

[illegible]

AlphaBASIC PLUS® User's Manual

© 1997 Alpha Microsystems

| REVISIONS INCORPORATED | |
|------------------------|------|
| REVISION | DATE |

| | |
|----|----------------|
| 00 | September 1989 |
| 01 | April 1991 |
| 02 | September 1996 |
| 03 | May 1997 |
| 04 | December 1997 |

AlphaBASIC PLUS User's Manual

To re-order this document, request part number DSO-00045-00.

This document applies to AMOS 2.3A, PR 12/97 and later.

The information contained in this manual is believed to be accurate and reliable. However, no responsibility for the accuracy, completeness or use of this information is assumed by Alpha Microsystems.

This document may contain references to products covered under U.S. Patent Number 4,530,048.

The following are registered trademarks of Alpha Microsystems, Santa Ana, CA 92799:

| | | | |
|-----------------|-----------|-------------|-----------------|
| AMIGOS | AMOS | Alpha Micro | AlphaACCOUNTING |
| AlphaBASIC | AlphaCALC | AlphaCOBOL | AlphaDDE |
| AlphaFORTRAN 77 | AlphaLAN | AlphaLEDGER | AlphaMAIL |
| AlphaMATE | AlphaNET | AlphaPASCAL | AlphaRJE |
| AlphaWRITE | CASELODE | OmniBASIC | VER-A-TEL |
| VIDEOTRAX | | | |

The following are trademarks of Alpha Microsystems, Santa Ana, CA 92799:

| | | | |
|-------------------|--------------|-------|-------------------|
| AlphaBASIC PLUS | AlphaVUE | AM-PC | AMTEC |
| AlphaDDE | AlphaCONNECT | DART | <i>inSight/am</i> |
| <i>inFront/am</i> | ESP | MULTI | |

All other copyrights and trademarks are the property of their respective holders.

ALPHA MICROSYSTEMS
2722 S. Fairview St.
P.O. Box 25059
Santa Ana, CA 92799

TABLE OF CONTENTS

PART ONE - INTRODUCTION

CHAPTER 1 - INTRODUCTION

| | |
|---|-----|
| 1.1 WHO IS THIS MANUAL WRITTEN FOR? | 1-1 |
| 1.2 WHAT IS BASIC? | 1-2 |
| 1.3 HOW IS ALPHABASIC PLUS SPECIAL? | 1-2 |
| 1.4 WHAT CAN ALPHABASIC PLUS DO? | 1-3 |
| 1.5 WHAT TYPES OF DATA DOES ALPHABASIC PLUS SUPPORT? | 1-4 |
| 1.6 THE ALPHABASIC PLUS SYSTEM | 1-4 |
| 1.7 EXTENSIONS USED WITH ALPHABASIC PLUS | 1-5 |
| 1.8 WHAT DOES AN ALPHABASIC PLUS PROGRAM LOOK LIKE? | 1-5 |
| 1.9 GRAPHICS CONVENTIONS | 1-6 |

PART TWO - THE AlphaBASIC PLUS SYSTEM

CHAPTER 2 - GENERAL INFORMATION

| | |
|------------------------------------|-----|
| 2.1 LINE NUMBERS | 2-1 |
| 2.2 LABELS | 2-2 |
| 2.3 COMMENTS (REM AND !) | 2-2 |
| 2.4 MULTIPLE STATEMENT LINES | 2-3 |
| 2.5 CONTINUATION LINES | 2-3 |
| 2.6 MEMORY ALLOCATION | 2-4 |
| 2.7 SEARCHING FOR A LIBRARY | 2-4 |

CHAPTER 3 - INTERACTIVE MODE

| | |
|---|-----|
| 3.1 WHAT IS INTERACTIVE MODE? | 3-1 |
| 3.2 DIRECT STATEMENTS | 3-3 |
| 3.3 CLEARING MEMORY FOR A NEW PROGRAM (NEW) | 3-3 |
| 3.4 CREATING A PROGRAM | 3-4 |
| 3.5 DELETING PROGRAM LINES (DELETE) | 3-4 |
| 3.6 SAVING A PROGRAM (SAVE) | 3-5 |
| 3.7 LISTING A PROGRAM (LIST) | 3-7 |
| 3.8 COMPILING A PROGRAM | 3-7 |
| 3.8.1 Compiler Options | 3-8 |
| 3.9 RUNNING A PROGRAM (RUN) | 3-8 |
| 3.10 EXITING FROM INTERACTIVE MODE (BYE) | 3-9 |
| 3.11 A HINT FOR THE SYSTEM OPERATOR | 3-9 |

CHAPTER 4 - MORE INTERACTIVE COMMANDS

| | |
|--|-----|
| 4.1 INTERRUPTING A PROGRAM RUN (BREAK) | 4-1 |
| 4.2 CONTINUING AN INTERRUPTED PROGRAM (CONT) | 4-2 |
| 4.3 STOPPING A PROGRAM RUN | 4-3 |
| 4.4 LOADING A PROGRAM INTO MEMORY (LOAD) | 4-3 |
| 4.5 THE SINGLE-STEP FEATURE | 4-4 |

CHAPTER 5 - WRITING PROGRAM FILES

| | |
|---|------|
| 5.1 CREATING A PROGRAM | 5-1 |
| 5.2 PROGRAM FORM | 5-1 |
| 5.3 COMPILING A PROGRAM | 5-3 |
| 5.3.1 Compiler Options | 5-3 |
| 5.4 COMPILATION CONTROL | 5-5 |
| 5.4.1 Including Other Files in Your Program | 5-7 |
| 5.4.2 Conditional Compilation | 5-8 |
| 5.4.3 ++PRAGMA - Setting Compiler Options | 5-9 |
| 5.4.3.1 ++PRAGMA and Command Line Switches. | 5-9 |
| 5.4.3.2 Setting String Work Area Size | 5-10 |
| 5.5 RUNNING A PROGRAM | 5-11 |
| 5.5.1 Runtime Options | 5-11 |

CHAPTER 6 - VARIABLES

| | |
|---|------|
| 6.1 NAMES | 6-1 |
| 6.1.1 Variable, Label, and User Defined Function Names | 6-1 |
| 6.1.1.1 Variable Names | 6-2 |
| 6.1.1.2 Label Names | 6-3 |
| 6.1.1.3 User Defined Function Names | 6-3 |
| 6.1.2 External Assembly Language Subroutine (XCALL) Names ... | 6-3 |
| 6.1.3 Subprogram Names | 6-3 |
| 6.2 VARIABLE TYPES | 6-4 |
| 6.2.1 Numeric Variables | 6-4 |
| 6.2.1.1 Integers | 6-4 |
| 6.2.1.1.1 True Integers | 6-5 |
| 6.2.1.1.2 Binary Variables | 6-5 |
| 6.2.1.1.3 Other Integer Representations | 6-5 |
| 6.2.1.2 Real Numbers | 6-6 |
| 6.2.1.2.1 AMOS Floating Point Numbers | 6-6 |
| 6.2.1.2.2 IEEE Floating Point Numbers | 6-6 |
| 6.2.1.3 Real Number Characteristics | 6-7 |
| 6.2.1.4 Real Number Mathematics | 6-7 |
| 6.2.2 String Variables | 6-9 |
| 6.2.3 Unformatted Variables | 6-10 |
| 6.2.4 Literals and Constants | 6-10 |
| 6.2.4.1 Integer Literals and Character Constants | 6-10 |
| 6.2.4.2 Alternative Radix Constants | 6-11 |
| 6.2.4.3 Floating Point Literals | 6-12 |
| 6.2.4.4 String Literals | 6-12 |
| 6.2.5 Default Data Type | 6-13 |

| | |
|--|------|
| 6.3 ARRAY VARIABLES | 6-13 |
| CHAPTER 7 - EXPRESSIONS AND EVALUATION | |
| 7.1 WHAT IS AN EXPRESSION? | 7-1 |
| 7.2 MATHEMATICAL EXPRESSIONS | 7-1 |
| 7.3 LOGICAL OPERATORS | 7-2 |
| 7.4 OPERATOR PRECEDENCE | 7-3 |
| 7.5 RULES OF CALCULATION | 7-4 |
| 7.6 MIXED TYPE RULES | 7-6 |
| 7.7 DIVISION AND MODULUS OPERATIONS | 7-7 |
| 7.8 THE SIGNIFICANCE OF SIGNIFICANCE | 7-8 |
| CHAPTER 8 - WORKING WITH STRING VARIABLES | |
| 8.1 WHAT IS A SUBSTRING MODIFIER? | 8-1 |
| 8.2 HOW DO YOU USE A SUBSTRING MODIFIER? | 8-1 |
| 8.3 SETTING THE SIZE OF THE STRING WORK AREA | 8-4 |
| PART THREE - COMMANDS AND FUNCTIONS | |
| CHAPTER 9 - PROGRAM STATEMENTS | |
| 9.1 AMOS | 9-1 |
| 9.2 CALL | 9-2 |
| 9.3 CASE | 9-2 |
| 9.4 CHAIN | 9-2 |
| 9.5 DATA | 9-3 |
| 9.6 DEFAULT | 9-3 |
| 9.7 DEFINE | 9-3 |
| 9.8 DIM | 9-4 |
| 9.9 DIVIDE'BY'0 | 9-5 |
| 9.10 DO WHILE/UNTIL LOOP | 9-5 |
| 9.11 ECHO | 9-6 |
| 9.12 ELSE | 9-6 |
| 9.13 END | 9-6 |
| 9.14 ENDSWITCH | 9-6 |
| 9.15 EXIT | 9-7 |
| 9.16 FOR, TO, NEXT AND STEP | 9-7 |
| 9.17 GOSUB (OR CALL) AND RETURN | 9-9 |
| 9.18 GOTO | 9-12 |
| 9.19 IF, THEN AND ELSE | 9-12 |
| 9.20 INPUT | 9-14 |
| 9.21 INPUT LINE | 9-17 |
| 9.22 INPUT RAW | 9-18 |
| 9.23 LET | 9-18 |
| 9.24 LOOP | 9-19 |
| 9.25 NEXT | 9-19 |
| 9.26 NO'DIVIDE'BY'0 | 9-19 |

| | | |
|------|--------------------------|------|
| 9.27 | NOECHO | 9-19 |
| 9.28 | ON CTRLC GOTO AND RESUME | 9-19 |
| 9.29 | ON ERROR GOTO AND RESUME | 9-20 |
| 9.30 | ON GOSUB (CALL) | 9-20 |
| 9.31 | ON - GOTO | 9-21 |
| 9.32 | PRINT | 9-21 |
| 9.33 | PRINT USING | 9-23 |
| 9.34 | PROGRAM | 9-24 |
| 9.35 | RANDOMIZE | 9-24 |
| 9.36 | READ, RESTORE, AND DATA | 9-25 |
| 9.37 | RENAME | 9-26 |
| 9.38 | REPEAT | 9-26 |
| 9.39 | RESTORE | 9-27 |
| 9.40 | RESUME | 9-27 |
| 9.41 | RETURN | 9-27 |
| 9.42 | SCALE | 9-27 |
| 9.43 | SIGNIFICANCE | 9-27 |
| 9.44 | SLEEP | 9-28 |
| 9.45 | STEP | 9-28 |
| 9.46 | STOP | 9-29 |
| 9.47 | STRSIZ | 9-29 |
| 9.48 | SWITCH/CASE | 9-29 |
| 9.49 | THEN | 9-31 |
| 9.50 | TO | 9-31 |
| 9.51 | UNTIL | 9-31 |
| 9.52 | USING | 9-31 |
| 9.53 | WHILE | 9-31 |
| 9.54 | XCALL | 9-31 |

