

July 1977

This manual describes the use of the BASIC-PLUS-2 Compiler on the RSTS/E system. It includes descriptions of the run-time systems, libraries, Task Builder, RMS Record I/O, and the Translator.

## **BASIC-PLUS-2**

### **RSTS/E User's Guide**

Order No. AA-0154A-TC

**OPERATING SYSTEM AND VERSION:** RSTS/E V06B

**SOFTWARE VERSION:** PDP-11 BASIC-PLUS-2 V01

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

**digital equipment corporation • maynard, massachusetts**

First Printing, July 1977

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 be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1977 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

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

## CONTENTS

	Page
PREFACE	vii
CHAPTER 1 BASIC-PLUS-2 ON RSTS/E	1-1
1.1 ACCESSING THE RSTS/E SYSTEM	1-1
1.1.1 Filename Specifications	1-3
1.1.1.1 Devices	1-3
1.1.1.2 Extensions	1-4
1.1.1.3 Protection Codes	1-5
1.1.1.4 Option Switches	1-6
1.1.2 Concise Command Language	1-8
1.2 BASIC-PLUS-2 COMPILER	1-8
1.2.1 Commands	1-8
1.2.1.1 APPEND Command	1-11
1.2.1.2 BUILD Command	1-12
1.2.1.3 COMPILE Command	1-14
1.2.1.4 DELETE Command	1-17
1.2.1.5 EXIT Command	1-17
1.2.1.6 HISEG Command	1-17
1.2.1.7 IDENTIFY Command	1-18
1.2.1.8 LIST Command	1-19
1.2.1.9 NEW Command	1-19
1.2.1.10 OLD Command	1-20
1.2.1.11 RENAME Command	1-20
1.2.1.12 REPLACE Command	1-21
1.2.1.13 RUN Command	1-21
1.2.1.14 SAVE Command	1-22
1.2.1.15 SCALE Command	1-22
1.2.1.16 UNSAVE Command	1-22
1.2.2 Editing	1-23
1.2.3 Debugging	1-24
1.2.3.1 BREAK and UNBREAK Commands	1-25
1.2.3.2 STEP Command	1-27
1.2.3.3 PRINT and LET Commands	1-27
1.2.3.4 TRACE and UNTRACE Commands	1-28
1.3 BASIC-PLUS-2 PROGRAMS	1-28
1.3.1 Source Lines	1-28
1.3.2 BASIC-PLUS-2 RSTS/E Sample Program	1-29
CHAPTER 2 TASK BUILDER	2-1
2.1 INVOKING THE TASK BUILDER	2-1
2.1.1 Task Build Command Line	2-2
2.1.1.1 Task File Extensions	2-3
2.2 TASK BUILDER INPUT	2-3
2.2.1 Switches	2-4
2.2.2 Options	2-5
2.2.2.1 ABORT Option	2-6
2.2.2.2 ABSPAT Option	2-7
2.2.2.3 ASG Option	2-7
2.2.2.4 EXTTSK Option	2-7

# CONTENTS (Cont.)

		Page
2.2.2.5	GBLDEF Option	2-8
2.2.2.6	HISEG Option	2-8
2.2.2.7	STACK Option	2-8
2.2.2.8	TASK Option	2-9
2.2.2.9	UNITS Option	2-9
2.3	TASK BUILDER OUTPUT	2-9
2.3.1	Listings	2-10
2.4	PROGRAM SEGMENTATION	2-10
2.4.1	Overlays	2-10
2.5	EXECUTING THE TASK	2-14
CHAPTER 3	RUN-TIME SYSTEMS AND LIBRARIES	3-1
3.1	BASIC-PLUS-2 RUN-TIME SYSTEMS	3-1
3.1.1	BASIC2 RTS	3-2
3.1.2	BP2COM RTS	3-2
3.2	BASIC-PLUS-2 LIBRARIES	3-2
3.2.1	Librarian Utility Program	3-3
3.2.1.1	Create Switch (/CR)	3-4
3.2.1.2	Insert Switch (/IN)	3-5
3.2.1.3	Extract (/EX) and Replace (/RP) Switches	3-5
3.2.1.4	List Switch (/LI, /LE, and /FU)	3-7
3.2.1.5	Delete (/DE) and Compress (/CO) Switches	3-8
3.3	MACRO SUBROUTINES	3-10
3.3.1	Subroutine Linkage	3-10
3.3.2	Subroutine Register Usage	3-11
3.3.3	Subroutine Calls	3-11
CHAPTER 4	FILES	4-1
4.1	FILE CREATION	4-1
4.1.1	Virtual Files	4-2
4.2	INTRODUCTION TO RMS	4-4
4.2.1	Sequential Files	4-5
4.2.2	Relative Files	4-8
4.2.3	Indexed Files	4-10
4.2.3.1	Primary and Alternate Key Record Access	4-14
4.2.4	File Sharing	4-15
4.2.5	RMS Memory Allocation	4-17
4.3	RECORD ACCESS METHODS	4-17
4.3.1	Sequential Access	4-18
4.3.2	Random Access	4-19
4.4	RECORD FORMAT	4-21
4.4.1	Fixed-Length Records	4-22
4.4.2	Variable-Length Records	4-23
4.4.3	Stream-Format Records	4-23
4.5	DATA STRUCTURE	4-24
4.5.1	Blocks	4-25
4.5.2	Buckets	4-25
4.5.2.1	Bucket Size	4-26
4.6	RECORD MAPPING	4-29
4.7	RMS UTILITIES	4-30
CHAPTER 5	TRANSLATOR UTILITY	5-1
5.1	ITEMS FOR TRANSLATION	5-1
5.2	USING THE TRANSLATOR	5-4
5.2.1	Variable Name Specification	5-5
5.2.2	Translator Sample Run	5-6
5.2.3	Translator Warning Messages	5-8



# CONTENTS (Cont.)

		Page
APPENDIX A	COMPATIBILITY	A-1
A.1	TRANSLATABLE ISSUES	A-1
A.1.1	PRINT USING String Format	A-1
A.1.2	Quoted String Literals	A-3
A.1.3	Multiple Assignment Statement	A-3
A.1.4	Ambiguous Constants	A-4
A.1.5	DEF Statements	A-4
A.1.6	POS Function	A-5
A.1.7	DATA Statement String Literals	A-5
A.1.8	Multi-Statement Lines	A-6
A.1.9	Comment Separator	A-6
A.1.10	Continuations	A-6
A.1.11	PRINT Synonym	A-6
A.1.12	Long Variable Names	A-7
A.1.13	CHAIN Statement	A-7
A.1.14	SYS Functions	A-8
A.1.15	INPUT and PRINT Statement Punctuation	A-8
A.2	NONTRANSLATABLE ISSUES	A-8
A.2.1	Transfer into FOR NEXT Loops	A-9
A.2.2	Debugging	A-9
A.2.3	CALL Statements	A-9
A.2.4	Compile-Time Errors	A-9
A.2.5	Array Subscripts	A-9
A.2.6	Record I/O	A-10
APPENDIX B	BASIC-PLUS-2 LANGUAGE ELEMENTS	B-1
B.1	LINE AND DATA FORMAT	B-1
B.2	COMMANDS	B-4
B.2.1	Control Characters	B-6
B.3	FUNCTIONS	B-7
B.4	RESERVED KEYWORDS	B-10
APPENDIX C	ERROR MESSAGES	C-1
C.1	TRACEBACK	C-2
C.2	BASIC-PLUS-2 COMPILE-TIME ERROR MESSAGES	C-3
C.3	ERROR CODES	C-9
C.4	RUN-TIME ERROR MESSAGES	C-12
APPENDIX D	ASCII CODES AND DATA REPRESENTATION	D-1
D.1	ASCII CHARACTER CODES	D-1
D.2	RADIX-50 CHARACTER SET	D-4
D.3	INTEGER FORMAT	D-6
D.4	FLOATING-POINT FORMATS	D-6
D.4.1	REAL Format (2-Word Floating-Point)	D-7
D.4.2	DOUBLE-PRECISION Format (4-Word Floating-Point)	D-7
D.5	STRING AND ARRAY FORMAT	D-7
D.5.1	Dynamic String Format	D-7
D.5.2	Array Format	D-8
D.5.3	Array Descriptor Word	D-9

# CONTENTS (Cont.)

Page

## FIGURES

FIGURE	2-1	Memory Allocation Map	2-11
	2-2	Overlay Structure	2-12
	2-3	Overlay Path	2-14
	3-1	Argument List Format	3-11
	3-2	CALL Statement	3-12
	3-3	CALL BY REF Statement	3-13

## TABLES

TABLE	1-1	Device Specifications	1-4
	1-2	BASIC-PLUS-2 Extensions	1-5
	1-3	Protection Codes	1-5
	1-4	BASIC-PLUS-2 Commands	1-9
	2-1	Default File Types	2-3
	2-2	Task Builder Switches	2-4
	2-3	Task Builder Options	2-6
	4-1	Comparison of File Organizations	4-5
	4-2	Access Methods	4-18
	4-3	Record Formats	4-22
	4-4	Relative File Default Bucket Size	4-27
	4-5	Indexed File Default Bucket Size	4-28
	B-1	Arithmetic Operators	B-2
	B-2	Logical Operators	B-2
	B-3	Relational Operators	B-3
	B-4	Reserved Keywords	B-11
	B-5	System Reserved Keywords	B-12
	C-1	Recoverable Error Codes	C-9
	D-1	ASCII/Radix-50 Equivalents	D-5
	D-2	Array Descriptor Word	D-9

## PREFACE

The BASIC-PLUS-2 RSTS/E User's Guide describes how to use the BASIC-PLUS-2 Language Processor on the RSTS/E operating system.

Chapter 1 summarizes the procedures for accessing the RSTS/E system. The chapter also describes the BASIC-PLUS-2 command format and the building of programs for execution as load modules.

Chapter 2 contains information on the Task Builder that is used to link object modules into an executable task. The chapter also describes Task Builder switches and options and a procedure for building overlays.

Chapter 3 contains information on the BASIC-PLUS-2 Run-Time Systems (RTS), the Librarian Utility Program, and subroutine calling conventions.

Chapter 4 explains RMS (Record Management Services) file handling and Record I/O.

Chapter 5 describes the Translator utility that is used to convert BASIC-PLUS programs to BASIC-PLUS-2.


The manual also contains appendixes that describe compatibility issues, 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 the RSTS/E operating system and the BASIC-PLUS-2 language before reading this user's guide. Manuals that can provide this background are the RSTS/E System User's Guide and the BASIC-PLUS-2 Language Manual. Note that information on RSTS/E documentation can be found in the RSTS/E Documentation Directory. 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, or otherwise ease the exposition of material. The symbols and conventions used are explained below.

Convention	Meaning
	The return key 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.
RUN	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 and 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 indicate that, of several elements shown, one is chosen. Brackets indicate user options.
{ braces }	
[ brackets ]	

## CHAPTER 1

### BASIC-PLUS-2 ON RSTS/E

This chapter contains information on the RSTS/E operating system and the use of BASIC-PLUS-2 on that system. You will find, in this chapter, information on establishing communication with RSTS/E, the RSTS/E file specification format, and a description of the RSTS/E Concise Command Language. Note that the RSTS/E specific information in this manual is a summary only; you are expected to be familiar with the RSTS/E system and with the information found in the RSTS/E System User's Guide.

Chapter 1 also describes the syntax and use of commands, editing of BASIC programs, debugging aids, and creation of source files.

#### 1.1 ACCESSING THE RSTS/E SYSTEM

RSTS/E (Resource Sharing Time Sharing/Extended) is a time-sharing system. A time-sharing environment permits many users to process data on the same system. RSTS/E schedules the execution of each user's job and attempts to minimize the time used to process individual programs.

The resource sharing facility of RSTS/E means that each user has access to system peripherals and devices. These devices can be terminals, disks, line printers, card readers, etc. Throughout this manual, a terminal is assumed to be the input and output device.

You gain access to the RSTS/E system by typing:

```
HELLO  (RET)
```

on the terminal keyboard. In response, the system runs the LOGIN program that prints an identification line and then issues a request for your identification. LOGIN indicates a request for information by printing a prompt on the terminal. In this case, the prompt is a number sign (#).

Because many users can share RSTS/E, you must have a system identity to distinguish yourself from other users. The system manager assigns this identification to you in the form of a project, programmer number and password. When you attempt to access RSTS/E, LOGIN asks for this identification. A typical access dialogue consists of a series of prompts and responses, and appears as follows:

## BASIC-PLUS-2 ON RSTS/E

HELLO

RSTS V06B-02 TIMESHARING RJE JOB 20 KB1 07-APR-77 12:49 PM

#26,11

PASSWORD:

07-APR-77

THE TAPE DRIVES ARE NOW AVAILABLE FOR GENERAL  
USE. THESE DRIVES ARE LABELED MT2 AND MT3.

READY

The LOGIN prompt, #, represents a request for you to type your project and programmer numbers separated by a comma. In the above example, 26 is the project number and 11 is the programmer number. The system manager assigns these numbers to you to identify and protect your programs and files. When you create programs or files, the system stores them in your numbered account on the public structure.

The system manager also assigns you a unique password. You type this password in response to the PASSWORD: prompt. LOGIN checks the password you type against a master directory of permitted project, programmer number and password combinations. If your password matches the one on the master directory, you are allowed access. Notice that when you type the password it is not printed on the terminal. This prevents unauthorized persons from copying it.

The system prints READY on the terminal to signify that you have successfully gained access. This means that RSTS/E is now prepared to accept input.

After you complete the access procedure, the LOGIN program may print messages written by the system manager. These messages often contain helpful information such as the message concerning tape drives that is included in the above dialogue.

You type BYE on the terminal to end system use. When you type BYE, the system runs the LOGOUT program, saves any programs you have stored on the public structure, and prints a message indicating how much storage space is left in your account.

Consider the following example:

```
BYE
CONFIRM: Y
SAVED ALL DISK FILES; 328 BLOCKS IN USE, 4672 FREE
JOB 20 USER 26,11 LOGGED OFF KB1 AT 07-APR-77 01:01 PM
SYSTEM RSTS V06B-02 TIMESHARING RJE
RUN TIME WAS .6 SECONDS
ELAPSED TIME WAS 2 MINUTES, 30 SECONDS
GOOD AFTERNOON
```

In this example, you type BYE on the terminal and LOGOUT asks you to confirm your intention to log off. If you type Y (for Yes), the program logs you off. It also prints information on the amount of disk space you have used, the current date and time, and the amount of time you were on the system.

If, in response to the CONFIRM: prompt, you type N (for No), LOGOUT ignores the exit, prints a READY, and allows you to type further input. If you type I (for Inspect), LOGOUT prints a list of each file or program name in your account. This list is printed, one name at a time, and affords you the opportunity to retain or delete each program. To retain the program, type the RETURN key following the

## BASIC-PLUS-2 ON RSTS/E

name. To delete a program, type the letter K (for Kill) after the name. If you type a question mark (?) in response to CONFIRM:, the LOGOUT program prints a list of exit options. Finally, if you type BYE/F instead of BYE, LOGOUT performs a fast exit without printing a CONFIRM: or any additional information.

### 1.1.1.1 Filename Specifications

The RSTS/E system accepts a filename specification of the form:

dev:[p,pn]filnam.ext<prot>/sw

where:

dev:	can be a 1- to 6-character logical name, or a 2-character device code followed by a unit number. If the device code is omitted, the default is the public structure. Devices and device codes are summarized in Section 1.1.1.1.
[p,pn]	is a project (p), programmer number (pn) that identifies the owner of a program or file. These numbers are assigned to you by the system manager and are used as part of the access procedure.
filnam	is any 1- to 6-character alphanumeric name that identifies a program or file.
.ext	is a 1- to 3-character alphanumeric extension code (preceded by a period) denoting the type of program or file. If an extension is not specified, RSTS/E and BASIC supply a default extension. Extensions are summarized in Section 1.1.1.2.
<prot>	is a protection code that is used to restrict access to a program or file. Protection codes are assigned when a file is created or renamed. They are summarized in Section 1.1.1.3.
/sw	is a system file specification option. There are four switch options available: /CLUSTERSIZE, /FILESIZE, /MODE, and /RONLY. If you specify an option switch, it must be the final RSTS/E file designation and must precede any BASIC-PLUS-2 switch specification. RSTS/E switch options are summarized in Section 1.1.1.4.

**1.1.1.1.1 Devices** - You indicate devices in filename specifications to specify the use of a particular medium for input or output. The physical device names that can be used in filename specifications are listed in Table 1-1. If you do not specify a device, the default is the public structure, SY:. For more information on device specifications and the assignment of logical device names, refer to the RSTS/E System User's Guide.

# BASIC-PLUS-2 ON RSTS/E

Table 1-1  
Device Specifications

Device	Code
Card reader	CR:
CD-11, CR-11, or CM11 card reader	CD:
RP02 or RP03 disk (n=0 to 7)	DPn:
RP04, RP05, and RP06 disk (n=0 to 7)	DBn:
RS03 or RS04 fixed-head disk (n=0 to 7)	DSn:
RC11 fixed-head disk	DC0:
RF11 fixed-head disk drive	DF0:
RK05 or RK05F disk cartridge drive (n=0 to 7)	DKn:
RK06 disk cartridge drive (n=0 to 7)	DMn:
Public disk structure (default storage)	SY:
DECTape (n=0 to 7)	DTn:
RX01 floppy disk (n=0 to 7)	DXn:
Line printer (n=0 to 7)	LPn:
TU16, TU45 magtape (n=0 to 7)	MMn:
TU10 or TS03 magtape (n=0 to 7)	MTn:
Null device	NL:
High speed paper tape punch	PP:
High speed paper tape reader	PR:
Unit from which system was bootstrapped	SY0:
Terminal n in the system (if KBn:) or current user terminal	KBn:, KB:, TTn:, TT:, or TI:

Note that RMS files (see Chapter 4) are not allowed on DECTape (DT), RX01 floppy disk (DX), or null devices (NL).

**1.1.1.2 Extensions** - A filename extension (also called a file type) can be used to indicate the use or internal characteristic of the program or file. For instance, a BASIC-PLUS-2 source program has an extension of .B2S.

The most common BASIC-PLUS-2 extensions, with their meanings, are listed in Table 1-2.



# BASIC-PLUS-2 ON RSTS/E

Table 1-2  
BASIC-PLUS-2 Extensions

Extension	Meaning
.B2S	Source program (stream ASCII format)
.CMD	Command file
.DAT	Data file
.DIR	Directory file
.LST	Listing file
.MAC	Compiler macro source file
.MAP	Task Builder memory map
.OBJ	Object module
.ODL	Overlay Description Language file
.TMP	Temporary file
.TSK	Executable task image (binary format)

For more information on filename extensions, refer to the RSTS/E System User's Guide.

1.1.1.3 Protection Codes - Protection codes, <prot>, designate which class of user can access your program or file and the types of access allowed. There are three classes of users:

1. Owner
2. Group - all users having the same project number as the owner
3. Others - all other users not in the owner's group

Access is denied to one or more of these classes by specifying a code as shown in Table 1-3.

Table 1-3  
Protection Codes

Code	Meaning
1	Read protect against owner
2	Write protect against owner
4	Read protect against other users with owner's project number
8	Write protect against other users with owner's project number

(Continued on next page)

# BASIC-PLUS-2 ON RSTS/E

Table 1-3 (Cont.)  
Protection Codes

Code	Meaning
16	Read protect against all others who are not assigned owner's project number
32	Write protect against all others who are not assigned owner's project number
64	Compiled, run-only files
128	Temporarily privileged program (compiled, run-only file)

You can combine these codes as desired and enter them in the file specification line to provide different degrees of protection. When you specify a code, enclose the number within angle brackets as part of the file specification. For example, a protection code of <48> is the result of combining code 32 (write protect against class 3) with code 16 (read protect against class 3). Thus, when you specify code <48> in the file specification, read or write access to that file is denied to anyone logged on the system with a different project number.

When you create a task image (i.e., extension .TSK), RSTS/E supplies a protection code of <124> by default. This allows you to execute the task and to have read/write access, and disallows execution, read, or write access to all others. That is, 124 is the combination of 64 (compiled file), 32 and 16 (execution, write, and read protect against all other projects), and 8 and 4 (execution, write, and read protect against others in the owner's project).

When a program is built and saved (i.e., a source file) or compiled as a MACRO file or object module, the default is <60>. This code protects the file from others in the project (codes 8 and 4) and others outside the project (codes 16 and 32).

Protection codes need only be specified when the protection assigned by default is not sufficient.

For more information on protection codes, refer to the RSTS/E System User's Guide.

**1.1.1.4 Option Switches** - A RSTS/E file specification can include an optional switch as the last system element in the specification. These switches are specific to the RSTS/E system and must not be interspersed with any BASIC specification. If a file specification line contains both system switches and BASIC specifiers, the system switches must precede the BASIC specifiers. The switches and their use are summarized below; refer to the RSTS/E System User's Guide for more complete information.

The /FILESIZE option allows you to create a disk file of a specified size prior to any read or write operation. The option switch is written as follows:

```
/FI[LESIZE]:[#]n[.]
```

## BASIC-PLUS-2 ON RSTS/E

where:

- /FILESIZE is the switch name and can be abbreviated to a 2-character minimum, i.e., /FI.
- # the optional number sign converts the argument, n, to an octal value.
- n is a decimal number in the range of 0 to 32767 and indicates the number of blocks in the pre-extended file.
- . the optional trailing decimal point ensures that n is interpreted as a decimal number.

The /CLUSTERSIZE option establishes the minimum cluster size for a disk file. A cluster is a number of contiguous blocks that are treated as a unit; RSTS/E permits file cluster sizes of 1, 2, 4, 8, 16, 32, 64, 128, or 256 blocks.

The /CLUSTERSIZE switch has the same general format as /FILESIZE with the exceptions noted below:

/CL[USTERSIZE]:[-][#]n[.]

where:

- is an optional minus sign that specifies a negative cluster size. This option is used to avoid an error resulting from an illegal cluster size specification.
- n is the cluster size specification in blocks of 1, 2, 4, 8, 16, 32, 64, 128, or 256.

The /MODE option allows you to pass up to 15 (decimal) bits of information to the device driver at file open time. The meaning of these bits is device dependent and you should refer to the RSTS/E Programming Manual for this information.

The /MODE switch has the same general format as /FILESIZE with the exception noted below:

/MO[DE]:[#]n[.]

where:

- n specifies a mode setting in the range of 0 to 32767.

The /ROONLY option allows you to set the read only MODE value for a disk file. The switch accepts no arguments and has the following format:

/RO[NLY]

These switches do not apply to RMS files. RMS does support the function of the /FILESIZE and /CLUSTERSIZE options, but only with file attributes specified in the OPEN statement (see Section 4.2).

### 1.1.2 Concise Command Language

The Concise Command Language (CCL) provides an alternative method for invoking RSTS/E system programs such as the Peripheral Interchange Program (PIP) and Directory (DIRECT). CCL commands allow you to run a system program by means of a single command specification.

The CCL is composed of a set of commands with specific functions. CCL commands are installed as part of the system and the system manager has the option of defining the commands and their functions. You should contact the system manager to determine the current status of CCL on the system.

To use CCL, type on one line the CCL command and the optional command argument string as defined for your system. RSTS/E loads the specified CCL program, inputs the argument string (if present), and executes the program command. Note that when you exit from a CCL-invoked system program, you return to the default run-time system.

For more information on CCL, refer to the RSTS/E System User's Guide.

## 1.2 BASIC-PLUS-2 COMPILER

The RSTS/E operating system uses a BASIC-PLUS-2 Language Processor composed of a Compiler and a Run-Time System/Library. The RTS/Library is discussed in Chapter 3.

There are two types of operations available with the BASIC Compiler. The primary type of operation produces an executable load module in task image format from your source program. The compiler checks each program line for syntax errors as you enter it and returns an appropriate error message if one is found. You can then correct the program (if necessary), compile, and execute it. This mode of operation is detailed in the rest of this chapter.

The second type of operation produces programs as linkable object modules suitable for input to the Task Builder. The Task Builder must process object modules into a task image before they can be executed. Object modules need only be produced when the source program's requirements exceed the in-core addressability limit of 32K words, a CALL statement is used to access external routines, or RMS is used to perform file I/O. The Task Builder is discussed in Chapter 2.

RSTS/E is capable of supporting multiple run-time systems. If BASIC-PLUS-2 is not the default, type:

```
RUN $BASIC2
```

on the terminal. This procedure clears the current contents of memory and invokes the BASIC-PLUS-2 Compiler. If access to BASIC-PLUS-2 is successful, an identification line (see Section 1.2.1.7) followed by the READY prompt print on the terminal. READY indicates that the compiler is prepared to accept input.

### 1.2.1 Commands

Input to the BASIC Compiler can be a command or a source program line. BASIC source programs are described in Section 1.3. This section and the subsections that follow describe the BASIC commands.

## BASIC-PLUS-2 ON RSTS/E

Commands are used to perform various functions outside the context of programs. That is, they require no line numbers and you type them directly on the terminal, along with any required arguments. Table 1-4 lists the BASIC commands with brief explanations of their use. Succeeding sections describe each command in detail. The commands listed in Table 1-4 can be used individually or combined in a user-created command file. The command file allows you to execute a series of BASIC-PLUS-2 commands by means of a single command file specification.

Although many of the RSTS/E system commands are listed in Table 1-4, BASIC-PLUS-2 does not support the complete system command set. BASIC-PLUS-2 on RSTS/E only supports the commands listed in Table 1-4 and those installed as CCL commands. If you attempt to use any other command, such as ASSIGN, BASIC returns an error message (i.e., ?WHAT?) and ignores the command.

Table 1-4  
BASIC-PLUS-2 Commands

Command	Function
APPEND	Merges the current program with a previously saved program.
BUILD	Creates an indirect command file and an overlay description file. These files can be used to specify input to the Task Builder program.
COMPILE	Translates a BASIC source program into an executable file in task image format with a default extension of .TSK. If the program cannot be compiled as an executable file, the command translates the program into an object module.
COMPILE /DEBUG	Translates a BASIC source program and enables the use of the BASIC debugging aid.
COMPILE /DOUBLE	Translates a BASIC source program and enables the double-precision (4-word floating-point) math package.
COMPILE /MACRO	Translates a BASIC source program into a MACRO source file.
COMPILE /NOCHAIN	Translates a BASIC source program and indicates that another program will only chain to the beginning of the compiled program.
COMPILE /NOLINE	Translates a BASIC source program and indicates that internal line headers will not be used for error processing. This command has the effect of saving at least two words of memory per program line.

(Continued on next page)

# BASIC-PLUS-2 ON RSTS/E

Table 1-4 (Cont.)  
BASIC-PLUS-2 Commands

Command	Function
COMPILE /OBJECT	Translates a BASIC source program into an object module. The .OBJ extension is appended to the filename by default.
COMPILE /TSK	Translates a BASIC source program into an executable file with the .TSK extension. If the program cannot be compiled as an executable file, a fatal error is generated.
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 and returns you to the default run-time system.
HISEG	Allows you to choose between one of two supplied BASIC-PLUS-2 run-time systems.
IDENTIFY	Causes BASIC-PLUS-2 to print an identification header on the terminal.
LIST	Prints a copy of the current program or its specified lines on the terminal.
NEW	Clears memory for the creation of a new program.
OLD	Loads a specified program into memory.
RENAME	Changes the name of the current program.
REPLACE	Stores the current program on the public structure or a specified device.
RUN	Executes a specified program or file.
SAVE	Copies and preserves a source program on the public structure or a specified device.
SCALE	Controls the scale factor for double-precision (4-word floating-point) format.
UNSAVE	Deletes a specified file.

## BASIC-PLUS-2 ON RSTS/E

These commands can be abbreviated to a minimum 3-letter specification. When this shortened form of the command is used, you must type at least the first three letters of the command. For example, the COMPILE /DEBUG command can be abbreviated to COM /DEB. Note that if the abbreviation NH is used with an abbreviated command that allows no header specifications (LIST and RUN), NH must be appended to the command abbreviation, i.e., RUNNH and LISNH. The specific abbreviations for each command are given in the appropriate subsection that follows.

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

```
APPEND  (RET)
```

on the terminal, 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
```

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

If both programs you wish to append 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 1.2.1.10).

If you do not specify a filename in the APPEND command but type only a carriage return, the compiler searches for a source program called NONAME.B2S. If no filename 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.

As an example of APPEND, consider the following procedure. You have built two programs named AP1 and AP2 and saved them on the public structure. These programs appear as follows:

10	LET B=5	35	LET D=A^C
20	LET C=2	40	PRINT A;D
30	LET A=B^C		
40	PRINT A		
50	END		
	AP1		AP2

## BASIC-PLUS-2 ON RSTS/E

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

READY

APP AP2

READY

LIST
AP1          01:54 PM      27-MAY-77
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.

**1.2.1.2 BUILD Command** - The BUILD (BUI) command accepts the names of one or more object modules as input and creates an indirect command file with the default extension .CMD. This file contains all of the Task Builder command input required to create an executable task image file with a default extension of .TSK and a memory allocation map with a default extension of .MAP. In addition to the command file, the BUILD command generates an overlay description language file (extension .ODL). You can edit the contents of this file to produce overlaid program segments. The procedure used to overlay the BUILD command output is described in Section 2.4.1.

An object module is a user program that is compiled by means of the COMPILE /OBJECT command (see Section 1.2.1.3). You create object modules and link them by means of the Task Builder for the following reasons:

1. To produce a memory allocation map - The map is a file that contains descriptions of program code storage allocation and global symbol definitions.
2. To link subprograms - User subprograms must be separately compiled as object modules and selectively linked with your program to create a single executable file.
3. To access RMS required code - Record I/O operations on sequential, relative, or indexed files (see Chapter 4) 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 and run-time system (see Section 1.2.1.6). Because the BUILD command automatically creates an indirect command file that contains all of this input, 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 a map file based on a single command file specification. Note that if you wish to link your program with special Task Builder options, the BUILD command output must be modified (see Section 2.4.1). The use of the Task Builder program and special options are described in Chapter 2.

To use the BUILD command, type:

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

on the terminal. In this example, main represents the name of a program compiled as an object module. This filename becomes the name of the indirect command file with the .CMD default extension 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. There is no specific limit on the number of modules contained 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 perform RMS record I/O operations, you must append the appropriate switch(es) to the end of the command line. The switches and their use are as follows:

/SEQ	links in the RMS code required for sequential file operations.
/REL	links in the RMS code required for relative file operations.
/IND	links in the RMS code required for indexed file operations.

You can use any combination of the RMS 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 sequential, relative, or indexed file, the appropriate switch(es) must be appended.

The command line shown above results in an overlay description file (MAIN.ODL) and an indirect command file named MAIN.CMD. If you input the command file to the Task Builder program, a task and map are generated. For example:

```
RUN $TKB
TKB> @MAIN
TKB> //
READY
```

Following successful task creation, the system prints the READY prompt on the terminal. This indicates that the operation is complete and that your user area contains an executable task image file (MAIN.TSK) composed of the linked modules you specified as input. Your area also contains a memory allocation map (MAIN.MAP). The filename for both the task image and map is the name of the first module appearing as input in the command line. The actual linking operation is handled by the Task Builder. For more information on the Task Builder, refer to Chapter 2 of this manual or to the RSTS/E Task Builder Reference Manual.

1.2.1.3 **COMPILE Command** - The COMPILE (COM) command translates a program currently in memory into executable machine language code. This command can be used in conjunction with the following optional switches: /DEBUG, /DOUBLE, /MACRO, /NOCHAIN, /NOLINE, /OBJECT, and /TSK. A LOCK command is also available that allows you to specify default switch settings.

When used alone, the COMPILE command translates the program into executable code in task image format and stores it on the public structure. If the program is a subroutine or contains a CALL statement that references an external subroutine or contains an RMS OPEN statement, the COMPILE command causes the automatic generation of an object module. The generation of an object module is indicated by one of the following printed messages on the terminal:

%CALL/SUB forces OBJ output

or

%RMS I/O forces OBJ output

The appropriate default extension, .TSK or .OBJ, is appended to the program name. The program is not executed; it is only compiled and saved. For example, if the program is currently in memory (i.e., at command level) and you type:

COMPILE

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

COMPILE filespec

where filespec is a RSTS/E file specification. This procedure compiles the current program, assigns the specified name, and appends .TSK or .OBJ to the name (if no other extension is specified). To compile a source program that is not in memory, you must bring it into memory by means of an OLD command (see Section 1.2.1.10) and then type COMPILE.

The COMPILE /DEBUG (COM /DEB) command translates the program into executable code and enables the use of the BASIC-PLUS-2 debugging aid. The debugging aid is described in Section 1.2.3. Note that the program must be compiled with the /DEBUG switch before the debugging aid can be used. Also, because the debugging aid is line-oriented, it does not allow the use of the BREAK command (see Section 1.2.3.1) in a program compiled with the /NOLINE switch.

The COMPILE /DOUBLE (COM /DOU) command translates the program into executable 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. The BASIC default is single-precision format (2-word) for floating-point operations.

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

The COMPILE /NOCHAIN (COM /NOC) and COMPILE /NOLINE (COM /NOL) commands translate the program and reduce the memory requirements of the output program. The /NOCHAIN switch disables line number table

storage in memory. The amount of required memory that is reduced by this switch depends on the content of the program. An attempted chain operation to a specified line in a program compiled with /NOCHAIN generates an error; however, any chain from the compiled program is allowed.

The /NOLINE switch disables program line headers in memory and reduces program requirements by the following amounts:

- Two words per line
- Two 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 program, 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

The COMPILE /OBJECT (COM /OBJ) command translates the program and saves it as an object module. The extension, .OBJ, is appended to the program name. Programs compiled as object modules must be linked into a task image before they can be executed. You can construct a task image by means of the Task Builder (see Chapter 2).

In most cases, the switches described above can be combined in the COMPILE command. The /TSK, /OBJECT, and /MACRO switches, however, cannot be combined with each other. For example:

COM /OBJ /DEB /DOU /NOL /NOC

is a legal specification, but:

COM /OBJ /MAC

is not.

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 a task image file by default.

## BASIC-PLUS-2 ON RSTS/E

Consider the following example:

```
LOCK /OBJ/NOL/NOC  
  
OLD PROG1  
  
READY  
  
COM  
  
READY  
  
OLD PROG2  
  
READY  
  
COM  
  
READY  
  
OLD PROG3  
  
READY  
  
COM /TSK  
  
READY  
  
LOCK  
  
OLD PROG4  
  
READY  
  
COM /MAC  
  
READY
```

In this example, four programs are brought into memory by means of OLD commands (see Section 1.2.1.10). The initial LOCK command sets the /OBJECT, /NOLINE, and /NOCHAIN compiler switches as the defaults. When you compile PROG1 and PROG2, they become object modules with /NOLINE and /NOCHAIN enabled. When you compile PROG3, however, the creation of a task image file is specified. This overrides the /OBJECT default and creates a compiled program with the .TSK extension. The /NOLINE and /NOCHAIN switches are left enabled. Finally, the LOCK command with no arguments disables all defaults and PROG4 is compiled as a MACRO file with no switches in force. The result of these four compilations is as follows:

```
PROG1.OBJ (NOLINE and NOCHAIN enabled)  
PROG2.OBJ (NOLINE and NOCHAIN enabled)  
PROG3.TSK (NOLINE and NOCHAIN enabled)  
PROG4.MAC (no switches enabled)
```

Note that if you specify COM /TSK to a program that cannot compile into a task image file, a fatal error is generated:

