated to an integer value and the compilation continues.

```
? Inconsistent subscript usage
```

FATAL - This error occurs when the same subscripted variable name appears with both 1 and 2 subscripts. Change the name of one of the arrays. Ensure that subscripted variables retain the same dimensions throughout the program.

```
? Logical operation on non-integer quantity
```

FATAL - The program contains an incorrect data type in a logical operation (e.g., A%=B AND C%,. Use integer data types in logical operations.

```
% Loop will not execute at line n statement m
```

WARNING - The program contains a FOR/NEXT loop whose parameters are constants. The compiler evaluates the parameters, and if the resulting

loop is unexecutable, prints the above message. The program compiles correctly, but with a branch around the loop at the line and statement contained in the message.

```
% MAT INV forces OBJ output
```

WARNING - An attempt is made to compile a program that contains a MATRIX INVERSION statement into a task image file. Programs that contain this statement must be compiled as object modules and linked by the Task Builder. The compiler automatically generates an object module when it encounters a MATRIX INVERSION statement.

```
? Missing FNEND
```

FATAL - The compiler encounters a multi-line DEF statement without a corresponding FNEND. Ensure that multi-line function definitions are terminated with an FNEND statement.

```
? Missing SUBEND
```

FATAL - The compiler encounters a subprogram that does not contain a corresponding SUBEND statement. Ensure that the subprogram is properly terminated.

```
? Multiply allocated variable
```

FATAL - A program variable is assigned conflicting values or is inconsistently used in a statement. For example, COM A,B,A, where the variable A is assigned to COMMON twice, can cause this error.

```
? Multiply defined SUB or recursive CALL
```

FATAL - An attempt is made to compile a subprogram that contains an illegal call to itself. Ensure that subprograms do not call themselves.

```
? NEXT without FOR
```

FATAL - The compiler encounters a NEXT statement without first encountering a corresponding FOR statement. A loop must be initialized with a FOR statement.

```
? NEXT without WHILE/UNTIL
```

FATAL - The program encounters an uninitialized conditional loop. Examine the program and ensure that each conditional loop NEXT statement corresponds to a prior WHILE or UNTIL statement.

```
? Program overflows
```

FATAL – An attempt is made to compile a program that exceeds the allowable memory space. Recompile the program as separate object modules.

```
% RESUME overrides /NOLINE
```

WARNING – A program, compiled with the /NOLINE switch, encounters a RESUME statement without a line number argument. The /NOLINE switch is nullified. The program compiles with /LINE in effect. The RESUME will work properly.

```
?? Stack error in x, compiling line n
```

FATAL – In this message, *x* is the name of the compiler module in which the error occurred and *n* is the currently compiling line number. This error causes an exit from BASIC, a return to operating system command level, and no code is output (this degree of severity is indicated by the double question marks). It is a compiler error and should not occur. In the event that it does, use a Software Performance Report to report the error to DIGITAL and include your source file.

```
? String array in CALL BY REF
```

FATAL – An attempt is made to use a string array argument in a CALL BY REF statement. CALL BY REF does not accept a string array argument. Use the CALL statement.

```
? SUBEND without SUB
```

FATAL – The compiler encounters a SUBEND statement without first encountering a SUB statement. Examine the program and ensure that a subprogram starts with a SUB statement and ends with a SUBEND statement.

```
? Syntax error
```

FATAL – A program line contains illegal syntax or illegal format. Correct the program line to conform with BASIC–PLUS–2 syntax.

```
? Thread x not in run-time system at line n
```

FATAL – The compiler encounters a reference to an object-time system module (thread) that is not present in the current operating system. In this message, *x* is the thread name and *n* is the program line that originated the call. This error should not occur with the system supplied by DIGITAL. In the event that it does, use a Software Performance Report to report the error to DIGITAL and include your source file.

```
? Too few arguments
```

FATAL – An attempt is made to call a function with fewer arguments than are defined for that function. Ensure that the number of arguments given in the function call agree with the function requirements.

```
? Too many arguments
```

FATAL – This error occurs when a function call contains too many arguments. Ensure that the function arguments agree with the function limits.

```
% Unaligned COM or MAP variable x in (y)
```

WARNING – The compiler encounters a numeric variable definition in a COMMON or MAP statement where the variable falls on an odd address. In this message, *x* is the variable name and *y* is the MAP or COMMON name. A string, composed of an odd number of characters, that precedes the numeric variable can cause the variable to fall on an odd address. The compiler aligns the variable to the next highest word boundary and continues with the compilation.

```
? Undefined function (x) called at line n
```