CHAPTER 10 - FUNCTIONS

| | | |
|---------|-------------------------|------|
| 10.1 | NUMERIC FUNCTIONS | 10-2 |
| 10.1.1 | ABS(X) | 10-2 |
| 10.1.2 | ASC(X) | 10-2 |
| 10.1.3 | EXP(X) | 10-2 |
| 10.1.4 | FACT(X) | 10-2 |
| 10.1.5 | FIX(X) | 10-3 |
| 10.1.6 | INT(X) | 10-3 |
| 10.1.7 | LOG(X) | 10-3 |
| 10.1.8 | LOG10(X) | 10-3 |
| 10.1.9 | RND(X) | 10-3 |
| 10.1.10 | RNDN(X) | 10-4 |
| 10.1.11 | SGN(X) | 10-4 |
| 10.1.12 | SQR(X) | 10-4 |
| 10.1.13 | VAL(A\$) | 10-4 |
| 10.2 | TRIGONOMETRIC FUNCTIONS | 10-4 |
| 10.3 | STRING FUNCTIONS | 10-5 |
| 10.3.1 | CHR(X) | 10-5 |
| 10.3.2 | EDIT\$(A\$,C) | 10-5 |
| 10.3.3 | FILL\$(A\$,L) | 10-6 |

| | |
|------------------------------|-------|
| 10.3.4 INSTR(X,A\$,B\$) | 10-6 |
| 10.3.5 LCS(A\$) | 10-7 |
| 10.3.6 LEFT(A\$,X) | 10-7 |
| 10.3.7 LEN(A\$) | 10-8 |
| 10.3.8 MID(A\$,X,Y) | 10-8 |
| 10.3.9 RIGHT(A\$,X) | 10-9 |
| 10.3.10 SPACE(X) | 10-9 |
| 10.3.11 STR(X) | 10-10 |
| 10.3.12 STRIP\$(A\$) | 10-10 |
| 10.3.13 TIME | 10-10 |
| 10.3.14 UCS(A\$) | 10-10 |
| 10.4 MISCELLANEOUS FUNCTIONS | 10-11 |
| 10.4.1 CMDLIN | 10-11 |
| 10.4.2 DITOS(X) | 10-11 |
| 10.4.3 DSTOI(X) | 10-11 |
| 10.4.4 ERRMSG(X) | 10-11 |
| 10.4.5 ERR(X) | 10-12 |
| 10.4.6 FILEBLOCK(X) | 10-12 |
| 10.4.7 GETKEY(X) | 10-12 |
| 10.4.8 ODTIM | 10-13 |

CHAPTER 11 - SYSTEM AND FILE FUNCTIONS

| | |
|------------------------------------|------|
| 11.1 BYTE(X), WORD(X), AND LONG(X) | 11-1 |
| 11.2 DATE | 11-2 |
| 11.3 IO(X) | 11-3 |
| 11.4 MEM(X) | 11-3 |
| 11.5 TIME | 11-4 |
| 11.6 LOOKUP | 11-4 |
| 11.7 KILL | 11-5 |
| 11.8 RENAME | 11-6 |
| 11.9 VER\$ | 11-6 |

CHAPTER 12 - FORMATTING OUTPUT

| | |
|---|------|
| 12.1 THE USING MODIFIER | 12-1 |
| 12.2 HOW TO SPECIFY FORMAT STRINGS | 12-3 |
| 12.2.1 Formatting a Numeric Field (#) | 12-3 |
| 12.2.2 Formatting a String Field (\) | 12-4 |
| 12.2.3 One-character String Fields (!) | 12-5 |
| 12.2.4 Using Decimal Points in Numeric Fields (.) | 12-5 |
| 12.2.5 Dollar Signs and Numeric Fields (\$\$) | 12-6 |
| 12.2.6 Putting a Comma Every Three Digits (,) | 12-7 |
| 12.2.7 Fill Leading Blanks with Asterisks (**) | 12-7 |
| 12.2.8 Fill Leading Blanks with Zeros (Z) | 12-7 |
| 12.2.9 Add a Trailing Minus Sign to a Number (-) | 12-7 |
| 12.2.10 Printing Numbers as Exponents (^^^) | 12-8 |
| 12.3 FORMATTING EXAMPLES AND HINTS | 12-8 |
| 12.4 TAB FUNCTIONS | 12-9 |

PART FOUR - ADVANCED PROGRAMMING**CHAPTER 13 - SCALED ARITHMETIC**

| | |
|---|------|
| 13.1 THE SCALE STATEMENT | 13-2 |
| 13.2 HOW THE SCALING FACTOR WORKS | 13-3 |

CHAPTER 14 - MAPPING VARIABLES

| | |
|--|-------|
| 14.1 WHAT IS VARIABLE MAPPING? | 14-1 |
| 14.2 THE FORMAT OF MAP STATEMENTS | 14-2 |
| 14.2.1 MAP Level | 14-3 |
| 14.2.2 Variable Names in MAP Statements | 14-4 |
| 14.2.3 Type | 14-5 |
| 14.2.3.1 Unformatted Data | 14-5 |
| 14.2.3.2 String Data | 14-5 |
| 14.2.3.3 Floating Point Data | 14-6 |
| 14.2.3.4 Binary Data | 14-6 |
| 14.2.3.5 Integer Data | 14-6 |
| 14.2.4 Size | 14-6 |
| 14.2.5 Value | 14-6 |
| 14.2.6 Origin | 14-7 |
| 14.3 USING MAP STATEMENTS | 14-9 |
| 14.3.1 Examples | 14-9 |
| 14.4 USING MAPPED DATA WITH DATA FILES | 14-11 |
| 14.5 HOW VARIABLES ARE ALLOCATED IN MEMORY | 14-12 |
| 14.6 LOCATING VARIABLES DURING DEBUGGING | 14-13 |
| 14.6.1 Examples | 14-14 |

CHAPTER 15 - THE FILE INPUT/OUTPUT SYSTEM

| | |
|---|-------|
| 15.1 WHAT IS A DISK FILE? | 15-1 |
| 15.2 FILE CHANNELS | 15-2 |
| 15.3 INPUT/OUTPUT STATEMENTS | 15-2 |
| 15.4 SEQUENTIAL FILES | 15-3 |
| 15.5 RANDOM FILES | 15-4 |
| 15.5.1 Logical Records | 15-5 |
| 15.5.2 Figuring Out How Many Blocks Your File Needs | 15-5 |
| 15.5.3 Accessing Random Files | 15-6 |
| 15.6 FILE LOCKING FOR SEQUENTIAL FILES | 15-8 |
| 15.7 FILE LOCKING FOR RANDOM FILES | 15-8 |
| 15.8 FILE LOCKING FOR ISAM PLUS FILES | 15-8 |
| 15.9 FILE STATEMENTS | 15-9 |
| 15.9.1 ALLOCATE | 15-9 |
| 15.9.2 CLOSE | 15-10 |
| 15.9.3 CLOSEK | 15-10 |
| 15.9.4 EOF(X) | 15-10 |
| 15.9.5 FILEBASE | 15-11 |
| 15.9.6 INPUT | 15-11 |
| 15.9.7 INPUT LINE | 15-13 |

| | |
|--|-------|
| 15.9.8 INPUT RAW | 15-14 |
| 15.9.9 KILL | 15-14 |
| 15.9.10 LOOKUP | 15-15 |
| 15.9.11 OPEN | 15-16 |
| 15.9.11.1 OPEN Modes | 15-18 |
| 15.9.12 PRINT | 15-19 |
| 15.9.13 READ | 15-19 |
| 15.9.14 READL | 15-20 |
| 15.9.15 READ'READ'ONLY | 15-20 |
| 15.9.16 UNLOKR | 15-21 |
| 15.9.17 WRITE | 15-21 |
| 15.9.18 WRITEL | 15-22 |
| 15.9.19 WRITEN | 15-22 |
| 15.9.20 WRITELN | 15-22 |
| CHAPTER 16 - CHAINING AND SUBPROGRAMS | |
| 16.1 THE CHAIN STATEMENT | 16-1 |
| 16.1.1 Chaining to another AlphaBASIC PLUS Program | 16-1 |
| 16.1.2 Chaining to System Functions | 16-2 |
| 16.2 WHAT IS A SUBPROGRAM? | 16-3 |
| 16.2.1 Subroutines, User Defined Functions, and Subprograms | 16-4 |
| 16.2.2 Types of Subprograms | 16-5 |
| 16.2.2.1 Considerations for Internal Subprograms | 16-5 |
| 16.2.3 Subprogram Structure | 16-5 |
| 16.2.4 Subprogram Components | 16-6 |
| 16.2.4.1 Subprogram Name | 16-7 |
| 16.2.4.2 Parameters | 16-7 |
| 16.2.4.3 Subprogram Code | 16-9 |
| 16.2.4.4 SUBEND | 16-9 |
| 16.2.5 Error Handling in Subprograms | 16-9 |
| 16.2.6 Using Subprograms | 16-10 |
| 16.2.7 Saved Runtime Environment | 16-11 |
| CHAPTER 17 - ERROR TRAPPING | |
| 17.1 WHAT IS ERROR TRAPPING? | 17-1 |
| 17.2 THE ON ERROR GOTO STATEMENT | 17-1 |
| 17.3 ERR(X) FUNCTION | 17-2 |
| 17.3.1 Error Codes Returned by ERR(0) | 17-2 |
| 17.4 THE RESUME STATEMENT | 17-3 |
| 17.5 CONTROL-C TRAPPING | 17-4 |
| 17.6 A SAMPLE ERROR RECOVERY ROUTINE | 17-4 |