```
?TSK output not possible
```

## BASIC-PLUS-2 ON RSTS/E

**1.2.1.4 DELETE Command** - The DELETE (DEL) command removes a specified line or lines from the program currently in memory.

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:

DEL 50

removes line 50 from the program.

DEL 50, 80

removes lines 50 and 80 from the program.

DEL 50-80

removes lines 50 through 80 from the program.

DEL 50, 60, 90-110

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

If you do not specify a line in the DELETE command, 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. That is, 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. 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.

**1.2.1.5 EXIT Command** - The EXIT (EXI) command terminates access to BASIC-PLUS-2 and returns you to the default run-time system. This command is the only means of leaving BASIC-PLUS-2 that ensures the immediate return of control to the default run-time system.

**1.2.1.6 HISEG Command** - BASIC-PLUS-2 is supplied with two separate run-time systems and libraries called BASIC2 and BP2COM, respectively. BASIC2 is installed as the default (see Section 1.2) and contains all of the run-time support routines that are required to generate an executable task image file. BP2COM is a minimum run-time system and uses its library to store most of the run-time system support routines.

The HISEG command is used to switch run-time systems and allows you to include BP2COM in the command file that is generated by the BUILD command (see Section 1.2.1.2). The decision to use BP2COM rather than the BASIC2 default is based on time and space considerations.

The BASIC2 run-time system is 16K words long and the associated library is minimal. Because all of the routines required by a BASIC program are present in the run-time system (with the exception of RMS I/O), BASIC2 is used to generate an executable task image file (i.e., COM /TSK). BASIC2 also minimizes task image size and program linkage time.

## BASIC-PLUS-2 ON RSTS/E

The BP2COM run-time system is 4K words long and the library is used to contain the BASIC routines. BP2COM cannot be used to directly generate an executable file; it must be included in the command file that is generated by the BUILD command. This command file is specified as input to the Task Builder, which selects the BASIC routines required by the program. The selected routines are then linked to the program to produce an executable file.

With the BP2COM run-time system and library included in the Task Builder input, you can produce a program with a maximum size of 28K words. However, because the required BASIC routines are included, they cannot be shared with other users.

To use the HISEG command, type:

```
HISEG  (RET)
```

In response to the command, BASIC prompts for the name of the desired run-time system and the account number under which it is stored. For example:

```
HISEG
Name-- BP2COM
Account- (1,1)
```

This example causes BP2COM to be used in all succeeding BUILD commands. If the command is successful, READY is printed at the terminal. If you follow this procedure with a BUILD command, the generated command file contains the BP2COM run-time system and library as well as the specified object modules (see Section 1.2.1.2).

BP2COM remains the BUILD command default run-time system until you replace it by means of another HISEG command or exit from BASIC-PLUS-2.

**1.2.1.7 IDENTIFY Command** - The IDENTIFY (IDE) command prints a BASIC-PLUS-2 header on the terminal. The header consists of the BASIC-PLUS-2 name and version number. IDE eliminates confusion as to what BASIC is currently in effect. That is, an identifying header is printed in response to this command only when the current run-time system is BASIC-PLUS-2.

Consider the following example:

```
IDE
BASIC-PLUS-2      V01-00
READY
EXIT
READY
IDE
?WHAT?
```

In this example, the current availability of BASIC-PLUS-2 is confirmed by the result of typing the IDE command. After you type EXIT (see Section 1.2.1.5), BASIC-PLUS-2 is replaced by the default run-time system. An IDE command on the default system (assuming that it is not

## BASIC-PLUS-2 ON RSTS/E

BASIC-PLUS-2) 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 (i.e., type RUN \$BASIC2).

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

If you type:

```
LIS (RET)
```

the entire program is printed, along with header material containing 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:

```
LISNH (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 1.2.1.4). For example:

```
LIS 30, 70
```

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

```
LISNH 30-70
```

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

**1.2.1.9 NEW Command** - The NEW command reserves space for building programs by creating a temporary file. When you type NEW on the terminal, any name and source code currently in the compiler's buffer or in a temporary file are deleted. After you type the command, the system 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 the system 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
```

the system assigns the name PROG1 to the program you create.

## BASIC-PLUS-2 ON RSTS/E

In all cases, NEW creates space for source files, so the default extension is .B2S. If you specify any other extension in the NEW command, it is ignored.

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

OLD (RET)

the system 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 specified program, with a .B2S extension, to be read into memory and become the current program. The program is now ready for processing (i.e., editing, execution, compiling, etc.).

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

OLD filespec

where filespec is the program name. If you specify a program name that does not exist, or if NONAME.B2S cannot be found, the system 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 the system reads in the specified program, it performs no syntax check on the contents.

1.2.1.11 RENAME Command - The RENAME (REN) command changes the name of a program currently in memory. For example, if you have a program in memory named FILE1 and you type:

REN FILE2

the name FILE1 is erased and replaced with the name FILE2. If you type SAVE (see Section 1.2.1.14), the program is stored on the public structure with the name FILE2.

If you bring a saved program named FILE1 into memory with an OLD command and type:

REN FILE2

the program is named FILE2 on the temporary file but retains the name FILE1 on the public structure.



## BASIC-PLUS-2 ON RSTS/E

1.2.1.12 **REPLACE Command** - The REPLACE (REP) command replaces a program on the public structure 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:

REP     RET

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

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

REP FILE1

where FILE1 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 the public structure. 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 public structure.

1.2.1.13 **RUN Command** - The RUN command causes the execution of a program. If a program is currently in memory and you type:

RUN     RET

the program is compiled and executed. If the program is saved on the public structure, you must specify the name of the program in the RUN command. For example:

RUN PROG1

causes the specified program, PROG1, to be executed.

When you type only the name of a specific program (i.e., no extension) with the RUN command, the system searches for and executes a compiled program of that name. This program need not be a BASIC-PLUS-2 executable file. If such a program cannot be found, the system attempts to OLD the specified source program (extension .B2S), compile it, and then execute the program. However, the compiled image of an executed source program is not saved. Obviously, it takes more time to run a source program because it must be compiled first. It saves time if you compile your completed program before execution (see Section 1.2.1.3). Note that if you execute a non-BASIC-PLUS-2 program, completion of the program returns you to the default run-time system.

When you issue the RUN command, the executed program includes a header as part of the output. This situation parallels that found with the LIST command (see Section 1.2.1.8). To run the program without a header, you type RUNNH.

## BASIC-PLUS-2 ON RSTS/E

**1.2.1.14 SAVE Command** - The SAVE (SAV) command preserves a completed source program by transferring it from memory into a file on a specified device or the default public structure. For example, if you have a program in memory and type:

SAV    

the line numbers of the program are sequenced, and the program is stored on the public structure as source code under the current name with a .B2S extension. If you wish to specify a particular storage device, extension, or program name, type:

SAV filespec

where filespec is a RSTS/E file specification that contains the desired name or device (see Section 1.1.1). If you have built an unnamed program, a SAVE command with no specification stores the program as NONAME.B2S.

If you attempt to save a program that has the same name 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 1.2.1.11 and 1.2.1.12.

**1.2.1.15 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 errors in fractional computations performed when floating-point format is enabled. SCALE enables 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. 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 SCALE command factor of 0. Note that a SCALE command with no factor specification causes the system to print the current scale factor. Refer to the RSTS/E System User's Guide for a detailed explanation of the SCALE command.

**1.2.1.16 UNSAVE Command** - The UNSAVE (UNS) command deletes a file from the public structure. For example, if you type:

UNS    

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

UNS filespec

the specified file, filespec, is deleted from the public structure or specified device whether or not it is currently in memory. This

## BASIC-PLUS-2 ON RSTS/E

command is useful for erasing unwanted files from the public structure or other specified devices or accounts.

The UNSAVE command causes the system to search for and delete a specified source program. If the program is not found, the system 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 extension. For example:

```
UNS FILE.TSK
```

### 1.2.2 Editing

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 built, or a saved program after it is brought into memory by means of an OLD command. You cannot edit a compiled program or an object module.

As you create new programs, you can erase misspelled or incorrect characters with the RUBOUT key and type corrections at the terminal. Note that RUBOUT is labeled the DELETE 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 RUBOUT key and retype as follows:

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

Press the RUBOUT 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 RUBOUT 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) RFT  
?Syntax error
```

## BASIC-PLUS-2 ON RSTS/E

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
```

Also, you can use the DELETE command to perform the same function (see Section 1.2.1.4).

### 1.2.3 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 and CONTINUE and their use is permitted only on programs or subroutines that are compiled with the /DEBUG switch (see Section 1.2.1.3). 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 acts to conserve memory.

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

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 on the terminal:

```
DEBUG:  prog name
```

where prog name is the name of the program or subroutine 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. To reinitiate program execution and cause the specified command action, type the CONTINUE (CON) command as follows:

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

## BASIC-PLUS-2 ON RSTS/E

In this instance, the CON command reinitiates program execution as specified by the BREAK command, i.e., program 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 subroutine. This message has the form:

command AT LINE n [,name]

where:

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

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

name is the name of the currently executing subroutine. 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 1.2.1.5). This command terminates the debugger and returns you to the default run-time system.

**1.2.3.1 BREAK and UNBREAK Commands** - You type the BREAK command in response to a debugging aid prompt. The command is typed at the terminal as follows:

# BREAK [arg]

where arg is a command argument that causes a halt at a specified point in a program or subroutine 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 subroutine.

BREAK n;name where n is a line number. The semicolon followed by a routine name (name) specifies that line number n is a breakpoint only in the named program or subroutine.

## BASIC-PLUS-2 ON RSTS/E

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 semicolon or 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 subroutine.
2. Line 300 in the currently executing routine.
3. Line 310 in the routine 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 subroutine.

UNBREAK n;name    disables the breakpoint set at line number n in the named routine.

Note that, as in the BREAK command, you can specify a maximum of 10 breakpoints separated by commas or semicolons 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}
```

where:

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

DEF            causes a halt in execution each time a user-defined function is entered. The halt occurs immediately before the execution of the function.

LOOP           causes a halt in execution each time a FOR, WHILE, or UNTIL statement or modifier is encountered. Halts occur after the loop is initialized, immediately before execution of the loop body, and after exit from the loop.

## BASIC-PLUS-2 ON RSTS/E

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

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

```
# STEP [n]
```

where:

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

n           specifies the number of statements to be executed.

As with other debugging commands, the STEP command has effect only on programs or subroutines that are compiled with the /DEBUG switch. Therefore, the statement executed by the STEP command is the first statement encountered in a /DEBUG enabled routine.

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

**1.2.3.3 PRINT and LET Commands** - The PRINT and LET commands allow you to examine and change the contents of variables in programs and routines 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 and LET 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 PRINT and LET debugging commands allow constants or variables as arguments, but they do not allow expressions.

1.2.3.4 **TRACE and UNTRACE Commands** - The TRACE command allows you to track the execution of a program or subroutine 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
```

where:

```
TRACE          the command with no argument prints the number of
                each line as it is executed.
```

To disable the TRACE command, type UNTRACE in response to the # prompt.

### 1.3 BASIC-PLUS-2 PROGRAMS

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

```
line-number text (RET)
```

where the symbol (RET) represents a carriage return line terminator. In addition to carriage return, BASIC-PLUS-2 accepts a line feed, form feed, vertical tab, or escape 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:

```
?Syntax error
```

A line number with no text is considered to be a line deletion (see Section 1.2.2). Text with no line number (except for legal commands and continuation lines) is ignored and the system 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.

#### 1.3.1 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 means of 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:

```
!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 alphabetic in line text as upper case, but lower-case alphabetic 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  $.29 \times 10^{-38}$  to  $.17 \times 10^{39}$ . Double-precision (4-word) floating-point values are rounded down to 17 digits of accuracy and lie in the range of  $.29 \times 10^{-38}$  to  $.17 \times 10^{39}$ . For more information on data representation, see Appendix D.

### 1.3.2 BASIC-PLUS-2 RSTS/E Sample Program

The following example summarizes the building of BASIC source programs on the RSTS/E system.

# BASIC-PLUS-2 ON RSTS/E

RUN \$BASIC2

READY

NEW

NEW FILE NAME---SORT02

READY

```

10      DIM SORT(100)                ! MAX NUMBER OF ELEMENTS
20      INPUT "NUMBER OF ENTRIES"; CNTZ ! GET NUMBER OF ELEMENTS
\      IF CNTZ < 2% OR CNTZ > 100%      ! CHECK CORRECT NUMBER
\      THEN PRINT 'LIMITS - 2 TO 100'  ! WRONG - INFORM USER
\      GO TO 20                        ! TRY AGAIN
\      ELSE INPUT SORT(IZ) FOR IZ=1% TO CNTZ
30      REM
          B U B B L E   S O R T

          CHECK EACH PAIR OF ELEMENTS
          IF IN WRONG ORDER, SWITCH THEM
          SORT.FLG IS SET TO FALSE (0) WHEN A SWITCH IS MADE
          PASS OVER THE ENTIRE LIST UNTIL NO SWITCH IS MADE

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

```

SAVE

READY

COMPILE

READY

RUNNH SORT02

NUMBER OF ENTRIES? 6

? 0

? -5.5

? 10

? 20

? -5.6

? -100

20	10	0	-5.5	-5.6	-100
----	----	---	------	------	------

READY

BYE

CONFIRM: Y

SAVED ALL DISK FILES; 120 BLOCKS IN USE, 4880 FREE

JOB 9 USER 26,11 LOGGED OFF KB60 AT 23-APR-77 09:54 AM

SYSTEM RSTS V06B-02 TIMESHARING RJE

RUN TIME WAS 1 SECOND

ELAPSED TIME WAS 11 MINUTES, 29 SECONDS

GOOD MORNING

## BASIC-PLUS-2 ON RSTS/E

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

Command	Explanation
RUN \$BASIC2	The RUN \$BASIC2 command (see Section 1.2) invokes the BASIC-PLUS-2 compiler and run-time system.
NEW NEW FILE NAME--SORT02	The NEW command (see Section 1.2.1.9) 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 system's prompt (NEW FILE NAME--), you assign the name SORT02 to your program.
READY	READY is printed by the run-time system 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 1.2.1.14) copies and preserves the program on the system device. The program now resides on the system as a source program (extension .B2S) named SORT02.
COMPILE	The COMPILE command (see Section 1.2.1.3) translates the program into executable code. The default extension, .TSK, is appended to the program name.
RUNNH	The RUNNH command (see Section 1.2.1.13) causes the execution of the program.
BYE	The BYE command ends the session and causes you to exit from the system. You are asked to confirm that you wish to leave. If the answer is yes (Y), the system prints additional information and then logs you off (see Section 1.1).



## CHAPTER 2

### TASK BUILDER

The Task Builder is a system program that is used to link 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 `COMPILE /OBJ` command (see Section 1.2.1.3). Programs created as object modules have the `.OBJ` extension appended to the filename. 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 executable code. You must use the Task Builder if the program:

1. Contains a `CALL` statement that references an external subroutine.
2. Is to be linked to library modules.
3. Is to use a Task Builder option.
4. Is to create or access an RMS file.

#### 2.1 INVOKING THE TASK BUILDER

The Task Builder is a system-library program. You invoke it by typing on the terminal:

```
RUN $TKB
```

The Task Builder indicates that it has been successfully invoked and is ready to accept input by printing the following prompt on the terminal:

```
TKB>
```

Note that when you invoke a system program such as the Task Builder, you exit from the current run-time system. An exit from the system program clears memory and returns you to the default run-time system.

The system manager has the option of installing the Task Builder as a CCL command (see Section 1.1.2). If this is the case, you can type `TKB` to invoke the Task Builder. You should refer to the system manager for this capability.

## TASK BUILDER

### 2.1.1 Task Build Command Line

In response to the TKB> prompt, you can specify a command line that contains from zero to two output files and any number of input files, switches, and Task Builder options. The command line has the form:

```
TKB>task,map=input/sw
TKB>/
```

where:

task	is the first, optional output file specification and represents the task image file with a .TSK default extension.
map	is the second, optional output file specification, separated from the task by a comma. This specification represents the memory allocation map with a .MAP default extension.
input	is one or more object modules with a filename specification as shown in Section 1.1.1. Input is separated from output by an equal sign and each specified object module is separated by commas.
/sw	is one or more optional switches as described in Section 2.2.1.
/	is a slash that causes the Task Builder to solicit one or more options. (See Section 2.2.2.)

The Task Builder continues to prompt for input until you type a double slash (//). Note that the single slash to solicit options must be typed on a separate command line. When you type the double slash, the Task Builder builds the task image, outputs a task image file and a map (if these files are requested), and resolves any specified switches and options.

Consider the following example:

```
RUN $TKB

TKB>PROG,PROG=OBJ1,OBJ2,OBJ3,LIB/LB/sw
TKB>/
ENTER OPTIONS:
TKB> opt1=option
TKB> opt2=option
TKB> //

READY
```

In this example, the first output specification calls for the building of a task image file named PROG.TSK that contains the modules and options specified as input. The second output specification, PROG.MAP, calls for the building of a task memory allocation map that is stored in the specified file. Note that if no output specification is made, the Task Builder only performs an error check on the input specifications and issues appropriate diagnostic messages.

The input specification contains three object modules built by the compiler (COMPILE /OBJ). The modules are followed by library and optional switch specifications and a carriage return. The carriage return causes a continuation of the TKB> prompt. A single slash causes the Task Builder to prompt for system options by printing ENTER

## TASK BUILDER

**OPTIONS:** on the terminal followed by the TKB> prompt. In response, you type the desired option. Note that only one option can be entered on a line. Task build input is terminated by a double slash. This causes the creation of the specified output files and an exit from the system program.

The BUILD command (see Section 1.2.1.2) offers you an alternative procedure for specifying TKB 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 command input. For example:

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

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

```
TKB> @MOD1
```

the Task Builder generates a task image file (MOD1.TSK) and a map (MOD1.MAP). Note that you cannot use the unmodified output of the BUILD command when you desire Task Builder options. In these cases, you must specify the complete Task Builder command line or build your own command file with an editor.

For more information on the Task Builder specifications, refer to the RSTS/E Task Builder Reference Manual.

**2.1.1.1 Task File Extensions** - The files that are processed by the Task Builder are assigned file type extensions by default. Table 2-1 lists these extensions and the applicable file.

Table 2-1  
Default File Types

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

## 2.2 TASK BUILDER INPUT

Input to the Task Builder program consists of one or more object modules, any required libraries, optional switches, and program options. Note that the HISEG option is required in order to associate the task with the BASIC-PLUS-2 run-time system (see Section 2.2.2.6).

## TASK BUILDER

The object modules can be input as RSTS/E filename specifications (without RSTS/E switch specifications or wild card characters) or the filenames alone. When you type the complete filename specification, the Task Builder transfers any project-programmer number, device, protection code, and extension to the task image. If you specify the filename alone, the system defaults are used.

The switches and options also have default settings. In most cases you can change these by specifying the desired setting in the command line. The switches and options, their defaults, and functions are discussed in the following sections. For additional information on these specifications, refer to the RSTS/E Task Builder Reference Manual.

### 2.2.1 Switches

Switch action is specified by a slash followed by a 2-letter mnemonic that indicates the switch name. No specification is needed when switch action is the default; however, you can negate switch action by specifying the switch, preceded by a minus sign or NO. For example:

/SW specifies switch action  
/-SW negates switch action  
/NOSW negates switch action

Table 2-2 lists the switches that you can specify to the Task Builder. The table also shows the action caused by the switch, the file it applies to, and its default.

Table 2-2  
Task Builder Switches

Mnemonic	Meaning	File	Default
/DA	Task contains debugging aid	T,I	/-DA
/LB	Input file is a library file	I	/-LB
/MA	Includes a file in the map	M,I	/MA
/MP	Input file contains overlay description	I	/-MP
/SH	Short memory allocation map is output	M	/SH
/WI	Causes a wide listing	M	/WI
/XT:n	Task Builder exits after n diagnostics	T	/-XT

T-task image file

M-memory allocation map

S-symbol definition file

I-input file



## TASK BUILDER

When you specify a switch in the command line, it must be associated with the appropriate file. That is, a switch that applies to a task, map, or input file must appear with that file in the Task Builder command line. A switch that is associated with an inappropriate file causes an error (i.e., Illegal switch).

The /DA switch indicates that the task includes a debugging aid. If you specify this switch to an input file, a user-built program that controls execution is input to the task. If you apply this switch to the task image file, the Task Builder automatically includes the system debugging aid SY:[1,1]ODT.OBJ in the task image. The default does not include a debugging aid.

The /LB switch has two forms: without arguments (/LB) and with arguments (/LB:module-1:module-2:...module-8). If you do not specify arguments with this switch, the input file is assumed to be a library file containing modules that resolve undefined global references. The Task Builder searches the file and includes the global defining modules in the task image file. If you specify arguments with this switch, the input file is assumed to be a library file from which the named modules are taken and included in the task image. You can specify from one to eight modules as arguments to this switch. The default identifies input as an object module and not as a library file.

The /MA switch causes a module to be included in the memory allocation map. When you apply this switch to the input specification, the input file is included in the memory allocation map. The default includes the input file in the map. When you specify this switch in the output map file, any resident library symbol definitions are included in the map. For output files, the default negates this switch and excludes symbol definitions from the map.

The /MP switch indicates that the input file contains a description of the task's overlay structure. The Task Builder allocates memory as directed by the overlay description file (with a default extension .ODL). The default does not include an overlay description.

The /SH switch produces a short version of the memory allocation map file. That is, the "File Contents" portion of the file is not produced. The default produces the short version of the map file.

The /WI switch causes a wide 132-column listing of the memory allocation map. When you negate this switch, a narrow 80-column listing is produced. The default produces a wide listing.

The /XT:n switch causes the Task Builder to exit (cease execution) after n error diagnostics are produced. The specified number (n) can be decimal, octal, or default to 1. If decimal, n must be followed by a decimal point; if octal, n must be preceded by a number sign (#). The default (/XT) is not to exit.

### 2.2.2 Options

The options are specified on 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 a name or number. The name or number assigned to the option is dependent on the desired characteristic. The following sections describe each option and its use.

## TASK BUILDER

Options can be divided into five categories:

1. Control - affects the execution of the Task Builder.
2. Identification - specifies task characteristics.
3. Allocation - modifies the task's memory allocation.
4. Alter - adapts file contents to global symbol usage.
5. Device - specifies task device requirements.

Table 2-3 lists the option keywords, their meanings, and the categories to which they apply.

Table 2-3  
Task Builder Options

Keyword	Meaning	Category
ABORT	Directs the Task Builder to terminate execution	Control
ABSPAT	Declares absolute patch values	Alter
ASG	Declares device assignments to logical units	Device
EXTTSK	Extends the amount of memory allocated to a task	Allocation
GBLDEF	Declares a global symbol definition	Alter
HISEG	Associates task with a run-time system	Allocation
STACK	Declares stack size	Allocation
TASK	Declares task name	Identification
UNITS	Declares the maximum number of units	Device

The following sections describe the format and use of the options you can specify to the Task Builder. For more information on these options, refer to the RSTS/E Task Builder Reference Manual.

**2.2.2.1 ABORT Option** - The ABORT option causes the Task Builder to cease building the task image. After you type the option and a carriage return, the Task Builder stops accepting input and returns to input level for a new task build. Any previous input is ignored.

The ABORT option has the form:

ABORT=n

where n is any integer value. This value is used solely to satisfy the option format and has no other significance.

## TASK BUILDER

ABORT causes termination of task input and a new TKB> prompt; the terminated task is not built. To end input and build the task, type a double slash (//) as shown in Section 2.1.1.

**2.2.2.2 ABSPAT Option** - The ABSPAT option declares a series of patches, which start at a specified base address, and allows you to introduce corrective code into the task image. You can specify up to eight patch values with this option.

The ABSPAT option has the form:

ABSPAT=seg-name:address:val-1:val-2:...val-8

where:

seg-name	is the 1- to 6-character Radix-50 name of the program segment.
address	is the octal address of the first patch. The address can be on a byte boundary; however, two bytes are always modified for each patch.
val-1	is an octal number in the range of 0 to 177777 assigned to the address.
val-2	is an octal number in the range of 0 to 177777 assigned to the address +2.
val-8	is an octal number in the range of 0 to 177777 assigned to the address +20.

All patches must be within task and segment memory limits or a fatal error is generated.

**2.2.2.3 ASG Option** - The ASG option assigns a specified physical device to one or more logical units. You can specify a maximum of 12 logical units in this option.

The ASG option has the form:

ASG=device name:unit 1:...unit 12

where:

device name	is a 2-character alphabetic device name.
unit	is a decimal integer that indicates the logical unit number.

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

If your program requires the use of units five and six, you must override the ASG default with an explicit ASG specification. Also, when the UNITS option (see Section 2.2.2.9) and ASG are part of the same input specification, UNITS must precede ASG.

**2.2.2.4 EXTTSK Option** - 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.

## TASK BUILDER

ABORT causes termination of task input and a new TKB> prompt; the terminated task is not built. To end input and build the task, type a double slash (//) as shown in Section 2.1.1.

**2.2.2.2 ABSPAT Option** - The ABSPAT option declares a series of patches, which start at a specified base address, and allows you to introduce corrective code into the task image. You can specify up to eight patch values with this option.

The ABSPAT option has the form:

ABSPAT=seg-name:address:val-1:val-2:...val-8

where:

seg-name	is the 1- to 6-character Radix-50 name of the program segment.
address	is the octal address of the first patch. The address can be on a byte boundary; however, two bytes are always modified for each patch.
val-1	is an octal number in the range of 0 to 177777 assigned to the address.
val-2	is an octal number in the range of 0 to 177777 assigned to the address +2.
val-8	is an octal number in the range of 0 to 177777 assigned to the address +20.

All patches must be within task and segment memory limits or a fatal error is generated.

**2.2.2.3 ASG Option** - The ASG option assigns a specified physical device to one or more logical units. You can specify a maximum of 12 logical units in this option.

The ASG option has the form:

ASG=device name:unit 1:...unit 12

where:

device name	is a 2-character alphabetic device name.
unit	is a decimal integer that indicates the logical unit number.

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

If your program requires the use of units five and six, you must override the ASG default with an explicit ASG specification. Also, when the UNITS option (see Section 2.2.2.9) and ASG are part of the same input specification, UNITS must precede ASG.

**2.2.2.4 EXTTSK Option** - 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.

## TASK BUILDER

The amount of memory available to the task is 32K words minus the size of the run-time system. If the 16K BASIC2 run-time system is used, the amount of memory available is 16K. If the 4K BP2COM run-time system is used, the amount of available memory is 28K.

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 private maximum task size, the system maximum swap size, or the run-time system maximum allocation (i.e., 16K or 28K), 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.

**2.2.2.5 GBLDEF Option** - The GBLDEF option defines a global symbol. Once specified, the symbol definition is considered permanent for the task.

The GBLDEF option has the form:

GBLDEF=symbol name:symbol value

where:

symbol name            is the 1- to 6-character Radix-50 symbol name.

symbol value           is an octal number in the range of 0 to 177777 assigned to the symbol.

**2.2.2.6 HISEG Option** - The HISEG option associates the task with a specific high segment in memory (i.e., run-time system). BASIC-PLUS-2 programs require the BASIC2 or BP2COM run-time systems, therefore, you must specify HISEG=BASIC2 or HISEG=BP2COM in the task build command line. The Task Builder automatically includes the symbol definition file of the high segment in the input file. That is, specifying BASIC2 in the HISEG option associates the task image with the BASIC-PLUS-2 run-time system and includes BASIC2.STB (located in account [1,1]) in the input file. Note that the BUILD command automatically includes HISEG=BASIC2 as part of the generated command file.

The HISEG option has the form:

HISEG=run-time system

where run-time system is a specified high segment. The Task Builder default high segment is RSX.

**2.2.2.7 STACK Option** - The STACK option declares the maximum stack size required by the task. The default is 256 decimal words.

## TASK BUILDER

The STACK option has the form:

STACK=stack size

where stack size is a decimal integer specifying the stack size in words.

**2.2.2.8 TASK Option** - The TASK option specifies the name of the task. The default name is the first six characters of the task image file.

The TASK option has the form:

TASK=task name

where task name is a 1- to 6-character Radix-50 name. The use of this option has no effect on the name of the executable task image file.

**2.2.2.9 UNITS Option** - 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 12.

The UNITS option has the form:

UNITS=max-units

where max-units is a decimal integer in the range of 0 to 12.

## 2.3 TASK BUILDER OUTPUT

The Task Builder can output two types of files: a task image file and a memory allocation map.

If you specify the task image file as output, an executable file is constructed. If you do not specify the task image, the Task Builder does not construct an executable task image file. The Task Builder does, however, check the input for errors and print an appropriate message.

If you specify a memory allocation map, the Task Builder produces an ASCII file that contains information on the allocation of task memory. If no map is specified, memory is allocated for the task but no memory content file is produced.

Because you type output files in a specific order, a space must be assigned in the output command line when a preceding file is omitted. For example, to specify output of a map file but no task, the task specification position is left blank but is followed by a comma:

TKB> ,map=input

## TASK BUILDER

### 2.3.1 Listings

The following example illustrates the command format used to produce a task image and memory allocation map. The input file is a command file (generated with the BUILD command) that contains the SORT02 program shown in Section 1.3.2. The command format is as follows:

```
RUN $TKE
```

```
TKB>@SORT02
```

```
READY
```

These command lines produce an executable task image file named SORT02.TSK and a map named SORT02.MAP. These files are stored in your account. Figure 2-1 shows a listing of the map file that the Task Builder produces from the SORT02 program.

### 2.4 PROGRAM SEGMENTATION

Programs can be logically broken into sections (subprograms) that are compiled and input to the Task Builder as object modules. These sections can then be overlaid, which allows you to create programs that would otherwise exceed the maximum in-core program limits.

A program requires overlays when in-core program needs exceed the system default maximum job size. Because sections of code may share the same memory space, the Task Builder prevents these sections from calling (overlying) each other. This constraint must be taken into consideration when you design program overlays.

#### 2.4.1 Overlays

When you use the Task Builder overlay facility, you can specify only one input file in the command line. This input file describes the overlay structure, the location of program sections, and the loading procedures.

Overlay structure is defined by means of the Overlay Description Language (ODL). This structure is analogous to a tree, with the main program being the root and the program sections representing the branches. The ODL directives are contained in a user-created file that is specified in the command string. The /MP switch (see Section 2.2.1) must be appended to the file specification to identify the file as an ODL file.

Note that you can use the output produced by the BUILD command (see Section 1.2.1.2) to create overlaid program segments. That is, the BUILD command produces a command file that contains all of the input required by the Task Builder and an ODL file with an appended /MP switch. You can examine the ODL file and use an editor to modify its content before the command file is input to the Task Builder.

Overlay loading must be performed automatically (autoload). A complete description of the ODL directives and loading procedures is given in the RSTS/E Task Builder Reference Manual.

# TASK BUILDER

PARTITION NAME : GEN  
 IDENTIFICATION : V01X00  
 TASK UIC : [26,11]  
 STACK LIMITS: 001000 001777 001000 00512.  
 PRG XFR ADDRESS: 014732  
 TOTAL ADDRESS WINDOWS: 2.  
 TASK IMAGE SIZE : 4128. WORDS  
 TASK ADDRESS LIMITS: 000000 020003

\*\*\* ROOT SEGMENT: SORT02

R/W MEM LIMITS: 000000 020003 020004 08196.  
 DISK BLK LIMITS: 000002 000022 000021 00017.

## MEMORY ALLOCATION SYNOPSIS:

SECTION	TITLE	IDENT	FILE
. BLK: (RW,I,LCL,REL,CON)	002000 012732 05594.		
	002000 000126 00086.	\$SSFLT	BASIC2.OLB
	002126 000116 00078.	\$IOEND	BASIC2.OLB
	002244 001044 00548.	\$BL2	BASIC2.OLB
	003310 001662 00946.	\$IORED	BASIC2.OLB
	005172 001124 00596.	\$IOWRT	BASIC2.OLB
	006316 001446 00806.	\$INIT	BASIC2.OLB
	007764 000722 00466.	\$IOBLK	BASIC2.OLB
	010706 003012 01546.	\$IOPRG	BASIC2.OLB
	013720 000610 00392.	\$IOOPN	BASIC2.OLB
	014530 000102 00066.	\$FENTR	BASIC2.OLB
	014632 000100 00064.	\$ICON1	BASIC2.OLB
\$ARRAY: (RW,D,LCL,REL,CON)	014732 000000 00000.		
	014732 000000 00000.	SORT02 V01X00	SORT02.OBJ
\$CODE: (RW,I,LCL,REL,CON)	014732 000650 00424.		
	014732 000650 00424.	SORT02 V01X00	SORT02.OBJ
\$FLAGR: (RW,D,GBL,REL,CON)	015602 000000 00000.		
	015602 000000 00000.	SORT02 V01X00	SORT02.OBJ
\$FLAGS: (RW,D,GBL,REL,CON)	015602 000002 00002.		
	015602 000002 00002.	SORT02 V01X00	SORT02.OBJ
\$FLAGT: (RW,D,GBL,REL,CON)	015604 000000 00000.		
	015604 000000 00000.	SORT02 V01X00	SORT02.OBJ
\$IDATA: (RW,D,LCL,REL,CON)	015604 002074 01084.		
	015604 002074 01084.	SORT02 V01X00	SORT02.OBJ
\$PDATA: (RW,D,LCL,REL,CON)	017700 000102 00066.		
	017700 000102 00066.	SORT02 V01X00	SORT02.OBJ
\$SAVSP: (RW,D,LCL,REL,CON)	020002 000002 00002.		
	020002 000002 00002.	SORT02 V01X00	SORT02.OBJ
\$STRNF: (RW,D,GBL,REL,CON)	020004 000000 00000.		
	020004 000000 00000.	SORT02 V01X00	SORT02.OBJ
\$STRNG: (RW,D,GBL,REL,CON)	020004 000000 00000.		
	020004 000000 00000.	SORT02 V01X00	SORT02.OBJ
\$STRNH: (RW,D,GBL,REL,CON)	020004 000000 00000.		
	020004 000000 00000.	SORT02 V01X00	SORT02.OBJ
\$TDATA: (RW,D,LCL,REL,CON)	020004 000000 00000.		
	020004 000000 00000.	SORT02 V01X00	SORT02.OBJ

## GLOBAL SYMBOLS:

A1F\$M	002000-R	ERSTOP	002724-R	DEG\$	002510-R	V1S.M	002002-R
A1F.M	002002-R	FLN\$	002252-R	PU	000200	V2F\$M	002064-R
A1S\$M	002000-R	FLUSH	006104-R	PUGLG	000006	V2F.M	002066-R
A1S.M	002002-R	FLUSHN	006116-R	PUFMT	010732-R	V2S\$M	002012-R
A2F\$M	002012-R	FMTID	005716-R	PUFMTI	010706-R	V2S.M	002014-R
A2F.M	002014-R	FTIM	000100	PVD\$SI	005172-R	\$DOIT	002726-R
A2S\$M	002012-R	GTUVRD	010152-R	PVF\$SI	005312-R	\$D.END	014556-R
A2S.M	002014-R	GTUVRW	010140-R	PVI\$SI	005630-R	\$F.END	014574-R
BCL\$	007764-R	IMP\$	002340-R	PVS\$AI	005452-R	\$F.ENT	014530-R
BLG\$	007774-R	IVD\$A	003310-R	RCT\$	002412-R	\$IC	014632-R
BLP\$	010026-R	IVF\$A	003310-R	READ	010612-R	\$ID	014632-R
BOP\$	013720-R	IVI\$A	003450-R	RSM\$	002436-R	\$INITM	006316-R
BPC\$	010070-R	IVS\$A	003636-R	RSU\$	002446-R	\$INITS	007424-R
CAL\$	007500-R	LINSW	002244-R	SBE\$	007702-R	\$IR	014646-R
CDI\$	014632-R	LIN\$	002252-R	SETUP	006054-R	\$OTSVA	015644-R
CFI\$	014646-R	LYN\$	002366-R	STS\$	002424-R	\$POPR3	014624-R
CLOSE	014240-R	NGI\$MS	002360-R	SWF\$	002346-R	\$POPR4	014612-R
END\$	002126-R	NGI\$PS	002352-R	VREAD	010524-R	\$POPR5	014612-R
ERL\$	002366-R	NGI\$SS	002362-R	V1F\$M	002050-R		
ERRTRP	002556-R	NUMFLG	000400	V1F.M	002052-R		
ERR\$	002400-R	DEA\$	002522-R	V1S\$M	002000-R		

## \*\*\* TASK BUILDER STATISTICS:

TOTAL WORK FILE REFERENCES: 51459.  
 WORK FILE READS: 0.  
 WORK FILE WRITES: 0.  
 SIZE OF CORE POOL: 4436. WORDS (25. PAGES)  
 SIZE OF WORK FILE: 4352. WORDS (17. PAGES)  
 ELAPSED TIME: 00:00:32

Figure 2-1 Memory Allocation Map



## TASK BUILDER

At a minimum, the overlay description must contain a .ROOT and an .END directive. The .ROOT directive declares the overlay tree structure and the .END directive signifies the end of input. Note that an overlay description can contain only one .ROOT directive, which limits the tree structure declaration to a single line of input. The Task Builder truncates an input line that exceeds 80 characters, but this limitation should not effect the majority of tree structure declarations. However, you can use the .FCTR directive to build large trees and extend the description beyond a single line. For a description of the .FCTR directive, refer to the RSTS/E Task Builder Reference Manual.