FATAL – The compiler encounters a user-defined function that is not defined with a corresponding DEF statement. In this message, *x* is the user-defined function name and *n* is the line number of the call. Examine the program and ensure that all user-defined functions are defined with an associated DEF statement.

```
% Undefined line number n
```

WARNING – The compiler encounters a control statement that directs the program to a nonexistent line (represented by *n*). The program statement is compiled. The next highest line number to the one specified is assumed to be the control destination.

```
% Undefined MAP (x) in OPEN at line n
```

WARNING – The compiler encounters a MAP clause in the OPEN statement that references a nonexistent map name. In this message, *x* is the name of the undefined map in the MAP clause and *n* is the OPEN statement line number. Each map reference in an OPEN statement must be associated with a defined MAP statement. The compiler ignores the MAP clause in the OPEN statement and continues the compilation.

```
? Unmapped variable x in key clause at line n
```

FATAL – The compiler encounters an indexed file key definition clause containing a reference to a variable that is not defined in a MAP statement. That is, a key must be defined in terms of its position and length in the record before it can be referenced in an OPEN statement KEY clause. The mechanism used to define a record key is the MAP statement. In this message, *x* is the name of the unmapped variable and *n* is the program line that contains the OPEN statement.

```
? Unterminated string
```

FATAL – A string that is not enclosed by single or double quotation marks or is inconsistently terminated causes this error. That is, "ABC and "ABC´ are both illegal; a properly terminated string would be as follows, "ABC" or ´ABC´.

```
? Variable or function name too long
```

FATAL – A variable name exceeds 30 characters (excluding a percent or dollar sign). A function name exceeds 30 characters (excluding FN and a percent or dollar sign). Either of these two occurrences can cause this error.

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

(continued on next page)

Table D-1: ASCII Codes (Cont.)

DECIMAL CODE	7-BIT OCTAL CODE	CHARACTER	REMARKS
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
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

(continued on next page)

Table D-1: ASCII Codes (Cont.)

DECIMAL CODE	7-BIT OCTAL CODE	CHARACTER	REMARKS
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

(continued on next page)

Table D-1: ASCII Codes (Cont.)

DECIMAL CODE	7-BIT OCTAL CODE	CHARACTER	REMARKS
67	103	C	Upper-case C
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

(continued on next page)

Table D-1: ASCII Codes (Cont.)

DECIMAL CODE	7-BIT OCTAL CODE	CHARACTER	REMARKS
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
116	164	t	Lower-case t

(continued on next page)

Table D-1: ASCII Codes (Cont.)

DECIMAL CODE	7-BIT OCTAL CODE	CHARACTER	REMARKS
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

NOTE:

1. Teleprinters manufactured by Teletype Corporation, Skokie, Illinois, have used codes 175 (ALT) and 176 for ESC. Programs may forgo the use of 175 and 176 in order to use these codes as ESC on older teleprinters.

2. ASCII is a 7-bit character code with an optional odd parity bit (200) added for many devices. Programs normally use just seven bits internally; the 200 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) is identical to ASCII except that number sign (043) is represented as # instead of £ and certain characters are reserved for national use.

D.2 Radix-50 Character Set

Many operating system specifications, such as file and program segment names, 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). The RAD$() function converts a Radix-50 word to its 3-character representation.

The complete set of characters capable of being represented in Radix–50 format, their ASCII octal equivalents, and the Radix–50 value by which each character is represented are as follows:

Character	ASCII Octal Equivalent	Radix–50 Octal Equivalent
space	40	0
A–Z	101–132	1–32
$	44	33
.	56	34
unused		35
0–9	60–71	36–47

The value of a character in its 2–byte Radix–50 representation depends on its position. To obtain the octal value of the character in the Radix–50 representation, you must multiply its Radix–50 octal equivalent by the appropriate power of 50(octal). To gain the full value of the Radix–50 representation of a 3–character string, the values of the 3 characters must be summed. For example, the maximum Radix–50 value (representing the character string 999) is as follows:

47*50^2+47*50^1+47*50^0=174777

Table D–2 provides a convenient means of translating between the ASCII character set and its Radix–50 equivalents based on position within a string.