CHAPTER 18 - EXTERNAL ASSEMBLY LANGUAGE SUBROUTINES

| | |
|---|------|
| 18.1 WHY USE ASSEMBLY LANGUAGE SUBROUTINES? | 18-1 |
| 18.2 AUTOMATIC SUBROUTINE LOADING | 18-2 |

CHAPTER 19 - USING ISAM PLUS FILES

| | |
|--|------|
| 19.1 WHAT IS ISAM PLUS? | 19-1 |
| 19.2 ISAM PLUS FILE STRUCTURE | 19-2 |
| 19.3 UPDATING THE RECORDS OF AN ISAM PLUS FILE | 19-2 |
| 19.3.1 OPEN Statement for ISAM PLUS Files | 19-3 |
| 19.3.2 GET, GET'LOCKED, and GET'READ'ONLY | 19-3 |
| 19.3.3 GET'NEXT, GET'NEXT'LOCKED, and GET'NEXT'READ'ONLY | 19-4 |
| 19.3.4 GET'PREV, GET'PREV'LOCKED and GET'PREV'READ'ONLY | 19-4 |
| 19.3.5 FIND, FIND'NEXT, and FIND'PREV | 19-5 |
| 19.3.6 UPDATE'RECORD | 19-5 |
| 19.3.7 CREATE'RECORD | 19-6 |
| 19.3.8 DELETE'RECORD | 19-6 |
| 19.3.9 RELEASE'RECORD | 19-6 |
| 19.3.10 RELEASE'ALL | 19-7 |
| 19.3.11 CLOSE | 19-7 |
| 19.3.12 CLOSEK | 19-7 |
| 19.3.13 UNLOKR | 19-7 |
| 19.4 CREATING AN ISAM PLUS FILE | 19-8 |
| 19.4.1 ALLOCATE'INDEXED | 19-8 |
| 19.5 RETRIEVING STATISTICAL INFORMATION | 19-8 |
| 19.5.1 INDEXED'STATS | 19-8 |
| 19.6 ERROR PROCESSING | 19-9 |

CHAPTER 20 - USER DEFINABLE FUNCTIONS

| | |
|---|------|
| 20.1 WHY USE DEFINED FUNCTIONS? | 20-1 |
| 20.2 SINGLE LINE FUNCTIONS | 20-1 |
| 20.3 MULTIPLE LINE FUNCTIONS | 20-2 |
| 20.4 LOCAL VARIABLE AND PARAMETER TYPES | 20-3 |
| 20.5 USING DEFINED FUNCTIONS | 20-4 |
| 20.6 EXAMPLES | 20-5 |

CHAPTER 21 - THE UNIFY RDBMS INTERFACE

| | |
|-------------------------------|------|
| 21.1 WHAT IS UNIFY? | 21-1 |
| 21.2 UNIFY XCALL FORMAT | 21-1 |

CHAPTER 22 - THE ALPHABASIC PLUS DEBUGGER

| | |
|--|------|
| 22.1 SETTING UP YOUR PROGRAM FOR DEBUGGING | 22-1 |
| 22.2 CALLING THE DEBUGGER | 22-1 |
| 22.3 WHAT YOU SEE | 22-1 |
| 22.4 CONTROLLING PROGRAM EXECUTION | 22-2 |
| 22.5 COMMAND MODE | 22-3 |
| 22.5.1 Print Variable Value | 22-4 |

| | |
|---|-------|
| 22.5.2 See Variable Information | 22-4 |
| 22.5.3 Set a Breakpoint | 22-4 |
| 22.5.4 Clear a Breakpoint | 22-4 |
| 22.5.5 Execute AMOS Command | 22-5 |
| 22.5.6 Find a Specific Label | 22-5 |
| 22.5.7 Trap a Verb | 22-5 |
| 22.5.8 Turn Off TRAP | 22-5 |
| 22.5.9 Send Program Display to Another Terminal | 22-6 |
| 22.5.10 Display File Channel Information | 22-6 |
| 22.5.11 Print Error Value | 22-6 |
| 22.5.12 Print Information on All File Channels | 22-6 |
| 22.5.13 Display Function Source Code | 22-7 |
| 22.5.14 Indent Text | 22-7 |
| 22.5.15 Display List of Labels | 22-7 |
| 22.5.16 Change a Variable Value | 22-7 |
| 22.5.17 Re-Start Debugging | 22-8 |
| 22.5.18 Run Program From Program Pointer | 22-8 |
| 22.5.19 Exit Debugger | 22-8 |
| 22.5.20 Clear Error | 22-8 |
| 22.5.21 Clear Error and Resume Execution | 22-8 |
| 22.5.22 Set Scale Factor | 22-8 |
| 22.5.23 Set Significance Value | 22-8 |
| 22.5.24 Break on Value Change | 22-9 |
| 22.5.25 Remove Tracepoint | 22-9 |
| 22.5.26 Display List of Variables | 22-9 |
| 22.5.27 Break at External Program | 22-9 |
| 22.5.28 Break at Subprogram | 22-9 |
| 22.6 DEBUGGING A SUBPROGRAM | 22-10 |

PART FIVE - APPENDICES

APPENDIX A - MESSAGES

APPENDIX B - RESERVED WORDS

APPENDIX C - PROGRAMMING HINTS

| | |
|--|-----|
| C.1 MANAGING MEMORY | C-1 |
| C.2 INCREASING EXECUTION SPEED | C-2 |
| C.3 MAKING YOUR SOURCE PROGRAMS EASIER TO READ | C-2 |

APPENDIX D - SCREEN HANDLING CODES

APPENDIX E - WRITING ASSEMBLY LANGUAGE SUBROUTINES

| | |
|---|-----|
| E.1 REGISTER PARAMETERS | E-2 |
| E.2 ARGUMENT LIST FORMAT | E-2 |
| E.3 CONVERTING ARGUMENTS TO BINARY FORMAT | E-3 |
| E.4 FREE MEMORY USAGE | E-4 |
| E.5 LOCATING OPEN FILES | E-4 |
| E.6 PROGRAM HEADERS | E-4 |

APPENDIX F - CHARACTER SETS**APPENDIX G - ERR(3) ERROR CODES****APPENDIX H - MEMORY**

| | |
|--|-----|
| H.1 MEMORY REQUIREMENTS | H-1 |
| H.2 MEMORY MANAGEMENT | H-2 |
| H.2.1 The .BPR file | H-2 |
| H.2.2 .BPR File Settings | H-2 |
| H.2.3 RUNP Memory Partition Layout | H-3 |
| H.2.4 RUNP'S Use of the Stacks, Heap and Free Area | H-4 |
| H.2.5 Calculating Runtime Requirements | H-6 |
| H.2.6 COMPLP and RUNP C Settings | H-6 |

APPENDIX I - THE STRING WORK AREA**APPENDIX J - COMPILER AND RUNTIME OPTIONS**

| | |
|--|-----|
| J.1 RESOLVING THE DILEMMA | J-1 |
| J.2 OPTION SPECIFICATION | J-1 |
| J.3 CHECKING AVAILABILITY OF OPTIONS | J-2 |
| J.4 ALTERNATIVE METHOD OF OPTION SPECIFICATION | J-2 |
| J.5 EFFECTS OF USING OPTIONS | J-4 |
| J.6 DEFINED OPTIONS | J-5 |

GLOSSARY**DOCUMENT HISTORY****INDEX**

CHAPTER 1

INTRODUCTION

This chapter introduces you to AlphaBASIC PLUS and discusses some of the concepts we'll be using throughout the manual. The topics are:

- Who is this manual written for?
- What is BASIC?
- How is AlphaBASIC PLUS special?
- What can AlphaBASIC PLUS do?
- The programs that make up the AlphaBASIC PLUS system
- What does an AlphaBASIC PLUS program look like?
- Graphic conventions we'll be using

1.1 WHO IS THIS MANUAL WRITTEN FOR?

This is a reference manual for the AlphaBASIC PLUS language. It is not meant to be a tutorial for the BASIC language, or to teach concepts of programming.

However, we have tried to make this manual easy to use and understand. In general, the text of this manual is written with the assumption you are inexperienced with BASIC.



A glossary is provided near the back of this manual. It defines terms used in AlphaBASIC PLUS, and other terms relating to this manual and to the Alpha Micro system.

If you are an experienced BASIC programmer, you may want to glance through sections 2 through 4 to familiarize yourself with the syntax and features of AlphaBASIC PLUS. As you program you will probably find the *AlphaBASIC PLUS Quick Reference Card* to be helpful to remind you of commands and syntax.

If you are new to or relatively inexperienced with the BASIC language, you should read Part I before beginning to use AlphaBASIC PLUS. You may also want to read a tutorial on BASIC, and/or a tutorial about programming concepts.

1.2 WHAT IS BASIC?

The acronym BASIC stands for: **B**eginners' **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.

BASIC is a higher-level programming language created to be a versatile tool for learning computer programming, and also to provide a relatively simple language for a wide variety of applications.

But today, BASIC is more than that. Most programming on small, interactive systems is done in BASIC. One of the reasons is BASIC is so much like the English language.

Over the years since its inception, BASIC has been added to and modified as new concepts of programming have emerged. Some versions of BASIC are more extensive than others; the use of extended versions provides the programmer with a wider range of applications, greater ease in programming, and greater efficiency and speed.

1.3 HOW IS ALPHABASIC PLUS SPECIAL?

AlphaBASIC PLUS is an extended version of the BASIC language, with several features not found in other BASIC implementations. These features not only enhance the performance of traditional uses of the language but also make business applications easier to program. Here are some of AlphaBASIC PLUS's features:

- MAP statements that make it easy to create hierarchical data structures, much like the structured data types available in the COBOL and C languages.
- The ability to call assembly language subroutines from within AlphaBASIC PLUS.
- Variable names that may be any number of alphanumeric characters, which helps improve program source readability.
- The ability to use labels instead of or in addition to line numbers. Eliminating line numbers makes updating programs much easier, and makes program source code easier to read and understand.
- The ability to define your own single and multi-line functions.
- The ability to use subprograms, allowing parts of programs to be compiled separately.
- The ability to use AMOS commands from within programs.