Suppose, for example, you have a program consisting of a main program and calls to three external subprograms. One subprogram does pre-processing of data. The second does primary processing. The third subprogram does post-processing. The main program and three subprograms are compiled as object modules named MAIN.OBJ, PRE.OBJ, PROC.OBJ, and POST.OBJ, respectively.

You can build an overlay structure that causes the main program to be resident in memory and the three subprograms to share the same memory location. The ODL directive that creates this structure has the form:

```
.ROOT MAIN-*(PRE,PROC,POST)
.END
```

In this example:

.ROOT MAIN	defines the root of the overlay structure as the object module named MAIN.OBJ.
-	the hyphen indicates that the following modules are concatenated to the preceding module.
*	the asterisk indicates that modules are loaded automatically (autoload). The asterisk must precede every such module. If all modules within parentheses are to be autoload, a single asterisk preceding the parentheses is used.
( )	parentheses group the descriptions of overlay sections.
PRE,PROC,POST	commas separating object modules contained in parentheses indicate that the named modules occupy the same storage area.

Figure 2-2 is a graphic illustration of the overlay structure specified above as it would appear in memory.

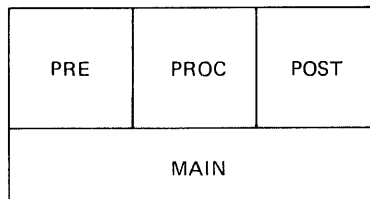


Figure 2-2 Overlay Structure

## TASK BUILDER

To create an overlaid program by means of the BUILD command, you edit the ODL file that is generated. That is, a BUILD command produces a command file (extension .CMD) and an overlay description language file (extension .ODL). The ODL file must be edited to reflect the desired overlay structure prior to input to the Task Builder.

For example, if the object modules described in Figure 2-2 (i.e., MAIN, PRE, PROC, and POST) are used as arguments in the BUILD command:

```
BUILD MAIN,PRE,PROC,POST
```

the result is a command file (MAIN.CMD) that appears as follows:

```
SY:MAIN,SY:MAIN=SY:MAIN/MF
HISEG=BASIC2
ASG = SY:5:6
//
```

The BUILD command also generates an overlay description file (MAIN.ODL) that appears as follows:

```
USER:      .ROOT  USER
USER:      .FCTR  MAIN-PRE-PROC-POST-LIBR
LIBR:      .FCTR  [1,1]BASIC2/LB
           .END
```

You can edit this ODL file to create an overlay as follows:

```
USER:      .ROOT  USER
USER:      .FCTR  MAIN-LIBR-*(PRE-LIBR,PROC-LIBR,POST-LIBR)
LIBR:      .FCTR  [1,1]BASIC2/LB
           .END
```

The overlay structure used in this example duplicates that shown in Figure 2-2. Note that each branch of the structure must be associated with the library. This procedure ensures that the correct routines are linked at run time.

The path of an overlay structure is from the root of the structure, along a series of branches, to the outermost section. The root section can call any overlay section. However, a subprogram in an overlay section can call another overlay section only if they share a common path. Therefore, in the previous example, MAIN can call PRE, PROC, and POST, but the three subprograms cannot call each other.

The concept of paths is better illustrated with a tree diagram. For example:

```
.ROOT A-B-*(C,D-(E,F,G))
.END
```

where A and B are two object modules representing root sections. C and D are the branches of A and B. E, F, and G are branches of D. A tree diagram of this structure appears in Figure 2-3.

## TASK BUILDER

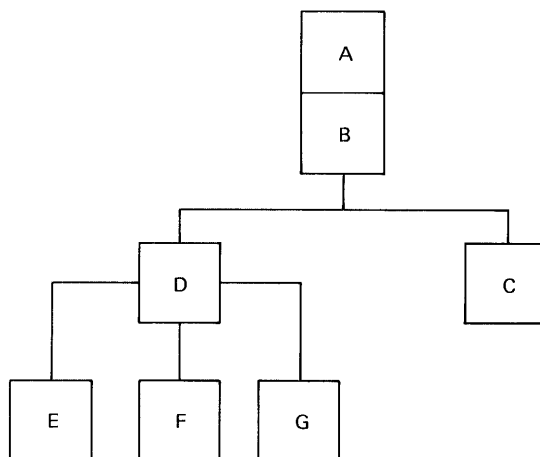


Figure 2-3 Overlay Path

The paths of this structure are: A-B-C, A-B-D-E, A-B-D-F, and A-B-D-G. Within this structure:

1. A and B can call all of the sections.
2. D can call E, F, and G.
3. C and D cannot directly call each other.
4. C cannot call E, F, and G.
5. E, F, and G cannot call each other.

Note that if A calls C, C in turn can call B. However, if B simultaneously calls D and then attempts to return to C, an error occurs. The error is due to B returning to an overlaid segment, i.e., D overlays C.

### 2.5 EXECUTING THE TASK

The Task Builder outputs executable code that can be invoked by means of the RUN command. The sequence of events leading up to task execution are:

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

## TASK BUILDER

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

```
RUN $TKB
```

```
TKB> MYPROG,MYPROG=MYPROG1,MYPROG2,C1,3JULIE/LB,C1,1JBASIC2/LB
TKB> /
ENTER OPTIONS:
TKB> HISEG=BASIC2
TKB> //
```

```
READY
```

These commands cause the output of a task image (MYPROG.TSK) and a memory allocation map (MYPROG.MAP). Input consists of two object modules (MYPROG1 and MYPROG2), a user-generated library, and a BASIC-PLUS-2 library. The library specifications contain the account number under which the library is stored, the library file specification, and the /LB switch. The HISEG option is used to associate the task with the BASIC-PLUS-2 run-time system.

```
OLD NONAME
```

```
READY
```

```
COM /OBJ
```

```
READY
```

```
BUILD NONAME/IND
```

```
READY
```

```
RUN $TKB
TKB>@NONAME
```

```
READY
```

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 required as input to the Task Builder as well as the switch (/IND) required to enable the use of RMS indexed I/O. The command file is used as input to the TKB prompt and the result is a map file and an executable task. Note that no additional switches or options can be associated with the command file input specification. For example, TKB>@NONAME/-WI is illegal. The use of an RMS switch (/SEQ, /REL, or /IND) causes the BUILD command to change the generated .ODL file as required for RMS Record 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 C1,1JBASIC2/LB
RMS:           .FCTR B10047
@SY:C1,1J      BASIC4
                .END
```



## CHAPTER 3

### RUN-TIME SYSTEMS AND LIBRARIES

BASIC-PLUS-2 on RSTS/E is distributed in the form of two separate run-time system and library combinations called BASIC2 and BP2COM, respectively. The BASIC2 run-time system is designed to be used with the BASIC2 library. The BP2COM run-time system is used with the BP2COM library. You can select the run-time system/library combination that is best suited to the needs of your program. This selection process is accomplished with the HISEG command (see Section 1.2.1.6).

The BASIC2 and BP2COM run-time system/library combinations both contain the same run-time support routines. These routines include:

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.

Note that BASIC2 contains all of these routines in the run-time system while BP2COM uses its associated library to contain most of the routines. This distinction is discussed in Section 3.2.

#### 3.1 BASIC-PLUS-2 RUN-TIME SYSTEMS

A run-time system (RTS) is a common area of code that can be shared by two or more user tasks. The code contained in the BASIC-PLUS-2 RTS serve the following purposes:

1. Provide an interface between your executable code and the monitor.
2. Control task execution when the monitor allows the task to run.
3. Contain the run-time support routines required by the executing task.

Because the RTS can be shared by multiple tasks, its use increases the efficiency of the task and the system. That is, each task need not contain its own copy of the common code.

## RUN-TIME SYSTEMS AND LIBRARIES

### 3.1.1 BASIC2 RTS

The BASIC2 run-time system is 16K words long. All of the shareable code required for BASIC-PLUS-2 run-time support routines is contained in the RTS. Because the maximum allowable memory for any task is 32K words, the use of BASIC2 RTS limits your program size to 16K words. Note that the BASIC2 RTS must be used if you specify the direct compilation of a task image file (i.e., COM/TSK).

The code required for RMS-structured I/O is not included in the RTS. To use RMS, you must append an /SEQ, /REL, or /IND switch in the BUILD command (see Section 1.2.1.2).

### 3.1.2 BP2COM RTS

The BP2COM run-time system is 4K words long. It contains some of the BASIC-PLUS-2 run-time support routines and uses the BP2COM library to contain the remaining routines. Because the BP2COM RTS uses 4K of the 32K allowable memory, your program's maximum size is 28K words.

When you create a task that uses BP2COM, the run-time support routines are selectively linked to your program from the BP2COM library. The Task Builder is used to perform the selection and linking process. Therefore, to use the BP2COM RTS and library, you must specify BP2COM in the HISEG command and then use the BUILD command to generate a command file for Task Builder input. For example:

```
HISEG
NAME---BP2COM
ACCOUNT--- (RIT)

READY

BUI NONAME

READY
```

This procedure results in a command file as follows:

```
NONAME,NONAME=NONAME/MP
HISEG=BP2COM
UNITS = 12
ASG = SY:5:6:7:8:9:10:11:12
//
```

This command file is specified as input to the Task Builder. When the Task Builder processes the file, it selects and links the required run-time support routines to your task from the BP2COM library. The use of the BUILD and HISEG commands, which make the BP2COM RTS and library available to the Task Builder, are described in Sections 1.2.1.2 (BUILD) and 1.2.1.6 (HISEG).

## 3.2 BASIC-PLUS-2 LIBRARIES

A library is a collection of modules that can be selectively linked to your program by the Task Builder. The Task Builder links only those modules required for program execution and omits unnecessary routines from the task image. This process conserves memory space.

## RUN-TIME SYSTEMS AND LIBRARIES

BASIC-PLUS-2 allows you to write your own subroutines and add them as modules to the BASIC2, BP2COM, or user libraries. These subroutines can be written in the BASIC-PLUS-2 language or in RSX-11 MACRO assembly language. The following sections describe the BASIC2 and BP2COM libraries and the use of the Librarian Utility Program (LBR) to add subroutines (modules) to the libraries. For information on writing BASIC-PLUS-2 subroutines, refer to the BASIC-PLUS-2 Language Manual.

As distributed, the BASIC2 library contains a minimal amount of code. However, when the Task Builder is used with the BASIC2 RTS, this library must be used to fulfill format requirements.

The BP2COM library is distributed with all of the run-time support routines not contained in the BP2COM RTS.

### 3.2.1 Librarian Utility Program

The Librarian Utility Program (LBR) allows you to create, update, list, and maintain BASIC-PLUS-2 object code library files.

To invoke the Librarian, type:

```
RUN $LBR
```

If the invocation is successful, the Librarian issues an input prompt as follows:

```
LBR>
```

In response to the prompt, you type a command string as follows:

```
LBR>outfile=infile
```

where outfile and infile are RSTS/E file specifications. These specifications represent the user-written subroutines (infile) and the library file generated by LBR (outfile). The Librarian places the following restrictions on file specifications:

1. The specification can contain only physical device names; logical device names are not allowed.
2. A project, programmer number, if specified, must follow the filename and extension and must be enclosed in brackets.
3. The filename must be explicit.
4. No RSTS/E switches can be appended to the filename specification.

To terminate the Librarian program, type CTRL/Z in response to the prompt:

```
LBR>^Z  
READY
```

The library files that you create with LBR consist of a header, an entry point table (EPT), a module name table (MNT), the library modules, and free space.

The library header describes the current status of the library. When the file is modified, LBR automatically updates the header. This



## RUN-TIME SYSTEMS AND LIBRARIES

allows LBR to use the header to access the information it needs to perform its functions.

The entry point table consists of elements, each of which contain an entry point name and a pointer to the module header that defines the entry point. The EPT is alphabetically ordered and is searched by LBR when a library module is referenced by one of its entry points.

The module name table also consists of elements, each of which contain a module name and a pointer to the module header. The MNT is alphabetically ordered and is searched by LBR when a library module is referenced by its module name, rather than by entry point.

LBR uses switches to perform various functions on the library. These switches and their use are described in the following sections. The switches described here are a summary of the LBR function set contained in the RSX-11 Utilities Procedures Manual. The switches contained in the following sections are used to create a library file (/CR), list the modules in the file (/LI, /LE, and /FU), modify a module (/EX and /RP), insert a module (/IN), and edit the file (/DE and /CO).

**3.2.1.1 Create Switch (/CR)** - The create switch is used to allocate a library file on a direct access device such as a disk. It initializes the library file header, entry point table, and module name table. You can append this switch only to the output file, as follows:

```
LBR>outfile/CR:size:ept:mnt
```

where:

outfile	is the file specification of the library file being created. The default file extension is .OLB if you are creating an object library (BASIC-PLUS-2).
/CR	is the create switch.
:size	is the size of the library file in 256-word blocks. The default size is 100 decimal blocks.
:ept	is the number of entry point tables to allocate. The default for object modules is 512 decimal. The maximum number of entries is 4096.
:mnt	is the number of module name table entries to allocate. The default value is 256 decimal and the maximum is 4096.

The values that you specify in the command line must be multiples of 64. If the specifications are not a multiple of 64, the EPT or MNT expands to the next block boundary.

Consider the following example:

```
LBR>ACTLIB/CR: :128.:64.:OBJ
```

In this example, you create a library file named ACTLIB.OLB in the default directory on SY0:. ACTLIB has the following attributes:

size	100 decimal blocks (default)
EPT	128 decimal entry points

## RUN-TIME SYSTEMS AND LIBRARIES

MNT            64 decimal module names  
type           .OBJ

**3.2.1.2 Insert Switch (/IN)** - The insert switch is used to insert object code or MACRO modules into a library file. You can specify any number of input files and these can contain any number of concatenated input modules. This switch is the default library file option and can only be appended to the library file specification (outfile).

If you attempt to insert a module that already exists in the library, the operation terminates and the following error message prints on the terminal:

```
LBR--*FATAL*--DUPLICATE MODULE NAME "name" IN filename
```

Also, if you attempt to insert a module whose entry point duplicates one that is in the EPT, the operation terminates and the following error message prints on the terminal:

```
LBR--*FATAL*--DUPLICATE ENTRY POINT "name" IN filename
```

The insert switch has the following format:

```
LBR>outfile/IN=infile 1,...,infile n
```

where:

outfile    is the file specification of the library that is to contain the inserted input modules. The file extension is .OLB if the default is an object library.

/IN        is the insert switch.

infile    is the file specification of the input file that contains the modules to be inserted in the library file (outfile). If outfile is an object library, the input file default extension is .OBJ.

Consider the following example:

```
LBR>ACTLIB/IN=DEBIT,INV,TOTAL
```

In this example, the modules contained in the input files (DEBIT, INV, and TOTAL) are inserted into the library file named ACTLIB. All of these files reside in the default directory on SY:. The input files have an .OBJ extension.

**3.2.1.3 Extract (/EX) and Replace (/RP) Switches** - The extract and replace switches each perform unique functions, but you can combine these functions to modify and update existing library file modules. The extract switch is used to read one or more modules from the library file and write them into an output file. Once in the output file, you can modify or patch the modules to reflect any desired changes.

The replace switch is used to replace existing modules in the library file with modified modules from the output file.

The extract switch has the following format:

```
LBR>outfile=infile/EX[:module name...:module name]
```

## RUN-TIME SYSTEMS AND LIBRARIES

where:

**outfile** is the file specification of the file that is to contain the extracted modules. The file extension defaults to the extension of the modules. The modules are automatically concatenated when they are extracted.

**infile** specifies the library file from which you desire to extract modules. The default extension depends on the current library file extension.

**/EX** is the extract switch.

**:module name** is the name(s) of the module(s) you wish to extract. Up to eight modules can be specified. If no module names are specified, all modules in the library file are extracted.

The extract operation has no physical effect on the library file. That is, the modules are copied into the output file; the library file remains intact.

The replace switch replaces modules in a library file with input modules of the same name. When a match occurs on a module name during a replace, the operation deletes the existing module and inserts the input module. As each module is replaced, the following message prints on the terminal:

MODULE "name" REPLACED

If a match is not made, LBR assumes that the input module is new and inserts it in the library. In this case, no message is printed.

The replace switch can be specified in one of two formats:

LBR>outfile/RP=infile 1[,infile 2...,infile n]

This format is global and specifies that all of the input files (infile) contain replacement modules.

LBR>outfile=infile 1[/RP][,infile n[/RP]]

This format is local and specifies that only those input files to which an /RP switch is appended contain replacement modules.

In both the global and local formats:

**outfile** is the library file specification.

**/RP** is the replace switch. When appended to the library file (outfile), the switch is global. When appended to a specific input file (infile), the switch is local for that file.

**infile** is the input file specification that contains the modules to be inserted or replaced.

Note that the global replace switch can be used even when one or more of the input files does not contain replacement modules. If you append a /-RP or /NORP switch to the non-replacement module file, LBR overrides the global /RP switch for that particular file.

## RUN-TIME SYSTEMS AND LIBRARIES

Consider the following examples of the extract and replace switches:

```
LBR>ACT.OBJ=DEBIT/EX:COM
```

This example extracts the object module COM from the file DEBIT.OBJ and places it in the file ACT.OBJ. Note that COM is only copied into ACT.OBJ; the object module still exists in DEBIT.OBJ. After you have modified COM, you can use the replace switch to update the file DEBIT.OBJ:

```
LBR>DEBIT/RP=ACT
```

This example replaces the module in DEBIT with the module of the same name contained in ACT, i.e., COM.

**3.2.1.4 List Switch (/LI, /LE, and /FU)** - The list switch is used to produce a printed listing of a library file's content. The switch can be appended only to the output file specification or to a list file if one is specified. If the list file is not specified, the contents of the library file are printed to the initiating terminal.

There are three types of listings you can obtain, depending on the switch that you use:

/LI	lists the names of all modules in the library file.
/LE	lists the names of all modules in the library file and the module entry points.
/FU	lists the module names and a description of the size, insertion date, and module dependent information for each module.

Note that these switches can be combined, for example:

```
LBR>DK1:[200,200]ACT,TT:/LE/FU
```

This example causes a list of module names, their entry points, and a description of each module to print on the current terminal. The modules listed reside in the file ACT on DK1: in account directory [200,200].

The following example is a listing obtained with the /FU switch. The library filename is ACTLIB.OLB and it contains a module named DEBIT.OBJ.

```
RUN $LBR
LBR>ACTLIB/FU
```

```
DIRECTORY OF FILE ACTLIB.OLB
OBJECT MODULE LIBRARY CREATED BY: LBR VX04.02
LAST INSERT OCCURRED 4-MAY-77 AT 10:02:25
MNT ENTRIES ALLOCATED: 64; AVAILABLE: 63
EPT ENTRIES ALLOCATED: 128; AVAILABLE: 127
FILE SPACE AVAILABLE: 24165 WORDS
```

```
DEBIT  SIZE:00411  INSERTED:4-MAY-77  IDENT:V01X02
```

```
LBR>^Z
```

```
READY
```

## RUN-TIME SYSTEMS AND LIBRARIES

**3.2.1.5 Delete (/DE) and Compress (/CO) Switches** - The delete switch is used to delete modules and their associated entry points from a library file. A maximum of 15 modules can be specified in a single delete switch command line.

When LBR begins to process the deletions, it prints:

MODULES DELETED:

As the modules are deleted, their names are printed on the terminal.

If a specified module is not found in the library file, the deletion process terminates and the following message is printed:

LBR--\*FATAL\*--NO MODULE NAMED "name"

Note that the delete switch makes the specified module inaccessible to you, but it does not physically remove it from the file. To reclaim the space caused by the deletion of the module(s) from the file, you must use the compress switch.

The delete switch has the form:

LBR>outfile/DE:module 1[:module 2...:module n]

where:

outfile is the library file specification.

/DE is the delete switch.

:module is the name of the module(s) to be deleted.

The compress switch physically removes all logically deleted modules from the file, puts all free space at the end of the file, and makes space available for the insertion of new modules.

A compress operation does not delete the old library file; rather, it creates a compressed copy of the file. For this reason, you must specify a unique output file name. The format of the /CO switch is similar to the create switch. The format is as follows:

LBR>outfile/CO:size:ept:mnt=infile

where:

outfile is the file specification of the file that is to be the new, compressed version of the input file. The name of the new file must differ from the name of the input file. The default extension is .OLB if the input file is an object library.

/CO is the compress switch and can be appended only to the output file.

:size is the size of the new library file in 256-word blocks. If the size is omitted, the default is the size of the old library file (infile).

## RUN-TIME SYSTEMS AND LIBRARIES

:ept is the number of entry point tables to allocate. If the specified value is not a multiple of 64 decimal, the next highest multiple of 64 is used. The maximum number of entries is 4096 decimal. The default is the number of EPT's in the old library file.

:mnt is the number of module name table entries to allocate. If the specified value is not a multiple of 64 decimal, the next highest multiple of 64 is used. The maximum number of entries is 4096 decimal. The default is the number of MNT's in the old library file.

infile is the file specification of the library file to be compressed. The default file extension is .OLB for object libraries.

Consider the following example:

```
RUN $LBR
LBR>ACTLIB/FU
```

```
DIRECTORY OF FILE ACTLIB.OLB
OBJECT MODULE LIBRARY CREATED BY: LBR VX04.02
LAST INSERT OCCURRED 4-MAY-77 AT 16:32:55
MNT ENTRIES ALLOCATED: 64; AVAILABLE: 61
EPT ENTRIES ALLOCATED: 128; AVAILABLE: 125
FILE SPACE AVAILABLE: 23601 WORDS
```

```
DEBIT  SIZE:00411  INSERTED:4-MAY-77  IDENT:V01X02
SB1    SIZE:00282  INSERTED:4-MAY-77  IDENT:V01X02
SB2    SIZE:00282  INSERTED:4-MAY-77  IDENT:V01X02
```

```
LBR>NEWACT/CO:50.:192.:128.=ACTLIB
LBR>NEWACT/FU
```

```
DIRECTORY OF FILE NEWACT.OLB
OBJECT MODULE LIBRARY CREATED BY: LBR VX04.02
LAST INSERT OCCURRED 4-MAY-77 AT 16:32:55
MNT ENTRIES ALLOCATED: 128; AVAILABLE: 125
EPT ENTRIES ALLOCATED: 192; AVAILABLE: 189
FILE SPACE AVAILABLE: 10289 WORDS
```

```
DEBIT  SIZE:00411  INSERTED:4-MAY-77  IDENT:V01X02
SB1    SIZE:00282  INSERTED:4-MAY-77  IDENT:V01X02
SB2    SIZE:00282  INSERTED:4-MAY-77  IDENT:V01X02
```

```
LBR>TZ
```

In this example, a full listing of the file ACTLIB is shown. This file is then compressed and renamed to NEWACT. The compress switch command line is also used to change the size, EPT, and MNT values. These changes are reflected in the full listing shown for the new file NEWACT. Note that the old file, ACTLIB, still exists.

## 3.3 MACRO SUBROUTINES

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

MACRO subroutines on a BASIC-PLUS-2 system are subject to the following restrictions:

1. MACRO subroutines cannot call BASIC-PLUS-2 subroutines, perform input/output operations, or execute monitor operations.
2. Virtual arrays cannot be passed to MACRO subroutines.
3. MACRO subroutines cannot create strings.

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

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

1. BASIC-PLUS-2 can call a BASIC-PLUS-2 subroutine with a CALL statement. BASIC-PLUS-2 can call a MACRO subroutine with a CALL or CALL BY REF statement.
2. BASIC-PLUS-2 can call a MACRO subroutine that is also callable from FORTRAN with a CALL BY REF statement.
3. BASIC-PLUS-2 cannot call a FORTRAN subroutine.
4. BASIC-PLUS-2 cannot be called by a MACRO or FORTRAN subroutine.

### 3.3.1 Subroutine Linkage

BASIC-PLUS-2 programs call MACRO subroutines with the following instruction:

```
JSR    PC,routine
```

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

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

```
RTS    PC
```

Where RTS is the Return from Subroutine instruction.

Arguments are passed from BASIC-PLUS-2 programs to MACRO subroutines in the form of an argument list. When the MACRO subroutine starts, register 5 (R5) contains the address of an argument list as shown in Figure 3-1.

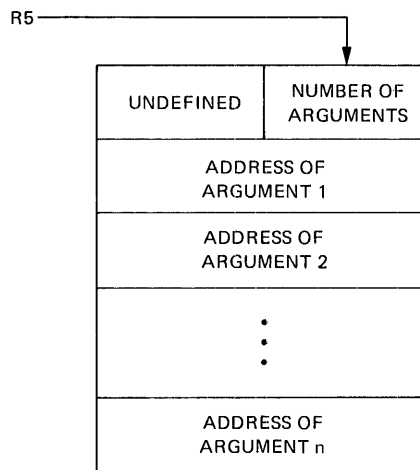


Figure 3-1 Argument List Format

### 3.3.2 Subroutine Register Usage

A MACRO subroutine 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 subroutine. That is, each "push" onto the stack must be matched by a "pop" from the stack before the subroutine returns control to the BASIC-PLUS-2 program.

### 3.3.3 Subroutine Calls

Arguments are passed to a MACRO subroutine by means of 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.



## RUN-TIME SYSTEMS AND LIBRARIES

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.

Consider Figures 3-2 and 3-3. These figures are examples of two MACRO subroutines and illustrate the methods used to pass arguments to subroutines. Figure 3-2 is an example of the use of the CALL statement. Figure 3-3 is an example of the use of the CALL BY REF 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
;
; EFFECTS: THIS SUBROUTINE 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::
    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 OF 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 3-2 CALL Statement

# RUN-TIME SYSTEMS AND LIBRARIES

```

.TITLE INSRT
;
;          CALL INSRT BY REF(A$,LEN(A$),B$,LEN(B$),CZ)
;
; INPUTS:  ARG1 = ADDRESS OF A$
;          ARG2 = ADDRESS OF LENGTH OF A$
;          ARG3 = ADDRESS OF B$
;          ARG4 = ADDRESS OF LENGTH OF B$
;          ARG5 = ADDRESS OF CZ
;
; OUTPUTS: CZ = 0 IF OPERATION WAS SUCCESSFUL
;          = -1 IF OPERATION FAILED
; EFFECTS: THIS SUBROUTINE OVERWRITES THE SUBSTRING B$ INTO THE
;          STRING A$ BEGINNING AT CHARACTER POSITION CZ.
;          RETURNS 0 IN CZ IF THE OPERATION WAS SUCCESSFUL.
;          RETURNS -1 IN CZ IF THE OPERATION FAILED.
;
INSRT::
    MOV     @12(R5),R2          ; R2 = CZ
    BLE     ERREX               ; BR TO ERROR IF CZ <= 0
    ADD     @10(R5),R2          ; R2 = CZ 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 = CZ
    DEC     R2                  ; R2 = CZ MINUS ONE
    ADD     R2,R0               ; R0 = ADDRESS OF A$ PLUS CZ
    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 CZ TO 0 (OPERATION SUCCESSFUL)
    RETURN

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

```

Figure 3-3 CALL BY REF Statement



## CHAPTER 4

### FILES

You can perform efficient input/output operations on large amounts of related data by collecting that data into files. BASIC-PLUS-2 Record Management Services (RMS) can increase this efficiency by allowing you to organize the 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 files and their records on BASIC-PLUS-2. This chapter describes RSTS/E block I/O, the use of RMS, the file organizations available under RMS, and the operations allowed on each type of organization. The chapter also contains a summary of the RMS utilities.

For additional information on the BASIC-PLUS-2 syntax used to create and manipulate files, refer to the BASIC-PLUS-2 Language Manual. For information on the RMS utilities, refer to the RSTS/E RMS-11 Utilities Manual.

#### 4.1 FILE CREATION

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 organization; 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] AS FILE #num-exp
                   { SEQUENTIAL
                   , [ORGANIZATION] { RELATIVE
                                     { INDEXED
                                     { VIRTUAL
                                     }
                                     }
                                     }
                   [,attributes]
```

where:

filename is a RSTS/E filename specification as shown in Section 1.1.1. Note that RSTS/E switch options are not allowed.

FOR OUTPUT indicates the creation of a new file.

## FILES

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 four 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).

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 contain one or more virtual arrays. This file organization permits RSTS/E file handling and block I/O operations, but it does not allow RMS record operations.

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 magtape, or unit record devices (e.g., line printers, terminals, etc.).

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. You can access a record sequentially or by reference to a key. Indexed files are allowed only on disk media.

Note that BASIC also allows you to create a terminal-format file. A terminal-format file is a collection of ASCII characters stored in lines of varying length. The length of the line is determined by the presence of a line terminator. Information in a terminal-format file is accessed sequentially. To create a terminal-format file, use the OPEN statement with no ORGANIZATION specification. For information on terminal-format files, refer to Section 4.4.3 and to the BASIC-PLUS-2 Language Manual.

### 4.1.1 Virtual Files

The virtual file organization specifies a file that is compatible with RSTS/E BASIC-PLUS. Virtual files contain binary data that is stored and accessed in the manner of elements in a virtual array. With the virtual organization, you can create and use a block I/O structured file that supports RSTS/E file handling statements such as FIELD, LSET, RSET, and CVT functions.

Virtual file organization is the default on RSTS/E and, because it is similar to an array in its method of storing data, it must be

## FILES

dimensioned with a DIM # statement. This statement is described in the BASIC-PLUS-2 Language Manual.

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)>]

[,RECORDSIZE<num-exp>]

where:

,ORGANIZATION VIRTUAL

specifies the creation or access of a virtual file and allows the use of RSTS/E block I/O. Virtual is the RSTS/E default file organization. The ORGANIZATION keyword is optional.

,ACCESS specifies the operations that the current user will perform on the file. MODIFY is the default. Refer to Section 4.2.4.

,ALLOW specifies the operations that the current user will permit other programs to perform on the file. READ is the default. Refer to Section 4.2.4.

,MAP references a MAP statement and can be used to define record size (see Section 4.6). Note that MAP must not 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 4.5.

When you specify a RECORDSIZE that exceeds the default minimum of 512 bytes, the specification must be a multiple of 518. During I/O operations on a virtual file, 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.

The virtual organization allows block I/O file operations such as CVT, LSET, RSET, and FIELD, 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
```

## FILES

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 attribute to MODIFY and the ALLOW attribute to NONE. Note that ALLOW NONE is the equivalent of ALLOW READ (see Section 4.2.4).

### 4.2 INTRODUCTION TO RMS

Record Management Services (RMS) is a set of system library routines. These routines effect the transmission of data between files and BASIC programs. Files are composed of records that act as storage and transmission 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 stream-format records.
4. Data structure - Data is maintained in records that are contained on 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 4-1 illustrates the record access methods and operation types allowed on each file organization.

## FILES

Table 4-1  
Comparison of File Organizations

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

The following subsections describe each file organization in detail.

### 4.2.1 Sequential Files

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

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

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

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

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

```
[,MAP<(map-name)>]
```

```
[,RECORDSIZE<num-exp>]
```

```
[,NOSPAN]
```

```
[,SPAN]
```

```
[,CLUSTER SIZE<num-exp>]
```

```
[,BLOCKSIZE<num-exp>]
```

```
[,CONTIGUOUS]
```

```
[,NOREWIND]
```



## FILES

where:

### ,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. Refer to Section 4.4.

### ,ACCESS

specifies the operations that the current user will perform on the file. MODIFY is the default. Refer to Section 4.2.4.

### ,ALLOW

specifies the operations that the current user 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 4.2.4.

### ,MAP

references a MAP statement that can be used to define record size. Refer to Section 4.6.

### ,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. Refer to Section 4.5.

,NOSPAN  
,SPAN

SPAN is the default and allows records to cross block boundaries. Refer to Section 4.5.1.

### ,CLUSTERSIZE

specifies a contiguous unit of blocks on disk devices. Refer to Section 4.5.1 and to the RSTS/E System User's Guide.

### ,BLOCKSIZE

specifies the number of records contained in a block on magnetic tape. Refer to Section 4.5.1.

### ,CONTIGUOUS

specifies that the contents of the file are contiguous on disk devices.

## FILES

### ,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.

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, RECORDSIZE 58%, 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 4.6). The RECORDSIZE attribute defines the maximum record size for this file as 58 bytes. The NOSPAN attribute overrides the SPAN default and prohibits records from crossing block boundaries (see Section 4.5).

A sequentially organized file maintains a strict relationship among the records on the file. The file is structured such that each record you write to the file physically follows the record that precedes it. Therefore, 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 first 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.
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 located by a GET or FIND before the UPDATE is made.

## FILES

4. SCRATCH operations erase the contents of the file beginning at the program's current file position up to the end of the file. The current file position is established as the end of the file. Exclusive file access is required for SCRATCH operations. If an erasure of the entire file is desired, 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.

### 4.2.2 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.

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>]

[,CLUSTERSIZE<num-exp>]

[,BUCKETSIZE<num-exp>]

where:

,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 4.4.

,ACCESS

specifies the operations that the current user will perform on the file. MODIFY is the default. Refer to Section 4.2.4.

## FILES

### ,ALLOW

specifies the operations that the current user will permit other programs to perform on the file. READ is the default. Refer to Section 4.2.4.

### ,MAP

references a MAP statement and can be used to define record size. Refer to Section 4.6.

### ,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 4.5.

### ,CLUSTERSIZE

specifies a contiguous unit of blocks on disk devices.

### ,BUCKETSIZE

specifies the size of a bucket in terms of the number of records. Refer to Section 4.5.2.

Consider the following example:

```
10      OPEN "RMSIVX.FIX" FOR OUTPUT AS FILE #3,      &
      ORGANIZATION RELATIVE FIXED, ACCESS             &
      MODIFY, ALLOW NONE, MAP MAP1,                   &
      RECORDSIZE 58%
```

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 4.2.4). 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 4.6). Because the file contains fixed-length records, the RECORDSIZE attribute defines the size of each record in the file as 58 bytes.

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 two 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

## FILES

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. 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 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.

### 4.2.3 Indexed Files

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

## FILES

[,ORGANIZATION] INDEXED[{FIXED  
VARIABLE}]

[,ACCESS{READ  
WRITE  
MODIFY}]

[,ALLOW{NONE  
READ  
WRITE  
MODIFY}]

[,MAP<(map-name)>]

[,RECORDSIZE<num-exp>]

[,CLUSTERSIZE]

[,BUCKETSIZE<num-exp>]

,PRIMARY[KEY]<name>

[,ALTERNATE[KEY]name]

[NODUPPLICATES NOCHANGES]

[DUPLICATES CHANGES]

where:

,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  
4.4.

,ACCESS

specifies the operations that the current user will  
perform on the file. MODIFY is the default. Refer to  
Section 4.2.4.

,ALLOW

specifies the operations that the current user will  
permit other programs to perform on the file. READ is  
the default. Refer to Section 4.2.4.

,MAP

references a MAP statement and can be used to define  
record size. Refer to Section 4.6.

## FILES

### ,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 4.5.

### ,CLUSTERSIZE

specifies a contiguous unit of blocks on disk devices.

### ,BUCKETSIZE

specifies the size of a bucket in terms of the number of records. Refer to Section 4.5.2.

### ,PRIMARY

defines the primary key for a particular record. This attribute is required. Refer to Section 4.2.3.1.

### ,ALTERNATE

allows you to define up to 254 alternate keys. This attribute is optional. Refer to Section 4.2.3.1.

### NODUPLICATES DUPLICATES

specifies the use of a duplicate key in the file. NODUPLICATES is the default. Refer to Section 4.2.3.1.

### NOCHANGES CHANGES

specifies the use of a key field change in the file. NOCHANGES is the default. Refer to Section 4.2.3.1.

Consider the following example:

```
10      OPEN "RMSIXV.VAR" FOR OUTPUT AS FILE #3  &  
        ORGANIZATION INDEXED VARIABLE, ACCESS  &  
        MODIFY, ALLOW NONE, MAP MAP1,          &  
        RECORDSIZE 58Z, PRIMARY NAME$  
  
20      MAP (MAP1) NAME$=30Z,IDZ,HRWAGE,FILL,FILLZ,FILL$
```

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 4.2.4). The OPEN statement also contains a map attribute that references a MAP statement named MAP1. The MAP statement (line 20) defines the content of records in the file (see Section 4.6). 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 RECORDSIZE attribute in the OPEN statement defines the size of the file's largest record as 58 bytes. The PRIMARY attribute associates the primary index key with NAME\$, which is defined in the MAP statement on line 20.

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

## FILES

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 record as a whole.

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. RMS stores the defined key in a primary index table along with a pointer to the location of the record. To access the record, you provide the BASIC program with a key number and key value that reference the index and key, respectively. RMS searches the index for that key, finds the value, and uses the pointer to locate the record.

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.

Like 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 index table to search. UPDATE, PUT, and DELETE operations do not require a key of reference specification.

Every index table contains a series of entries composed of key data fields copied from records. The table also contains pointers associated with each key that indicate the records' location in the file. When a new record is written in the file (PUT operation), its key values are placed in the appropriate indices. When a random GET operation is specified, a key number is included in the statement. The number causes RMS to search a specified index table and locate the



## FILES

record pointer. Because the index table is maintained in sequential order (understood by RMS), 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.

**4.2.3.1 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 arranges primary key values in index tables by ASCII collating sequence. Alternate keys are also arranged in individual tables by ASCII sequence except where duplicate keys are present. Duplicate keys are arranged in tables according to the order that the records were input to the file.

RMS allows you to 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 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. When a key changes, RMS automatically updates the appropriate index by replacing the old value with the modified version. 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 rel 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 define rel as 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
```

## FILES

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.