A 3–character string is stored left to right in the Radix–50 word. For example, given the ASCII string X2B, the Radix–50 representation is computed as follows.

X = 113000(octal)
2 = 002400(octal)
B = 000002(octal)
X2B = 115402(octal)

Note that addition is done in octal.

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. Thus, for the character X, its representation 30(octal) is multiplied by 50^2 to give 113000(octal), the value shown in Table D–2 for X when it is the first character. The second character in a string is stored in the next position to the right. For the character 2 (in the second position), its representation 40(octal) is multiplied by 50^1 to give 002400, the value shown in Table D–2 for 2 when it is the second character. The third character in a 3–character

string is stored in the rightmost position. For the character B (in the third position), its representation is multiplied by 50^0 (which is 1) to give 000002, the value shown in Table D-2 for B when it is the third character. The full octal value of the Radix-50 word is finally gained by adding the value of each character by its position in the string.

Table D-2: ASCII/Radix-50 Equivalents

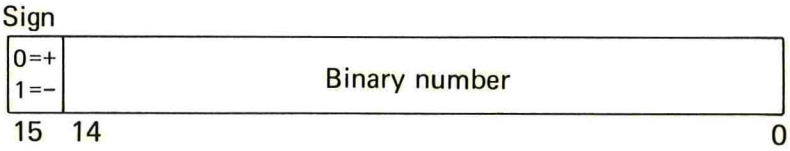
First or Single Character	Second Character	Third Character
space 000000	space 000000	space 000000
A 003100	A 000050	A 000001
B 006200	B 000120	B 000002
C 011300	C 000170	C 000003
D 014400	D 000240	D 000004
E 017500	E 000310	E 000005
F 022600	F 000360	F 000006
G 025700	G 000430	G 000007
H 031000	H 000500	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

(continued on next page)

Table D-2: ASCII/Radix-50 Equivalents (Cont.)

First or Single Character	Second Character	Third Character
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
unused 132500	unused 002210	unused 000035
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
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



Integers are stored in a 2's complement representation. Integer constants must lie in the range -32767 to +32767. For example:

+22 = 000026(octal)
-7 = 177771(octal)

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 0 through 255 (0 through 377 (octal)). Fractions are represented in sign-magnitude notation with the binary radix point to the left. Numbers are assumed to be normalized and, because of redundancy, the most significant bit is not stored (this is called hidden bit normalization). This bit is assumed to be a 1 unless the exponent is 0 (corresponding to 2-128) in which case it is assumed to be 0. The value 0 is represented by two or four words of 0s. For example, +1.0 would be represented by:

```
40200
0
```

in the 2-word format, or:

```
40200
0
0
0
```

in the 4-word format. -5 would be:

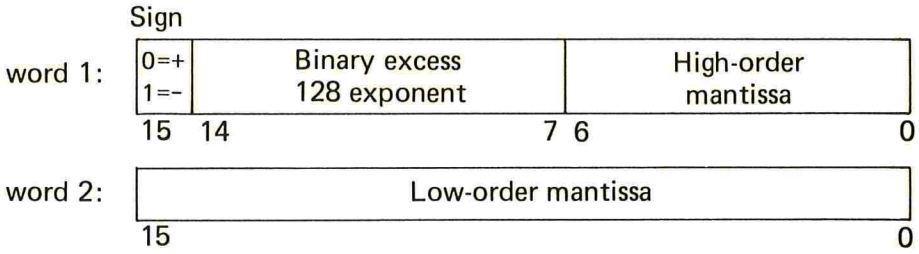
```
140640
0
```

in the 2-word format, or:

```
140640
0
0
0
```

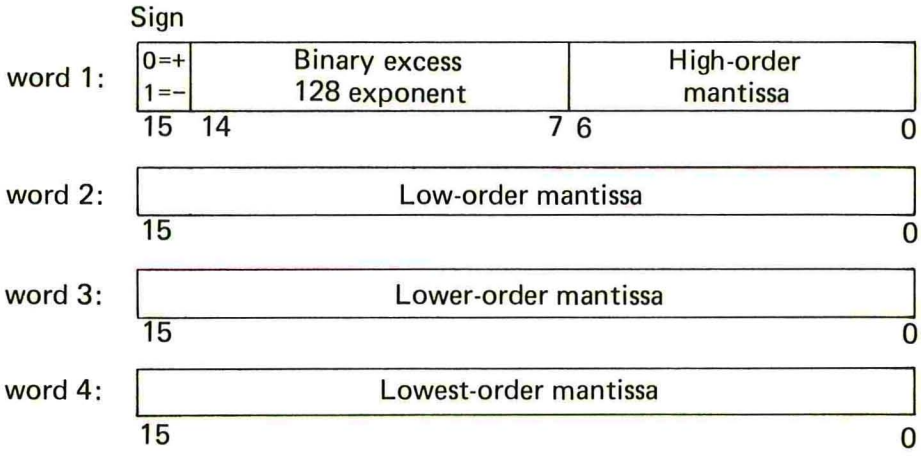
in the 4-word format.