- A full range of control structures.

1.4 WHAT CAN ALPHABASIC PLUS DO?

AlphaBASIC PLUS can be used in two different ways—either in interactive mode, or by using disk files, a compiler, and a run-time package.

Interactive mode operates much like a traditional interactive interpreter; that is, you create, alter and test your program while it is in your memory partition. This mode is convenient for the creation and debugging of new programs. It also makes it easy to quickly test a program by altering lines or data within the program.

Using disk files, the compiler, and a run-time package is more useful for programs which are to be put into production use, or for testing programs which are too large to fit in memory in the interactive mode.

Using AlphaBASIC PLUS in this way, you create the program file using the AlphaXED or AlphaVUE program editor (or use a disk file saved from an interactive mode session), then you compile the program at monitor level (using the COMPLP program). Finally, you use the run-time package (the RUNP program) to execute the program.

One of the advantages of using AlphaBASIC PLUS in this way is AlphaXED makes it easy to write and edit your program. See Chapter 5 for more information about creating and editing programs with AlphaXED.

During the actual running of the compiled program, only the object code and the run-time execution package are in memory. This conserves memory on the computer.

The compiler and the run-time package are both written as re-entrant programs. This means that, in a timesharing environment, any or all users who are running or debugging programs may share one copy of the compiler and the run-time package. The way to do this is to load COMPLP.LIT (the compiler program) and RUNP.LIT (the run-time package) into system memory, optionally along with the files COMPLP.RTI and RUNP.RTI. See your *System Operator's Guide* for information on loading files into system memory.

Once created by the compiler, an object program (also known as a compiled program) is also re-entrant and sharable, and may be placed in system memory or in the AlphaBASIC PLUS program account BP: (DSK0:[7,35]) so it may be shared by all users on your system.

1.5 WHAT TYPES OF DATA DOES ALPHABASIC PLUS SUPPORT?

AlphaBASIC PLUS supports the following data formats:

| | |
|----------------|--|
| FLOATING POINT | Numbers including a decimal fraction |
| STRING | ASCII alphanumeric characters |
| BINARY | Whole numbers stored in binary, computed as floating point |
| UNFORMATTED | Special variables set up by MAP statements |
| INTEGER | Whole numbers stored and computed in binary |

All data formats may be simple variables or array structures. In addition, the unique memory mapping system allows you to specify the ordering of variables in special groups. This "grouping" of variables makes them easier to use, and makes your program more efficient.

The MAP statement is similar to the data formatting capabilities of the COBOL language or a C structure, and lends itself well to business applications where the grouping of related information is important.

Variable names are not limited to the single character and single digit format of many BASICs, but may be any reasonable number of alphanumeric characters in length (since lines are limited to just under 1000 characters, this defines the outer limits of variable size), as long as the first character is alphabetic. Apostrophes and underscores can also be used. This is another feature which makes AlphaBASIC PLUS well suited for business applications.

Since the source code is compiled and need not be in memory when the program is run, the length of the variable name is not a significant concern. Label names may also be used to identify points in the program for GOTO and GOSUB branches. Label names are alphanumeric and help to clarify the program structure. Some examples of label names:

```
EXIT'ERROR:
EVALUATE'ANSWER:
PROCESS_DATA
```

Variable names and data formats are discussed in more detail in Chapter 6.

1.6 THE ALPHABASIC PLUS SYSTEM

The AlphaBASIC PLUS system consists of three programs:

| | |
|------------|--|
| BASICP.LIT | A BASIC PLUS module that combines an interactive compiler with a run-time package to provide an interactive BASIC environment. |
| COMPLP.LIT | AMOS level disk-based AlphaBASIC PLUS compiler. |
| RUNP.LIT | AMOS level AlphaBASIC PLUS run-time package. |

Each of these programs has a corresponding .RTI file which is used for internal initialization when the .LIT file executes for the first time.

You use RUNP and COMPLP from monitor command level to run and compile AlphaBASIC PLUS programs that exist as disk files. You use the BASICP command when you want to use AlphaBASIC PLUS in interactive mode.

With these three programs you can use AlphaBASIC PLUS in two different ways—either as an interactive system (interactive mode) or as a disk-file/compiler system.

Your choice of how you use AlphaBASIC PLUS depends on several factors:

- The amount of memory you have in your user partition
- What stage of development your program is in
- The physical form of your program
- The size of your program
- Your personal preference

Chapters 3 and 4 tell you more about the Interactive mode, and Chapter 5 explains how to use AlphaXED, the compiler, and the run-time package together.

1.7 EXTENSIONS USED WITH ALPHABASIC PLUS

As with all AMOS files, your AlphaBASIC PLUS programs can have names made up of from 1 to 6 alphanumeric characters. The default file extension for AlphaBASIC PLUS source programs is .BP. If your source program has this extension, you don't need to specify the extension when you compile the program.

The COMPLP program automatically creates a file with the same name as your source file, but with a .RP extension. This extension is the default for the RUNP program.

Other extensions you may use with AlphaBASIC PLUS include .BPI (an include file), .SPG (a compiled external subprogram), and .XBR (an assembly language subroutine). These are discussed later in this manual.

1.8 WHAT DOES AN ALPHABASIC PLUS PROGRAM LOOK LIKE?

Here is a small sample program written in AlphaBASIC PLUS:

```
PRINT "Hello!"
INPUT "Type in a number you would like squared: ",NUMBER
ANSWER = NUMBER * NUMBER
PRINT "The square of"; NUMBER; "is:" ;ANSWER
END
```














Notice how the AlphaBASIC PLUS statements used in the program above are English words (PRINT, INPUT, END) which give you a good idea of what they do. Because of the way AlphaBASIC PLUS handles variable names, you can give your variables names that give you an idea of what they are for (i.e., NUMBER and ANSWER). You can learn more about the above statements later in the manual.

1.9[∞]GRAPHICS CONVENTIONS

This manual conforms to the other Alpha Micro publications in its use of a standard set of graphics conventions. We hope these graphics simplify our examples and make them easier for you to use. Unless stated otherwise, all examples of commands are assumed to be entered at AMOS command level.

| SYMBOL | MEANING |
|----------|---|
| devn: | Device-Name. The "dev" is the three letter physical device code, and the "n" is the logical unit number. Examples of device names are DSK0:, DSK5:, WIN1:, and MTU0:. Usually, device names indicate disk drives, but they can also refer to magnetic tape drives and video cassette recorders. |
| filespec | File Specification. A file specification identifies a specific file within an account. A complete filespec is made up of the devn:, the filename, the file extension, and the project-programmer number. For example: devn:filename.ext[p,pn] -or- DSK0:SYSTEM.INI[1,4] |
| [p,pn] | This abbreviation represents an account on a disk where you can store files and data. An actual disk account number looks like this: [100,2] or [1,4]. Disk account specifications are sometimes referred to as "Project-programmer numbers." |
| { } | Braces are used in some examples to indicate optional elements of a command line. In the example: DIR{/switch} the braces tell you "/switch" is not a required portion of the DIR command line. |

(continued)

| SYMBOL | MEANING |
|---|--|
| / | <p>The slash symbol precedes a command line switch or "option request." For example:</p> <p>DIR/WIDE:3 </p> <p>This command requests a directory display of the disk account you are currently logged into. The switch (/WIDE:3) indicates you want the display to be three columns wide.</p> |
| TEXT | Bold text in an example of user/computer communication represents the characters you type. |
| TEXT | Text like this in an example of user/computer communication represents characters the computer displays on your terminal screen. |
|  | <p>In our examples, the key symbol appears whenever you need to press a certain key on your terminal keyboard. The name of the key you need to press appears inside the key symbol, like this: . If you need to press the TAB key, you would see , or the ESCAPE key, . (Sometimes the ESCAPE key is labeled ESC or ALT MODE.)</p> |
|  /  | This indicates a control sequence you press on the keyboard. Press  and hold it down while the indicated key is pressed. |
| ^ | This symbol in front of a capital letter means the letter is a "control character." For example, when you press  /  , it appears on your screen as ^C. (^C is the control character that cancels most programs and returns you to AMOS command level.) |
|  | This symbol means "halt!" It indicates an important note you should read carefully before going further in the documentation. Usually, text next to this symbol contains instructions for something you MUST or MUST NOT do, so read it carefully. |
|  | This symbol means "hint." It indicates a helpful bit of information, or a "short cut" that could save you time or trouble. |
|  | This symbol means "remember." It indicates something you should keep in mind while you are following a set of instructions. |

CHAPTER 2

GENERAL INFORMATION

This chapter gives general information about the form your AlphaBASIC PLUS programs may take. We discuss:

- Lines with more than one BASIC statement
- Statements using more than one line
- Numbering lines
- How to put comments in your programs
- Using labels in programs
- How AlphaBASIC PLUS gives programs memory
- Reserved word spacing
- The case of letters in programs
- AlphaBASIC PLUS libraries

2.1 LINE NUMBERS

Program line numbers may range from 1 to 65535. Programs used in interactive mode **MUST** contain line numbers. Programs to be compiled at AMOS level do not require line numbers. You may use line numbers only for certain lines in a compiled program if you like—just because you use line numbers does not mean all the lines have to have numbers.



If you have a line number on the line at which an error occurs, AlphaBASIC PLUS is able to tell you where that error occurred by displaying that line number. Without line numbers, it can only report an error occurred, but not where.

You may want to use line numbers in special places when debugging the program to help locate errors. A disadvantage of line numbers is they occupy space in memory, making your program slightly larger and slower.

2.2[∞]LABELS

AlphaBASIC PLUS allows the use of labels to identify locations in a program. The exact rules for label names are described in Chapter 6. Briefly, a label must be the first thing on a line (apart from an optional line number), it must start with a letter, contain only alphanumeric characters, apostrophes, and underscores, and be terminated by a colon. For example:

```
ERROR'MESSAGE:
Addition_Routine:
START:
```



It is important to remember you may not place a space between the label and its colon; doing so causes AlphaBASIC PLUS to think you have entered a multi-statement line rather than a label.