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.

If you specify:

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

where:

#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.

### 4.2.4 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 writer. Relative and indexed files can be shared by multiple readers and multiple writers.

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

## FILES

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 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 4.5.)

You can explicitly disable the locking mechanism by specifying an UNLOCK statement. For example:

```
70      UNLOCK #1%
```

causes the records in the specified file 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
```

## FILES

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.

If your program creates a relative or indexed file that is to be extended during the life of that program, READ must be specified in the ALLOW attribute. Because shared files cannot be extended, an error is generated when a PUT operation attempts to extend a file that contains a WRITE or MODIFY specification in the ALLOW attribute. The error is a protection violation. Note that if you specify FOR OUTPUT in the OPEN statement, the file is created with the ALLOW READ default specification.

### 4.2.5 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. Space is initially allocated when a file organization is specified and additional space is allocated at run time for each channel that the program opens.

The space that is initially allocated is as follows:

Sequential files - 3.9K words

Relative files - 4.0K words

Indexed files - 5.4K words

All file organizations - 6.0K words

Note that this space is allocated when the organization first appears in the program and not for each open file.

In addition, each open file in the program is allocated space as determined by the following algorithms:

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

### 4.3 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.

## FILES

RMS allows you to specify two types of record access; sequential and random. If you specify sequential access, records are accessed in serial order as established by the file organization. If you specify random access, record operations can take place at any point in the file.

Table 4-2 shows the relationship between file organization and record access.

Table 4-2  
Access Methods

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

The following subsections discuss each type of record access.

### 4.3.1 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.

Sequentially organized files allow 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). 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.

## FILES

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 are creating a new file or appending 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.

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.

### 4.3.2 Random Access

Random access allows the BASIC program, rather than file organization, to control the order of record access. The predecessor-successor relationship has no effect on random 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 physical 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. If the operation is a PUT and a record already exists in the specified location, RMS also returns an error.

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.).

## FILES

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. But 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 that indicates 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 generic key in the program by indicating an initial portion of a key field. These three methods are described in Section 4.2.3.1.

Consider the following example:

```

5      ON ERROR GO TO 200
10     MAP (PDATA) NAME$=30%,ID$=6%,JOBDES$=20
20     OPEN "PFILE.DAT" FOR OUTPUT AS FILE #1%      &
        ,ORGANIZATION INDEXED FIXED,ACCESS MODIFY  &
        ,ALLOW NONE,MAP PDATA,RECORDSIZE 56%      &
        ,PRIMARY NAME$,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 MODIFY  &
        ,ALLOW NONE,MAP PDATA,RECORDSIZE 56%      &
        ,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
200    PRINT "ERROR ";ERR," AT LINE ";ERL
210    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 200      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.

## FILES

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 at the terminal 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 to the terminal.

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 210 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 target 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.

### 4.4 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 three formats. These are:

Fixed	the file contains records of equal and fixed length.
Variable	the file may contain records of different lengths.



## FILES

**Stream** the file contains a contiguous series of ASCII characters. A record is defined as a set of characters delimited by a form feed, vertical tab, line feed, escape, or carriage return/line feed combination.

The file organization determines which of the formats you can select. Table 4-3 shows the relationship between file organization and record format.

Table 4-3  
Record Formats

File Organization	Format		
	Fixed	Variable	Stream
Sequential	yes	yes	yes
Relative	yes	yes	no
Indexed	yes	yes	no

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    } { STREAM }
```

Stream format is supported only for sequential files on disk. A stream specification is used for BASIC-PLUS compatibility and creates a file containing ASCII stream data. If you attempt to create a stream ASCII file on a non-disk device, an error is returned (i.e., Illegal record format). 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.

### 4.4.1 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

## FILES

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.

### 4.4.2 Variable-Length Records

Variable-length format 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.

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.

When you create sequential files with variable-length format, RMS appends a count field in front of each record written to the file. The count field indicates the number of character positions present in the record. When the record is requested by a program, RMS releases a record whose length is that specified by the count field.

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 RECORDSIZE 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.

### 4.4.3 Stream-Format Records

Stream format describes a file that contains a group of contiguous ASCII characters. RMS considers a record in such a file to be a set of characters delimited by one of the following:

## FILES

1. Line feed (LF)
2. Form feed (FF)
3. Vertical tab (VT)
4. Carriage return/line feed combination (CR/LF)
5. Escape (ESC)

Stream format is allowed only on files with sequential organization that reside on disk. The stream attribute is provided on RSTS/E to allow BASIC-PLUS file compatibility and to permit the use of the OPEN statement to create terminal I/O files. That is, stream-format files contain only ASCII data as opposed to the binary record data that appears in other formats.

To create a stream-format file, you can open a sequential file with no organization specification (i.e., terminal format) or you can specify the STREAM attribute in the SEQUENTIAL clause, as follows:

```
OPEN filename [FOR OUTPUT] AS FILE #num-exp  
  ,[ORGANIZATION] SEQUENTIAL STREAM
```

Stream-format records can be of fixed or variable length and no count field precedes the record. For output operations (PUT), RMS examines the last character of the record that your BASIC program constructed. If the last character is a LF, VT, ESC, or FF, RMS writes the record to the file. If the last character is not a LF, VT, ESC, or FF, RMS appends a CR followed by a LF to the end of the record and then writes it to the file.

For input operations (GET), RMS scans the file for the first occurrence of a LF, VT, FF, ESC, or CR/LF. If the scan is terminated by a LF, VT, ESC, or FF, RMS passes the entire string (including the terminator) to the program. If the scan encounters a CR/LF, RMS removes these two characters and passes the preceding string to the program. Each successive input operation causes the scan to resume at the character following the last LF, VT, FF, ESC, or CR/LF encountered.

### 4.5 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 4.5.1.

A bucket is a logical data structure that is composed of blocks. Buckets are used for 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 4.5.2.

## FILES

### 4.5.1 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-labelled tapes, you can specify this size in the BLOCKSIZE clause as a positive integer. 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
```

where:

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.

### 4.5.2 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 15. 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 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.

## FILES

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 4.5.2.1.

**4.5.2.1 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,

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

## FILES

where:

- Bnum
is the number of blocks per bucket in a range of 1 to 15 blocks. The bucket size is rounded up to the next highest integer, where required.
- Rlen
is the length in bytes of the file's fixed-length records as defined in the RECORDSIZE clause.
- Rmax
is the length in bytes of the largest variable-length record in the file as defined in the RECORDSIZE clause.
- Rnum
is the number of records 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 4-4 shows the default bucket sizes selected by BASIC when the number of records is undefined (i.e., the bucket contains only one record).

Table 4-4  
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,

$$\text{Bnum} = (22 + \text{Rlen}) / 512$$

Fixed-length records with BUCKETSIZE specified,

$$\text{Bnum} = ((7 + \text{Rlen}) * \text{Rnum}) + 15 / 512$$

## FILES

Variable-length records with no BUCKETSIZE specification,

$$\text{Bnum} = (24 + \text{Rmax}) / 512$$

Variable-length records with BUCKETSIZE specified,

$$\text{Bnum} = ((9 + \text{Rmax}) * \text{Rnum}) + 15 / 512$$

where:

- Bnum
is the number of blocks per bucket in a range of 1 to 15 blocks. The bucket size is rounded up to the next highest integer, where required.
- Rlen
is the length in bytes of the file's fixed-length records as defined in the RECORDSIZE clause.
- Rmax
is the length in bytes of the largest variable-length record in the file as defined in the RECORDSIZE clause.
- Rnum
is the number of records you 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 4-5 shows the default bucket sizes selected by BASIC when the number of records is undefined (i.e., the bucket contains only one record).

Table 4-5  
Indexed File Default Bucket Size

Bnum	Rlen	Rmax
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

## FILES

When you specify a bucket size for files in your program, you should keep in mind the space versus speed trade-offs 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.

### 4.6 RECORD MAPPING

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 4.5).

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 allocation of buffer space and the content of the records in those buffers is determined by the program 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. 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 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.

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 elements 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    } { STREAM
, MAP map-name
, RECORDSIZE num-exp
```



## FILES

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

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,RECORDSIZE 41%
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 RECORDSIZE specification and MAP reference are 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 can also be used in the OPEN statement to define RECORDSIZE. In addition, a map reference and a RECORDSIZE specification can both appear in the same OPEN statement. This enables you to specify a smaller record (buffer) for a particular operation when the record length format is variable. Note that when both a map reference and a RECORDSIZE specification are used, the RECORDSIZE specification takes precedence.

### NOTE

Because a RECORDSIZE specification overrides a MAP, it is possible to define a record size and cause a record operation to overwrite mapped areas. You should exercise caution when specifying a RECORDSIZE that is larger than previously defined MAP statements for the same file.

## 4.7 RMS UTILITIES

RMS provides a collection of utility programs that can be used to initialize, manipulate, and maintain RMS files at the system command level.

For the most part, the utility programs operate on existing RMS files; however, these programs are not designed to substitute for the creation and use of RMS files with BASIC-PLUS-2. The rest of this section contains an introductory explanation of the RMS utilities. For a detailed description of their format and use, refer to the RSTS/E RMS-11 Utilities User's Guide.

## FILES

The utility programs are composed of five separate operations, as follows:

1. RMSBCK - creates copies of RMS files.
2. RMSRST - returns copied files to their original state.
3. RMSCNV - moves data from one file to another.
4. RMSDFN - creates RMS files.
5. RMSDSP - lists attributes of RMS files.

The RMSBCK utility allows you to create copies of one or more RMS files and store the copies on a specified medium (disk or magtape). This process ensures that data in a file will not be lost because of a software or hardware problem. The copies created by RMSBCK are specially formatted such that user programs cannot access them. Also, the attributes, header information, and protection code associated with the original file are preserved on the copy.

The RMSRST utility reverses the process initiated by RMSBCK. The RMSRST utility accepts RMSBCK files as input and outputs standard files, i.e., the file as it was in its original state. The structure, content, and attributes of a file can thus be protected against loss with these two utilities.

The RMSCNV utility reads records from a specified input file and writes them onto a specified output file. The method used to transfer records depends on the organization of the specified files and on the desired utility options.

The RMSDFN utility creates an RMS file with a user-specified filename and attribute assignment. The utility does not allocate file space nor does it write records to the file. These operations must be done by means of a user program or the RMSCNV utility.

The RMSDSP utility allows you to list the attributes of one or more specified files. This utility is especially useful to RSTS/E BASIC-PLUS-2 programmers who wish to examine the attributes of existing files. The utility lists the filename, extension, creation date, file organization, protection codes, space allocation, revision dates, record format, record size, primary and alternate key definitions (if applicable), and bucket size (for relative and indexed files).



## CHAPTER 5

### TRANSLATOR UTILITY

The Translator utility is a system program that converts a BASIC-PLUS user program to BASIC-PLUS-2 and thus preserves existing RSTS/E BASIC-PLUS applications. The Translator accepts a BASIC-PLUS source program as input, converts it to BASIC-PLUS-2 while maintaining the intent of the program, and outputs the converted program. In addition, the Translator prints an appropriate warning for any potential trouble spots that it detects in your BASIC-PLUS program. The Translator also allows you to convert BASIC-PLUS programs written in EXTEND and NOEXTEND mode. The Translator does not, however, alter the format of your program (indentation, blanks, tabs, etc.) except for those items listed in Section 5.1 that are actually converted.

#### 5.1 ITEMS FOR TRANSLATION

The following are the BASIC-PLUS syntax items that are converted to BASIC-PLUS-2 syntax by the Translator. Most of these items are automatically converted but some are changed at your discretion. A complete description of these language functions and compatibility issues is given in Appendix A.

##### Continuation Lines

BASIC-PLUS program lines are continued with a line feed or an ampersand (&) followed by a line terminator. BASIC-PLUS-2 continuations can only be made with an ampersand (&) followed by a line terminator. This change is made automatically by the Translator. The Translator adds a period on the end of BASIC-PLUS lines that contain an ampersand as the last character, thus avoiding any ambiguity.

##### DEF Statements

BASIC-PLUS and BASIC-PLUS-2 user-defined functions differ in the method used for passing arguments. To ensure compatibility, BASIC-PLUS-2 on RSTS/E also supports the BASIC-PLUS method. The Translator adds an asterisk to BASIC-PLUS DEF statements (i.e., DEF\*) to mark them as using the BASIC-PLUS argument passing method.

##### PRINT Synonym

BASIC-PLUS accepts an ampersand as a synonym for PRINT. This conflicts with the BASIC-PLUS-2 continuation character. The Translator automatically changes all BASIC-PLUS ampersands to PRINT statements (except for ampersands contained in string literals and comments or those used for continuation in EXTEND mode).

## TRANSLATOR UTILITY

### CHAIN Statements

BASIC-PLUS-2 requires that the keyword LINE precede a specified line number in CHAIN statements. The Translator automatically inserts the keyword LINE before line numbers when they are specified in BASIC-PLUS CHAIN statements.

### Statement Separators

BASIC-PLUS allows multiple statements to be separated by a colon or a backslash. BASIC-PLUS-2 only allows a backslash as the separator. The Translator automatically converts all statement separators to backslashes in BASIC-PLUS programs.

### PRINT USING

BASIC-PLUS syntax for specifying the output format of a character string is an exclamation point for one character and n-2 spaces enclosed by backslashes for more than one character (where n is the desired number of characters). BASIC-PLUS-2 syntax is a single quote or an exclamation point for one character and a single quote plus an L for more than one (where the number of Ls plus 1 is the desired number of characters). The letter L represents left-justified output and can be replaced by R for right-justified, C for centered, and E for extended. To ensure compatibility, BASIC-PLUS-2 on RSTS/E supports both forms of string format. The Translator automatically converts BASIC-PLUS syntax to BASIC-PLUS-2 where possible, and issues a warning where not possible.

### DATA Statements

BASIC-PLUS ignores embedded spaces and tabs in unquoted string literals. BASIC-PLUS-2 considers them significant. The Translator resolves this conflict by automatically removing embedded spaces and tabs.

### Position Function

BASIC-PLUS-2 supports the BASIC-PLUS POS function but changes the function name to CCPOS by means of the Translator. The name is converted because BASIC-PLUS-2 supports a function that locates the position of a substring within a string. This function is named POS and is compatible with Dartmouth BASIC.

### Long Variable Names

BASIC-PLUS-2 allows variable names of up to 30 characters (not including the optional characters FN, \$, and %). The first character must be alphabetic and can be followed by up to 29 letters, numbers, or periods. You can use the Translator to change specified BASIC-PLUS variable names to the long format. You can also create a command file containing BASIC-PLUS and BASIC-PLUS-2 names and make the name changes from the file.

### Spaces

Because BASIC-PLUS-2 allows long variable names, spaces or tabs are required between keywords, symbols, and literals to avoid ambiguity. The Translator automatically inserts spaces between BASIC-PLUS lexical elements.

## TRANSLATOR UTILITY

### Comment Separators

BASIC-PLUS requires that comments begin with an exclamation point and end with a line terminator. BASIC-PLUS-2 allows comments to begin and end with an exclamation point, which permits you to write comments at any point on a program line. The Translator automatically removes all but the first exclamation point from BASIC-PLUS lines and replaces them with asterisks to preserve spacing. This does not apply to string literals. BASIC-PLUS-2 comments cannot be continued; the Translator inserts exclamation points to convert BASIC-PLUS continued comments.

### Unterminated String Literals

BASIC-PLUS allows string literals to be delimited by a line terminator. BASIC-PLUS-2 requires matching single or double quotation marks on both sides of the string. The Translator automatically adds the proper terminators to BASIC-PLUS string literals.

### SYS Functions

BASIC-PLUS supports SYS functions and the PEEK function. To ensure compatibility, BASIC-PLUS-2 supports these functions on RSTS/E only. The Translator flags SYS and PEEK functions as errors on all systems but RSTS/E.

### Multiple Assignment Statements

BASIC-PLUS evaluates multiple assignment statements in right-to-left order. BASIC-PLUS-2 evaluates them from left to right and makes assignments from right to left. The Translator resolves this conflict by internally restructuring the statements such that each assignment is evaluated as a separate statement and issues a warning for potential evaluation errors.

### Ambiguous Constants

When an ambiguous constant (i.e., 100 as opposed to 100% or 100.) is used in a BASIC-PLUS arithmetic expression, it is treated as an integer if an integer value appears to the left of the constant in the expression. If such an integer is not present, BASIC-PLUS treats the ambiguous constant as a floating-point number.

BASIC-PLUS-2 always treats an ambiguous constant as a floating-point number.

The Translator simulates the BASIC-PLUS treatment. It adds a per cent sign to an ambiguous constant if an integer value appears to the constant's left in the expression.

### Semicolons

In certain cases, BASIC-PLUS allows an implied semicolon in PRINT and INPUT statements. BASIC-PLUS-2 requires a comma or semicolon between items in a PRINT statement list and assumes a comma by default in INPUT statements. For example:

```
10 PRINT "SN" X
```

is legal in BASIC-PLUS, but returns a syntax error in BASIC-PLUS-2.

The Translator automatically inserts semicolons in BASIC-PLUS PRINT and INPUT statements, as required.

## TRANSLATOR UTILITY

### Line numbers

Per cent signs on line numbers are superfluous. The Translator automatically removes them from the BASIC-PLUS program. For example:

```
100% PRINT
```

is converted to:

```
100 PRINT
```

### Numeric Constants

BASIC-PLUS allows blanks and tabs to appear within a numeric constant; BASIC-PLUS-2 does not. For example:

```
100 PRINT 5 32
```

in a BASIC-PLUS program outputs the constant 532. The same line in a BASIC-PLUS-2 program returns a syntax error. The Translator automatically compresses all blanks and tabs that appear within a numeric constant. Therefore, when line 100 (above) is translated, it appears in the program as follows:

```
100 PRINT 532
```

### Null Arguments in Functions

BASIC-PLUS allows a null argument in a user-defined function; BASIC-PLUS-2 does not. The Translator resolves this issue by automatically removing null arguments in functions. For example:

```
10 Y=FNA()
```

in the BASIC-PLUS program is converted to:

```
10 Y=FNA
```

## 5.2 USING THE TRANSLATOR

The Translator utility program is stored in the system library account. To access it, type:

```
RUN $TRANS
```

The Translator signifies successful access by printing an identification header on the terminal. Because the Translator utility can be present on a number of operating systems, it follows the identification header with the following prompt:

```
TARGET SYSTEM?
```

To answer, type a name that represents the system you are using. In the case of RSTS/E PDP-11, you type RSTS. Note that RSTS is also the Translator utility default. The Translator then prints:

```
INPUT FILE?
```

This is a prompt for the name and extension of the BASIC-PLUS program you wish to convert. The Translator requires the name of an existing BASIC-PLUS program; however, if no extension is specified, it searches for a source program (.BAS) by default. In addition, the

## TRANSLATOR UTILITY

specified input program must be error free. If the program contains immediate mode statements, the Translator removes them and issues a message to that effect.

Note that only one program can be input in response to the INPUT FILE prompt. Following this prompt, the Translator prints:

OUTPUT FILE?

This prompt allows you to assign a different name and extension to the converted program. If you do not type a new name and extension, the Translator assigns the input filename with a .B2S extension to the output. If you specify the input name and its extension for the converted BASIC-PLUS-2 program, the old BASIC-PLUS program is deleted following the conversion.

### NOTE

It is recommended that you take the necessary steps to save the old BASIC-PLUS program. In the event of an incorrect translation, a filename change protects against permanent program loss.

After a conversion is complete, the Translator again prompts for input. When you have made all of the desired conversions, type CTRL/Z (^Z) in response to the INPUT FILE? prompt. This causes an exit from the Translator.

Following the OUTPUT FILE? prompt, the Translator prints:

EXTEND MODE?

If the BASIC-PLUS program is written in EXTEND mode (a RSTS/E version 6B feature), you respond to this prompt with YES. This ensures that any syntactic differences between BASIC-PLUS EXTEND and BASIC-PLUS-2 are converted to BASIC-PLUS-2. If you respond with a carriage return, or any answer but YES, the program is assumed to be NOEXTEND. The Translator automatically shifts to the proper mode if it encounters an EXTEND or NOEXTEND statement in the input program.

### 5.2.1 Variable Name Specification

After you type the name of the BASIC-PLUS program and indicate NOEXTEND mode, the Translator requests the BASIC-PLUS variable name(s) that you wish to change.

For example:

EXTEND MODE? NO

OLD NAME?

In response to this prompt, type the existing BASIC-PLUS variable name. The Translator then asks you to enter the new name, which can be a BASIC-PLUS-2 name of up to 30 characters. Note that you do not include function, string, or integer designations in the new name (i.e., FN, \$, or %). For example:

OLD NAME? A\$

NEW NAME? ACCOUNTS



## TRANSLATOR UTILITY

This dialogue continues until you have specified all the variable names you wish to convert. When you have typed all the names, or if you do not wish to change any, type the RETURN key.

When the Translator executes, the specified variable name changes are made but the variable type and any subscripts are unchanged. For example:

```
OLD NAME?  T1%  
NEW NAME?  TEMP1
```

changes all occurrences of T1% in the BASIC-PLUS program to TEMP1% in the BASIC-PLUS-2 program.

You can also build a command file containing the BASIC-PLUS and BASIC-PLUS-2 variable names you wish to convert. If you specify:

```
OLD NAME?  @filspc
```

where @filspc is an at sign (@) followed by a specified file containing the variable names, all conversions are made from the file. You can prepare this file by means of an editor. The file should be formatted so that the old variable name is followed on the next line by the new name. For example:

```
Al%(  
NEW.ARRAY  
FNW$  
NEW.FUNCTION
```

Note that the OLD NAME prompt will not appear if you have specified an EXTEND mode BASIC-PLUS program.

### 5.2.2 Translator Sample Run

The following is a BASIC-PLUS program named FUNC:

```
10      DEF FNF(X)=X^2-2*X-3  
100     & "PROGRAM TO PLOT FUNCTION FNF(X)  
110     & "TYPE VALUES FOR THE FOLLOWING  
120     INPUT "MIN VALUE OF X"X1  
130     INPUT "MAX VALUE OF X"X2  
140     INPUT "SUBINTERVALS ON X-AXIS" S  
150     INPUT "MIN VALUE OF Y"Y1  
160     INPUT "MAX VALUE OF Y"Y2  
200     LET L=(X2-X1)/40      :  
        LET A=Y1-L/2        :  
        LET B=Y2-L/2  
300     & "Y-AXIS : " Y1 "TO" Y2 "IN INTERVALS OF" L      :  
        & TAB(15%); "-----"  
400     FOR X=X1 TO X2 STEPS      :  
        Y=FNF(X)                :  
        IF Y<A OR Y>=B THEN      :  
            & X; TAB(14%); " "      :  
        GOTO 420  
410     & X; TAB(14%) " " TAB((Y-Y1)/L+15.5); " " :  
420     NEXT X  
500     STOP  
510     & X; TAB(14%); " " :  
520     GOTO 420  
32767   END
```

# TRANSLATOR UTILITY

The following dialogue converts this program to BASIC-PLUS-2. The option to change variable names is also used.

```

READY

RUN $TRANS

TRANS Y01-01

TARGET SYSTEM? RSTS

INPUT FILE? FUNC.BAS

OUTPUT FILE? FUNC.B2S

EXTEND MODE? NO

OLD NAME? A
NEW NAME? MIN
OLD NAME? B
NEW NAME? MAX
OLD NAME? RET

INPUT FILE? ^Z

READY

```

If you list the converted program (FUNC.B2S), it appears as:

```

10      DEF *FNF(X)=X^2-2*X-3
100     PRINT "PROGRAM TO PLOT FUNCTION FNF(X)"
110     PRINT "TYPE VALUES FOR THE FOLLOWING"
120     INPUT "MIN VALUE OF X";X1
130     INPUT "MAX VALUE OF X";X2
140     INPUT "SUBINTERVALS ON X-AXIS";S
150     INPUT "MIN VALUE OF Y";Y1
160     INPUT "MAX VALUE OF Y";Y2
200     LET L=(X2-X1)/40
        LET MIN=Y1-L/2
        LET MAX=Y2-L/2
300     PRINT "Y-AXIS : " ;Y1 ;"TO" ;Y2 ;"IN INTERVALS OF" ;L
        PRINT TAB(15%); "-----"
400     FOR X=X1 TO X2 STEPS
        Y=FNF(X)
        IF Y<MIN OR Y>MAX THEN
            PRINT X; TAB(14%); " "
        GOTO 420
410     PRINT X; TAB(14%) ;" " ;TAB((Y-Y1)/L+15.5); " "
420     NEXT X
500     STOP
510     PRINT X; TAB(14%); " "
520     GOTO 420
32767  END

```

Note that the Translator adds an asterisk to the DEF statement (line 10) to mark it as BASIC-PLUS compatible. Also, the variable names have been changed as requested and the colon statement separators have been replaced by backslashes.

## TRANSLATOR UTILITY

### 5.2.3 Translator Warning Messages

The Translator prints a warning message when it detects an error in the conversion process or to notify you of potential trouble spots in the converted program. Errors that occur during the conversion process generate an immediate message. Warning messages that indicate potential trouble spots are printed at the end of the conversion.

The messages and their meanings are as follows:

#### ? TRANS - Unexpected error x at line y

This message is generated because of an error in the BASIC-PLUS input program. In this message, x represents an error code as described in Section C.3 and y represents the input program line that contains the error. The conversion is aborted. You must ensure that the BASIC-PLUS input program is error free.

#### % TRANS - System dependent function at line y

This message occurs when you attempt to convert a BASIC-PLUS program that contains SYS or PEEK functions. Because RSTS/E supports these functions, this message does not appear if RSTS/E is the target system. In all cases the conversion is made, but for non-RSTS/E systems, you are advised that the function is not supported.

#### % TRANS - Expression sequence check at line y

This message is issued when a multiple assignment statement is encountered. It advises you that program evaluation of multiple assignment statements may be out of sequence. The conversion is made, but you should examine the evaluation made at line number y.

#### % TRANS - OPEN for terminal I/O at line y

This message is generated by a program that attempts to create a virtual array file on a non-RSTS/E system. The conversion is made, but if the target system is not RSTS/E, you are advised that the desired file capability may not be supported.

#### % TRANS - PRINT USING format variable at line y

This message indicates a potential PRINT USING format error in the converted program at line number y. A possible cause can be a PRINT USING string literal that contains an apostrophe. In certain cases, the Translator interprets the apostrophe as a space. The conversion is made, but you should examine the program.

#### % TRANS - No variable substitutions

This message is generated when a variable name change error is detected. The Translator does not convert any BASIC-PLUS variable names.

## TRANSLATOR UTILITY

### % TRANS - Immediate mode statements removed

Because BASIC-PLUS-2 does not support immediate mode, the Translator removes immediate mode statements from the BASIC-PLUS conversion. When the conversion is completed, the Translator prints this message followed by a list of the removed statements.

### % TRANS - Entry already exists

This message results from an attempt to enter duplicate variable names in response to the variable name change dialogue. The Translator prints the message and reinitiates the dialogue.

### % TRANS - Invalid variable name

This message results from an attempt to enter an illegal variable name in response to the variable name change dialogue. The Translator prints the message and reinitiates the dialogue.



1

2



3

4



## APPENDIX A

### COMPATIBILITY

BASIC-PLUS-2 is a new language containing many improvements over the BASIC-PLUS language while continuing to support BASIC-PLUS usage. There are, however, some syntactic and semantic differences between BASIC-PLUS-2 and BASIC-PLUS. The majority of these differences are resolved by the Translator utility described in Chapter 5. Thus, most BASIC-PLUS user applications are transportable to BASIC-PLUS-2.

The following sections detail the compatibility issues existing between the two languages. Those issues resolved by the Translator program are included in Section A.1. You can resolve the remaining issues by redesigning your current BASIC-PLUS program to conform with BASIC-PLUS-2 (see Section A.2).

#### A.1 TRANSLATABLE ISSUES

The following subsections describe the BASIC-PLUS-2/BASIC-PLUS compatibility items that are acceptable input to the Translator program. Most of these items are automatically converted when you translate a BASIC-PLUS program. Variable names (see Section A.1.12), however, are converted at your option. The description associated with each item explains the compatibility issue involved and the method used by the Translator for resolving it.

##### A.1.1 PRINT USING String Format

The format specifications for string fields in the BASIC-PLUS-2 PRINT USING statement differ syntactically from the BASIC-PLUS specifications. These differences are tabulated below. You can resolve these differences with the Translator utility. The Translator either converts the BASIC-PLUS syntax to BASIC-PLUS-2 or issues a warning message. There is no difference between BASIC-PLUS-2 and BASIC-PLUS numeric field format specifications, so no conversion is necessary for them.

## COMPATIBILITY

### NOTE

It is possible to create a string field format specification in a BASIC-PLUS PRINT USING statement at run time. In this case, the Translator cannot make the conversion. Therefore, to ensure compatibility, BASIC-PLUS-2 on RSTS/E supports both types of string format. Conversion is recommended, however, because this support is offered only on RSTS/E and programs with the BASIC-PLUS format will not be transportable to other systems.

BASIC-PLUS PRINT USING string format specifications consist of the following symbols:

- ! Exclamation point. This symbol identifies a 1-character field and causes the output of the first character in a string.
- \\ Spaces enclosed by backslashes. The number of enclosed spaces (plus two for the backslashes) indicates the number of characters in the string to be printed.

The BASIC-PLUS-2 string format specifications consist of the following symbols:

- ! ' Apostrophe or exclamation point. When the apostrophe or exclamation point is used alone, it indicates the output of the first character in a string. The apostrophe must appear when any of the following symbols are used.
- C An upper-case C indicates that string output is to be centered. The number of string characters printed equals the number of Cs plus one (for the apostrophe) in the specification.
- L An upper-case L indicates left-justified output. The number of string characters printed equals the number of Ls plus one (for the apostrophe) in the specification.
- R An upper-case R indicates right-justified output. The number of string characters printed equals the number of Rs plus one (for the apostrophe) in the specification.
- E An upper-case E indicates left-justified output and extends the field so that all string characters are printed.

The following example illustrates the use and format of BASIC-PLUS-2 PRINT USING string format specifications:

## COMPATIBILITY

```
20      PRINT USING " 'LLLLLLLLLL", "THIS TEXT"
30      PRINT USING " 'LLLLLLLLLLLLLLLL", 'SHOULD PRINT '
40      PRINT USING " 'LLLLLLLLLLLLLLLL", "AT LEFT MARGIN"
50      PRINT USING " 'RRRR", "1,2,3,4"
60      PRINT USING " 'RRRR", "1,2,3"
70      PRINT USING " 'RRRR", "1,2"
80      PRINT USING " 'RRRR", "1"
90      PRINT USING " 'CCCCCCCC", "A"
100     PRINT USING " 'CCCCCCCC", "ABC"
110     PRINT USING " 'CCCCCCCC", "ABCDE"
120     PRINT USING " 'CCCCCCCC", "ABCDEFG"
130     PRINT USING " 'CCCCCCCC", "ABCDEFGH"
140     PRINT USING " 'LLLLLLLLLLLLLLLL", "YOU ONLY SEE HALF OF THE LINE"
150     PRINT USING " 'E", "YOU CAN SEE ALL OF THE LINE WHEN EXTENDED"
160     END
```

READY

```
RUNNH
THIS TEXT
SHOULD PRINT
AT LEFT MARGIN
1,2,3
1,2,3
  1,2
    1
      A
        ABC
          ABCDE
            ABCDEFG
              ABCDEFGH
                YOU ONLY SEE HALF
                  YOU CAN SEE ALL OF THE LINE WHEN EXTENDED
```

READY

### A.1.2 Quoted String Literals

BASIC-PLUS-2 requires that string literals be enclosed by double or single quotation marks on the beginning and end of the string. Thus, a string enclosed by double quotation marks can contain single quotation marks within the string. The converse is also true. BASIC-PLUS-2 considers spaces in delimited strings significant. The following are legally delimited BASIC-PLUS-2 strings:

"A STRING"

' "A STRING", HE SAID '

BASIC-PLUS allows string literals to begin with a single or double quotation mark and to end with a line terminator. Such a string in BASIC-PLUS-2 generates an error at compile time:

?Unterminated string

The Translator utility converts unterminated strings in BASIC-PLUS programs by adding the proper delimiter.

### A.1.3 Multiple Assignment Statement

BASIC-PLUS-2 evaluates a multiple assignment statement in left-to-right order. That is, as BASIC-PLUS-2 scans the line from



## COMPATIBILITY

left to right, all expressions on the left side of an assignment are evaluated. This order differs from the BASIC-PLUS order of evaluation, which is from right to left. BASIC-PLUS-2, however, makes assignments in right-to-left order.

The Translator utility resolves this issue by internally restructuring the assignments into separate statements.

For example, the lines:

```
LET I=5
LET A(I), I=10
```

are interpreted by the Translator as follows:

```
LET I=5
TEMP=10
I=TEMP
A(I)=TEMP
```

### A.1.4 Ambiguous Constants

BASIC-PLUS-2 treats ambiguous constants (i.e., 100 as opposed to 100% or 100.) as floating-point numbers.

BASIC-PLUS treats ambiguous constants as floating-point unless an integer appears to the constant's left in a statement. For example, in the expression  $A\% = B\% / 100$ , BASIC-PLUS treats 100 as an integer while BASIC-PLUS-2 treats it as a floating-point number. The Translator inserts per cent signs in converted BASIC-PLUS programs where required to maintain the context of expressions.

### A.1.5 DEF Statements

BASIC-PLUS-2 handles user-defined functions differently than BASIC-PLUS. In BASIC-PLUS-2 programs, DEF function parameters are maintained within the function routine as local variables. When these variables are referenced in the routine, they are referenced locally. When a reference is made to a function parameter from outside the function (such as in an ON ERROR GOTO statement), the local variable value is not accessible.

BASIC-PLUS makes all references to variables, both inside and outside the routine, as globals.

This difference may cause the values printed by identical BASIC-PLUS-2 and BASIC-PLUS user-defined functions to vary. Consider the following example:

```
10 DEF FNX(A)
20 IF A<3 GOTO 40
30 LET A=6\GOTO 100
40 FNXD
50 LET A=3
60 LET C=FNX(4)
70 LET D=FNX(2)
80 PRINT A
90 STOP
100 PRINT A
110 GOTO 40
120 END
```

## COMPATIBILITY

When this program is run on BASIC-PLUS-2, the printed values are 3 and 3. However, when the same program is run on BASIC-PLUS, the printed values are 6 and 3.

To ensure compatibility, BASIC-PLUS-2 supports both methods of handling user-defined functions. However, this support is provided only for RSTS/E and the DECsystem-20. Therefore, programs that reference parameters outside the function and use the BASIC-PLUS method of handling user-defined functions are not transportable to other systems.

The Translator utility marks all DEF statements in BASIC-PLUS programs with an asterisk (\*). This identifies that they are written with the BASIC-PLUS method.

### A.1.6 POS Function

The BASIC-PLUS position (POS) function returns a numeric value that indicates the pointer's current column position on a specified I/O channel. The function has the form:

POS(x)

where x is an I/O channel number. (A specification of 0 always represents the user terminal.) For example:

```
10 PRINT "X";TAB(10);POS(0)
```

prints on the terminal:

```
      X          10
```

This indicates that the pointer is currently at the tenth column on the terminal output line.

The BASIC-PLUS-2 POS function locates substrings within BASIC-PLUS-2 strings. The use of this function parallels the use of the BASIC-PLUS INSTR function and ensures compatibility with Dartmouth BASIC. For example:

POS(A\$,B\$,X%)

locates the position of the first occurrence of a specified substring (B\$) within a string (A\$), beginning at a specified position (X%).

Because the use of the POS function differs on BASIC-PLUS-2 and BASIC-PLUS, the name of the BASIC-PLUS POS function has been changed to CCPOS. The Translator utility makes this change automatically by changing all POS keywords in BASIC-PLUS programs to CCPOS.

### A.1.7 DATA Statement String Literals

BASIC-PLUS-2 does not require that DATA statement string literals be enclosed by quotation marks and treats embedded spaces and tabs in unquoted strings as significant. This method differs from BASIC-PLUS usage, which disregards embedded spaces and tabs in unquoted string literals. However, both BASIC-PLUS-2 and BASIC-PLUS require quotation marks if commas or leading and trailing spaces are to be considered significant.

## COMPATIBILITY

The Translator utility removes embedded spaces and tabs from BASIC-PLUS program DATA strings and thus avoids any conflict between BASIC-PLUS-2 and BASIC-PLUS.

### A.1.8 Multi-Statement Lines

BASIC-PLUS-2 allows more than one statement on a program line as long as you separate all the statements (except the last) with backslashes. BASIC-PLUS, however, allows either a colon or a backslash as a statement separator.

The Translator converts all statement separators to backslashes in BASIC-PLUS programs, with the exception of colons in string literals.

### A.1.9 Comment Separator

BASIC-PLUS-2 allows comments to begin and end with an exclamation point (!). This permits you to insert comments at any point on a program line. For example:

```
10 LET A=1!COMMENT!\B=3
20 !COMMENT! PRINT A+B!COMMENT
```

This method differs from the BASIC-PLUS method, which requires that comments begin with an exclamation point and end with a line terminator.

The Translator utility resolves this issue by removing all but the first exclamation point from BASIC-PLUS program comments. The removed exclamation points are replaced by asterisks to maintain the spacing of the line.

### A.1.10 Continuations

BASIC-PLUS-2 program lines are continued when you type an ampersand (&) followed by a line terminator (a carriage return, line feed, form feed, vertical tab, or escape) at the end of the line to be continued. The printed ampersand on a listed program identifies the line as a continuation. The BASIC-PLUS method of continuing lines (a line feed) leaves no indication that the line is a continuation.

The Translator utility changes all BASIC-PLUS continued lines to the BASIC-PLUS-2 format. If an existing BASIC-PLUS line has an ampersand as the last character, the Translator adds a period after the ampersand. This prevents any ambiguity over whether or not the line is a continuation.

Comments with the exclamation point delimiter cannot be continued to another line. To continue a long comment line, write it as a REM statement.

### A.1.11 PRINT Synonym

BASIC-PLUS allows an ampersand (&) to be used as a synonym for the PRINT statement. Because this conflicts with BASIC-PLUS-2's use of ampersand as a continuation character, BASIC-PLUS-2 does not support the BASIC-PLUS usage.

## COMPATIBILITY

The Translator utility resolves this conflict by converting all ampersands to PRINT statements in BASIC-PLUS programs, with the exception of ampersands in string literals or those used to represent continuations in EXTEND mode.

### A.1.12 Long Variable Names

BASIC-PLUS-2 allows variable and function names of up to 30 characters. The size restriction of 30 characters does not include FN, %, or \$. That is, a user-defined integer function can begin with FN, contain 30 characters, and end with a percent (%), for a total of 33 characters. The 30 characters can include alphanumerics and periods. However, you must avoid using a BASIC keyword as a variable name (see Section B.4). Because long variable names could be ambiguous, spaces or tabs are required between keywords, symbols, and literals. For example:

```
10 LETHAL=29
```

assigns the value 29 to the variable LETHAL. If you intend to name the variable HAL, the correct BASIC-PLUS-2 line is written as:

```
10 LET HAL=29
```

BASIC-PLUS (in NOEXTEND mode) does not support variable names of more than two characters and so does not require spaces between language elements. For example:

```
10 FOR I=STOP
```

is understandable input for BASIC-PLUS, which compiles the above line as:

```
10 FOR I=S TO P
```

The Translator utility inserts spaces between language elements in BASIC-PLUS programs. Thus, any ambiguity in converting programs from BASIC-PLUS to BASIC-PLUS-2 is resolved. You also have the option of using the Translator to change one or more specified BASIC-PLUS variable names to the long form.

### A.1.13 CHAIN Statement

The BASIC-PLUS-2 CHAIN statement syntax differs from that of BASIC-PLUS. BASIC-PLUS-2 requires the keyword LINE between the filename specified in the CHAIN statement and the line number where the chaining is to begin. A BASIC-PLUS-2 CHAIN statement has the following format:

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

The BASIC-PLUS CHAIN statement is written as:

```
100 CHAIN file-expression [num-exp]
```

The Translator utility makes these syntactic changes to BASIC-PLUS programs. The Translator inserts the required keyword LINE in all BASIC-PLUS program CHAIN statements that contain line number specifications.

## COMPATIBILITY

Note that BASIC-PLUS-2 completes the output of any pending buffers before the files are closed by a CHAIN statement. This differs from BASIC-PLUS CHAIN statements, which close the files without necessarily completing buffer output.

### A.1.14 SYS Functions

The BASIC-PLUS RSTS/E SYS and PEEK functions are supported by BASIC-PLUS-2 on the RSTS/E system only. Their use on other systems is flagged as an error and they are marked as such by the Translator. The use of these functions in a RSTS/E BASIC-PLUS-2 program can cause the program to be non-transportable to other systems.

### A.1.15 INPUT and PRINT Statement Punctuation

The BASIC-PLUS-2 INPUT and PRINT statements require a comma or semicolon between a quoted string and a variable. For example, the BASIC-PLUS-2 INPUT line:

```
10 INPUT "YOUR NAME";A$
```

outputs on the terminal as:

```
YOUR NAME ?
```

BASIC-PLUS-2 does not assume a semicolon between the quoted string and the variable. The default for INPUT statements without a comma or semicolon is to insert a carriage return/line feed combination following the string and output the INPUT prompt on the next terminal line. A PRINT statement without a comma or semicolon between the string and variable causes an error during compilation. The Translator automatically inserts semicolons in BASIC-PLUS PRINT and INPUT statements, as required.

Also, BASIC-PLUS-2 assumes commas when spaces separate elements of the INPUT list. For example:

```
10 INPUT A B
```

reserves space for the assignment of two values.

```
10 INPUT AB
```

reserves space for one value.

## A.2 NONTRANSLATABLE ISSUES

The following subsections describe the BASIC-PLUS-2/BASIC-PLUS issues that are not resolved by the Translator program. Where conflicts exist, you must redesign your BASIC-PLUS program to conform with BASIC-PLUS-2. The description associated with each item explains the issue involved and the BASIC-PLUS-2 usage.

## COMPATIBILITY

### A.2.1 Transfer Into FOR NEXT Loops

BASIC-PLUS-2 requires that the FOR statement in a FOR NEXT loop be executed before the program transfers into that loop by a GOTO or other such statement. The BASIC-PLUS-2 Compiler does not return an error message when an illegal transfer is made but the result of the transfer is undefined.

BASIC-PLUS also considers a transfer into the middle of a FOR NEXT loop illegal. But the outcome of such a loop in BASIC-PLUS may be different from that encountered under BASIC-PLUS-2.

### A.2.2 Debugging

BASIC-PLUS-2 provides several commands that are useful for debugging programs. These commands include BREAK, PRINT, CONTINUE, TRACE, LET, and STEP. Section 1.2.3 describes these BASIC-PLUS-2 features.

### A.2.3 CALL Statements

BASIC-PLUS-2 provides CALL and CALL BY REF statements that allow a program to access an external subroutine. Note that the subroutine must be compiled separately. Refer to Section 3.3 for information on these statements.

The CALL statement also indicates that the program is an object module and is linked by the Task Builder for execution. See Chapter 2 for information on the Task Builder.

### A.2.4 Compile-Time Errors

Because BASIC-PLUS is an interpreter, a program can contain certain errors that are not detected until you attempt execution. These errors include undefined line numbers, undefined functions, and illegal loop nesting. BASIC-PLUS-2, however, is a compiler and detects these errors at program compilation. A complete list of these errors and their causes is given in Section C.2.

### A.2.5 Array Subscripts

In a two-dimensional array, BASIC-PLUS checks the sum of both subscripts and returns an error if the range is exceeded. BASIC-PLUS-2 checks the subscripts individually and returns an error if either is out of range. For example:

```
100 DIM A$(20,20)
130 PRINT A$(1,21)
```

generates an error in BASIC-PLUS-2, but does not return an error in BASIC-PLUS.

## COMPATIBILITY

### A.2.6 Record I/O

BASIC-PLUS-2 Record I/O operations are controlled through the RMS (Record Management Service) file structure. This method of Record I/O supports sequential, relative, and indexed file organizations. RMS features also include primary and alternate keys, fixed- and variable-length records, and record mapping. A full explanation of Record I/O is given in Chapter 4.

RMS operations differ from the block I/O operations available under BASIC-PLUS RSTS/E. The BASIC-PLUS file system allows you to perform many types of operations at varying times on the same file. RMS, on the other hand, evaluates a requested operation against file attributes declared at the file's creation. Thus, many operations that would be legal under BASIC-PLUS on RSTS/E may be disallowed under BASIC-PLUS-2 RMS.

To ensure compatibility between BASIC-PLUS and BASIC-PLUS-2 I/O, RSTS/E supports block I/O operations by means of the virtual file organization. These operations include CVT (convert), LSET, RSET, and FIELD statements.

For example, on the BASIC-PLUS file structure, only string quantities are stored in I/O buffers. The CVT function is used to map binary data into string data for buffer storage. On the RMS file structure, integers and real numbers are directly stored and accessed as variables. Logical and physical Record I/O buffers are defined at compile time by the MAP statement, which also allows direct access of the numbers.

To create a RSTS/E block I/O structured file, you specify ORGANIZATION VIRTUAL in the OPEN statement (see Section 4.1.1). This file structure permits the use of BASIC-PLUS block I/O in the program and allows you to do any type of I/O mode mixing.

## APPENDIX B

### BASIC-PLUS-2 LANGUAGE ELEMENTS

This appendix summarizes the BASIC-PLUS-2 commands and functions that are supported on RSTS/E. If you desire more information on the language elements, refer to the BASIC-PLUS-2 Language 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.

#### B.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 per cent 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 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



## BASIC-PLUS-2 LANGUAGE ELEMENTS

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, form feed, vertical tab, or 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 characters are accepted in string literals but are ignored outside of strings and generate warnings. The computer 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. The following tables illustrate these operators and their use.

Table B-1  
Arithmetic Operators

Operator	Use	Meaning
$\wedge$ or **	$A^B$ or $A**B$	Exponentiation
*	$A*B$	Multiplication
/	$A/B$	Division
+	$A+B$	Addition, unary +
-	$A-B$	Subtraction, unary -
+	$A\$B\$$	String concatenation

Table B-2  
Logical Operators

Operator	Use	Meaning
NOT	NOT A	Logical opposite of A
AND	A AND B	Logical product of A and B
OR	A OR B	Logical sum of A and B
XOR	A XOR B	Logical exclusive OR of A and B
EQV	A EQV B	Logical equivalence of A and B
IMP	A IMP B	Logical implication of A and B

## BASIC-PLUS-2 LANGUAGE ELEMENTS

Table B-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\$==B\$	A is approximately equal to B; A\$ is exactly equal to B\$

### NOTE

A is approximately equal to B (A==B) if A and B are the same when rounded to six characters (i.e., printed). If A\$ and B\$ are strings, the relation (==) is true if the contents of A\$ and B\$ are the same in length and composition.

### 3. Constants

BASIC-PLUS-2 accepts three types of constants: numeric, string, and integer. Numeric constants are decimal digits in the range of  $10^{-38}$  to  $10^{+38}$  (where n is the constant), which includes E notation. Integer constants are also decimal digits 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: numeric, string, and integer. Numeric 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 per cent sign. If a per cent 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. Keywords are reserved and their use as variable names will produce an error during compilation (see Section B.4).

You designate an array by specifying a numeric, integer, or string variable followed by subscripts in parentheses. Subscripts are in the range of 0 to 32767 and a maximum of

## BASIC-PLUS-2 LANGUAGE ELEMENTS

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 B.3. 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 per cent sign or dollar sign terminator is also allowed for integer and string functions, respectively.

## B.2 COMMANDS

Commands allow you to perform certain operations on the system outside the context of a program. That is, commands do not require a line number. You type them directly to the system along with any legal arguments.

The following is a brief description of the commands, their format, and use. For a more detailed explanation, refer to the RSTS/E System User's Guide or specified sections of this manual.

Command Format	Use	Section
APPEND filespec	Merges a previously saved source program (filespec) with one in memory	1.2.1.1
BREAK line number(s)	Debugging aid used to halt execution at specified program lines.	1.2.3.1
BUILD	Produces a command file from specified object modules. This file contains all of the Task Builder input required to create a task image and memory allocation map.	1.2.1.2
BYE	Terminates session, closes and stores all saved files.	1.1

# BASIC-PLUS-2 LANGUAGE ELEMENTS

Command Format	Use	Section
COMPILE filespec COMPILE/sw	Stores the current program (filespec) as a task image file. This command can be combined with certain switches. If a filename is specified, the program is compiled under that name.	1.2.1.3
CONTINUE	Resumes execution of the program after a halt caused by a debugging aid.	1.2.3
DELETE line number(s)	Erases specified lines from the current program.	1.2.1.4
EXIT	Replaces the current BASIC-PLUS-2 run-time system with the default run-time system.	1.2.1.5
HELLO	Initiates system access.	1.1
HISEG	Switches from a 16K BASIC-PLUS-2 run-time system to one of 4K.	1.2.1.6
IDENTIFY	Prints a header identifying the BASIC-PLUS-2 run-time system.	1.2.1.7
LET var=exp	Debugging aid used to change the contents of variables.	1.2.3.3
LIST[NH]line number(s)	Prints a copy of the current program. If you specify LISTNH, header material is deleted from the copy. If you specify line numbers, only those program lines are printed.	1.2.1.8
LOCK	Sets the COMPILE switch specifications as defaults.	1.2.1.3
LOGIN	Same as HELLO.	
NEW filename	Clears your area for the creation of a program. If you specify a filename, the new program is assigned that name.	1.2.1.9
OLD filename	Recalls a saved program to command level.	1.2.1.10
PRINT var	Debugging aid used to print the current contents of variables.	1.2.3.3
RENAME filename	Changes the name of the current program to the specified name.	1.2.1.11

## BASIC-PLUS-2 LANGUAGE ELEMENTS

Command Format	Use	Section
REPLACE	Saves the current program and substitutes it for one of the same name in memory.	1.2.1.12
RUN[NH] filename	Executes the current program. If you specify RUNNH, header material is not included in the execution. If you specify the name of a saved program, that program is compiled and executed.	1.2.1.13
SAVE filespec	Stores the current program as source code. The program is saved under the current name unless another is specified.	1.2.1.14
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.	1.2.1.15
STEP	Debugging aid used to cause statement-by-statement execution of the program.	1.2.3.2
TRACE	Debugging aid used to track program execution on a line-by-line basis.	1.2.3.4
UNBREAK line number(s)	Debugging aid used to disable the BREAK command. If line numbers are specified, only those breaks are disabled.	1.2.3.1
UNSAVE filespec	Deletes a specified file from your area.	1.2.1.16
UNTRACE	Debugging aid used to disable the TRACE command.	1.2.3.4

### B.2.1 Control Characters

The following list describes the effect of using certain terminal key combinations. The most common combination is to simultaneously press the control key and a letter key such as C. This type of combination is designated as CTRL/x where x is a specified letter. Control keys of this type appear on the terminal output as an uparrow or circumflex followed by the letter. For example, CTRL/C is the same as ^C. For additional information on control characters, refer to the RSTS/E System User's Guide.

## BASIC-PLUS-2 LANGUAGE ELEMENTS

Key	Meaning
CTRL/C	Causes the system to halt the current operation and return to command level. If the operation is the execution of a debugging aid, CTRL/C returns to the debugging prompt. CTRL/C is useful as an exit from an infinite loop.
ESC or ALT MODE	Enters a typed line to the system and prints a dollar sign (\$) character.
CTRL/L	Causes a form feed and generates four line feeds at the terminal.
CTRL/O	Causes a temporary halt of terminal output, but not of execution. If you type another CTRL/O following the first, output continues.
CTRL/Q	Causes output, which was halted by CTRL/S, to resume on a device. Output continues from the point of interruption.
RETURN	Enters a typed line to the system and generates a carriage return/line feed combination.
RUBOUT	Deletes the last character on the current line. One character is deleted each time the key is pressed. On some terminals, this key is labeled DELETE. See Section 1.2.2.
CTRL/S	Causes a temporary halt of device output. Output resumes if you type any character. Note that the typed character, with the exception of CTRL/Q, is also printed.
TAB	Causes a tabulation of eight spaces (also CTRL/I).
CTRL/U	Deletes the current line and performs a carriage return/line feed combination.

### B.3 FUNCTIONS

#### MATH FUNCTIONS

Keyword	Usage
ABS(x)	Returns absolute value of x.
ATN(x)	Returns arctangent of x in radians.
COS(x)	Returns cosine of x in radians.
EXP(x)	Returns value of $e^x$ where $e=2.71828$ .
FIX(x)	Returns truncated value of x.
INT(x)	Returns greatest integer that is less than, or equal to, x.
LOG(x)	Returns natural logarithm of x.

## BASIC-PLUS-2 LANGUAGE ELEMENTS

Keyword	Usage
LOG10(x)	Returns common logarithm of x.
PI	Is a constant value, 3.14159.
RND	Returns random number between 0 and 1. An argument is optional.
SGN(x)	Returns 1 if x is positive, 0 if x is zero, -1 if x is negative.
SIN(x)	Returns sine of x in radians.
SQR(x)	Returns square root of x.
TAN(x)	Returns tangent of x in radians.

### PRINT FUNCTIONS

Keyword	Usage
POS(x\$,y\$,z%)	Returns position of substring y\$ in main string x\$ beginning at position z%.
CCPOS%(x)	Returns current column position for channel x (0 is the user device).
TAB(x%)	Moves print head to position x%.

### STRING FUNCTIONS

Keyword	Usage
ASCII(x\$)	Returns decimal ASCII value for a specified code or the first character of a specified string.
CHR\$(x%)	Returns string equivalent of the ASCII value x%.
COMP%(x\$,y\$)	Compares two numeric strings; returns -1 if x\$ is less than y\$, 0 if x\$ and y\$ are equal, +1 if x\$ is greater than y\$.
DIF\$(x\$,y\$)	Returns difference between two numeric strings (x\$-y\$).
INSTR(x%,y\$,z\$)	Searches for substring z\$ in main string y\$ beginning at position x%; returns 0 or the position of the first character if z\$ is not found.
LEFT(x\$,y%)	Returns a substring of x\$ beginning at the leftmost position for a total length of y% characters (also LEFT\$(x\$,y%).
LEN(x\$)	Returns number of characters in x\$.
MID(x\$,y%,z%)	Returns a substring of x\$ beginning at position y% with a length of z%, also MID\$(x\$,y%,z%).

## BASIC-PLUS-2 LANGUAGE ELEMENTS

Keyword	Usage
NUM	After input of a matrix, NUM contains the number of rows for 2-dimensional arrays or the number of elements for 1-dimensional arrays.
NUM2	After input of a matrix, NUM2 contains the number of elements entered for the last row.
NUM\$(x)	Indicates a string of numeric characters representing the value of x.
NUM1\$(x)	Same as NUM\$ except that E format and spaces are not returned.
PLACE\$(x\$,y%)	Returns a numeric string equal to x\$ but rounded to y% places. Positive y rounds to y significant digits on the right of the decimal, negative y rounds to y significant places on the left of the decimal. For example, where B\$="10364.79306", PLACE\$ returns:  <div style="margin-left: 40px;">PLACE\$(B\$,2%)=10364.79  PLACE\$(B\$,1%)=10364.8  PLACE\$(B\$,-1%)=1037</div>
PROD\$(x\$,y\$,z%)	Returns a numeric string that is the product of x\$*y\$ rounded by z% as shown in PLACE\$.
QUO\$(x\$,y\$,z%)	Returns a numeric string that is the quotient of x\$/y\$ rounded by z% as shown in PLACE\$.
RAD\$(x%)	Converts the integer expression x% from Radix-50 to a string of three ASCII characters.
RIGHT(x\$,y%)	Returns a substring of x\$ that ranges from the character y% to the end of the string, also RIGHT\$(x\$,y%).
SEG\$(x\$,y%,z%)	Returns a substring of x\$ that ranges from character y% to character z%.
SPACE\$(x%)	Produces a string of x% spaces.
STR\$(x)	Returns a string representing the numeric value of x, also NUM\$(x).
STRING\$(x,y)	Creates a string of x length whose characters represent the ASCII value of y.
SUM\$(x\$,y\$)	Creates a string that is the sum of numeric strings x\$+y\$.
TRM\$(x\$)	Returns x\$ with trailing blanks deleted.
VAL(x\$)	Computes the numeric value of the numeric string x\$; x\$ must be acceptable numeric input.
XLATE(x\$,y\$)	Translates x\$ by means of the table string y\$.



## BASIC-PLUS-2 LANGUAGE ELEMENTS

### SYSTEM FUNCTIONS

Keyword	Usage
DATE\$(0%)	Returns the current date in the form dd-mmm-yy.
DATE\$(x%)	Returns date according to the following formula: $x = \text{day of the year} + (\text{years since 1970} * 1000)$ .
ERR	Returns value associated with the last error trapped.
ERN\$	Returns routine name where last error occurred.
ERL	Returns line number of the last error trapped.
MAGTAPE(I1,I2,I3)	Allows program control over magtape operations.
RECOUNT	Returns number of characters provided for a preceding input operation.
TIME\$(0%)	Returns current time.
TIME\$(x%)	Returns time at x minutes before midnight.
TIME(0)	Returns clock time in seconds since midnight.
TIME(1%)	Returns used Central Processing Unit (CPU) time in tenths of seconds.
TIME(2%)	Returns connect time in minutes.
TIME(3%)	Returns kilo-core ticks used by current job.
TIME(4%)	Returns used device time in minutes.

### B.4 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 B-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.

# BASIC-PLUS-2 LANGUAGE ELEMENTS

Table B-4  
Reserved Keywords

ABS	EQ	MAGTAPE	RND
ACCESS	EQV	MAP	RSET
ALLOW	ERL	MAT	SCRATCH
ALTERNATE	ERN\$	MID	SEG\$
AND	ERR	MOD	SEQUENTIAL
APPEND	ERROR	MODE	SGN
AS	ESC	MODIFY	SI
ASCII	EXP	MOVE	SIN
ATN	EXTEND	NAME	SLEEP
ATN2	FF	NEXT	SO
BEL	FIELD	NOCHANGES	SP
BLOCKSIZE	FILE	NODUPPLICATES	SPACE\$
BS	FILESIZE	NONE	SQR
BUCKETSIZE	FILL	NOREWIND	STATUS
BUFSIZ	FILL%	NOSPAN	STEP
CALL	FILL\$	NOT	STOP
CCPOS	FIND	NULL	STR\$
CHAIN	FIX	NUM	STREAM
CHANGE	FIXED	NUM\$	STRING\$
CHANGES	FNEND	NUM1\$	SUB
CHR\$	FOR	NUM2	SUBEND
CLK	FROM	ON	SUM\$
CLOSE	GE	ONERROR	SWAP%
CLUSTERSIZE	GET	OPEN	SYS
COM	GO	OR	TAB
COMMON	GOSUB	ORGANIZATION	TAN
COMP%	GOTO	OUTPUT	THEN
CON	GT	PEEK	TIME
CONTIGUOUS	HT	PI	TIME\$
COS	IND	PLACE\$	TO
COUNT	IF	POS	TRM\$
CR	IMP	PRIMARY	TRN
CVT\$	INDEXED	PRINT	TYP
CVT\$\$	INPUT	PROD\$	TYPE
CVT\$F	INSTR	PUT	UNLESS
CVT%\$	INT	QUO\$	UNLOCK
CVTF\$	INV	RAD\$	UNTIL
DAT	KEY	RANDOM	UPDATE
DATA	KILL	RANDOMIZE	USAGE
DATES	LEFT	READ	USING
DEF	LEN	RECORD	USR
DELETE	LET	RECORDSIZE	VAL
DENSITY	LF	RECOUNT	VARIABLE
DET	LINE	REF	VIRTUAL
DIF\$	LINPUT	RELATIVE	VT
DIM	LOC	REM	WAIT
DIMENSION	LOCKED	RESTORE	WHILE
DUPLICATES	LOF	RESUME	WRITE
EDIT\$	LOG	RETURN	XLATE
ELSE	LOG10	RIGHT	XOR
END	LSET		ZER

# BASIC-PLUS-2 LANGUAGE ELEMENTS

Table B-5 contains a list of the BASIC-PLUS-2 keywords that are used on other systems (i.e., DECsystem-20). To ensure transportability, these language elements are also reserved.

Table B-5  
System Reserved Keywords

ABORT	DAT\$	MAR	QUOTE
ACCESS%	DEL	MAR%	RCTRLC
ALIGNED	DELIMIT	MARGIN	RCTRLLO
ALL	DOUBLEBUF	MID\$	RESET
BACK	ECHO	MOD%	RIGHT\$
BIN\$	FNEXIT	NODATA	SPAN
BINARY	FORCEIN	NOECHO	SQRT
BIT	HANGUP	NOPAGE	SUBEXIT
BROADCAST	IFEND	NOQUOTE	TAPE
BUFFER	IFMORE	NOTAPE	TERMINAL
BUFFERSIZE	IMAGE	NUL\$	TIM
BY	INIMAGE	OCT\$	TYPE\$
CLK\$	INVALID	ONECHR	UNALIGNED
COT	LEFT\$	ONENDFILE	USAGE\$
CTRLC	LINO	PAGE	USR\$
CVT%	LOCK	POS%	VPS%
	LSA	PPS%	WITH

## APPENDIX C

### ERROR MESSAGES

BASIC-PLUS-2 prints a diagnostic message on the terminal 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. 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 error messages printed at the terminal are preceded, in most cases, by either a question mark (?) or a per cent 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.

There are two exceptions to this general format.

If the error occurs during the execution of an INPUT statement, BASIC-PLUS-2 prints a message and automatically reinitiates execution. That is, the INPUT prompt reappears on the terminal.

The other exception is error trapping. Associated with each error is an error variable called ERR (see Section C.3). If error trapping is enabled (i.e., an ONERROR GOTO routine is in effect), an ERR value specification can be used to transfer control to the line number specified by the routine. For information on error handling routines, refer to the BASIC-PLUS-2 Language Manual.

Section C.2 contains the BASIC-PLUS-2 error messages printed at the terminal during compilation. Section C.4 contains the error messages printed at the terminal at run time. 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).

## ERROR MESSAGES

### 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     AZ=FNSZ(3%)
1100     RETURN
1200     DEF FNSZ(E%)
1210     CALL SUBR(E%)
1220     FNSZ=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=FNZZ
2200     RETURN
3000     DEF FNZZ
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
2100     FNE=1/0
2200     FNEND
32767    SUBEND
```

## ERROR MESSAGES

```
RUNNH MAIN
LINE 100
LINE 1100 IN SUBR
LINE 110 IN SUBR2
ZDivision 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 "
```

Ready

Note that the routine SUBR is compiled with the /NOLINE switch enabled.

### C.2 BASIC-PLUS-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.

#### % 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. The program continues and the routine processes the error.

## ERROR MESSAGES

### ?? 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 and a return to 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 all pertinent output.

### ? 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.

### ? FNEND without DEF

FATAL - The compiler encounters an FNEND statement without first encountering a DEF statement. Ensure that the desired function is defined before inserting an 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 subroutine name exceeds six characters or contains a percent sign. 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 FILL specification

FATAL - An attempt is made to define an integer or floating-point length in a 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.

## ERROR MESSAGES

### ? Illegal MAP redefinition

FATAL - An attempt is made to redefine a map. A defined map can be referred to only by its assigned name. Use a different map name for each map definition.

### ? 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\$ for concatenation 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 a subscripted variable utilizes dimensions that differ from those originally defined for it. Ensure that subscripted variables retain the same dimensions throughout the program.



## ERROR MESSAGES

### ? Logical operation on non-integer quantity

FATAL - The program contains an incorrect data type in a logical operation (e.g., A%=B AND C%, where B must be an integer). Use consistent data types in logical operations.

### ? 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 program that contains an illegal transfer into a subprogram. Ensure that subprograms are not nested and do not contain recursive calls.

### ? 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 continues at the line that contains the initial error.

### % RMS I/O forces OBJ output

WARNING - An attempt is made to compile a program that contains an RMS-structured OPEN statement into a task image file. Programs that handle sequential, relative, or indexed files must be compiled as object modules and linked by the Task Builder. The compiler automatically generates an object module when it encounters an RMS-structured OPEN statement.

## ERROR MESSAGES

### ?? 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 a return to 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 all pertinent output.

### ? 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 logic and ensure that each SUBEND statement corresponds to a prior SUB 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 run-time 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 run-time system supplied by DIGITAL. In the event that it does, use a Software Performance Report to report the error to DIGITAL and include all pertinent output.

### ? 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.

### ? TSK output not possible

FATAL - An attempt is made to specify the compilation of a task image file (COM /TSK) where the target file contains a CALL, SUB, or RMS-structured OPEN statement. Recompile the program as an object module.

## ERROR MESSAGES

### % 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.

## ERROR MESSAGES

### C.3 ERROR CODES

Table C-1 contains a list of errors that are recoverable by means of an error handling routine. Included in the table is the value of the ERR variable that is associated with each error. A program can use these error values to differentiate errors. The table also describes the severity of the error and the message that is printed on the terminal. Refer to the alphabetized list in Section C.4 for a description of the error cause and recovery procedures.

Note that four of the errors listed in Table C-1 (34, 36, 37, and 38) cannot be trapped by a program. These errors involve special conditions, which the run-time system cannot control. If these errors occur, you should notify the system manager.

Table C-1  
Recoverable Error Codes

ERR #	Error Class	Message
1	FATAL	?Bad directory for device
2	FATAL	?Illegal file name
3	FATAL	?Account or device in use
4	FATAL	?No room for user on device
5	FATAL	?Can't find file or account
6	FATAL	?Not a valid device
7	FATAL	?I/O channel already open
8	FATAL	?Device not available
9	FATAL	?I/O channel not open
10	FATAL	?Protection violation
11	FATAL	?End of file on device
12	FATAL	?Fatal system I/O failure
13	FATAL	?User data error on device
14	FATAL	?Device hung or write locked
15	FATAL	?Keyboard wait exhausted
16	FATAL	?Name or account now exists
17	FATAL	?Too many open files on unit
18	FATAL	?Illegal SYS() usage
19	FATAL	?Disk block is interlocked
20	FATAL	?Pack ID's don't match
21	FATAL	?Disk pack is not mounted
22	FATAL	?Disk pack is locked out
23	FATAL	?Illegal cluster size
24	FATAL	?Disk pack is private
25	FATAL	?Disk pack needs 'CLEANING'
26	FATAL	?Fatal disk pack mount error
27	FATAL	?I/O to detached keyboard
28	FATAL	?Programmable ^C trap
29	FATAL	?Corrupted file structure
30	FATAL	?Device not file structured
31	FATAL	?Illegal byte count for I/O
32	FATAL	?No buffer space available
33	FATAL	?UNIBUS timeout fatal trap
34	FATAL	?Reserved instruction trap
35	FATAL	?Memory management violation
36	FATAL	?SP (R6) stack overflow
37	FATAL	?Disk error during swap
38	FATAL	?Memory parity failure

(Continued on next page)

# ERROR MESSAGES

Table C-1 (Cont.)  
Recoverable Error Codes

ERR #	Error Class	Message
39	FATAL	?Magtape select error
40	FATAL	?Magtape record length error
41	FATAL	?Non-res run-time system
42	RESERVED	
43	FATAL	?Virtual array not on disk
44	FATAL	?Matrix or array too big
45	FATAL	?Virtual array not yet open
46	FATAL	?Illegal I/O channel
47	FATAL	?Line too long
48	WARNING	%Floating point error
49	WARNING	%Argument too large in EXP
50	WARNING	%Data format error
51	WARNING	%Integer error
52	WARNING	%Illegal number
53	WARNING	%Illegal argument in LOG
54	WARNING	%Imaginary square roots
55	FATAL	?Subscript out of range
56	FATAL	?Can't invert matrix
57	FATAL	?Out of data
58	FATAL	?ON statement out of range
59	FATAL	?Not enough data in record
60	FATAL	?Integer overflow, FOR loop
61	WARNING	%Division by 0
62	FATAL	?No run-time system
63	FATAL	?FIELD overflows buffer
64	FATAL	?Not a random access device
65	FATAL	?Illegal MAGTAPE() usage
66	FATAL	?Missing special feature
67	FATAL	?Illegal switch usage
71	FATAL	?Statement not found
72	FATAL	?RETURN without GOSUB
73	FATAL	?FNEND without function call
74	FATAL	?Undefined function called
75	FATAL	?Illegal symbol
76	FATAL	?Illegal verb
77	FATAL	?Illegal expression
79	FATAL	?Illegal IF statement
80	FATAL	?Illegal conditional clause
81	FATAL	?Illegal function name
82	FATAL	?Illegal dummy variable
83	FATAL	?Illegal FN redefinition
84	FATAL	?Illegal line number(s)
85	FATAL	?Modifier error
86	FATAL	?Can't compile statement
87	FATAL	?Expression too complicated
88	FATAL	?Arguments don't match
89	FATAL	?Too many arguments
90	WARNING	%Inconsistent function usage
91	FATAL	?Illegal DEF nesting
92	FATAL	?FOR without NEXT
94	FATAL	?DEF without FNEND
96	FATAL	?Literal string needed
97	FATAL	?Too few arguments
98	FATAL	?Syntax error

(Continued on next page)

# ERROR MESSAGES

Table C-1 (Cont.)  
Recoverable Error Codes

ERR #	Error Class	Message
99	FATAL	?String is needed
100	FATAL	?Number is needed
101	FATAL	?Data type error
102	FATAL	?1 or 2 dimensions only
103	FATAL	?Program lost-Sorry
104	FATAL	?RESUME and no error
105	FATAL	?Redimensioned array
106	FATAL	?Inconsistent subscript usage
107	FATAL	?ON statement needs GOTO
108	FATAL	?End of statement not seen
109	FATAL	?What?
110	FATAL	?Bad line number pair
111	FATAL	?Not enough available memory
112	FATAL	?Execute only file
113	FATAL	?Please use the RUN command
114	FATAL	?Can't CONTINUE
115	FATAL	?File exists-RENAME/REPLACE
116	FATAL	?PRINT-USING format error
117	FATAL	?Matrix or array without DIM
118	FATAL	?Bad number in PRINT-USING
119	FATAL	?Illegal in immediate mode
120	FATAL	?PRINT-USING buffer overflow
121	FATAL	?Illegal statement
122	FATAL	?Illegal FIELD variable
123	MESSAGE	Stop
124	FATAL	?Matrix dimension error
125	FATAL	?Wrong math package
126	FATAL	?Maximum memory exceeded
127	WARNING	%SCALE factor interlock
128	FATAL	?Tape records not ANSI
129	RESERVED	
130	WARNING	%Key not changeable
131	WARNING	%No current record
132	RESERVED	
133	FATAL	?Illegal usage for device
134	WARNING	%Duplicate key detected
135	FATAL	?Illegal usage
136	FATAL	?Illegal or illogical access
137	FATAL	?Illegal key attributes
138	WARNING	%File is locked
139	RESERVED	
140	FATAL	?Index not initialized
141	FATAL	?Illegal operation
142	FATAL	?Illegal record on file
143	WARNING	%Bad record identifier
144	WARNING	%Invalid key of reference
145	WARNING	%Key size is too large
146	FATAL	?Tape not ANSI labeled
147	WARNING	%RECORD number exceeds maximum
148	FATAL	?Bad RECORDSIZE value on OPEN
149	FATAL	?Not at end of file
150	FATAL	?No primary key specified
151	FATAL	?Key field beyond end of record
152	RESERVED	

(Continued on next page)

## ERROR MESSAGES

Table C-1 (Cont.)  
Recoverable Error Codes

ERR #	Error Class	Message
153	WARNING	%Record already exists
154	WARNING	%Record/bucket locked
155	WARNING	%Record not found
156	WARNING	%Size of record invalid
157	FATAL	?Record on file too big
158	WARNING	%Primary key out of sequence
159	FATAL	?Key larger than record
160	FATAL	?File attributes not matched
161	FATAL	?Move overflows buffer
162	FATAL	?Cannot open file
163	RESERVED	
164	WARNING	%Terminal format file required
165	FATAL	?Cannot position to EOF
166	WARNING	%Negative fill or string length
230	RESERVED	
231	RESERVED	
232	RESERVED	
233	RESERVED	
234	RESERVED	
235	RESERVED	
236	RESERVED	
237	WARNING	%1st arg to SEG\$ > 2nd
238	FATAL	?Arrays must be same dimension
239	FATAL	?Arrays must be square
240	RESERVED	
241	WARNING	%Floating overflow
242	WARNING	%Floating underflow
243	FATAL	?CHAIN to nonexistent line no.
244	WARNING	%Exponentiation error
248	FATAL	?Illegal return from subroutine
249	RESERVED	
250	FATAL	?Not implemented
251	FATAL	?Recursive subroutine call

### C.4 RUN-TIME ERROR MESSAGES

The following alphabetized list describes the error messages printed at the terminal at run time. The description includes the general cause of the error and steps you can take to recover from it. The severity of the error is also noted.