D.4.1 Real Format (2-Word Floating-Point)



Because the high-order bit of the mantissa is always 1, it is discarded, giving an effective precision of 24 bits (or approximately 7 digits of accuracy). The magnitude range lies between approximately .29 X 10^-38 and .17 X 10^39.

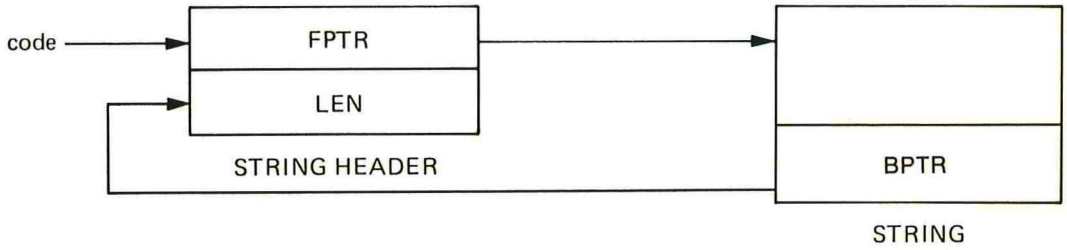
D.4.2 Double-Precision Format (4–Word Floating-Point)



The effective precision is 56 bits (or approximately 17 decimal digits of accuracy). The magnitude range lies between .29 X 10^-38 and .17 X 10^39.

D.5 String and Array Format

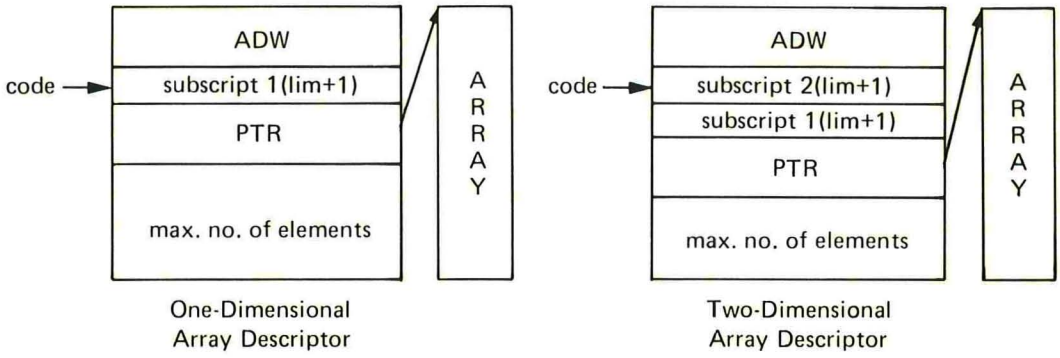
D.5.1 Dynamic String Format



The code for dynamic strings contains 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

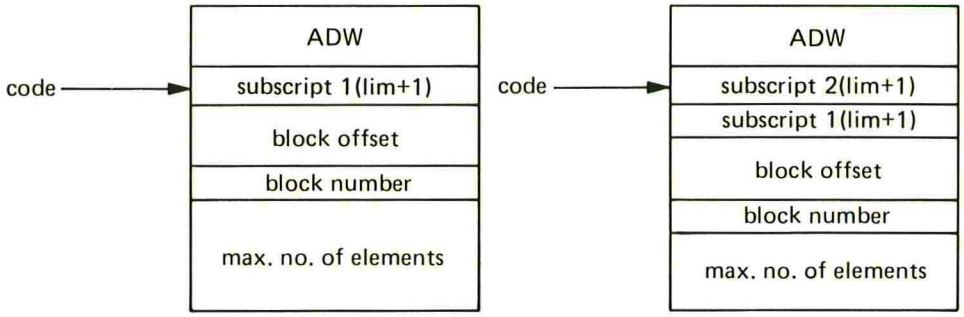
Arrays in Memory:



One-Dimensional Array Descriptor

Two-Dimensional Array Descriptor

Virtual Arrays:



One-Dimensional Array Descriptor

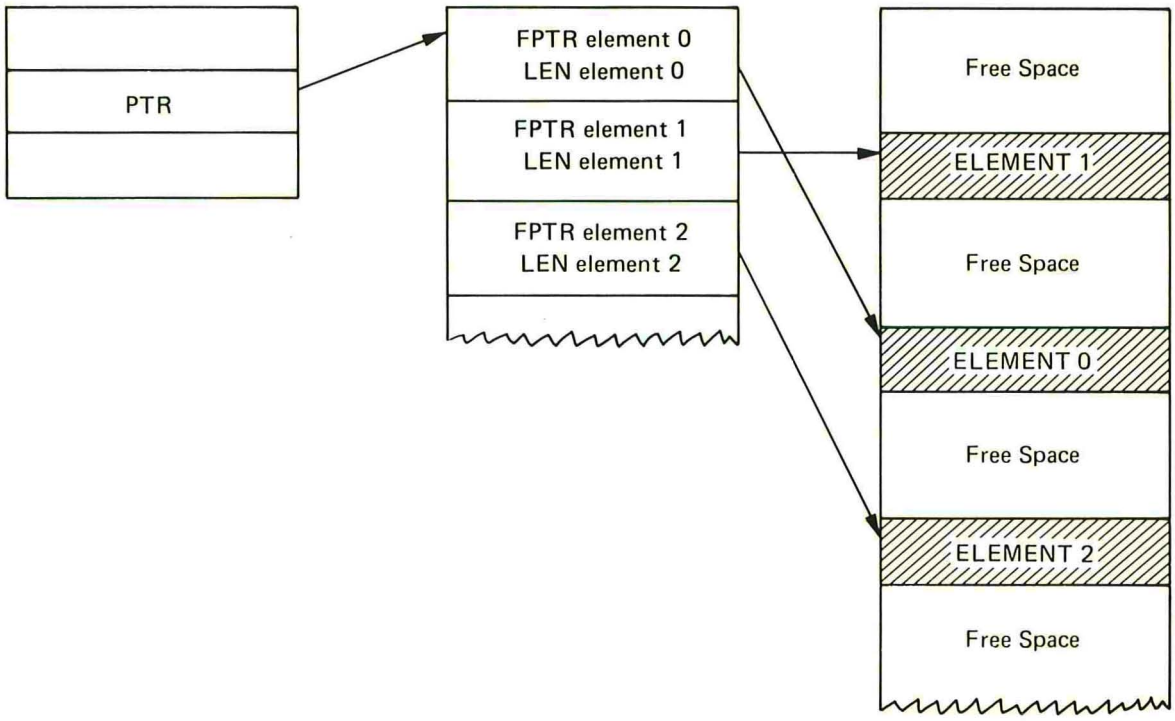
Two-Dimensional Array Descriptor

ADW is the Array Descriptor Word and is explained in Section D.5.3. Subscript is a word that represents the limits defined by the array subscripts plus 1.

The offset into the block and the block number specify the starting position of the array in the file. Block number represents the block that contains the first element of the array (block 1 is the first block of the file, block 2 is the second, etc.). The offset is the offset of the first element of the array in bytes from the beginning of the block that is referenced in block number (byte 0 is the first byte in the block). For example, the first array in a file is represented as block number 1 and the offset is into block 0 in the array descriptor.

The maximum number of elements is only present in the array descriptor 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.

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:



D.5.3 Array Descriptor Word

Table D-3: 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							

T - Data Type
S - Number of subscripts minus 1 (0 is one-dimensional, 1 is two-dimensional)
L - Location (memory or common)

The array descriptor word (ADW) is a 16-bit word as represented in Table D-3. Each type of array causes the bits to be set in an individual manner as follows:

Numeric memory	Bits 0 through 9 are set to 0. Bits 10 and 11 set the data type (i.e., 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; 1 if the array is in 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 LOG2 (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.

Index

BASIC-PLUS-2
RSX-11M/IAS
User's Guide
AA-0157B-TC

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_____

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML 5-5/E39
Maynard, Massachusetts 01754

digital equipment corporation

Printed in U.S.A.