A label may be followed by a program statement on the same line, or it may be the only item on the line. The use of labels is similar to the use of line numbers with GOTO and GOSUB statements, and makes the program easier to document. Here is a program that uses labels:

```
START'PROGRAM:
    INPUT "Enter two numbers to get the sum: ",A,B
    PRINT A; "+"; B; "="; A + B
    IF A + B <> 0 GOTO Sum_not_Zero
    PRINT "The sum is zero"
    GOTO END'PROGRAM
Sum_not_Zero:
    PRINT "The sum is not zero"
END'PROGRAM:
    END
```

where START'PROGRAM:, Sum_not_Zero:, and END'PROGRAM: are labels. Note a reference to a label (such as GOTO END'PROGRAM) does not have to be terminated by a colon. The reference must be identical to the actual label in its case (upper and/or lower) and in the placement of delimiters. For example, GOTO ENDPROGRAM or GOTO End'Program causes an error message when referring to END'PROGRAM. For more information on GOTO, see Chapter 9.

You may use AlphaBASIC PLUS reserved words as labels (END:, REM:, etc).

2.3[∞]COMMENTS (REM AND !)

AlphaBASIC PLUS allows you to insert comments into your source program in two different ways. The keyword REM may appear alone on a line followed by the comment, or may be inserted on the same line as a statement, to comment on the purpose of the statement. You may follow the REM (or "remarks") keyword with anything you want. For example:

```
REM  ANYTHING YOU WISH TO SAY
PRINT A      REM -->  VARIABLE A MEANS "ALLOWANCE"
```

Note any statement following a REM keyword on a line is NOT executed by AlphaBASIC PLUS. When the program is compiled, everything following the REM statement on the line is ignored.

The comment symbol ! is an abbreviation of the REM statement, and is used the same way. For instance:

```
PRINT "TRY ANOTHER TIME"      ! IF THEY MISS BETWEEN
GOTO AGAIN                    ! ONE AND THREE TIMES.
```

Like the REM statement, anything following the ! symbol on the line is ignored. You may "comment out" a section of a program containing comments. For example:

```
REM  PRINT "TRY ANOTHER TIME"  ! IF THEY MISS BETWEEN
REM  GOTO AGAIN                ! ONE AND THREE TIMES.
```

2.4[∞]MULTIPLE STATEMENT LINES

AlphaBASIC PLUS supports multiple statement lines. Multiple statement lines can be formed by using colons to separate the statements. For example, these lines:

```
FOR I = 1 TO 10
    PRINT "This is a loop."
NEXT I
```

could be written as:

```
FOR I = 1 TO 10 : PRINT "This is a loop." : NEXT I
```

The exception is a DATA statement cannot contain other statements on the same line, and no other statements may follow a comment (designated by REM or !). Direct statements in interactive mode may also be multiple statement lines.

2.5[∞]CONTINUATION LINES

COMPLP allows the use of continuation lines within the source program. That is, statements may be continued on the next line by using the ampersand (&) symbol as the last character on the line. A comment may come after the ampersand (but it must begin with !, not REM), since it is not part of the code. For example:

```
IF STOCK'NUMBER > 599 THEN &          ! Compare stock #
GOTO OVER'STOCKED
```

Notice the example above, though appearing on two lines in your source file, is actually treated as one statement by AlphaBASIC PLUS. Therefore, do not separate a line

being continued from the line before by a label or line number. For example, this code causes an error:

```
                IF STOCK'NUMBER > 599 THEN &  
END'LOOP:                GOTO OVER'STOCKED
```

You may also use the ampersand symbol after a colon.

Since any statement line may be indented as you please when your program is a disk file, use of continuation lines and indentation, plus eliminating line numbers (as discussed in the next section) allows you to give your source program a more structured look than allowed by more conventional BASICs. Here's an example of a "structured" code format:

```
TIME'OF'DAY = TIME/3600  
IF TIME'OF'DAY > 12 &  
    AND TIME'OF'DAY < 13  
    THEN PRINT "It's Lunch Time"  
    ELSE PRINT "The day's journey is half over " &  
        "but you still have work to do."  
ENDIF
```

The maximum size of any AlphaBASIC PLUS statement, including blanks, tabs, ampersands, and carriage return/line feeds, is just under 1000 characters.

2.6[∞]MEMORY ALLOCATION

In interactive mode, memory is allocated dynamically as you edit your program, and also during its compilation and execution. Checks are made to tell you if you have run out of memory. If you do, you get an error message. If you run out of memory while COMPLP is compiling a disk program, you see an error message and the compilation stops. See Appendix I for more information on AlphaBASIC PLUS memory allocation.

2.7[∞]SEARCHING FOR A LIBRARY

Whenever a program or subprogram (called with a RUNP command, a CHAIN statement, or a SUBCALL statement) or a subroutine (called with an XCALL statement) is requested, AlphaBASIC PLUS tries to execute the requested program/subroutine.

If the program/subroutine you request is not specified with a complete file specification, AlphaBASIC PLUS looks for it assuming a number of default specifications. AlphaBASIC PLUS follows a specific pattern in looking for the requested run or subroutine module. If you specify an account, then AlphaBASIC PLUS uses the current default device and the specified account. If you don't specify an account, the search sequence is:

1. ∞ Your memory partition
2. ∞ System memory
3. ∞ Your disk:[Your account]
4. ∞ Your disk:[Your Library account (P,0)]
5. ∞ The AlphaBASIC PLUS Library account, DSK0:[7,35]

By "your account" we mean the account you are currently logged into. If you specify a device, AlphaBASIC PLUS does not search in memory but proceeds directly to that device, using steps 3 and 4.

CHAPTER 3

INTERACTIVE MODE

This chapter discusses the AlphaBASIC PLUS interactive mode. The topics are:

- What is Interactive Mode?
- Clearing memory for a new program
- Creating a program
- Saving a program
- Compiling a program
- Running a program
- Direct statements

3.1 WHAT IS INTERACTIVE MODE?

We call it "interactive mode" because you interact directly with the AlphaBASIC PLUS interpreter as you write or edit your program. As you enter each line and press **RETURN**, AlphaBASIC PLUS looks at the line and determines if it has been entered correctly—whether what you enter is a legal AlphaBASIC PLUS statement. Interactive mode is like being "inside" BASIC, and gives you a number of advantages:

- Since AlphaBASIC PLUS checks each line as you enter it, it can help you learn how to use the AlphaBASIC PLUS programming statements.
- You can easily change lines or variables between program runs, making testing various parts of your program or variable ranges and values simple.
- Interactive mode allows you to run your program one line at a time. This is called the "single-step" feature.
- You can also set "breakpoints"—places inside your program, which cause the run to pause. These pauses, combined with the "single-step" feature, can help you locate errors in your programs.

There are also a few disadvantages to interactive mode:

- Because you are sharing memory with the AlphaBASIC PLUS interpreter program, your programs cannot be as large as if they were created with AlphaVUE (we'll discuss that method of writing programs in Chapter 5).
- You must use line numbers, making it more difficult to read and modify your program. Having line numbers restricts your program to a maximum size of 65,535 lines.
- It is more difficult to edit your program, because you must retype an entire line to change something.

Therefore, you will probably use interactive mode if:

- You are new to AlphaBASIC PLUS, and want to try out BASIC statements or small programs.
- Your program is in the early stage of its development, and you want to use the debugging features.
- You want to use direct statements to do mathematical calculations or test short processes.

To use interactive mode, first make sure you are at monitor level (where you see the AMOS prompt symbol). If you do not see a prompt, press **CTRL/C**. If the prompt still does not appear, see your System Operator. When you see the prompt, enter:

```
BASICP RETURN
```

You then see:

```
AlphaBASIC PLUS Version X.X(XXX)  
READY
```

READY is the AlphaBASIC PLUS prompt symbol. You are now in interactive mode.

You can specify many of the switches used with the AMOS level COMPLP command when you enter the BASICP command. These switches are then in effect for the entire interactive session. For example:

```
BASICP/I RETURN
```

To leave interactive mode and return to AMOS, enter:

```
BYE RETURN
```

3.2[∞]DIRECT STATEMENTS

Program statements that do not begin with a line number are considered direct statements, and AlphaBASIC PLUS executes them immediately. For example:

```
READY
A = 5 RETURN
PRINT A + 4 RETURN
9
```

The first direct statement, `A = 5`, stores 5 as the value of the variable A in memory. The second, `PRINT A + 4`, adds 4 to the value of A and, since this is a direct statement, it prints out the answer, 9. Each direct statement is compiled and executed when it is entered. You can define variables and change variable values using direct statements.

Certain statements are meaningless as direct statements, and so are not allowed (for example, `RESUME`, `GOSUB`, etc.). If you enter an illegal statement, you see: `?Illegal in immediate mode.`

AlphaBASIC PLUS allows multi-statement lines as direct statements. Multi-statement lines are lines which contain more than one statement; the statements are separated by colons. As you enter direct statements, AlphaBASIC PLUS checks them to see they are in proper form. If they are not, you see an error message. Continuation lines are not permitted.

3.3[∞]CLEARING MEMORY FOR A NEW PROGRAM (NEW)

When you first enter interactive mode, there is nothing in memory, so you can simply type in your program. After being in Interactive Mode for a while, if you want to create or load in a new program, you should first make sure there is nothing in memory. To clear memory, enter:

```
READY
NEW RETURN

READY
```

`NEW` erases the program currently in memory. If you do not erase the existing program, AlphaBASIC PLUS merges the new program into whatever is in memory. If any line numbers from the new program duplicate line numbers of an old program in memory, the new lines replace the old.

3.4[∞]CREATING A PROGRAM

To create a program in interactive mode, first make sure you have nothing in memory (see NEW above), and then begin entering program lines. Each line must begin with a line number. You do not have to enter the lines in numerical order—AlphaBASIC PLUS puts them in order for you. For example, you might enter:

```
10 INPUT "Enter a number: ", NUM RETURN
30 PRINT NUM RETURN
20 PRINT RETURN
```

If you make a mistake when you enter a line, you see:

```
40 A = A X A RETURN
      ^
?Syntax Error
```

This means AlphaBASIC PLUS did not understand your input. The caret points to the part of the line that caused the error. Re-type the correct line:

```
40 A = A * A RETURN
```



Subroutines (called by SUBCALL) may be used in Interactive mode, but after the subroutine is done, it exits you to AMOS command level. If you are going to use subroutines, keep this in mind and be sure to SAVE your program before running it. To edit a program line, you must re-type the entire line as we did above. You may use ++INCLUDE in interactive mode, but not as a direct statement.