#### ? Account or device in use

FATAL - An operation (such as zeroing or deleting) on a specified device or account cannot be performed because one or more users have current access. Check the current status of the device or account.

#### % Argument too large in EXP

WARNING - The argument values for the EXP function exceed the range of -89 to +88. The value returned is 0.

## ERROR MESSAGES

### ? Arguments don't match

FATAL - The function call arguments differ in quantity or type from those defined for the function. Check the function definition. Change the arguments or definition to conform.

### ? Arrays must be same dimension

FATAL - A matrix addition or subtraction operation is attempted on two or more arrays. The arrays are of different dimensions where the operation requires that they be the same. Redimension the arrays and try the operation again.

### ? Arrays must be square

FATAL - A matrix multiplication operation is attempted on two or more arrays that are not square. The operation requires square arrays. Redimension the arrays and try the operation again.

### ? Bad directory for device

FATAL - A device is referenced that contains a directory in unreadable format. This error can be caused by a magtape label format that differs from the system default or from the format specified in an OPEN statement. Change the format specification in the MODE clause of the OPEN statement or access a different device.

### ? Bad line number pair

FATAL - This error occurs when incorrectly formatted line numbers are specified in a LIST or DELETE command. Correct the desired line number specification.

### ? Bad number in PRINT-USING

FATAL - The format specification in the PRINT-USING string is incorrect. The printing of desired values is disallowed. The correct PRINT-USING string format is described in Section A.1.1.

### % Bad record identifier

WARNING - A random access operation that specifies an illegal record identifier generates this error. The two occurrences that can cause this error are:

1. A random access operation on a relative file contains a zero or negative record number specification. Ensure that a non-zero positive number is used.
2. An indexed file FIND or GET operation contains a null string as a key value specification.

### ? Bad RECORDSIZE value on OPEN

FATAL - The program attempts to open a file with a maximum RECORDSIZE specification of zero. All RMS files require a non-zero RECORDSIZE specification.

### ? Cannot open file

FATAL - This message indicates an operating system error during the open. This error should not occur, but in the event that it does, use a Software Performance Report to report the error to DIGITAL and include all pertinent output.



## ERROR MESSAGES

### ? Cannot position to EOF

FATAL - This error occurs when a sequential file is opened for APPEND access and the operating system is unable to locate the end of the file. One possible cause of this error is a corrupted target file.

### ? Can't compile statement

FATAL - An attempt is made to compile a program containing an excessively complex statement. This may occur, for example, when a statement has too many nesting levels. Examine the program logic and use multi-statement lines to resolve the complexity.

### ? Can't CONTINUE

FATAL - The program halts or ends at a point from which the CONT command is unable to resume execution. Examine the program logic. Reposition the STOP statement(s), where necessary.

### ? Can't find file or account

FATAL - The specified file or account number is not found on the specified device. Ensure that the file and device specifications are correct.

### ? Can't invert matrix

FATAL - This error occurs when an invert operation is attempted on a singular matrix. Examine the program logic for matrix construction type.

### ? CHAIN to nonexistent line no.

FATAL - A chain operation references a line number that does not exist. One possible cause of this error is an attempted chain operation to a program that is compiled with the /NOLINE switch enabled.

### ? Corrupted file structure

FATAL - An error occurs while a CLEAN operation is being performed on a disk. Check the condition of the specified disk pack and replace it, if necessary.

### % Data format error

WARNING - The data specification in a READ or INPUT statement is incorrectly formatted. For example, 1.3 is illegal floating-point number format, 1.2 is illegal integer format, and x" is illegal string format. Examine the data specified in the program.

### ? Data type error

FATAL - The program contains inconsistent or ambiguous data types, i.e., constant or variable data types (floating-point, integer, or character string) that are incorrect for a particular use. Check the program logic. Resolve inconsistent data type usage.

## ERROR MESSAGES

### ? DEF without FNEND

FATAL - The program contains a DEF statement without a corresponding FNEND. Ensure that all multi-line user-defined functions in the program are properly terminated with an FNEND statement.

### ? Device hung or write locked

FATAL - A specified device is unavailable for a requested operation. Possible causes include a line printer out of paper, or a high-speed reader off line. Check the hardware condition of the requested device.

### ? Device not available

FATAL - A requested device currently is reserved by another user or is privileged. Check device assignment status or use another device.

### ? Device not file structured

FATAL - File-structured access is attempted on a non-file structured device (i.e., a device other than disk, DECTape, or magtape). This error occurs, for example, when a directory listing is requested from a non-directory device. Use another device or reformat the program request.

### ? Disk block is interlocked

FATAL - A request for access is made to a locked disk block segment. The segment is locked because it is assigned to another user. Check the status of the desired disk.

### ? Disk error during swap

FATAL - A non-trappable hardware error on the disk occurs while a job is being swapped into, or out of, memory. The content of the job area is lost. However, the job remains logged into the system and is re-initialized to run the NONAME program. Notify the system manager to check the error logging report.

### ? Disk pack is locked out

FATAL - A specified disk pack is loaded in a drive but is temporarily disabled. Check the status of the disk pack or use another drive.

### ? Disk pack is not mounted

FATAL - A disk drive is specified that does not contain a disk or contains a disk that is not logically mounted. Mount a disk pack in the desired drive or use a loaded drive.

### ? Disk pack is private

FATAL - You do not have access privileges to the specified disk. You cannot create a file on the disk without obtaining privileged access. If protection codes permit, you can read files on the disk.

## ERROR MESSAGES

### ? Disk pack needs 'CLEANING'

FATAL - This error indicates that the CLEAN operation in UTILITY must be used on the specified disk after mounting. The disk is mounted, but you should not attempt access until a CLEAN operation resolves any problems in the directory structure.

### % Division by 0

WARNING - This error occurs when the program attempts to divide a quantity by 0. If an error handling routine is present, control is transferred to the specified line number. Otherwise, 0 is returned as the result.

### % Duplicate key detected

WARNING - A PUT operation on an indexed file attempts to write a record that contains one or more key fields that duplicate other record key fields in the file. Because duplicate values were not allowed for one or more of these keys at file creation, the operation fails and this message is generated.

### ? End of file on device

FATAL - An attempt is made to input beyond the end of a data file. Check the length of the data file.

### ? End of statement not seen

FATAL - A program statement contains too many elements and does not process correctly. Simplify the statement by rearranging the elements or using more than one statement line.

### ? Execute only file

FATAL - An attempt is made to add, delete, or list a statement in a compiled file. Compiled files can only be executed. Use a source file for additions, deletions, or listing.

### % Exponentiation error

WARNING - An attempt is made to perform an illegal exponentiation operation. For example, an attempt to raise a number to a power that is outside the legal range (-89 to +88) generates this error. The result of the operation is set to zero and the program continues.

### ? Expression too complicated

FATAL - An expression contains excessive levels of nested parentheses. The allowable depth is dependent on the individual expression. Break up the expression into fewer nested levels.

### ? Fatal disk pack mount error

FATAL - There is a problem with the disk pack such that it cannot be mounted. The disk may have been created on an operating system whose on-disk structure is incompatible with RSTS/E.

## ERROR MESSAGES

### ? Fatal system I/O failure

FATAL - An I/O error that occurs at the system level generates this message. You have no guarantee that the last requested operation was performed. This error should not happen. In the event that it does, use a Software Performance Report to report the error to DIGITAL and enclose all pertinent output.

### ? FIELD overflows buffer

FATAL - An attempt is made to allocate more space with the FIELD statement than exists in the specified buffer. Examine the program and adapt the FIELD specification to available buffer space.

### ? File attributes not matched

FATAL - The file attributes that you specified in the OPEN statement do not match those of the existing target file. This error applies to an inconsistency in one of the following specifications: ORGANIZATION, BUCKETSIZE, BLOCKSIZE, RECORDSIZE, record format, or the number, position, and length of indexed file keys.

### ? File exists-RENAME/REPLACE

FATAL - This error occurs when a SAVE command is given on a file and signifies that the specified filename already exists on the storage medium. Rename the specified file and reinitiate the SAVE or use the REPLACE command.

### % File is locked

WARNING - This error occurs when you attempt to access a file that is locked by another user or by the system. Your program cannot open the file for operations because it is already opened by a program that does not permit shared access.

### % 1st arg to SEG\$>2nd

WARNING - The SEG\$ function contains a beginning string position in the first argument that is greater than the ending string position in the second argument. Reformat the SEG\$ function arguments.

### % Floating overflow

WARNING - During an arithmetic operation, a real value exceeds the largest representable real number (1E38). The result of the operation is set to zero.

### % Floating point error

WARNING - A floating-point underflow has occurred. This happens when the computed floating-point number exceeds the range of 1E-38 to 1E38. If an error handling routine is present, control is transferred to the specified line number. Otherwise, 0 is returned as the value.

### % Floating underflow

WARNING - During an arithmetic operation, a real value becomes less than the smallest representable real number (1E-38). The result of the operation is set to zero.

## ERROR MESSAGES

### ? FNEND without function call

FATAL - The program encounters an FNEND statement without first entering a function. Check that the function is not included among executable statements and ensure that there is a DEF statement for each FNEND in the program.

### ? FOR without NEXT

FATAL - The program encounters a FOR statement without a corresponding NEXT. Ensure that all program FOR loops are terminated by a NEXT statement.

### ? I/O channel already open

FATAL - An attempt is made to open an I/O channel that is already open. Check the specified I/O channel number and the ordering of program statements.

### ? I/O channel not open

FATAL - An attempt is made to perform an input/output operation on a channel that is not open. Examine program statement ordering and include an OPEN statement before the desired I/O request.

### ? I/O to detached keyboard

FATAL - An I/O operation is attempted to a detached keyboard or data set. Check for a hung, or off-line, condition on the hardware.

### % Illegal argument in LOG

WARNING - This error occurs when a negative number or 0 is used as an argument in the LOG function. The argument specified to the function is returned as a value.

### ? Illegal byte count for I/O

FATAL - The specified buffer size is not compatible with the attempted operation. This error occurs when the buffer size specified in the OPEN statement's RECORDSIZE clause (or the PUT statement's COUNT clause) is not a multiple of the accessed I/O device's block size. Examine the specified size and adapt it to the device requirements.

### ? Illegal cluster size

FATAL - The specified cluster size is unacceptable. The cluster size must be a power of 2. For a file cluster, the size must be equal to or greater than the pack cluster size and must not exceed 256. For a pack cluster, the size must be equal to or greater than the device cluster size and must not exceed 16. The device cluster size is fixed by type.

### ? Illegal conditional clause

FATAL - The program encounters an incorrectly formatted conditional clause. Examine the program and ensure that all statements are correct.

## ERROR MESSAGES

### ? Illegal DEF nesting

FATAL - This error occurs when the range of one function definition crosses the range of another. Examine the program and ensure that all DEF ranges are self-contained.

### ? Illegal dummy variable

FATAL - A dummy variable in the DEF statement variable list is not a legal variable name. Examine the program and ensure that all variable names are correct.

### ? Illegal expression

FATAL - The program encounters an expression with illegal characters or format. For example, expressions containing double operators, missing operators, or mismatched parentheses can cause this error. Examine the program and ensure that all expressions are correct.

### ? Illegal FIELD variable

FATAL - The FIELD statement requires a string variable in the argument specification. This error occurs when an unacceptable variable is specified. Examine the program and ensure that the correct variable type is specified.

### ? Illegal file name

FATAL - The filename specification contains illegal characters, embedded blanks, or violates system conventions. Examine the specification for syntax errors and ensure that the specified file exists.

### ? Illegal FN redefinition

FATAL - An attempt is made to redefine a user function. A user-defined function can be defined only once in a program. Use a different function name for each function definition.

### ? Illegal function name

FATAL - The program encounters an illegal function name when attempting to define the function. Check the function definition and ensure that the name is correctly formatted. All user-defined function names must begin with the letters FN.

### ? Illegal IF statement

FATAL - This error occurs when the program encounters an incorrectly formatted IF statement. Check the program and ensure that all IF statements are correct.

### ? Illegal in immediate mode

FATAL - An attempt is made to issue a statement for execution in immediate mode. The issued statement requires a line number (i.e., be part of a program) before it can be executed. Reformat the desired operation to conform with the language requirements.

### ? Illegal I/O channel

FATAL - An attempt is made to open a file on a channel that is outside the legal range. A legal channel number is an integer in the range of 1 to 12.

## ERROR MESSAGES

### ? Illegal key attribute

FATAL - This error occurs when you specify an illegal combination of key characteristics. That is, a NODUPPLICATES and CHANGES specification causes this error. You cannot specify CHANGES without also specifying DUPLICATES.

### ? Illegal line number(s)

FATAL - This error occurs when the specified line number is not in the legal range. The legal line numbers are positive numbers in the range of 1 to 32767.

### ? Illegal MAGTAPE() usage

FATAL - The MAGTAPE function is used improperly. For example, the function cannot be used to perform a file structured operation on a non-file structured device. Ensure that the desired operation and function arguments are compatible.

### % 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 integer and 1E38 to 1E-38 for floating-point.

### ? Illegal operation

FATAL - There are a number of occurrences that can generate this error. These occurrences include:

1. The program attempts a DELETE operation on a sequential file.
2. The program attempts an UPDATE operation on a magnetic-tape file.
3. The program attempts a RSTS/E block I/O operation on an RMS-structured file. Block I/O operations require a VIRTUAL organization file.
4. The program attempts an RMS operation on a RSTS/E block I/O structured file. For RMS operations, the file must be sequential, relative, or indexed.

### ? Illegal or illogical access

FATAL - This error can be caused by one of three occurrences:

1. The attempted record operation is not compatible with the ACCESS clause in the OPEN statement.
2. The ACCESS clause specification is incorrect for the file organization.
3. The READ or APPEND attribute is specified in the ACCESS clause when file creation is attempted.

### ? Illegal record on file

FATAL - The program encounters an illegal record in a sequential file. This error occurs because of an invalid count field in the record.

## ERROR MESSAGES

### ? Illegal return from subroutine

FATAL - The program encounters an external subroutine RETURN statement without previously executing a CALL statement. Examine the program logic and ensure that entrance and exit from routines are correct.

### ? Illegal statement

FATAL - This message occurs when you attempt to execute an incorrect statement. Statements must compile without errors before they can be executed.

### ? Illegal switch usage

FATAL - An attempt is made to specify a switch operation that is illegal or the specification is in illegal format. Possible causes include an error in a CCL command that contains an otherwise valid CCL switch, a file specification switch that is not the last element in the specification, or a missing colon or argument in a file specification.

### ? Illegal symbol

FATAL - The program encounters a line containing an invalid character. For example, a program line containing a commercial at character as an operator (A\$=B\$@C\$) generates this error. Examine the program and ensure that lines contain correct syntax.

### ? Illegal SYS( ) usage

FATAL - An attempt is made to use a SYS system function in an illegal manner. For example, a nonprivileged user attempting a SYS function call for which privilege is required can cause this error. Check the privilege status of the function call.

### ? Illegal usage

FATAL - This error is caused by an inconsistent file attribute specification. The two occurrences that can generate this error are:

1. An attempt is made to open a file whose organization was never declared.
2. A record operation is specified that was never stated in the ACCESS clause.

### ? Illegal usage for device

FATAL - This error can be caused by one of three occurrences:

1. The operation's device specification is in illegal syntax.
2. The specified device does not exist.
3. The specified device is inappropriate for the current operation. For example, an attempt to create an indexed file on magnetic tape generates this error.



## ERROR MESSAGES

### ? Illegal verb

FATAL - A line of input does not contain a valid BASIC verb. Use the correct language element in the line.

### % Imaginary square roots

WARNING - This error occurs when a number less than 0 is the specified SQR function argument. The returned value is the square root of the argument's absolute value.

### ? Inconsistent function usage

FATAL - This error occurs when a function is redefined. The specified argument number or type is inconsistent with the existing calls to that function. Ensure that the redefined function argument is consistent with those that already exist in the program.

### ? Inconsistent subscript usage

FATAL - This error occurs when a subscripted variable utilizes dimensions that differ from those originally defined for it. Ensure that subscripted variables retain the same dimensions throughout the program.

### ? Index not initialized

FATAL - This error occurs on indexed file GET or FIND operations that contain key specifications. The error is caused by an attempted operation on an empty indexed file.

### % Integer error

WARNING - A computed integer exceeds the range of -32767 to +32767. If an error handling routine is present, control is transferred to the specified line number. Otherwise, 0 is returned as the integer value.

### ? Integer overflow, FOR loop

FATAL - The FOR loop integer index exceeds the range of -32767 to +32766. Check the program logic and ensure that the index is within allowable limits.

### % Invalid key of reference

WARNING - This error occurs when the program specifies an indexed file record operation that references an invalid key field. The operations that can generate this error are GET, FIND, and RESTORE.

### ? Key field beyond end of record

FATAL - This error occurs when the program defines a key field record position that exceeds the maximum size of the record. The entire record key must be locatable within the record.

### ? Key larger than record

FATAL - An attempt is made to create an indexed file with a key specification that exceeds the maximum record size.

## ERROR MESSAGES

### % Key not changeable

WARNING - An UPDATE operation is attempted on an indexed file where the replacement record contains one or more key fields that duplicate other record key fields in the file. Because duplicate values were not allowed for one or more of these replacement keys at file creation, the operation fails and this error is generated. To change a key field on UPDATE, you must specify CHANGES for that key in the OPEN statement.

### % Key size too large

WARNING - This error occurs during FIND and GET operations when you specify a key length that equals zero or is larger than the key length defined for the target record.

### ? Keyboard wait exhausted

FATAL - The time specified in a WAIT statement is exhausted with no input received from the specified keyboard.

### ? Line too long

FATAL - This error occurs when a line of input exceeds 255 characters (including line terminators). A line greater than 255 characters causes the buffer to overflow. Shorten or continue the line to conform with buffer limits.

### ? Literal string needed

FATAL - Input of a variable where a numeric or character string is required generates this error. Ensure that string and variable data are used correctly.

### ? Magtape record length error

FATAL - This error occurs when input is performed from magtape. The record on magtape is larger than the buffer designated to receive it. Adapt the buffer size to the length of the desired record.

### ? Magtape select error

FATAL - Access to a magtape drive is denied because the specified unit is off-line. Bring the unit on-line or use another drive.

### ? Matrix dimension error

FATAL - An attempt is made to specify more than two dimensions in a matrix. A matrix requires two dimensions. A syntax error in the DIM statement can also cause this message. Ensure that the matrix dimension is correct in syntax and quantity.

### ? Matrix or array too big

FATAL - This error occurs when an array in memory exceeds the allowed size. Redimension the array to conform with memory size restrictions.

### ? Matrix or array without DIM

FATAL - A reference is made to an array or matrix element outside the range of an implicitly dimensioned matrix. Ensure that element references conform to array or matrix dimensions.

## ERROR MESSAGES

### ? Maximum memory exceeded

FATAL - This error occurs when your program grows too large to run or compile in the memory space assigned to you. You can try to obtain more space from the system manager, build your program as linkable object modules, or reduce program memory requirements.

### ? Memory management violation

FATAL - This is a hardware error and should not occur. In the event that it does, use a Software Performance Report to report the error to DIGITAL and enclose all pertinent output.

### ? Memory parity failure

FATAL - A non-trappable hardware parity error is detected in the program's memory area. Notify the system manager to check the error logging report.

### ? Missing special feature

FATAL - An attempt is made to specify a feature that is not installed on the system. See the system manager for the status of the desired feature.

### ? Modifier error

FATAL - This error occurs when a statement modifier (FOR, WHILE, UNTIL, IF, or UNLESS) is used incorrectly. Check the syntax of these modifiers in your program.

### ? Move overflows buffer

FATAL - The combined length of the MOVE statement I/O list elements exceeds the RECORDSIZE defined for the file. The error occurs when the MOVE statement attempts to place elements in the buffer.

### ? Name or account now exists

FATAL - An attempt is made to rename a file with the name of an existing file. This error also occurs when the system manager attempts to insert an account number that already exists in the system. You should select a different filename and the system manager should use a different account number.

### % Negative fill or string length

WARNING - A MOVE statement I/O list FILL element that is less than zero generates this error. Reformat the MOVE statement I/O list.

### ? No buffer space available

FATAL - File access under RSTS/E file control requires one small buffer for completion of the request. This error occurs when a buffer is not available. If the program is sending messages, two conditions are possible. First, a message is sent, but the receiving program has exceeded the pending message limit. Second, a sending program attempts to send a message, but a small buffer is not available for the operation.

## ERROR MESSAGES

### % No current record

WARNING - This error occurs when a PUT or UPDATE operation is not immediately preceded by a successful GET or FIND operation.

### No logins

This message can be printed on the terminal by the system when it is full, or by the system manager when further logins are disabled. It signifies that no additional users are allowed on the system.

### ? No primary key specified

FATAL - An attempt is made to create an indexed file that does not contain a defined primary key. An indexed file requires the definition of a primary key and allows the definition of up to 254 alternate keys.

### ? No room for user on device

FATAL - A specified device is too full to accept further data, or the device storage space allotted to you is exhausted. Delete unnecessary files from the device.

### ? No run-time system

FATAL - A run-time or object-time system is requested that is not present on the system. Check with the system manager for the availability of the desired RTS or OTS.

### ? Non-res run-time system

FATAL - The specified run-time system is not loaded in memory. Allow the system time to load the run-time system and retry the operation.

### ? Not a random access device

FATAL - This error occurs when random access I/O is attempted on a non-random access device. Use another device, if possible.

### ? Not a valid device

FATAL - The device specification references an illegal or nonexistent device. Check the device specification.

### ? Not at end of file

FATAL - This error occurs when you attempt a sequential file PUT operation without first positioning to the end of the file. An end-of-file position is required prior to a sequential file PUT operation. Note that this error can occur when an existing file is opened for WRITE access.

### ? Not enough available memory

FATAL - This error is caused by the attempted execution of a compiled program that is too large for the memory space assigned to you. You can try to obtain more space from the system manager, recompile the program as linkable object modules, or reduce program memory requirements.

## ERROR MESSAGES

### ? Not enough data in record

FATAL - An INPUT statement is unable to find enough data in one line to satisfy all specified variables. Check the program logic to ensure that there is enough data or delete unnecessary variables.

### ? Not implemented

FATAL - The desired feature is not implemented in the current version of BASIC-PLUS-2 or is not implemented on the current operating system.

### ? Number is needed

FATAL - An attempt is made to use a character string or variable data where a number is required. Check your program for the correct data type usage.

### ? ON statement needs GOTO

FATAL - The program encounters a statement beginning with ON that does not contain a corresponding GOTO or GOSUB clause. Correct the program line.

### ? ON statement out of range

FATAL - An ON GOTO or ON GOSUB index value is less than 1 or greater than the number of listed lines. Examine the program logic. Add lines to the list or adapt the index value to the number of listed lines.

### ? 1 or 2 dimensions only

FATAL - This error results from specifying more than two dimensions to a matrix or more than one to a list. Ensure that the correct subscript is used for the desired array type.

### ? Out of data

FATAL - A READ statement requested more data than is present in the DATA list. Check the program logic. Add more data to the list or use a RESTORE statement.

### ? Pack ID's don't match

FATAL - The specified identification code for a requested disk pack does not match the code stored on the pack. Check the desired disk pack's identification code.

### Please say HELLO

FATAL - This message is printed when you attempt to type anything, other than a legal logged-out command, on the terminal without first logging on the system. Follow the procedures for logging on the system before attempting input.

### ? Please use the RUN command

FATAL - This error occurs when you attempt a transfer of control (as in a GOTO, GOSUB, etc.) in immediate mode that cannot be performed. Incorporate the statement into a program and use the RUN command for execution.

## ERROR MESSAGES

### % Primary key out of sequence

WARNING - This error occurs during a sequential access PUT operation on an indexed file. An attempt is made to write a record that contains a key value less than the previous record's key value.

### ? PRINT-USING buffer overflow

FATAL - An attempt is made to specify a PRINT-USING field format that is too large for the statement. Redesign the field specification to conform with statement limits.

### ? PRINT-USING format error

FATAL - This error results from the use of an incorrect string construction to supply PRINT-USING output format. Examine the program and correct the output format specification.

### ? Program lost-Sorry

FATAL - A fatal system error has occurred that causes the current program to be lost. If you are in EXTEND mode, this error shifts you to NOEXTEND. A hardware problem or the use of an improperly compiled program can cause this error. See the system manager for the status of the system.

### ? Programmable ^C trap

FATAL - A CTRL/C combination is typed while an ON ERROR GOTO statement is in effect and programmable CTRL/C trapping is enabled. The program uses this error to perform special processing. Refer to the RSTS/E Programming Manual for information on CTRL/C trapping.

### ? Protection violation

FATAL - The requested operation can not be performed by the current user. This can be caused by an illegal operation (such as an input request from a line printer) or a privilege violation (such as an attempted deletion of a protected file).

### % Record already exists

WARNING - During a random access PUT operation on a relative file, the program encounters an existing record in the specified position. Relative files allow you to insert records only in unoccupied positions.

### % Record/bucket locked

WARNING - This error occurs when the program attempts an operation on a locked bucket. The target of the operation is locked by another program.

### % Record not found

WARNING - The record specified in a random access GET or FIND operation does not exist. This error applies only to relative and indexed files. Possible causes include a target record that was never written or a previously deleted target record.

## ERROR MESSAGES

### % Record number exceeds maximum

WARNING - There are two conditions that can cause this error:

1. The program attempts to create a relative file where the maximum record number is a negative value.
2. A random access operation on a relative file specifies a record number that exceeds the maximum number of records defined for that file.

### % Record on file too big

WARNING - This error occurs when an accessed record is larger than the buffer area reserved for that record. That is, a GET operation is performed on a record that is larger than the MAP area. Note that RMS considers the operation a success and allows a succeeding UPDATE or DELETE operation.

### ? Recursive subroutine call

FATAL - A subroutine that attempts to call itself generates this error. The recursive call can be direct or through a series of other subroutine calls.

### ? Redimensioned array

FATAL - The array or matrix usage in your program causes an implicit array redimension. Examine the program and restructure the array or matrix usage.

### ? Reserved instruction trap

FATAL - This non-trappable hardware error occurs when you attempt to execute an illegal or reserved instruction. An attempt to execute a Floating Point Processor (FPP) instruction, when floating-point hardware is not available, also causes this error. Check the hardware options available to you.

### ? RESUME and no error

FATAL - This message occurs when the program encounters an error handling routine RESUME statement without first encountering an error. Examine the program to determine the cause of transfer into the routine.

### ? RETURN without GOSUB

FATAL - The program encounters a subprogram RETURN statement without previously executing a GOSUB statement. Examine the main program and ensure a legal entry to the subprogram.

### % SCALE factor interlock

WARNING - An attempt is made to execute a program or source statement with the current SCALE factor. The program executes, but the system uses the SCALE factor of the program in memory. Use the REPLACE and OLD commands, or recompile the program, to execute with the current SCALE factor.

## ERROR MESSAGES

### \* Size of record invalid

WARNING - A PUT or UPDATE operation contains an invalid COUNT specification. The specification is invalid for one of the following reasons:

1. The COUNT specification equals zero.
2. The COUNT specification exceeds the maximum size defined for that file at its creation.
3. The COUNT specification conflicts with the actual size of the current record during a sequential file UPDATE operation on disk.
4. The COUNT specification does not equal the maximum size for fixed-format records.

### ? SP (R6) stack overflow

FATAL - The system attempts to extend the hardware stack beyond the legal size. This is a non-trappable hardware error. Notify the system manager.

### ? Statement not found

FATAL - Reference is made to a program line that does not exist in the program. Examine the program and ensure that all statement references to line numbers are correct.

### Stop

This message appears on the terminal when a STOP statement is executed. Program execution temporarily halts but can be resumed with a CONT (continue) command.

### ? String is needed

FATAL - A variable name or number is used where a character string is required. Examine the program to determine the proper response to statement requirements.

### ? Subscript out of range

FATAL - A reference is made to an array element that is greater than the number of elements dimensioned for the array. Check the program logic and redimension the array, if necessary.

### ? Syntax error

FATAL - A statement is input to the system that is illegal, or incorrectly formatted. Use the proper statement format in program lines.

### ? Tape not ANSI labeled

FATAL - An OPEN statement is attempted on a magnetic tape file that is not ANSI labeled. BASIC-PLUS-2 supports only ANSI-labeled magnetic tape.

### ? Tape records not ANSI

FATAL - This error occurs when you attempt a GET operation on variable-length records from a file that resides on magnetic tape. The records must be in ANSI D format.



## ERROR MESSAGES

### ? Terminal format file required

FATAL - A PRINT or INPUT operation on a non-terminal format file generates this error. PRINT and INPUT operations require a terminal format file.

### ? Too few arguments

FATAL - - An attempt is made to call a function with fewer arguments than are assigned in the DEF statement. Ensure that the number of arguments given in the function call agree with the number specified in the DEF statement.

### ? Too many arguments

FATAL - This error occurs when a user-defined function contains too many arguments. User-defined functions can have a maximum of eight arguments.

### ? Too many open files on unit

FATAL - The number of open files exceeds the number allowed for a particular device unit. On both DECtape and magtape drives, only one open file is allowed. Close the current file and reissue the OPEN statement.

### ? Undefined function called

FATAL - A statement is interpreted as a function call for which there is no defined function. The interpreted call may be attempted to either a system or user-defined function. Examine the program for an illegal statement or missing DEF statement.

### ? UNIBUS timeout fatal trap

FATAL - This is a hardware error and occurs when you attempt to address nonexistent memory or an odd address with the PEEK function. Should there be any other cause, use a Software Performance Report to report the error to DIGITAL and enclose all pertinent output.

### ? User data error on device

FATAL - One or more characters of data are incorrectly transmitted. Possible causes of this message are a parity error, bad punch combination on a card, or similar occurrence.

### ? Virtual array not on disk

FATAL - A virtual array is referenced on a channel containing an open, non-disk device. Virtual arrays are allowed only on disk devices.

### ? Virtual array not yet open

FATAL - An attempt is made to use a virtual array before opening the corresponding disk file. Check the program logic. Ensure that the associated file is open before accessing the array.

## ERROR MESSAGES

### ? What ?

This message appears on the terminal when an illegal or improper command is typed. The command cannot be processed in immediate mode, usually because of a format error. Check the format of the attempted command.

### ? Wrong math package

FATAL - This error occurs when you attempt to compile a program requiring RSTS/E processing that is incompatible with the current system status. An attempt to use a 4-word math package system to execute a program compiled on a system with a 2-word math package can cause this error. Check with the system manager for the availability of the required processing. The program must be recompiled.



# APPENDIX D ASCII CODES AND DATA REPRESENTATION

## D.1 ASCII CHARACTER CODES

Decimal Code	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed, shift, ^P
1	001	SOH	Start of heading, start of message, ^A
2	002	STX	Start of text, end of address, ^B
3	003	ETX	End of text, end of message, ^C
4	004	EOT	End of transmission, shuts off TWX machine, ^D
5	005	ENQ	Enquiry, WRU, ^E
6	006	ACK	Acknowledge, RU, ^F
7	007	BEL	Bell, ^G
8	010	BS	Backspace, format effector, ^H
9	011	HT	Horizontal tab, ^I
10	012	LF	Line feed, ^J
11	013	VT	Vertical tab, ^K
12	014	FF	Form feed, page, ^L
13	015	CR	Carriage return, ^M
14	016	SO	Shift out, ^N
15	017	SI	Shift in, ^O
16	020	DLE	Data link escape, ^P
17	021	DC1	Device control 1, ^Q
18	022	DC2	Device control 2, ^R
19	023	DC3	Device control 3, ^S
20	024	DC4	Device control 4, ^T
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

(Continued on next page)

## ASCII CODES AND DATA REPRESENTATION

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 items in RSTS/E, such as filenames and extensions, are stored in Radix-50 format. This format allows 3 characters of data to be stored as a 2-byte integer (one 16-bit word). The RAD\$() function converts a Radix-50 word to its 3-character representation. Also, the filename string scan SYS calls convert 3-character strings to Radix-50 format.

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 Equivalent (octal)
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-1 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.

$$\begin{aligned} X &= 113000(\text{octal}) \\ 2 &= 002400(\text{octal}) \\ B &= 000002(\text{octal}) \\ X2B &= 115402(\text{octal}) \end{aligned}$$

Note that addition is done in octal.

# ASCII CODES AND DATA REPRESENTATION

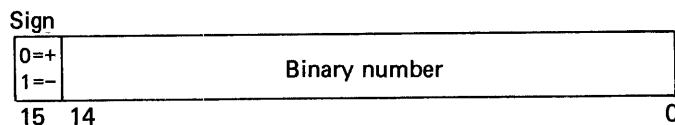
To represent a 3-character string in Radix-50 format, the first character of a string (or a single character) is placed in the leftmost position of the Radix-50 word. Thus, for the character X, its representation 30(octal) is multiplied by  $50^2$  to give 113000(octal), the value shown in Table D-1 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-1 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-1 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-1  
ASCII/Radix-50 Equivalents

First or Single Character	Second Character	Third Character
space 000000	space 000000	space 000000
A 003100	A 000050	A 000001
B 006200	B 000120	B 000002
C 011300	C 000170	C 000003
D 014400	D 000240	D 000004
E 017500	E 000310	E 000005
F 022600	F 000360	F 000006
G 025700	G 000430	G 000007
H 031000	H 000500	H 000010
I 034100	I 000550	I 000011
J 037200	J 000620	J 000012
K 042300	K 000670	K 000013
L 045400	L 000740	L 000014
M 050500	M 001010	M 000015
N 053600	N 001060	N 000016
O 056700	O 001130	O 000017
P 062000	P 001200	P 000020
Q 065100	Q 001250	Q 000021
R 070200	R 001320	R 000022
S 073300	S 001370	S 000023
T 076400	T 001440	T 000024
U 101500	U 001510	U 000025
V 104600	V 001560	V 000026
W 107700	W 001630	W 000027
X 113000	X 001700	X 000030
Y 116100	Y 001750	Y 000031
Z 121200	Z 002020	Z 000032
\$ 124300	\$ 002070	\$ 000033
. 127400	. 002140	. 000034
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

## ASCII CODES AND DATA REPRESENTATION

### 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 0's. 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

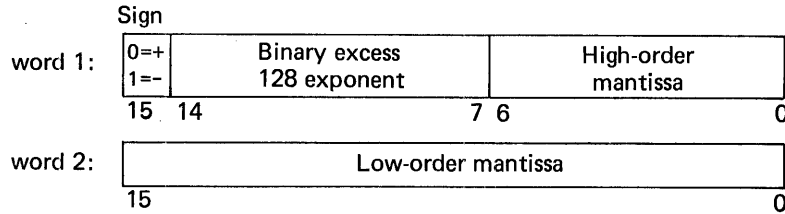
in the 2-word format, or:

140640  
0  
0  
0

in the 4-word format.