3.5[∞]DELETING PROGRAM LINES (DELETE)

The DELETE command is used to delete lines from the program text. You can delete a line from the program by entering DELETE, and the line number. For example:

```
READY
DELETE 40 RETURN
```

You can also delete a group of line numbers by specifying a beginning and ending number. For example, to delete all lines between (and including) 40 and 150:

```
READY
DELETE 40,150 RETURN
```



Although you usually separate the two line numbers with a comma, you can also use a dash, space, or some other non-numeric characters. The symbols \, &, \$, +, and % are used by AlphaBASIC PLUS, and won't be accepted.

Here is an example of a program listing before and after a DELETE:

```

LIST RETURN
10  FOR I = 1 TO 10
20      PRINT TAB(I) "ONE"
30      PRINT TAB(I) "TWO"
40      PRINT TAB(I) "SIX"
50      PRINT TAB(I) "TEN"
60  NEXT I

```

READY

```
DELETE 20,40 RETURN
```

READY

```

LIST RETURN
10  FOR I = 1 TO 10
50      PRINT TAB(I) "TEN"
60  NEXT I

```

READY

3.6[∞]SAVING A PROGRAM (SAVE)

Once you have your program entered (or even when it is partially entered), you may want to save the program on the disk. In interactive mode, your program is only in temporary memory, and it is erased when you leave interactive mode. To save a program, enter SAVE and a name for the program. For example:

```

READY
SAVE NEWPRG RETURN

```

READY

The command above saves the source program NEWPRG.BP as a disk file in the account you are logged into (.BP is the default extension).



All program names must be made up of one to six alphanumeric characters. All file extensions must be made up of one to three alphanumeric characters.

Programs are saved as sequential ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) files. Since this is a standard format, you can use the AlphaVUE or AlphaXED text editor to edit your file later.

If you want to SAVE the program to another account, or with a different extension, you can specify a full file specification. For example:

```

READY
SAVE NEWPRG.SAM[125,3] RETURN

```

or:

```
READY
SAVE DSK2:PAYROL.BP[50,1] 
```

You can save the compiled version of that program by specifying the .RP extension (we discuss compiling programs and .RP files later):

```
READY
SAVE NEWPRG.RP 
```

If you have not previously compiled the source program, or if you have changed the program since the last time you compiled it, AlphaBASIC PLUS automatically compiles it for you when you save an .RP file to ensure you are saving the most current version.

If you try to save a file when there is no source program in memory, AlphaBASIC PLUS displays: ?No source program in text buffer.

If a previous version of the program (one with the same name) already exists on the disk in the account you are writing the file to, that program is first deleted before the new program is saved. AlphaBASIC PLUS doesn't automatically create a backup file, so be careful when choosing a name for a program. You can use the LOOKUP command as an interactive direct statement to see if the file exists on the disk. For example:

```
LOOKUP "TEST.RP",result
PRINT result
```

If the number printed is not zero, the file exists. See Chapter 15 for more information on LOOKUP. The resulting object program (.RP file) is re-usable, and re-entrant (it may be loaded into system memory so everyone on your system can use the program—even at the same time).

In the interests of security, AlphaBASIC PLUS doesn't let you save a program in an account not within the same project as the account you are logged into. For example, if you are logged into DSK2:[100,2] and you try to save a program in DSK2:[340,1], you see: ?Cannot open NEWPRG.BP[340,1] - protection violation.

Since AlphaBASIC PLUS cannot convert an object file back to a source program file, you probably want to save the .BP version of your program.



After saving a program from within interactive mode, the source program still exists in interactive memory until you use NEW or until you exit interactive mode.

3.7[∞]LISTING A PROGRAM (LIST)

If you have a program in memory, you can display it by entering LIST:

```
READY
LIST 
10 INPUT "Enter a number: ", NUM
20 PRINT
30 PRINT NUM
40 INPUT "Enter a second number: ", NUM2
50 PRINT
60 PRINT NUM, NUM2, (NUM * NUM2)
```

If the program is longer than fits on your screen, use the NO SCRL key (or press / and /) to stop and resume the display, or press / to interrupt the listing. If you only want to see part of your list, you can ask for a section by adding the beginning and ending line numbers to the LIST command. For example:

```
READY
LIST 30,50 
30 PRINT NUM
40 INPUT "Enter a second number: ", NUM2
50 PRINT
```

Or, if you want to see a single line:

```
READY
LIST 60 
60 PRINT NUM, NUM2, (NUM * NUM2)
```



You may use a blank space or a hyphen in the LIST command where a comma is used (for example: LIST 30 50 or LIST 30-50).

3.8[∞]COMPILING A PROGRAM

Each statement you enter is checked when you press . If the statement is correct, nothing happens. If there is an error in the statement, you see an error message.

When you run a program (explained below) AlphaBASIC PLUS automatically compiles the program if it is not currently compiled. You can also use the COMPILE command to compile the program. For example:

```
READY
COMPILE 
COMPILING
Compile time was 0.003 seconds, elapsed time was 0 seconds

READY
```

In interactive mode, you may compile only the program currently in memory.

The resulting object program (.RP file) is re-usable (it can be used over and over again), and re-entrant (it may be loaded into system memory so everyone on your system can use the program—even at the same time).

Once the program is compiled, the object code resides in memory along with the source program. Compiling a program sets all variables to zero and deletes all variables left over because of direct statements. If no program is in memory, you see the message:
?No source program in text buffer.

3.8.1[∞]Compiler Options

You may specify any or all of the compiler options the system level COMPLP program allows when using the BASICP command. These are explained in Chapter 5. You may specify options when you use BASICP to enter interactive mode—these options then become the default for that session. You may also use the options with the COMPILE statement to effect only that specific compilation.

3.9[∞]RUNNING A PROGRAM (RUN)

To execute a program, enter RUN:

```
READY
RUN 
```

```
( The program begins running )
```

RUN resets all of the variables to zero and all string variables to a NULL value (a string containing no characters) before running the program. If you wish to stop the program during the run, press /.

If the program was changed since it was last compiled, RUN automatically recompiles it before executing it. Therefore, if you need to compile the source program and then run it, just use RUN. For example:

```
READY
10 REM This is a small test program 
20 FOR I = 1 TO 3 
30 PRINT "Little tasks make large return." 
40 NEXT I 
RUN 
COMPILING
Compile time was 0.003 seconds, elapsed time was 0 seconds
Little tasks make large return.
Little tasks make large return.
Little tasks make large return.
```



```
CPU time was 0.004 seconds, elapsed time was 0 seconds  
READY
```

You can't run a program other than the one currently in memory while in interactive mode.

3.10[∞]EXITING FROM INTERACTIVE MODE (BYE)



If you have a program in memory when you use BYE, it is erased. So, if you wish to keep a copy of the program, use SAVE before using BYE.

To exit from interactive mode:

```
READY  
BYE RETURN
```

3.11[∞]A HINT FOR THE SYSTEM OPERATOR

Because BASICP.LIT is re-entrant, you may load it in system memory to save room in user partitions. However, since it is a fairly large program, you probably only want to put it in system memory if many users use AlphaBASIC PLUS in interactive mode.

CHAPTER 4

MORE INTERACTIVE COMMANDS

This chapter discusses:

- Using the BREAK command to interrupt program runs
- Continuing a program after a BREAK
- Exiting from a program run using Control-C
- Loading programs into BASIC from the disk
- Using the single-step feature to debug programs

4.1 INTERRUPTING A PROGRAM RUN (BREAK)

The BREAK command can aid you in finding errors in your programs. It can be used to set, clear, and display breakpoints. For example:

| | |
|----------------|--|
| BREAK | Lists all breakpoints currently set |
| BREAK 100 | Sets a breakpoint at line 100 |
| BREAK 30, 50 | Sets breakpoints at lines 30 and 50 |
| BREAK -30 | Clears the breakpoint at line 30 |
| BREAK 20, -120 | Sets a breakpoint at line 20, clears the one at line 120 |

When you set a breakpoint, you are asking interactive AlphaBASIC PLUS to stop just before the line you specified. When the program is run, it proceeds until it hits the breakpoint. Then you see: `Break at line x.`

At this point you can enter interactive mode commands, proceed line by line using the single-step feature (see below), or enter direct statements. The value of being able to enter direct statements is you can find out what values the variables have, or even change those values. For example:

```
READY
RUN RETURN
Break at line 40
PRINT A RETURN
5
```

Here the value of the variable A is 5 before line 40 is executed. If your program is not giving you the answer you expect, you can use breakpoints at key points in your program to determine where the calculations are going wrong.

You can also change the value of a variable. Say in the example above you wanted the value of A to be 4. By entering as a direct statement `A = 4`, you can change the value for the duration of the program run.

There is no limit to the number of breakpoints that may be set in one program. The program doesn't run any slower with breakpoints in it than normally.

Compiling a program does not clear breakpoints. If you want to see what breakpoints are set, enter **BREAK**:

```
READY
BREAK RETURN
No breakpoints set

READY
BREAK RETURN
30      60
```

You may start the program over again by using the **RUN** command; it again breaks at the first breakpoint set.

4.2[∞]CONTINUING AN INTERRUPTED PROGRAM (CONT)

CONT, which stands for "continue," causes a suspended program to continue execution from the point at which it suspended. You may suspend a program by using a **BREAK** command prior to program execution or by using a **STOP** statement within the program. You may not continue a program after it has finished. The following is an example of **CONT** after a **STOP** statement has suspended a program:

```
Program stop in line 700
READY
CONT RETURN
```

The program continues by next executing the first line numbered higher than line 700. **CONT** also continues a program which you have partially executed using the single-step feature (see below).

4.3[∞]STOPPING A PROGRAM RUN

Pressing **CTRL/C** stops a running program. Depending on the operation being done, **^C** may display on your terminal. The line number of the source program at which the program was interrupted is displayed in the message. For example:

```

CTRL/C
^C   ( or nothing )
Operator interrupt in line 70
READY

```

You cannot continue a program after you have entered a **CTRL/C**.