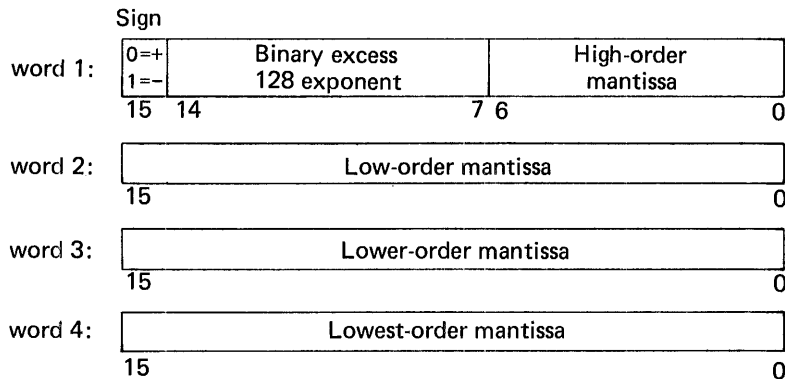
## ASCII CODES AND DATA REPRESENTATION

### 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  $.29 \times 10^{-38}$  and  $.17 \times 10^{39}$ .

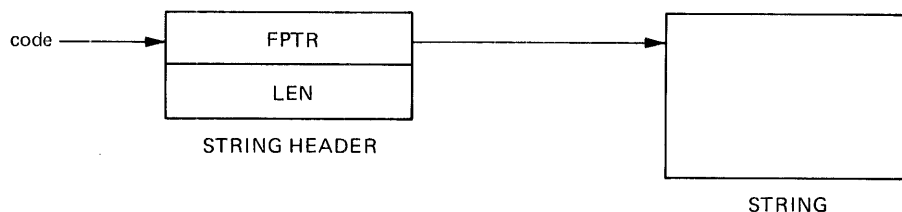
### D.4.2 DOUBLE-PRECISION Format (4-Word Floating-point)



The effective precision is 56 bits (or approximately 17 decimal digits of accuracy). The magnitude range lies between  $.29 \times 10^{-38}$  and  $.17 \times 10^{39}$ .

## D.5 STRING AND ARRAY FORMAT

### D.5.1 Dynamic String Format



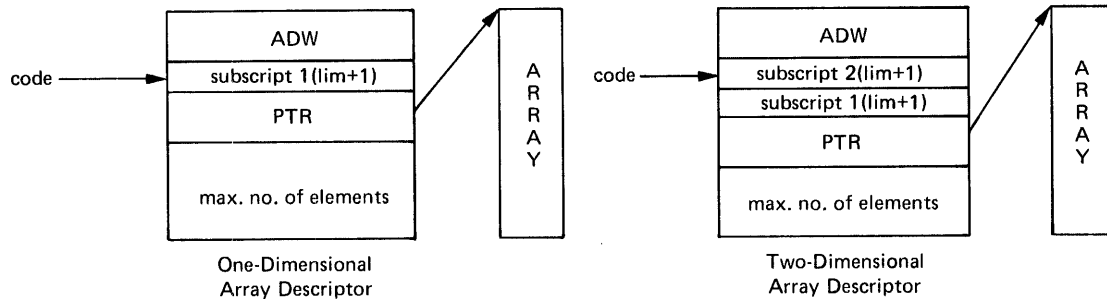
The code for dynamic strings contains a 2-word string header. The first word is a forward pointer (FPTR) that points to the first byte of the string. The second word represents the length (LEN) of the string in bytes.



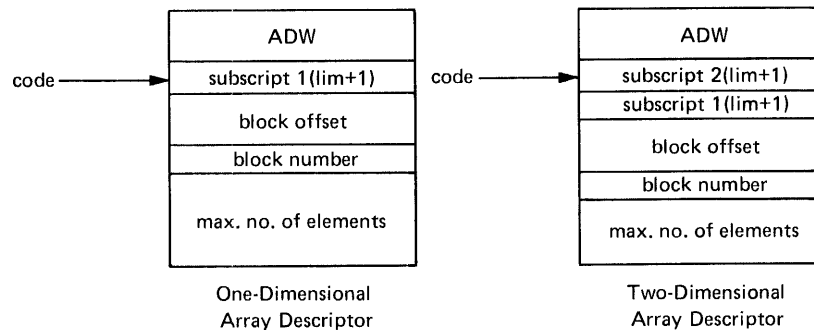
## ASCII CODES AND DATA REPRESENTATION

### D.5.2 Array Format

#### Arrays in Memory:



#### Virtual Arrays:



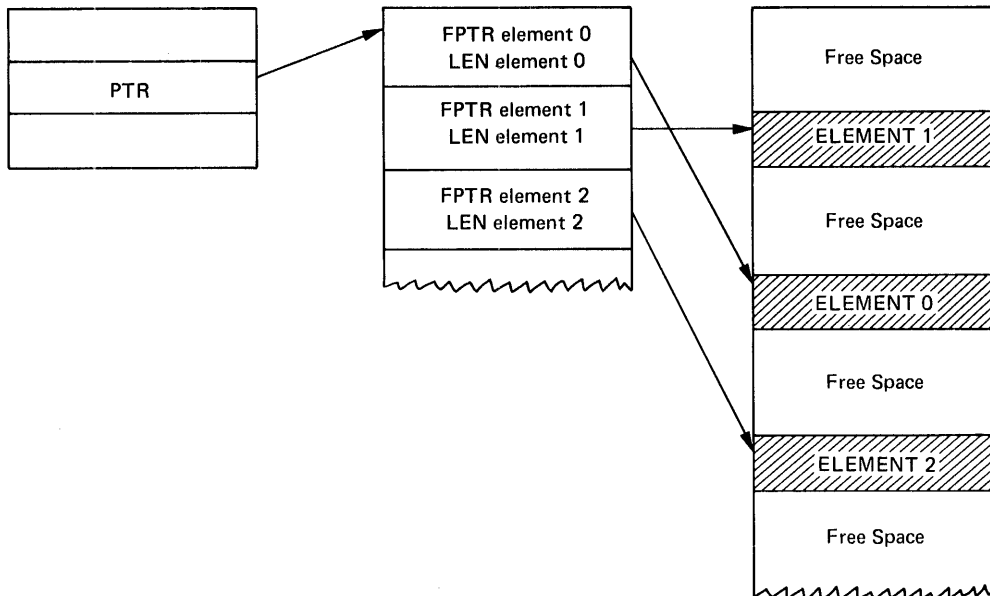
ADW is the Array Descriptor Word and is explained in Section D.5.3. Subscript is a word that represents the limits defined by the array subscripts plus 1.

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:

## ASCII CODES AND DATA REPRESENTATION



### D.5.3 Array Descriptor Word

Table D-2  
Array Descriptor Word

Array Type	Bits															
	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 <sub>2</sub> (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-2. 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.

## ASCII CODES AND DATA REPRESENTATION

- 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

Note that an underlined page number points to an entry's primary or definitive reference.

- Abbreviated BASIC-PLUS-2
  - commands, 1-11
  - NH, 1-11
- ABORT option, 2-6
  - format, 2-6
  - see also Task Builder options
- ABS function, B-7
- ABSPAT option, 2-7
  - format, 2-7
  - see also Task Builder options
- ACCESS file attribute, 4-6, 4-8, 4-11
- Access,
  - example of generic key, 4-15
  - example of record, 4-20
  - indexed file record, 4-14
  - random, 4-5
  - random record, 4-18, 4-19
  - record, 4-4
  - sequential, 4-5
  - sequential record, 4-18
  - shifting record, 4-21
  - virtual file data, 4-2
- ACCESS APPEND, 4-16
- ACCESS MODIFY, 4-16
- ACCESS READ, 4-16
- ACCESS SCRATCH, 4-16
- ACCESS WRITE, 4-16
- ALLOW file attribute, 4-3, 4-5, 4-8, 4-11
- ALLOW MODIFY, 4-16
- ALLOW NONE, 4-16
- ALLOW READ, 4-16
- ALLOW WRITE, 4-16
- ALT mode, B-7
  - see also ESC key
- ALTERNATE file attribute, 4-11, 4-12
- Alternate key,
  - definition, 4-13
  - duplicate key, 4-14
  - modification of, 4-14
  - numbering of, 4-13
  - record access by, 4-14
  - values, 4-14
- Ambiguous constant, 5-3, A-4
  - BASIC-PLUS, 5-3, A-4
  - BASIC-PLUS-2, 5-3, A-4
  - translation of, 5-3
- Ampersand continuation character, 1-28
- ANSI magnetic tape, 4-23
- Apostrophe,
  - PRINT USING, A-2
- APPEND file attribute, 4-16
  - ACCESS, 4-16
- APPEND (APP) command, 1-9, 1-11, B-4
  - example of, 1-11
  - use of, 1-11
- APPEND FILE NAME prompt, 1-11
- Approximate key, 4-14, 4-20
  - GE for, 4-14
  - GT for, 4-14
  - specification, 4-14, 4-15
- Argument list format, 3-11
- Argument passing,
  - array, 3-12
  - double, 3-11
  - integer, 3-11
  - real, 3-11
  - string, 3-11
  - subroutine, 3-10
- Argument range,
  - STEP command, 1-27
- Argument specification,
  - BREAK command, 1-26
- Arguments,
  - BREAK command maximum, 1-26
  - /LB switch, 2-5
  - STEP command, 1-27
- Arguments in functions,
  - null, 5-4
- Arithmetic operators, B-2
  - table of, B-2
- Array argument passing, 3-12
- Array descriptor word, D-8, D-9
- Array format, D-7, D-8
- Array subscripts, A-9
  - evaluation of, A-9
- Arrays,
  - format of virtual, D-8
  - format in memory, D-8
  - virtual, 4-2
- ASCII character codes, D-1
  - to D-3
- ASCII character set, B-2
- ASCII collating sequence, 4-14
- ASCII file,
  - stream, 4-22
  - stream data, 4-22
- ASCII function, B-8
- ASCII/Radix-50 equivalents,
  - table of, D-5
- ASG option, 2-7
  - format, 2-7
  - see also Task Builder options
- Assignment statement,
  - multiple, 5-3, A-3
- ATN function, B-7
- Attributes,
  - virtual file, 4-3

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Backslash statement separator, B-1
- Backslashes, PRINT USING, A-2
- BASIC-PLUS,
  - ambiguous constant, 5-3, A-4
  - CCPOS function, A-5
  - CHAIN statement, 5-2, A-7
  - comment, 5-3
  - continued line, 5-1, A-6
  - DATA statement, 5-2, A-5
  - DEF statement, 5-1, A-4
  - file compatibility, 4-24
  - POS function, 5-2, A-5
  - preserving programs, 5-1
  - PRINT USING format, 5-2
  - statement separator, 5-2
  - string literal, 5-3, A-3
  - syntax items, 5-1
  - SYS functions, 5-3
  - user-defined function, A-5
  - variable name, 5-2, A-7
- BASIC-PLUS-2,
  - ambiguous constant, 5-3, A-4
  - CHAIN statement, 5-2, A-7
  - command file, 1-9
  - comments, 5-3
  - continued line, 5-1, A-6
  - DATA statement, 5-2, A-5
  - DEF statement, 5-1, A-4
  - line number, 1-28
  - line terminator, 1-28
  - POS function, 5-2, A-5
  - PRINT USING format, 5-2, A-1
  - source program, 1-28, B-1
  - statement separator, 5-2
  - string literal, 5-3, A-3
  - syntax, 5-1
  - user-defined function, A-5, B-1
  - variable name, 5-2, A-7
- BASIC-PLUS-2 commands, 1-8
  - abbreviated, 1-11
  - table of, 1-9, 1-10, B-7
- BASIC-PLUS-2 compiler, 1-8
  - header, 1-18
  - invocation of, 1-8
  - leaving, 1-17
  - syntax check, 1-23
  - use of, 1-8
- BASIC-PLUS-2 debugging aid, 1-14
- BASIC-PLUS-2 editing methods, 1-23
- BASIC-PLUS-2 extensions, table of, 1-5
- BASIC-PLUS-2 function names, A-7
- BASIC-PLUS-2 library, 3-1, 3-2
- BASIC-PLUS-2 run-time system (RTS), 1-17, 3-1
  - purpose of, 3-1
- BASIC-PLUS-2 subroutines, 3-3
  - restrictions, 3-10
- BASIC-PLUS-2 Record I/O operations, A-10
- BASIC2 run-time system, 1-17, 2-8, 3-1, 3-2
  - content of, 1-17, 3-1
  - library, 3-1
  - program size limit, 3-2
  - size of, 1-17
  - use of RMS with, 3-2
- Block,
  - length of, 4-25
- Block boundaries, 4-25
  - crossing, 4-25
- Block I/O,
  - file, 4-1
  - operations, 4-2, A-10
- Block length definition, 4-25
- Block modification, 4-16
- Block size,
  - on disk, 4-24
  - on magnetic tape, 4-24
- Blocks,
  - contiguous unit of, 4-6
  - records on, 4-25
- BLOCKSIZE file attribute, 4-5, 4-6, 4-25
  - specification, 4-25
- BP2COM run-time system, 1-17, 2-8, 3-1, 3-2
  - access to, 3-2
  - contents of, 1-18, 3-1
  - library, 3-1, 3-2
  - program size limit, 3-2
  - size of, 1-18
  - use of, 3-2
- BREAK debugging command, 1-25, B-4
  - argument, 1-25
  - argument specification, 1-26
  - error message, 1-26
  - halt on CALL, 1-26
  - halt on DEF, 1-26
  - halt on LOOP, 1-26
  - maximum arguments, 1-26
- BREAK ON command, 1-27
- Breakpoints, 1-25
- Bucket, 4-16, 4-24, 4-25
  - composition of, 4-25
  - locked, 4-16
  - size of, 4-24
- Bucket overhead, RMS, 4-28

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Bucket size, 4-25
  - default, 4-25 to 4-28
  - definition, 4-25
  - establishing, 4-26
  - for indexed files, 4-27
  - for relative files, 4-26
  - formula, 4-26 to 4-28
  - indexed file default, 4-28
  - large, 4-29
  - relative file default, 4-27
  - small, 4-29
- BUCKETSIZE file attribute,
  - 4-8, 4-11
  - specification, 4-26
- Buffer name, 4-29
- Buffer space,
  - allocation of, 4-29
  - requirements, 4-26
- Buffers, 4-29
  - content of, 4-29
- BUILD (BUI) command, 1-9,
  - 1-12, 2-3, 2-8, B-4
  - contents of file, 1-12
  - default run-time system, 1-18
  - format, 1-13
  - overlay creation, 2-10, 2-13
  - RMS switches, 1-13
- CALL BY REF statement, 3-11, A-9
  - example, 3-13
- Call instructions,
  - subroutine, 3-10, 3-11
- CALL statement, 3-11, A-9
  - BREAK command halt on, 1-26
  - example, 3-12
- Calling conventions,
  - subroutine, 3-10
- Calls to overlay sections, 2-13
- CCL command, 1-8
  - Task Builder, 2-1
  - use of, 1-8
- CCPOS function, 5-2, B-8
  - BASIC-PLUS, A-5
- CHAIN statement, 5-2, A-7
  - BASIC-PLUS, 5-2, A-7
  - BASIC-PLUS-2, 5-2, A-7
  - translation of, 5-2
- CHANGES file attribute, 4-11,
  - 4-12, 4-14
  - key, 4-14
- Channel number range, 4-2
- Character codes,
  - ASCII, D-1 to D-3
- Character set, B-2
  - ASCII, B-2
  - RADIX-50, D-4
- CHR\$ function, B-8
- Circumflex, viii
- Cluster size,
  - file, 1-7
  - negative, 1-7
- CLUSTERSIZE file attribute, 4-5,
  - 4-6, 4-8, 4-9, 4-12
  - /CLUSTERSIZE switch option, 1-7
- Codes,
  - ASCII character, D-1 to D-3
  - combination of protection, 1-6
  - default protection, 1-6
  - device, 1-3
  - protection, 1-5
  - specification of protection, 1-6
  - table of protection, 1-5, 1-6
  - table of recoverable error, C-9 to C-12
- Collating sequence,
  - ASCII, 4-14
- Command,
  - APPEND, 1-9, 1-11, B-4
  - BREAK debugging, 1-25, B-4
  - BREAK ON, 1-27
  - BUILD, 1-9, 1-12, 2-3, 2-8, B-4
  - BYE, 1-2, B-4
  - COMPILE, 1-9, 1-14, B-5
  - COMPILE /DEBUG, 1-14
  - COMPILE /DOUBLE, 1-14
  - COMPILE /MACRO, 1-14
  - COMPILE /NOCHAIN, 1-14
  - COMPILE /NOLINE, 1-14
  - COMPILE /OBJECT, 1-12, 1-15
  - COMPILE /TSK, 1-14
  - CONTINUE, 1-24, B-5
  - DELETE, 1-10, 1-17, 1-24, B-5
  - EXIT, 1-10, 1-17, B-5
  - HELLO, 1-1, B-5
  - HISEG, 1-10, 1-17, 3-1, 3-2, B-5
  - IDENTIFY, 1-10, 1-18, B-5
  - LET debugging, 1-27, B-5
  - LIST, 1-10, 1-19, B-5
  - LOCK, 1-10, 1-15, B-5
  - LOGIN, B-5
  - NEW, 1-10, 1-19, B-5
  - OLD, 1-10, 1-20, B-5
  - PRINT debugging, 1-27, B-5
  - RENAME, 1-10, 1-20, B-5
  - REPLACE, 1-10, 1-21, B-6
  - RUN, 1-10, 1-21, B-6
  - SAVE, 1-10, 1-22, B-6
  - SCALE, 1-10, 1-22, B-6
  - STEP debugging, 1-27, B-6
  - TRACE debugging, 1-28, B-6

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Command (Cont.),
  - UNBREAK debugging, 1-25, 1-26, B-6
  - UNSAVE, 1-10, 1-22, B-6
  - UNTRACE debugging, 1-28, B-6
- Command file,
  - BASIC-PLUS-2, 1-9
  - contents of BUILD, 1-12
  - translator, 5-6
- Command file extension, 2-3
- Command set,
  - support of system, 1-9
- Command summary, B-4 to B-6
- Commands, B-4
  - abbreviated BASIC-PLUS-2, 1-11
  - BASIC-PLUS-2, 1-8, B-1
  - CCL, 1-8
  - debugging, 1-24
  - debugging aid, 1-24
  - summary of, B-1
  - table of BASIC-PLUS-2, 1-9, 1-10
  - use of CCL, 1-8
- Comment delimiter, B-1
- Comment separator, 1-29, A-6
  - translation of, 5-3
- Comments, 1-29, B-1
  - BASIC-PLUS, 5-3
  - BASIC-PLUS-2, 5-3
- COMP% function, B-8
- Compatibility,
  - BASIC, A-1
  - BASIC-PLUS file, 4-24
  - nontranslatable issues, A-8
- Compilation,
  - program, 1-14
- COMPILE /DEBUG command, 1-14
- COMPILE /DOUBLE command, 1-14
- COMPILE /MACRO command, 1-14
- COMPILE /NOCHAIN command, 1-14
- COMPILE /NOLINE command, 1-14
- COMPILE /OBJECT command, 1-12, 1-15
- COMPILE /TSK command, 1-14
- COMPILE command, 1-9, 1-14, B-5
  - switches, 1-9, 1-14
  - switches combined, 1-15
  - warning message, 1-14
- Compile-time error messages, C-1, C-3
- Compile-time errors, A-9
  - summary of, C-3 to C-8
- Compiler,
  - BASIC-PLUS-2, 1-8
  - syntax check, 1-23
  - use of, 1-8
- Compress switch (/CO), 3-8
  - example of, 3-9
  - format of, 3-8
  - see also Librarian
  - specification, 3-8
- Concise Command Language (CCL), 1-8
- Constant,
  - ambiguous, 5-3, A-4
  - BASIC-PLUS, A-4
  - BASIC-PLUS-2, A-4
  - integer, B-3
  - numeric, 5-4, B-3
  - range of integer, B-3
  - range of numeric, B-3
  - spaces in numeric, 5-4
  - string, B-3
  - tabs in numeric, 5-4
  - translation of ambiguous, 5-3
  - translation of numeric, 5-4
- CONTIGUOUS file attribute, 4-5, 4-6
- Continuation character,
  - ampersand, 1-28
- Continuation lines, 1-28, 5-1, B-1
  - BASIC-PLUS, 5-1, A-6
  - BASIC-PLUS-2, 5-1, A-6
  - translation of, 5-1
- CONTINUE (CON) command, 1-24, B-5
- Control characters, B-6
  - summary of, B-7
- COS function, B-7
- Count field, 4-23, 4-27
  - on disk, 4-23
  - on magnetic tape, 4-23
  - word alignment, 4-23
- Create switch (/CR), 3-4
  - example of, 3-4
  - format, 3-4
  - see also Librarian
- CTRL/C, B-7
- CTRL/L, B-7
- CTRL/O, B-7
- CTRL/Q, B-7
- CTRL/S, B-7
- CTRL/U, 1-23, B-7
- CVT file operation, 4-3
- D format, 4-23
- /DA switch, 2-5
  - see also Task Builder switches
- Data,
  - ASCII stream, 4-22
  - assignment of, 4-4

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Data (Cont.),
  - stream-format file, 4-24
  - transmission of file, 4-4
  - virtual file, 4-2
- Data access,
  - virtual file, 4-2
- Data field, 4-13
  - as key, 4-13
  - secondary, 4-13
- Data format, B-1
- DATA statement, 5-2, A-5
  - BASIC-PLUS, 5-2, A-5
  - BASIC-PLUS-2, 5-2, A-5
  - translation of, 5-2
- Data storage,
  - virtual file, 4-2
- Data structure, 4-4, 4-24
- DATE\$ function, B-10
- /DEBUG switch, 1-14, 1-24
  - memory requirements, 1-24
- Debugging aid,
  - BASIC-PLUS-2, 1-14
  - commands, 1-24
  - execution halts, 1-26
  - message, 1-24
  - prompt, 1-24
  - termination, 1-25
  - variable change, 1-27
  - variable examination, 1-27
- Debugging command,
  - BREAK, 1-25
  - LET, 1-27
  - PRINT, 1-27
  - STEP, 1-27
  - TRACE, 1-28
  - UNBREAK, 1-25
  - UNTRACE, 1-28
- Debugging subroutines, 1-24
- DEF statement, 5-1, A-4
  - BASIC-PLUS, 5-1, A-4
  - BASIC-PLUS-2, 5-1, A-4
  - BREAK command halt on, 1-26
  - function references, A-4
  - translation of, 5-1
- Default bucket size, 4-25 to 4-28
  - indexed file, 4-28
  - relative file, 4-27
- Default device,
  - public structure, 1-3
- Default extension, 1-3
  - object module, 2-1
- Default high segment, 2-8
- Default protection codes, 1-6
- Default record format, 4-6, 4-22, 4-23
- Default run-time system, 1-17
  - BUILD command, 1-18
- Default switches,
  - Task Builder, 2-5
- DELETE (DEL) command, 1-10, 1-17, 1-24, B-5
  - use of, 1-17
- DELETE key, 1-23
- DELETE file operation, 4-19
  - indexed file, 4-13
  - relative file, 4-10
- Delete switch /DE, 3-8
  - format, 3-8
  - message, 3-8
  - see also Librarian
  - termination, 3-8
- Delimiter,
  - (CR/LF) record, 4-24
  - (ESC) record, 4-24
  - (FF) record, 4-24
  - (LF) record, 4-24
  - (VT) record, 4-24
  - comment, B-1
  - exclamation point, A-6
  - stream-format record, 4-24
  - string, 5-3
- Device,
  - not allowed for RMS, 1-4
  - public structure default, 1-3
  - use of, 1-3
- Device assignment,
  - Task Builder, 2-7
- Device codes, 1-3
- Device names,
  - logical, 1-3
  - physical, 1-3
- Device specifications,
  - table of, 1-4
- DIF\$ function, B-8
- DIM # statement, 4-3
- Directive,
  - .END, 2-12
  - .FCTR, 2-12
  - .ROOT, 2-12
- Directive format,
  - ODL, 2-10, 2-12
- Disk,
  - block size on, 4-24
  - count field on, 4-23
- Documentation conventions, vii
- Double precision,
  - format, 1-14, D-7
  - range of, 1-29
- /DOUBLE switch, 1-14
- Duplicate keys, 4-14
  - alternate, 4-14
  - primary, 4-14
- DUPLICATES file attribute, 4-11, 4-12, 4-14
- Dynamic string format, D-7



## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Editing,
  - ODL file, 2-13
  - BASIC-PLUS-2 program, 1-23
  - .END directive, 2-12
  - END statement, 1-28
  - ENTER OPTIONS: prompt, 2-3
  - Entry point table (EPT), 3-3, 3-4
  - EQ for exact key, 4-14
  - Erasing unwanted files, 1-23
  - Erasures,
    - RUBOUT key, 1-23
  - ERL function, B-10
  - ERN\$ function, B-10
  - ERR function, B-10, C-1
    - value specification, C-1
    - variable, C-9
  - Error,
    - fatal, C-1
  - Error codes,
    - table of recoverable, C-9
    - to C-12
  - Error handling routines, C-1
  - Error message,
    - BREAK command, 1-26
    - Insert switch, 3-5
    - /NOLINE diagnostic, 1-15
  - Error message format, C-1
  - Error messages, C-1
    - compile-time, C-1, C-3
    - run-time, C-12
  - Error trace in functions, C-2
  - Error trace in subroutines,
    - C-2
  - Error trapping, C-1
  - Errors,
    - compile-time, A-9
    - summary of compile-time, C-3
    - to C-8
    - summary of run-time, C-12
    - to C-31
  - ESC key, B-7
  - Exact key, 4-14, 4-20
    - EQ for, 4-14
    - specification, 4-14
  - Exclamation point,
    - PRINT USING, A-2
  - Exclamation point separator,
    - 1-29, A-6
  - Executable task,
    - creation of, 1-14
    - production of, 1-8
  - Execution,
    - examine the path of, 1-28
    - program, 1-21
    - task, 2-14
  - Execution halts,
    - debugging commands, 1-26
  - Existence byte RMS, 4-27
  - EXIT (EXI) command, 1-10, 1-17, B-5
    - Exit from system program, 1-8
  - EXP function, B-7
  - EXTEND mode, 5-5
    - conversion of, 5-1
    - prompt, 5-5
  - Extending indexed files, 4-17
  - Extending relative files, 4-17
  - Extension,
    - command file, 2-3
    - default, 1-3
    - filename, 1-4
    - input object module, 2-3
    - library file, 2-3
    - memory allocation map, 2-3
    - object library, 3-4
    - object module default, 2-1
    - overlay description file, 2-3
    - task image file, 2-3
  - Extensions,
    - table of BASIC-PLUS-2, 1-5
  - Extract switch (/EX), 3-5, 3-6
    - example of, 3-7
    - format, 3-5
    - see also Librarian
  - EXTTSK option, 2-7
    - format, 2-8
    - see also Task Builder options
  - Fatal error, C-1
  - .FCTR directive, 2-12
  - Field,
    - count, 4-23, 4-27
    - data, 4-13
    - secondary data, 4-13
  - FIELD file operation, 4-3
  - File,
    - BASIC-PLUS-2 command, 1-9
    - contents of BUILD command, 1-12
    - creation of indexed, 4-10
    - creation of relative, 4-8
    - creation of sequential, 4-5
    - creation of stream-format, 4-24
    - creation of terminal-format, 4-2
    - creation of virtual, 4-2
    - editing ODL, 2-13
    - example of indexed, 4-12
    - example of relative, 4-9
    - example of sequential, 4-7
    - example of virtual, 4-3
    - indexed, 4-2, 4-10
    - inserting modules in library, 3-5

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- File (Cont.),
  - listing of library, 3-7
  - listing of map, 2-10
  - MACRO source, 1-14
  - ODL, 2-10
  - organization, 4-2
  - overlay description, 2-5
  - overlay description language, 1-12
  - record I/O, 4-1
  - relative, 4-2, 4-8
  - sequential, 4-2, 4-5
  - storage of, 4-24
  - stream ASCII, 4-22
  - task image, 2-9
  - terminal-format, 4-2
  - translator command, 5-6
  - virtual, 4-2
- File attribute, 4-1
  - ACCESS, 4-3, 4-5, 4-8, 4-11
  - ALLOW, 4-3, 4-5, 4-8, 4-11
  - ALTERNATE, 4-11
  - BLOCKSIZE, 4-5
  - BUCKETSIZE, 4-8, 4-11
  - CHANGES, 4-11
  - CLUSTERSIZE, 4-5, 4-8, 4-11
  - CONTIGUOUS, 4-5
  - DUPLICATES, 4-11
  - MAP, 4-3, 4-5, 4-8, 4-11
  - NOCHANGES, 4-11
  - NODUPLICATES, 4-11
  - NOREWIND, 4-5
  - NOSPAN, 4-5
  - PRIMARY, 4-11
  - RECORDSIZE, 4-3, 4-5, 4-8, 4-11
  - SPAN, 4-5
- File cluster sizes, 1-7
- File compatibility,
  - BASIC-PLUS, 4-24, A-10
- File compression,
  - library, 3-8
- File description header, 4-22
- File extension,
  - command, 2-3
  - library, 2-3
  - overlay description, 2-3
  - task image, 2-3
- File format,
  - indexed, 4-10
  - relative, 4-8
  - sequential, 4-5
  - virtual, 4-3
- File keys,
  - definition of, 4-29
  - length of, 4-29
  - position of, 4-29
- File media restriction,
  - indexed, 4-2
  - relative, 4-2
  - sequential, 4-2
- File modification,
  - library, 3-5
  - ODT, 2-10
- File operation,
  - CVT, 4-3
  - FIELD, 4-3
  - LSET, 4-3
  - RSET, 4-3
- File organization, 4-1, 4-4
  - assignment of, 4-2
  - comparison of, 4-5
  - default, 4-2
  - indexed, 4-10
  - relative, 4-8
  - sequential, 4-5
  - space in memory, 4-17
  - specification of, 4-2
  - types of, 4-1
  - virtual, 4-2, A-10
- File restrictions,
  - sequential, 4-7
- File sharing, 4-15
  - control of, 4-15
  - sequential, 4-15
- Filename, 1-3
  - extension, 1-4
  - specification format, 1-3, 4-1
- Files, 4-1
  - bucket size for indexed, 4-27
  - bucket size for relative, 4-26
  - erasing unwanted, 1-23
  - extending indexed, 4-17
  - extending relative, 4-17
  - FIND operations on indexed, 4-20
  - GET operation on indexed, 4-20
  - library, 3-3
  - operations on relative, 4-10
  - random access on indexed, 4-20
  - random access on relative, 4-19
  - record access on sequential, 4-18
  - RMS memory allocation for, 4-17
  - sequential PUT on relative, 4-19
  - sharing virtual, 4-16
  - terminal I/O, 4-24
- Files on magnetic tape,
  - sequential, 4-25

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Files with variable format,
  - indexed, 4-23
  - relative, 4-23
- /FILESIZE switch option, 1-6, 1-7
- FIND operation, 4-18, 4-19
  - indexed file, 4-13, 4-20
  - relative file, 4-10
  - sequential file, 4-7
- First record position, 4-9
- FIX function, B-7
- Fixed-length format, 4-22
- Fixed-length records, 4-21, 4-22
- Floating-point formats, D-6
- Floating-point operations, 1-14
- FOR NEXT loops,
  - transfer into, A-9
- Formula,
  - bucket size, 4-26 to 4-28
  - RMS memory allocation, 4-17
- Function,
  - ABS, B-7
  - ASCII, B-8
  - ATN, B-7
  - BASIC-PLUS CCPOS, A-5
  - BASIC-PLUS POS, 5-2, A-5
  - BASIC-PLUS-2 POS, 5-2, A-5
  - BASIC-PLUS-2 CCPOS, 5-2, B-8
  - CHR\$, B-8
  - COMP%, B-8
  - COS, B-7
  - DATE\$, B-10
  - DIF\$, B-8
  - ERL, B-10
  - ERN\$, B-10
  - ERR, B-10
  - EXP, B-7
  - FIX, B-7
  - INSTR, B-8
  - INT, B-7
  - LEFT, B-8
  - LEN, B-8
  - LOG, B-7
  - LOG10, B-8
  - MAGTAPE, B-10
  - MID, B-8
  - NUM, B-9
  - NUM\$, B-9
  - NUM1\$, B-9
  - NUM2, B-9
  - PEEK, 5-3
  - PI, B-8
  - PLACE\$, B-9
  - POS, A-5, B-8
  - PROD\$, B-9
  - QUO\$, B-9
  - RAD\$, B-9
  - RECOUNT, B-10
- Function (Cont.),
  - RIGHT, B-9
  - RND, B-8
  - SEG\$, B-9
  - SGN, B-8
  - SIN, B-8
  - SPACE\$, B-9
  - SQR, B-8
  - STR\$, B-9
  - STRING\$, B-9
  - SUM\$, B-9
  - TAB, B-8
  - TAN, B-8
  - TIME, B-10
  - TIME\$, B-10
  - TRM\$, B-9
  - VAL, B-9
  - XLATE, B-9
- Functions,
  - BASIC-PLUS SYS, 5-3
  - BASIC-PLUS user-defined, A-5
  - BASIC-PLUS-2 user-defined, A-5
    - error trace in, C-2
    - format of, B-4
    - format of user-defined, B-4
    - math, B-7
    - null arguments in, 5-4
    - summary of, B-7 to B-10
  - SYS, 5-3, A-8
  - translation of, 5-3, 5-4
  - user-defined, B-4
- GBLDEF option, 2-8
  - format, 2-8
  - see also Task Builder options
- GE for approximate key, 4-14
- Generic key, 4-14, 4-20
  - example of, 4-15
  - search, 4-15
- GET operation, 4-18, 4-19
  - indexed files, 4-13, 4-20
  - relative files, 4-10
  - sequential files, 4-7
- GT for approximate key, 4-14
- Header,
  - BASIC-PLUS-2, 1-18
  - file description, 4-22
  - library, 3-3
  - RUN command, 1-21
  - translator identification, 5-4
- Headers,
  - string, D-8
- HELLO command, B-5

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- High segment, 2-8
  - default, 2-8
  - Task Builder, 2-8,
- HISEG (HIS) command, 1-10, 1-17, 3-1, 3-2, B-5
- HISEG option, 2-3, 2-8
  - format, 2-8
  - see also Task Builder options
- I/O,
  - block, 4-1
  - record, 4-1, A-10
  - terminal, 4-24
- I/O operations,
  - BASIC-PLUS-2 record, A-10
  - block, 4-2, 4-3, A-10
  - speed of, 4-26
- Identification header,
  - translator, 5-4
- IDENTIFY (IDE) command, 1-10, 1-18, B-5
  - example of, 1-18
- Immediate mode statements, 5-5
- /IND RMS switch, 1-13
- Index,
  - content of, 4-13
  - reference to, 4-13
- Index table,
  - key arrangement, 4-14
  - maintenance, 4-14
  - primary, 4-13
- INDEXED,
  - ORGANIZATION, 4-10
- Indexed file, 4-2, 4-10
  - bucket size for, 4-27
  - creation of, 4-10
  - default bucket size, 4-28
  - DELETE operation, 4-13
  - example of, 4-12
  - extending, 4-17
  - FIND operation, 4-13, 4-20
  - format of, 4-11
  - GET operation, 4-13, 4-20
  - PUT operation, 4-13
  - random access on, 4-20
  - record access, 4-14
  - record location, 4-12
  - RESTORE operation, 4-13
  - UPDATE operation, 4-13
- Indexed file keys,
  - definition of, 4-29
  - length of, 4-29
  - position of, 4-29
- Indexed file media restriction, 4-2
- Indexed file memory space, 4-17
- Indexed file with variable format, 4-23
- Initialization,
  - variable, B-4
- INPUT FILE? prompt, 5-4
- INPUT statement,
  - punctuation, 5-3, A-8
  - translation of, 5-3
- Insert switch (/IN), 3-5
  - error message, 3-5
  - example of, 3-5
  - format of, 3-5
  - see also Librarian
- INSTR function, B-8
- INT function, B-7
- Integer argument passing, 3-11
- Integer constants, B-3
  - range of, B-3
- Integer format, D-6
- Integer variables, B-3
- Integers, 1-29
  - range of, 1-29
- Intended audience, vii
- Key, 4-13
  - approximate, 4-14, 4-20
  - content of, 4-13
  - data fields as, 4-13
  - EQ for exact, 4-14
  - exact, 4-14, 4-20
  - GE for approximate, 4-14
  - generic, 4-14, 4-20
  - GT for approximate, 4-14
  - length of, 4-13
  - numbering of alternate, 4-13
  - pointer to, 4-13
  - position of, 4-13
  - primary, 4-13
  - record access by alternate, 4-14
  - record access by primary, 4-14
- Key access,
  - example of generic, 4-15
- Key arrangement,
  - index table, 4-14
- Key changes, 4-14
- Key definition,
  - alternate, 4-13
  - primary, 4-13
- Key modification,
  - alternate, 4-14
- Key number, 4-13
  - specification of, 4-14
- Key of reference, 4-13, 4-14