4.4[∞]LOADING A PROGRAM INTO MEMORY (LOAD)

If you have a program saved as a disk file, you can load it into temporary memory so you can edit or run it. To do this, enter **LOAD** and the name of the file. For example:

```

READY
LOAD NEWPRG.BP RETURN
READY

```

This program file could have been created using AlphaVUE, or might have been saved from a previous interactive mode session. If you do not supply a file extension, AlphaBASIC PLUS uses the default extension of **.BP**. AlphaBASIC PLUS assumes the account and device you are logged into. If AlphaBASIC PLUS can't find the file you want to load, it displays: **?Cannot open <filespec> - file not found."**

The **LOAD** command does not clear the text buffer before it loads the requested file, and therefore may be used to concatenate or merge several programs or subroutines together to be saved as a single program.

The separate routines must not duplicate line numbers in the other routines they are to be merged with or else the new line numbers overlay the old ones just as if the file had been typed in from your terminal.



If you want to load in a complete program, use the **NEW** command prior to any **LOAD** command to make sure memory is clear. If you don't, the contents of a program in memory may mix with the program you are loading.

Here are two examples of **LOAD**:

```

READY
LOAD NEWBAS RETURN
READY
LOAD DSK2:PWRS2.BP RETURN
READY

```

4.5 THE SINGLE-STEP FEATURE

The single-step function is a feature not found in many versions of BASIC, and is useful in debugging programs and in teaching the principles of BASIC programming to people who are new to BASIC. Once you have a program (or even part of a program) in memory, you can use the single-step command by pressing ↓ (the LINEFEED key, sometimes labeled LF). When you press ↓, the next line of the program displays on your terminal, and then executes. Here is an example:

```

LIST RETURN
10  CONSTANT = 7
20  FOR I = 1 to 2
30  PRINT : INPUT "Enter a number: ",NUM
40  VALUE = VALUE + (CONSTANT + NUM - I)
50  NEXT I
60  PRINT VALUE

READY
↓
COMPILING
Compile time was 0.21 seconds, elapsed time was 1 seconds
10  CONSTANT = 7
↓
20  FOR I = 1 to 2
↓
30  PRINT : INPUT "Enter a number: ",NUM
Enter a number: 4 RETURN
↓
40  VALUE = VALUE + (CONSTANT + NUM - I)
PRINT VALUE RETURN
10
↓
50  NEXT I
↓
30  PRINT : INPUT "Enter a number: ",NUM
Enter a number: 154.5 RETURN
↓
40  VALUE = VALUE + (CONSTANT + NUM - I)
PRINT VALUE RETURN
169.5
VALUE = 160 RETURN
↓
50  NEXT I
↓
60  PRINT VALUE
160
*** End of Program ***

```

Note line 30 is a multi-statement line. When single-stepping, all statements on a line are executed at the same time.

At each point after a line has been executed, you can use direct statements to look at or change variable values (as we did with the `PRINT VALUE` and `VALUE = 160` direct statements in the example) before you continue the run of the program by using `CONT` or by pressing ↓.

Note any change in the source program causes AlphaBASIC PLUS to recompile the program. The next single-step command then executes the first line of the program. When we changed the variable `VALUE` to 160, we didn't change any of the program lines—therefore, AlphaBASIC PLUS didn't recompile the program. Only a variable was changed.

The single-step feature can be used at the beginning of the program, after program `STOP` statements and breakpoint interrupts.

After partially single-stepping through a program, you may execute the remainder of it normally by using `CONT`. Also, you may start over at the beginning and execute it normally by using `RUN`. If you try to single-step past the `*** End of Program***` message, the first program statement executes again.

If you single-step a statement asking for input from the terminal, enter the input and press `RETURN` (as we did in the example above). Then you may proceed to the next statement by pressing ↓ again.

CHAPTER 5

WRITING PROGRAM FILES

In this chapter we discuss:

- Using AlphaXED or AlphaVUE to create a program
- The form of an AlphaBASIC PLUS program
- Compiling a program
- Running a program

Although interactive mode has many good features, the true power of AlphaBASIC PLUS is in using AlphaXED, COMPLP, and RUNP to create, compile, and run your programs as disk files.

5.1 CREATING A PROGRAM

There are two ways to create a source program: using interactive mode to type in the program, save the program on disk, and then exit BASICP; or using AlphaXED or AlphaVUE. The easiest way to create a program to be compiled with COMPLP is to use AlphaXED to create the source file (which usually has a .BP extension).

AlphaXED is a screen-oriented program editor that lets you see your program on the terminal screen as you type it in. You can move the cursor around on the screen and change or delete text at the current cursor position. It also has unique features designed especially to aid programmers in writing programs. See your *AlphaXED User's Manual* if you are not familiar with AlphaXED. AlphaVUE can also be used.

5.2 PROGRAM FORM

The form your program takes may differ somewhat depending on whether you use AlphaXED or the interactive mode in creating your program. If you create and save your source program in interactive mode, that program must contain line numbers (otherwise, the interactive interpreter assumes a direct statement).

The monitor level compiler (COMPLP) doesn't require a program to have line numbers. That means if you create your program using AlphaXED, you don't need line numbers. In addition, you may indent your program lines in any way you like.

By omitting line numbers and using labels, and by using indentation judiciously, you can make your source program easier to read. A "label" is a special name defined by you that identifies a location within a program.



Although COMPLP doesn't require your program to have line numbers, it does check for duplicate line numbers within your file. Even when the /O switch is in effect, the compiler reports duplicate line number errors. It does NOT, however, check for lines out of numeric order. See below for information on the /O option.

The compiler also allows the use of statements that start on one line and continue on to another line. These are called continuation lines and are discussed in section 2.4. For example:

```
IF Answer = Right'Number THEN &
    PRINT "Very Good! " &
ELSE &
    PRINT "Try again: "
```

As you may have noticed, the above example is only one AlphaBASIC PLUS statement, even though it is written on four lines. However, you may not want to use continuation lines if you want to work on your program in interactive mode. Interactive BASICP puts continuation lines back together to form one line. Remember also an AlphaBASIC PLUS statement cannot be longer than about 1000 characters, even if continuation symbols are used. Here is an example of a program using continuation lines, indentation, and labels, and without line numbers:

```
STRSIZ 20      ! Program to print a name in reverse.
START:
    PRINT
    INPUT LINE "Enter your name: ",NAME$
    IF LEN(NAME$) = 0 &
        THEN GOTO START
    COUNTER = LEN(NAME$)
    PRINT
LOOP:
    FOR COLUMN = COUNTER TO 1 STEP -1
        PRINT NAME$[COLUMN;1];
    NEXT
    PRINT : PRINT
    INPUT "Would you like to try another? ",QUERY$
    IF LCS(QUERY$[1;1]) = "y" &
        THEN GOTO START &
        ELSE PRINT : PRINT "All done."
END
```


5.3[∞]COMPILING A PROGRAM

Compiling an AlphaBASIC PLUS program is done by using the COMPLP program. Compiling a program translates your AlphaBASIC PLUS source program into a language the RUNP program can execute. To compile a program, make sure you are at AMOS command level, then enter COMPLP and the name of your AlphaBASIC PLUS program, plus any options you wish. For example:

```
COMPLP PAYROL RETURN
```

The default extension is .BP. That is, if you don't add an extension to the filename, COMPLP looks for a file with that name and a .BP extension. You then see a number of statistics on your terminal as it compiles your program. COMPLP tells you if any errors exist within your source program as it processes your file.

COMPLP.LIT, RUNP.LIT, COMPLP.RTI and RUNP.RTI may be loaded into system memory or into your user memory partition. The fastest access, if multiple people use AlphaBASIC PLUS on your computer, is if you load them into system memory. See the LOAD and SYSTEM command reference sheets in your *System Commands Reference Manual*.

COMPLP uses memory in your user memory partition to process your program. If there is not enough memory in your partition for COMPLP to complete its task, you see an error message, and COMPLP returns your terminal to monitor command level. If this occurs, you may want to remove files loaded into your partition (see the DEL command reference sheet in your *System Commands Reference Manual*). If there are no programs loaded into memory, or deleting files does not help, see your System Operator about increasing the size of your memory partition.

When COMPLP has finished processing your file, it returns you to monitor level and, if there are no syntax errors, writes the object program to the disk as a file with the name of your source program and an .RP extension (in this case, PAYROL.RP). If there are errors displayed, no .RP file is created. The resulting object program (.RP file) is re-entrant, and may be loaded into system memory for use by multiple users. The .RP file can be executed using the RUNP program (see below).

5.3.1[∞]Compiler Options

The compiler has a number of optional "switches" controlling how the compilation works. To choose an option, include a slash (/) at the end of the file specification you give COMPLP, followed by the appropriate letter. If you want to add more than one switch, add another slash and letter. Each letter must be preceded by a slash. For example:

```
COMPLP PAYROL/O/N RETURN
```

Options are active for the entire source file, including subprogram definitions (see Chapter 16).

Here are the options:

| | |
|-------------|--|
| /A | Increases the size of the area in memory used to calculate program addresses. Normally, the compiler uses 16 bits—/A increases this to 24 bits. This adds to the size of your program, but lets you transfer control (when you use GOTO, GOSUB, RESUME, etc.) across a span greater than 64K. |
| /C:name=exp | Defines a constant called <code>NAME</code> with a value of <code>EXP</code> . <code>NAME</code> is the name of the constant being defined, and <code>EXP</code> is an expression consisting of one or more constant values. The syntax is the same as that for <code>DEFINE</code> (see section 9.7) without using the <code>DEFINE</code> keyword. You can define more than one constant from the command line by using multiple <code>/C</code> switches, one for each constant. The definitions are stored in an internal buffer for processing before the first line of the source file is read, and are processed sequentially in the order they are defined on the command line. The size of the buffer is approximately 200 bytes. |
| /D | Creates an <code>.RPD</code> file for use with the AlphaBASIC PLUS debugger (see Chapter 22). |
| /E:{#} | Specifies number (default is 100) of errors to allow before aborting compile. |
| /F | Normally, user-defined functions begin with <code>FN</code> , and variables may not begin with <code>FN</code> . This switch changes this so functions must begin with <code>FN'</code> . If you convert a program with many variables that begin with <code>FN</code> to AlphaBASIC PLUS, this lets those variables remain as they are without generating an error. |
| /G | Allows you to compile a program which does a <code>GOTO</code> or <code>GOSUB</code> into a loop structure. You will receive a warning message but the compile will complete. |
| /I | Selects IEEE format for floating point numbers as the default. See Chapter Six for more information on IEEE. |
| /L | COMPLP prints a message after processing every 1,000 lines of source file. |
| /M | Causes an error message to be displayed when COMPLP finds an unmapped variable. Helpful for verifying all variables are mapped. See Chapter 14 for more on mapping variables. |
| /N | Cancels the COMPLP statistical display. You see appropriate error messages, but none of the COMPLP statistics. |
| /O | Removes line number references in your compiled object code file. It still reports duplicate line number errors. /O makes your object code file smaller and makes the program run faster, but error messages don't show the line where the error occurred. |

`/P:option#`
`{,option#...}` Requests that the compiler and runtime systems select a particular option for processing. An `option` can change the default operation of either or both COMPLP and RUNP. Each option is assigned a number 1 through 64. See Appendix J for details.