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Key specification,
  - approximate, 4-14, 4-15
  - exact, 4-14
  - format of, 4-14
  - primary, 4-13
- Key value, 4-13
  - alternate, 4-14
- Keys,
  - alternate, 4-13
  - duplicate, 4-14
  - duplicate alternate, 4-14
  - duplicate primary, 4-14
- Keywords, B-3
  - reserved, B-10
  - table of reserved, B-11
  - table of system reserved, B-12
  
- Language elements,
  - BASIC-PLUS-2, B-1
- Language processor,
  - BASIC-PLUS-2, 1-8
- /LB switch, 2-5
  - arguments, 2-5
  - see also Task Builder switches
- Leading spaces, B-1
- Leading zeroes, B-1
- Leaving BASIC-PLUS-2, 1-17
- LEFT function, B-8
- LEN function, B-8
- LET debugging command, 1-27, B-5
- Lexical elements,
  - spaces in, 5-2
- Librarian, 3-3
  - command string, 3-3
  - file specifications, 3-3
  - input prompt, 3-3
  - invocation of, 3-3
  - switches, 3-4
  - termination of, 3-3
  - use of, 3-3
- Librarian utility program
  - (LBR), 3-3
- Libraries,
  - BASIC-PLUS-2, 3-2
- Library,
  - adding subroutines to, 3-3
  - BASIC-PLUS-2, 3-1
  - BASIC2, 3-1, 3-2
  - BP2COM, 3-1, 3-2
- Library extension, 2-3
  - object, 3-4
- Library file,
  - compression of, 3-8
  - creation of, 3-4
  - listing of, 3-7
  
- Library file (Cont.),
  - modification of, 3-5
  - module deletion, 3-8
  - module insertion, 3-5
  - size of, 3-4
  - update of, 3-5
- Library header, 3-3
- Library modules, 3-3
- LINE,
  - keyword, 5-2
- Line,
  - logical program, B-2
  - multi-statement, A-6, B-1
  - physical program, B-2
- Line and data format, B-1
- Line deletion,
  - program, 1-17, 1-23
- Line length, B-2
- Line number,
  - BASIC-PLUS-2, 1-28
  - per cent on, 5-4
  - range of, 1-28, B-1
  - translation of, 5-4
- Line replacement,
  - program, 1-23
- Line terminator, B-2
  - BASIC-PLUS-2, 1-28
- Lines,
  - BASIC-PLUS continued, 5-1, A-6
  - BASIC-PLUS-2 continued, 5-1, A-6
  - comments in source, 1-29
  - continued source, 1-28
  - translation of continuation, 5-1
- Linkage,
  - subroutine, 3-10
- LIST (LIS) command, 1-10, 1-19, B-5
  - use of, 1-19
- List switch, 3-7
  - combination of, 3-7
  - example of, 3-7
  - format of, 3-11
  - /FU, 3-7
  - /LE, 3-7
  - /LI, 3-7
  - see also Librarian
- Literals,
  - BASIC-PLUS string, 5-3
  - BASIC-PLUS-2 string, 5-3
  - quoted string, A-3
  - translation of string, 5-3
  - unterminated string, 5-3
- LOCK command, 1-10, 1-15, B-5
  - example of, 1-16

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Locked bucket, 4-16
- Locking,
  - disabling, 4-16
  - record, 4-16
- LOG function, B-7
- LOG10 function, B-8
- Logical device names, 1-3
- Logical operators, B-2
  - table of, B-2
- Logical program line, B-2
- Logical units,
  - number of, 2-9
- LOGIN command, B-5
- Login procedure, 1-1
- LOGIN program, 1-1
  - prompt, 1-1, 1-2
- Logout procedure, 1-2
- LOGOUT program, 1-2
  - options, 1-2
- Long variable names, A-7
- Loops,
  - debugging halt in, 1-26
  - transfer into FOR NEXT, A-9
- LSET file operation, 4-3
  
- /MA switch, 2-5
  - see also Task Builder switches
- MACRO modules,
  - inserting, 3-5
- MACRO source file, 1-14
- MACRO subroutines, 3-3, 3-10, 3-11
  - restrictions, 3-10
- /MACRO switch, 1-14
- Magnetic tape,
  - ANSI, 4-23
  - block size on, 4-24
  - count field on, 4-23
  - sequential files on, 4-25
- MAGTAPE function, B-10
- Map,
  - memory allocation, 2-9
  - short memory allocation, 2-5
- MAP clause, 4-22, 4-23, 4-29
- Map extension,
  - memory allocation, 2-3
- MAP file attribute, 4-3, 4-5, 4-8, 4-10
- MAP name, 4-29
- MAP statement, 4-13, 4-14, 4-20, 4-29
  - example of, 4-29, 4-30
- Mapping,
  - record, 4-4, 4-29
- Math functions, B-7
  
- Media restriction,
  - indexed file, 4-2
  - relative file, 4-2
  - sequential file, 4-2
- Memory allocation,
  - formulas, 4-17
  - for open files, 4-17
  - RMS, 4-17
  - RMS initial, 4-17
- Memory allocation map, 2-9
  - extension, 2-3
  - output, 2-9
  - production of, 1-12
  - see also Map
  - short, 2-5
- Memory,
  - arrays in, D-8
- Memory reductions,
  - /NOCHAIN, 1-15
  - /NOLINE, 1-15
- Memory requirements,
  - /DEBUG switch, 1-24
- Memory space,
  - file organizations, 4-17
  - indexed files, 4-17
  - relative files, 4-17
  - sequential files, 4-17
- Merging source programs, 1-11
  - see also APPEND command
- MID function, B-8
- MODE switch option, 1-7
- MODIFY file attribute, 4-16
  - ACCESS, 4-16
  - ALLOW, 4-16
- Module,
  - library, 3-3
  - library file deletion, 3-8
  - library file insertion, 3-5
  - object, 1-14, 2-1, 2-4
- Module name table (MNT), 3-3, 3-4
- /MP switch, 2-5, 2-10
  - see also Task Builder switches
- Multi-statement line, A-6, B-1
- Multiple assignment statement,
  - 5-3, A-3
  - translation of, 5-3
  
- Name change,
  - program, 1-20
  - translator program, 5-5
- Names,
  - BASIC-PLUS variable, 5-2
  - BASIC-PLUS-2 function, A-7
  - BASIC-PLUS-2 variable, 5-2, A-7
  - logical device, 1-3

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Names (Cont.),
  - long variable, A-7
  - physical device, 1-3
  - translation of variable, 5-2, 5-5
  - variable, 5-2
- Negating Task Builder
  - switches, 2-4
- Negation of replace switch, 3-6
- Negative cluster size, 1-7
- NEW command, 1-10, 1-19, B-5
  - prompt, 1-19
- NEW FILE NAME-- prompt, 1-19
- No header,
  - command specification, 1-11
  - NH abbreviation, 1-11
- /NOCHAIN switch, 1-14
  - memory reductions, 1-15
- NOCHANGES file attribute, 4-10, 4-12
- NODUPLICATES file attribute, 4-10, 4-12
- NOEXTEND mode, A-7
- /NOLINE switch, 1-14, 1-15
  - COMPILE command, 1-14
  - diagnostic error, 1-15
  - memory reductions with, 1-15
  - reasons for override, 1-15
- Non-printing characters, 1-29, B-2
- NONE file attribute, 4-16
  - ALLOW, 4-16
- NOREWIND file attribute, 4-5, 4-7
- NOSPAN file attribute, 4-5, 4-6
- Null arguments in functions, 5-4
- Null characters, 1-29, B-2
- NUM function, B-9
- NUM\$ function, B-9
- NUM1\$ function, B-9
- NUM2 function, B-9
- Number of logical units, 2-9
- Numeric constants, 5-4, B-3
  - range of, B-3
  - spaces in, 5-4
  - tabs in, 5-4
  - translation of, 5-4
- Numeric variables, B-3
- Object library extension, 3-4
- Object module, 1-12, 2-1, 2-4
  - default extension, 2-1
  - generation of, 1-14
  - input to Task Builder, 2-3
  - programs compiled as, 1-15
  - reasons for, 1-12
- /OBJECT switch, 1-15
- ODL directive format, 2-12
- ODL directives, 2-10
- ODL file, 2-10
  - editing, 2-13
  - modification, 2-10
- OLD command, 1-10, 1-20, B-5
- OLD FILE NAME-- prompt, 1-20
- OLD NAME? prompt, 5-5
- ON CALL,
  - BREAK command, 1-26
- ON DEF,
  - BREAK command, 1-26
- ON LOOP,
  - BREAK command, 1-26
- Open files,
  - RMS memory allocation for, 4-17
- OPEN statement, 4-1, 4-7, 4-9, 4-10, 4-12, 4-21
- OPEN statement format,
  - indexed file, 4-11
  - relative file, 4-8
  - sequential file, 4-5
  - virtual file, 4-3
- Operators,
  - arithmetic, B-2
  - logical, B-2
  - relational, B-2
  - table of arithmetic, B-2
  - table of logical, B-2
  - table of relational, B-3
- Organization,
  - assignment of file, 4-2
  - comparison of file, 4-5
  - file, 4-1, 4-4
  - indexed file, 4-10
  - relative file, 4-8
  - sequential file, 4-5
  - specification of file, 4-2
  - types of file, 4-1
  - virtual file, 4-2, A-10
- Organization default, 4-2
- ORGANIZATION INDEXED, 4-10
- ORGANIZATION keyword, 4-2, 4-3
- ORGANIZATION RELATIVE, 4-8
- ORGANIZATION SEQUENTIAL, 4-6
- ORGANIZATION VIRTUAL, 4-3
- OUTPUT FILE? prompt, 5-5
- Output specification,
  - Task Builder, 2-2, 2-9
- Overhead,
  - RMS bucket, 4-28
- Overlay,
  - common path, 2-13
  - definition, 2-10
  - from BUILD output, 2-10
  - loading procedures, 2-10
  - reason for, 2-10

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Overlay description file, 2-5
  - extension, 2-3
- Overlay description language (ODL), 1-12, 2-10
- Overlay sections,
  - calls to, 2-13
- Overlay structure,
  - example of, 2-12
  - path of, 2-13
- Override of tape rewind, 4-7
  
- Passing arguments,
  - array, 3-12
  - double, 3-11
  - integer, 3-11
  - real, 3-11
  - string, 3-11
  - subroutine, 3-10
- Password, 1-1
- PASSWORD: prompt, 1-2
- Patches,
  - Task Builder, 2-7
- Path of execution,
  - examine the, 1-28
- Path of overlay structure, 2-13
- PEEK function, 5-3, A-8
  - translation of, 5-3
- Physical device names, 1-3
- Physical program line, B-2
- PI function, B-8
- PLACE\$ function, B-9
- Pointer to key, 4-13
- POS function, A-5, B-8
  - BASIC-PLUS, 5-2, A-5
  - BASIC-PLUS-2, 5-2, A-5
  - translation of, 5-2
- Precision,
  - range of double, 1-29
  - range of single, 1-29
- PRIMARY file attribute, 4-11, 4-12
- Primary index table, 4-13
- Primary key, 4-13
  - definition, 4-13
  - duplicate, 4-14
  - record access by, 4-14
  - see also Key
  - specification, 4-13
- PRINT debugging command, 1-27, B-5
- PRINT statement,
  - synonym for, 5-1
  - translation of, 5-3
- PRINT statement punctuation, 5-3, A-8
- PRINT USING, 5-2
  - apostrophe, A-2
  - backslashes, A-2
  - BASIC compatibility, A-1
  - example of, A-2
  - exclamation point, A-2
- PRINT USING format,
  - BASIC-PLUS, 5-2
  - BASIC-PLUS-2, 5-2
  - translation of, 5-2
- PRINT USING string format, A-1, A-2
- PROD\$ function, B-9
- Program,
  - access to source, 1-20
  - BASIC-PLUS-2 source, 1-28
  - deletion of compiled, 1-23
  - exit from system, 1-8
  - LOGIN, 1-1
  - LOGOUT, 1-2
  - print a copy of, 1-19
  - saving a, 1-22
  - translator input, 5-5
- Program compilation, 1-14
- Program creation, 1-19
- Program deletion, 1-22
- Program editing, 1-23
- Program example,
  - source, 1-30
- Program execution, 1-21
- Program line,
  - deletion, 1-23
  - logical, B-2
  - physical, B-2
  - replacement, 1-23
- Program name change, 1-20
  - translator, 5-5
- Program replacement, 1-21
- Program segmentation, 2-10
- Program size limit,
  - BASIC2, 3-2
  - BP2COM, 3-2
- Program storage, 1-22
- Programmer number, 1-3
  - assignment of, 1-2
- Programs,
  - merging source, 1-11
  - preserving BASIC-PLUS, 5-1
- Project number, 1-3
- Prompt,
  - APPEND FILE NAME, 1-11
  - continuation of TKB>, 2-2
  - debugging aid, 1-24
  - ENTER OPTIONS:, 2-3
  - EXTEND MODE?, 5-5
  - HISEG command, 1-18
  - INPUT FILE?, 5-4



# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

## Prompt (Cont.)

- librarian input, 3-3
- LOGIN, 1-1, 1-2
- NEW command, 1-19
- NEW FILE NAME--, 1-19
- OLD FILE NAME--, 1-20
- OLD NAME?, 5-5
- OUTPUT FILE?, 5-5
- PASSWORD:, 1-2
- READY, 1-2, 1-8
- TARGET SYSTEM?, 5-4
- Task Builder, 2-1
- Protection code, 1-3, 1-5
  - combination of, 1-6
  - default, 1-6
  - specification of, 1-6
  - table of, 1-5, 1-6
- Public structure default device, 1-3
- PUT operation, 4-18, 4-19
  - indexed file, 4-13
  - relative file, 4-10, 4-19
  - sequential file, 4-7, 4-19

- QUO\$ function, B-9
- Quoted string literals, A-3

- RAD\$ function, B-9
- Radix-50 character set, D-4
- Radix-50 conversion, D-4
- Radix-50 format, D-4
- Random access, 4-5
  - on indexed files, 4-20
  - on relative files, 4-19
  - record, 4-18, 4-19
- Range,
  - channel number, 4-2
  - of double precision, 1-29
  - of integer constants, B-3
  - of integers, 1-29
  - of line numbers, 1-28, B-1
  - of numeric constants, B-3
  - of single precision, 1-29
  - of subscript variables, 1-29
  - of subscripts, B-3
- READ file attribute, 4-16
  - ACCESS, 4-16
  - ALLOW, 4-16
- Reading stream-format records, 4-24
- READY prompt, 1-2, 1-8
- Real argument passing, 3-11
- Real format, D-7

## Record,

- accessing the, 4-13
  - location of, 4-13
- Record access, 4-4
  - by alternate key, 4-14
  - by primary key, 4-14
  - example of, 4-20
  - in sequential files, 4-18
  - indexed file, 4-14
  - random, 4-18, 4-19
  - sequential, 4-18
  - shifting, 4-21
- Record access methods, 4-17
- Record addition, 4-5
- Record assignment,
  - relative file, 4-8
- Record deletion, 4-5, 4-19
- Record delimiter,
  - (CR/LF), 4-24
  - (ESC), 4-24
  - (FF), 4-24
  - (LF), 4-24
  - (VT), 4-24
  - stream-format, 4-24
- Record format, 4-4, 4-21
  - default, 4-6, 4-22, 4-23
  - relative file, 4-23
  - specification, 4-22
  - table of, 4-22
- Record I/O file, 4-1
  - BASIC-PLUS-2, A-10
- Record identifier, 4-14
- Record insertion, 4-5
- Record length specification, 4-22
- Record location,
  - indexed file, 4-12
- Record locking, 4-16
- Record Management Services (RMS), 4-1
- Record mapping, 4-4, 4-29
- Record movement, 4-29
- Record number, 4-19
- Record operations, 4-18
  - RMS, 4-2
- Record position, 4-19
  - arrangement of, 4-9
  - empty, 4-9
  - first, 4-9
  - relative file, 4-9
- Record relationship,
  - sequential file, 4-7
- Record replacement, 4-5
- Record retrieval operations, 4-23
- Record size,
  - maximum, 4-23
- Record/position number, 4-9

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Records, 4-1, 4-4
  - access to, 4-17
  - fixed-length, 4-21, 4-22
  - format of, 4-6
  - reading stream-format, 4-24
  - relative file, 4-22
  - serial access of, 4-7
  - stream-format, 4-22, 4-23, 4-24
  - variable-length, 4-21, 4-23
  - writing stream-format, 4-24
- Records on blocks, 4-25
- RECORDSIZE clause, 4-6, 4-9, 4-12, 4-22, 4-25
- RECORDSIZE file attribute, 4-3, 4-5, 4-8, 4-10
- RECORDSIZE specification, 4-3
- RECOUNT function, B-10
- Recoverable error codes, table of, C-9 to C-12
- Register usage, subroutine, 3-11
- /REL RMS switch, 1-13
- Relational operators, B-2
  - table of, B-3
- RELATIVE, ORGANIZATION, 4-8
- Relative file, 4-2
  - bucket size for, 4-26
  - creation of, 4-8
  - default bucket size, 4-27
  - DELETE operation, 4-10
  - example of, 4-9
  - extending, 4-17
  - FIND operation, 4-10
  - format of, 4-8, 4-23
  - GET operation, 4-10
  - media restrictions, 4-2
  - memory space, 4-17
  - PUT operation, 4-10, 4-19
  - random access on, 4-19
  - record assignment, 4-8
  - record position, 4-9
  - RESTORE operation, 4-10
  - UPDATE operation, 4-10
  - with variable format, 4-23
- RENAME (REN) command, 1-10, 1-20, B-5
- REPLACE (REP) command, 1-10, 1-21, B-6
- Replace switch (/RP), 3-5
  - example of, 3-7
  - format, 3-6
    - global, 3-6
    - local, 3-6
  - negation of, 3-6
  - see also Librarian
- Reserved keywords, B-10
  - table of, B-11
  - table of system, B-12
- Resource sharing, RSTS/E, 1-1
- RESTORE operation, 4-18
  - indexed file, 4-13
  - relative file, 4-10
  - sequential file, 4-8
- Retrieval operations, record, 4-23
- RETURN key, viii, 5-6, B-7
- Rewind,
  - override of tape, 4-7
- RIGHT function, B-9
- RMS, 4-1, 4-4, A-10
  - access to code, 1-12
  - bucket overhead, 4-28
  - devices not allowed for, 1-4
  - existence byte, 4-27
  - introduction to, 4-4
  - memory allocation, 4-17
  - operations, 4-2, 4-5, A-10
- RMS switch,
  - BUILD command, 1-13
  - /IND, 1-13
  - /REL, 1-13
  - /SEQ, 1-13
- RMS utilities, 4-1, 4-30
- RMS with BASIC2,
  - use of, 3-2
- RMSBCK utility, 4-31
- RMSCNV utility, 4-31
- RMSDFN utility, 4-31
- RMSDSP utility, 4-31
- RMSRST utility, 4-31
- RND function, B-8
- /RONLY switch option, 1-7
- .ROOT directive, 2-12
- Routines,
  - error handling, C-1
  - run-time support, 3-1
- RSET file operation, 4-3
- RSTS/E switch options, 1-3, 1-6
- RSTS/E system,
  - accessing the, 1-1
- RTS purpose of,
  - BASIC-PLUS-2, 3-1
  - see also Run-Time system
- RUBOUT key, 1-23, B-7
  - erasures, 1-23
- RUN command, 1-10, 1-21, B-6
- RUN command header, 1-21
- Run-time error messages, C-12
- Run-time errors,
  - summary of, C-12 to C-31
- Run-time support routines, 3-1

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Run-time system (RTS),
  - 2-8, 3-1
  - BASIC-PLUS-2, 1-17, 3-1
  - BASIC2, 1-17, 2-8, 3-1, 3-2
  - BP2COM, 1-17, 2-8, 3-1, 3-2
  - BUILD command default, 1-18
  - default, 1-17
- SAVE (SAV) command, 1-10, 1-22, B-6
- SCALE (SCA) command, 1-10, 1-22, B-6
- Scale factor,
  - specification of, 1-22
- Scaled arithmetic, 1-22
- SCRATCH file attribute, 4-16
  - ACCESS, 4-16
  - sequential file, 4-8
- Secondary data fields, 4-13
- SEG\$ function, B-9
- Segment,
  - default high, 2-8
  - high, 2-8
  - Task Builder, 2-8
- Segmentation,
  - program, 2-10
- Semicolons, 5-3
- Separator,
  - backslash statement, B-1
  - comment, 1-29, A-6
  - exclamation point, 1-29
  - statement, 1-29, B-1
- Separators,
  - BASIC-PLUS statement, 5-2
  - BASIC-PLUS-2 statement, 5-2
  - comment, 5-3
  - statement, 5-2, A-6
  - translation of comment, 5-3
  - translation of statement, 5-2
- /SEQ RMS switch, 1-13
- SEQUENTIAL,
  - ORGANIZATION, 4-6
- Sequential access, 4-5
- Sequential file, 4-2
  - creation of, 4-5
  - example of, 4-7
  - FIND operation, 4-7
  - format of, 4-5
  - GET operation, 4-7
  - media restrictions, 4-2
  - memory space, 4-17
  - on magnetic tape, 4-25
  - OPEN statement format, 4-5
  - organization, 4-5
  - PUT operation, 4-7
- Sequential file (Cont.),
  - record access in, 4-18
  - record relationship, 4-7
  - RESTORE operation, 4-8
  - restrictions, 4-7
  - SCRATCH operation, 4-8
  - sharing, 4-15
  - UPDATE operation, 4-7
- Sequential PUT operation,
  - relative files, 4-19
- Sequential record access,
  - 4-18
- Serial access of records, 4-7
- Setting defaults,
  - COMPILE command, 1-15
- SGN function, B-8
- /SH switch, 2-5
  - see also Task Builder switches
- Sharing,
  - control of file, 4-15
  - file, 4-15
  - sequential file, 4-15
  - virtual file, 4-16
- Shifting record access, 4-21
- Short memory allocation map,
  - 2-5
- SIN function, B-8
- Single precision,
  - range of, 1-29
- Single-precision format, 1-14
- Small bucket size, 4-29
- Source file,
  - MACRO, 1-14
- Source lines, 1-28
  - comments in, 1-29
  - continued, 1-28
- Source program,
  - access to, 1-20
  - BASIC-PLUS-2, 1-28
  - example, 1-30
  - merging, 1-11
- SPACE\$ function, B-9
- Spaces,
  - in lexical elements, 5-2
  - in numeric constants, 5-4
  - leading, B-1
- SPAN file attribute, 4-5, 4-6
- Spanning, 4-25
- Specification format,
  - filename, 1-3
- Speed of I/O operations, 4-26
- SQR function, B-8
- STACK option, 2-8
  - format, 2-9
  - see also Task Builder options
- Stack size declaration,
  - Task Builder, 2-8

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Statement,
  - BASIC-PLUS DATA, 5-2, A-5
  - BASIC-PLUS-2 DATA, 5-2, A-5
  - BASIC-PLUS CHAIN, 5-2, A-7
  - BASIC-PLUS-2 CHAIN, 5-2, A-7
  - CALL, 3-11, 3-12, A-9
  - CALL BY REF, 3-11, 3-13, A-9
  - CHAIN, 5-2, A-7
  - DATA, 5-2, A-5
  - DEF, 5-1, A-4
  - DIM #, 4-3
  - END, 1-28
  - example of MAP, 4-30
  - immediate mode, 5-5
  - MAP, 4-13, 4-14, 4-20, 4-29
  - multiple, 1-28
  - multiple assignment, A-3
  - OPEN, 4-1, 4-3, 4-7 to 4-10, 4-12, 4-21
  - translation of CHAIN, 5-2
  - translation of DATA, 5-2
  - translation of DEF, 5-1
  - translation of INPUT, 5-3
  - translation of PRINT, 5-3
  - UNLOCK, 4-16
- Statement separator, 1-29, 5-2, A-6, B-1
- Statement separators,
  - BASIC-PLUS, 5-2
  - BASIC-PLUS-2, 5-2
  - translation of, 5-2
- STEP debugging command, 1-27, B-6
  - argument, 1-27
  - argument range, 1-27
- STR\$ function, B-9
- Stream ASCII file, 4-22
  - creation of, 4-24
  - data in, 4-24
  - record delimiter, 4-24
- Stream-format records, 4-22 to 4-24
  - reading, 4-24
  - writing, 4-24
- String,
  - BASIC-PLUS, A-3
  - BASIC-PLUS-2, A-3
- String argument passing, 3-11
- String constant, B-3
  - length of, B-3
- String delimiter, 5-3
- String format, D-7
  - dynamic, D-7
- PRINT USING, A-1
- String header, D-8
- String literal,
  - BASIC-PLUS, 5-3
  - BASIC-PLUS-2, 5-3
- String literal (Cont.),
  - quoted, A-3
  - translation of, 5-3
  - unterminated, 5-3
- String manipulation,
  - space for, 2-8
- String variable, B-3
- STRING\$ function, B-9
- Structure,
  - data, 4-4, 4-24
  - logical data, 4-24
  - overlay tree, 2-12
  - path of overlay, 2-13
  - physical storage, 4-24
- Subprograms, 2-10
  - execution of, 1-12
- Subroutine,
  - adding to library, 3-3
  - argument passing, 3-10
  - BASIC-PLUS-2, 3-3
  - call instructions, 3-10
  - calling conventions, 3-10
  - calls, 3-11
  - debugging, 1-24
  - error trace in, C-2
  - linkage, 3-10
  - MACRO, 3-3, 3-10, 3-11
  - passing mechanism, 3-11
  - register usage, 3-11
  - restrictions, 3-10
  - writing, 3-3
- Subscript variable, 1-29
  - range of, 1-29
- Subscripts, B-3
  - array, A-9
  - evaluation of array, A-9
  - range of, B-3
- SUM\$ function, B-9
- Summary of commands, B-1
- Summary of compile-time errors, C-3 to C-8
- Summary of control characters, B-7
- Summary of functions, B-7 to B-10
- Summary of run-time errors, C-12 to C-31
- Support routines,
  - run-time, 3-1
- Switch,
  - /CO compress, 3-8
  - combination of list, 3-7
  - /CR create, 3-4
  - /DA debugging, 2-5
  - /DE delete, 3-8
  - /DEBUG, 1-14, 1-24
  - /DOUBLE, 1-14
  - /EX extract, 3-6

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

### Switch (Cont.),

- example of compress, 3-9
- example of insert, 3-5
- example of create, 3-4
- example of extract, 3-7
- example of replace, 3-7
- /FU list, 3-7
- /IN insert, 3-5
- /IND RMS, 1-13
- /LB library, 2-5
- /LE list, 3-7
- /LI list, 3-7
- /MA map, 2-5
- /MACRO, 1-14
- /MP overlay, 2-5, 2-10
- negation of replace, 3-6
- /NOCHAIN, 1-14
- /NOLINE, 1-14, 1-15
- /OBJECT, 1-15
- /REL RMS, 1-13
- /RP replace, 3-6
- /SEQ RMS, 1-13
- /SH short, 2-5
- /TSK, 1-14
- /WI wide, 2-5
- /XT exit, 2-5
- Switch options, 1-7, 4-1
  - /CLUSTERSIZE, 1-7
  - /FILESIZE, 1-7
  - /MODE, 1-7
  - /RONLY, 1-7
  - RSTS/E, 1-3, 1-6
  - system, 1-3
- Switch specification,
  - Task Builder, 2-5
- Switches,
  - BUILD command RMS, 1-13
  - COMPILE command, 1-9, 1-14
  - librarian, 3-4
  - negating Task Builder, 2-4
  - table of Task Builder, 2-4
  - Task Builder, 2-4
  - Task Builder default, 2-5
- Switches combined,
  - COMPILE command, 1-15
- Symbol definition,
  - Task Builder, 2-8
- Synonym for PRINT, 5-1
- Syntax,
  - BASIC-PLUS-2, 5-1
- Syntax check,
  - BASIC Compiler, 1-23
- SYS functions, 5-3, A-8
  - BASIC-PLUS, 5-3
  - translation of, 5-3
- System,
  - accessing the RSTS/E, 1-1
  - BASIC-PLUS-2 run-time, 3-1

### System (Cont.),

- BASIC2 run-time, 1-17, 2-8,
  - 3-1, 3-2
- BP2COM run-time, 1-17, 2-8,
  - 3-1, 3-2
- BUILD command default run-time, 1-18
  - default run-time, 1-17
  - run-time, 2-8
- System command set,
  - support of, 1-9
- System debugging aid, 2-5
- System program,
  - exit from, 1-8
- System reserved keywords,
  - table of, B-12
- System switch options, 1-3
- TAB key, B-7
- TAB function, B-8
- Table,
  - of access methods, 4-18
  - of arithmetic operators, B-2
  - of ASCII/Radix-50 equivalents,
    - D-5
  - of BASIC-PLUS-2 commands,
    - 1-9, 1-10
  - of BASIC-PLUS-2 extensions, 1-5
  - of device specifications, 1-4
  - of logical operators, B-2
  - of protection codes, 1-5, 1-6
  - of record formats, 4-22
  - of recoverable error codes,
    - C-9 to C-12
  - of relational operators, B-3
  - of reserved keywords, B-11
  - of system reserved keywords,
    - B-12
  - of Task Builder options, 2-6
  - of Task Builder switches, 2-4
  - of task file types, 2-3
- Tabs in numeric constants, 5-4
- TAN function, B-8
- Tape,
  - ANSI magnetic, 4-23
  - block size on magnetic, 4-24
  - count field on magnetic, 4-23
  - sequential files on magnetic,
    - 4-25
- Tape rewind,
  - override of, 4-7
- TARGET SYSTEM? prompt, 5-4
- Task,
  - creation of executable, 1-14
  - memory available, 2-8
  - production of executable, 1-8

# INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- Task Builder, 2-1, 2-2
  - aborting, 2-6
  - CCL command, 2-1
  - command input, 1-12
  - device assignment, 2-7
  - filename specification, 2-4
  - high segment association, 2-8
  - invocation, 2-1
  - logical unit number, 2-9
  - memory extension, 2-7
  - name specification, 2-9
  - overlay facility, 2-10
  - patches, 2-7
  - prompt, 2-1
  - reasons for using, 2-1
  - stack size declaration, 2-8
  - symbol definition, 2-8
- Task Builder input,
  - specification, 2-2
  - termination of, 2-3
- Task Builder options, 2-2, 2-5
  - table of, 2-6
  - use of, 2-6
- Task Builder output specification, 2-2, 2-9
- Task Builder switches, 2-4
  - default, 2-5
  - negating, 2-4
  - specification, 2-5
  - table of, 2-4
- Task creation,
  - examples of, 2-15
  - execution, 2-14
- Task file extensions, 2-3
  - table of, 2-3
- Task image,
  - corrective code into, 2-7
  - file, 2-9
  - file extension, 2-3
  - file output, 2-9
- Task input,
  - termination of, 2-7
- Task memory,
  - allocation of, 2-9
- TASK option, 2-9
  - format, 2-9
- Terminal I/O files, 4-24
- Terminal-format file, 4-2
  - creation of, 4-2
- Terminator,
  - BASIC-PLUS-2 line, 1-28
  - line, B-2
- TIME function, B-10
- Time sharing,
  - RSTS/E, 1-1
- TIME\$ function, B-10
- TKB> prompt,
  - continuation of, 2-2
- TRACE debugging command, 1-28, B-6
- Traceback, C-2
  - errors in functions, C-2
  - errors in subroutines, C-2
  - example of, C-2
  - mechanism, C-2
  - text of, C-2
- Translation,
  - of ambiguous constants, 5-3
  - of CHAIN statement, 5-2
  - of comment separator, 5-3
  - of continuation line, 5-1
  - of DATA statement, 5-2
  - of DEF statement, 5-1
  - of functions, 5-4
  - of INPUT statement, 5-3
  - of line numbers, 5-4
  - of numeric constants, 5-4
  - of PEEK function, 5-3
  - of POS function, 5-2
  - of PRINT statement, 5-3
  - of PRINT synonym, 5-1
  - of PRINT USING format, 5-2
  - of statement separators, 5-2
  - of string literals, 5-3
  - of SYS functions, 5-3
  - of variable names, 5-2, 5-5
- Translator utility, 5-1
  - access to, 5-4
  - command file, 5-6
  - default, 5-4
  - dialogue, 5-4
  - input to, A-1
  - program input, 5-5
  - program name change, 5-5
  - sample run, 5-6
  - using the, 5-4
  - warning messages, 5-8
- Tree structure,
  - overlay, 2-12
- TRM\$ function, B-9
- /TSK switch, 1-14
- UNBREAK debugging command, 1-25, 1-26, B-6
- Unit number,
  - Task Builder logical, 2-9
- UNITS option, 2-9
  - format, 2-9
  - see also Task Builder options
- UNLOCK statement, 4-16
- UNSAVE (UNS) command, 1-10, 1-22, B-6
- UNTRACE debugging command, 1-28, B-6

## INDEX (Cont.)

Note that an underlined page number points to an entry's primary or definitive reference.

- UPDATE operation, 4-18, 4-19
  - indexed file, 4-13
  - relative file, 4-10
  - sequential file, 4-7
- User-defined functions, B-4
  - BASIC-PLUS, A-5
  - BASIC-PLUS-2, A-5
  - debugging, 1-26
  - format of, B-4
- Utilities,
  - RMS, 4-1, 4-30
  - RMSBCK, 4-31
  - RMSCNV, 4-31
  - RMSDFN, 4-31
  - RMSDSP, 4-31
  - RMSRST, 4-31
  - translator, 5-1
- Utility program (LBR),
  - Librarian, 3-3
- VAL function, B-9
- Variable,
  - ERR, C-9
  - error, C-1
- Variable format,
  - indexed files with, 4-23
  - relative files with, 4-23
- Variable initialization, B-4
- Variable names, 5-2, B-3
  - BASIC-PLUS, 5-2, A-7
  - BASIC-PLUS-2, 5-2, A-7
  - long, A-7
  - specification, 5-5
  - translation of, 5-2, 5-5
- Variable-length format, 4-23
- Variable-length records, 4-21,
  - 4-23
- Variables, B-3
  - integer, B-3
  - numeric, B-3
  - range of subscript, 1-29
  - string, B-3
  - subscript, 1-29
- Variables change,
  - debugging commands, 1-27
- Variables examine,
  - debugging commands, 1-27
- VIRTUAL,
  - ORGANIZATION, 4-3
- Virtual arrays, 4-2
  - format of, D-8
- Virtual file, 4-2
  - attributes, 4-3
  - data, 4-2
  - data access, 4-2
  - data storage, 4-2
  - example of, 4-3
  - format, 4-3
  - operations, 4-16
  - organization, 4-2, A-10
  - sharing, 4-16
- Warning message, C-1
  - COMPILE command, 1-14
  - translator, 5-8
- /WI switch, 2-5
  - see also Task Builder switches
- Word,
  - array descriptor, D-8, D-9
- Word alignment,
  - count field, 4-23
- WRITE file attribute, 4-16
  - ACCESS, 4-16
  - ALLOW, 4-16
- XLATE function, B-9
  - /XT switch, 2-5
  - see also Task Builder switches
- Zeroes,
  - leading, B-1

READER'S COMMENTS

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

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

---

---

---

---

---

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

---

---

---

---

---

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

---

---

---

---

---

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

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

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

Organization \_\_\_\_\_

Street \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip Code \_\_\_\_\_

or  
Country

Please cut along this line.



-----  
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 ML5-5/E39  
Maynard, Massachusetts 01754