`/PX{number}:`
`value` A shorthand way to enter a large number of options. `Number` is either 1 or 2. If `number` is 1, then `value` reflects settings for options 1 through 32. If `number` is 2, then `value` reflects the settings for options 33 through 64. See Appendix J for more details.



Specify as few options as possible. The compiler will make reasonable efforts to spot unneeded options and inform you. See Appendix J for details.

`/R` This switch can be used in two ways. First, to display features and available options of COMPLP. To do this, type:

COMPLP/R

Second, this switch can incorporate the time and date of compilation, together with COMPLP's version numbers, inside the run file. You can view the run file information by using RUNP's `/R` switch. To incorporate this information, type:

COMPLP filename/R



Note, this option will change the hash of the run file every time a compilation is done, even if no changes to the source code are made.



You can retrieve the compiler version number and time stamp information using the `VER$` function.

`/S` Changes the default XCALL extension from `.XBR` to `.SBR`. This is for compatibility purposes for systems running both AlphaBASIC and AlphaBASIC PLUS. It lets you place one XCALL subroutine in memory which can be called by both AlphaBASIC and AlphaBASIC PLUS programs.

`/T` Displays each line of your source program as it scans it. If a problem occurs during compilation, `/T` shows you the line where the problem occurs. You can also use `/T` to see how fast different statements compile.

`/X` Program uses traditional ISAM, not ISAM PLUS

`/?` Displays command line usage for COMPLP.

5.4[∞]COMPILATION CONTROL

AlphaBASIC PLUS has a number of commands that begin with `++`. These commands let you control how your program compiles. COMPLP supports all the following commands, while BASICP supports only the `++INCLUDE` command. The commands are:

| | |
|---------------------------------------|---|
| <code>++INCLUDE file {option}</code> | Brings another program into the program at the point of the command. See the section "Including Other Files in Your Program," below, for details. |
| <code>++ERROR string</code> | Causes a compilation error to print, and the compilation to stop. This is useful if used within an <code>++IF</code> clause, for instance, to signal a section of code as yet uncompleted. <code>string</code> is the error message to display. Note that <code>string</code> is printed out without being processed or interpreted in any way. |
| <code>++IF constant-expression</code> | Used like an IF-THEN statement, this lets you set a condition to be met before a section of the program is compiled. The constant-expression must use a defined value, not a variable. |
| <code>++ELSE</code> | Used with <code>++IF</code> to compile an alternate section of code if the constant-expression is false. |
| <code>++ELIF</code> | Short for "ELSE-IF"—lets you nest <code>++IF</code> clauses. |
| <code>++ENDIF</code> | Signals the end of the condition-compile section of code. |
| <code>++IFDEF variable</code> | Used like <code>++IF</code> , compiles the designation code if the variable is defined in the program. |
| <code>++IFNDEF symbol</code> | Used like <code>++IF</code> , compiles the designation code if the symbol is not defined in the program. |
| <code>++MESSAGE string</code> | Similar to <code>++ERROR</code> , but does not cause a compilation error to print or the compilation to stop. |
| <code>++PRAGMA setting {value}</code> | Affects the compilation process, allowing the setting of various options from within a source code file. Most <code>++PRAGMA</code> lines have the form: |

`++PRAGMA setting value`

which will cause the variable specified by `setting` to take on the value specified by `value`. See the section "`++PRAGMA` - Setting Compiler Options" below.

5.4.1[∞]Including Other Files in Your Program

The `++INCLUDE` statement lets you bring other files into your program at the point of the `++INCLUDE`. All the statements in the included file are compiled as part of your program. For example:

```
++INCLUDE INVEN.BPI
```

You may use `++INCLUDE` in interactive mode, but it cannot be a direct statement. Included files can be nested up to 16 deep.

You can specify these options with `++INCLUDE`

| | |
|---------------------|--|
| <code>/E:ext</code> | Changes the file extension that COMPLP uses as a default for <code>++INCLUDE</code> processing from <code>.BSI</code> to <code>.ext</code> for this <code>++INCLUDE</code> only. |
| <code>/P</code> | MAP1 statements are changed to PARAMETERs if the included file is to be used as a subprogram. Lets the same module be included either as program code or subprogram. |
| <code>/S:n</code> | Default search path. If <code>n=0</code> (the default), searches only your PPN. If <code>n=1</code> , searches your PPN and <code>[P,0]</code> account. If <code>n=2</code> , searches PPN, <code>[P,0]</code> , and <code>BP:.</code> |

The `/P` switch lets you use the same file in both programs and subprograms. For example, if you had a file called `LABEL.BPI` that contained:

```
MAP1  Label
      MAP2  Name
      MAP2  Street'address
      MAP2  City'state'zip
```

You could include it in a `.BP` file using `++INCLUDE LABEL.BPI`. If you wanted to include it in a subprogram file, however, normally it would not compile because subprograms use `PARAMETER` statements instead of `MAP1` statements. By specifying:

```
++INCLUDE LABEL.BPI/P
```

In your subprogram file, the `MAP1` statements would be compiled as `PARAMETER` statements, thus allowing them to work within the subprogram (this does not change the `LABEL.BPI` file).

There should be no spaces before the switches in an `++INCLUDE` line.

5.4.2[∞]Conditional Compilation

The ++IF statement lets you conditionally compile sections of code. In this example:

```

DEFINE FLAG = 1
.
.
++IF FLAG = 1
.      (section of AlphaBASIC PLUS code)
.
++ELSE
.
.      (section of AlphaBASIC PLUS code)
.
++ENDIF

```

The first section of code is compiled, since FLAG is defined as one. By redefining FLAG to 0, you can change to compiling only the second section of code by changing just the definition statement. Here's another example:

```

DEFINE FLAG = 1
.
.
++IF FLAG = 1
.      (section of AlphaBASIC PLUS code)
.
++ELSE
!
++ERROR Code incomplete. Change FLAG to 1 and re-compile.
!
++ENDIF

```

In this case, if FLAG is set to not equal 1, the compiler aborts with the message "Code incomplete. Change FLAG to 1 and re-compile." displayed.

AlphaBASIC PLUS defines two special constants that can only be used with conditional compilation directives: ++BASIC, and ++BASICP. ++BASIC always returns a FALSE value in AlphaBASIC PLUS: it is used by sections of code that should only be compiled under AlphaBASIC 1.4, a predecessor of AlphaBASIC PLUS. ++BASICP always returns a TRUE value in AlphaBASIC PLUS.

++IFDEF works much like ++IF, except it looks for whether or not a constant is defined. For example:

```

DEFINE FLAG = 1
.
.
++IFDEF FLAG
.
.    (section of AlphaBASIC PLUS code)
.
++ENDIF

```

The section of code following the ++IFDEF flag is compiled if FLAG is defined (as anything).

5.4.3[∞]++PRAGMA - Setting Compiler Options

++PRAGMA can cause the compiler to set or take a specified option. There are two groups of ++PRAGMA settings: command line switch settings and string work area settings.

5.4.3.1[∞]++PRAGMA and Command Line Switches.

Instead of specifying certain compiler options on the command line, ++PRAGMA settings can be used from within the source file. This can be a lot more convenient during the program's maintenance life cycle. The following ++PRAGMA settings correspond to command line switches

| ++PRAGMA Setting (case insensitive) | Command Line Switch | Note |
|--|------------------------|------|
| ERROR_IF_NOT_MAPPED | /M | 1 |
| FORCE_FN' (note apostrophe) | /F | 2 |
| FORCE_IEEE | /I | 1 |
| FORCE_OLD_ISAM | /X | 1 |
| FORCE_SBR_EXT | /S | 1 |
| FORCE_24BIT | /A | 2 |
| NO_LINENUMBERS | /O | 1 |
| TIMESTAMP | /R | 1 |
| FEATURE | /P | 2,3 |
| LIFESIGN | /L | 1 |

Notes:

1. This `++PRAGMA` can be put anywhere in the source file. The setting will take effect from that point. `++PRAGMA` can be used multiple times in the same file.
2. This `++PRAGMA` can only be put in the file once, at the top of the file before any object code is generated by the compiler.
3. This `++PRAGMA` option takes a list of comma-separated decimal values to turn on a particular option. See Appendix J for option values and definitions. Once selected, an option cannot be deselected.



Any `++PRAGMA` options set in a `++INCLUDEd` file will be active on return to the program that contained the `++INCLUDE` statement.

Each of the above `++PRAGMA` settings, except for `FEATURE`, takes a single parameter to tell the compiler to either turn the option on or off. The following case-insensitive values are recognized:

To turn the option on: "ON", 1, "TRUE", Yes-string, and Yes-character
(make it active) (each within double quotes) defined in the .LDF file of the job doing the compilation.

To turn the option off: "OFF", 0, "FALSE", No-string, and No-character
(make it inactive) (each within double quotes) defined in the .LDF file of the job doing the compilation.

`++PRAGMA` settings take effect immediately, and last for the rest of the source file, including any subprograms (see Chapter 16) defined in the file. If you specify both a `++PRAGMA` setting and its corresponding command line switch at compile time, the command line switch will override any and all corresponding `++PRAGMA` commands. This allows a default condition to be set up in the source file which can be overridden on an as-needed basis by the command line.

5.4.3.2 Setting String Work Area Size

The string work area is a runtime memory allocation which is used by RUNP's string processing. The size of this area can be set automatically by the compiler, but there are occasions where you need to adjust the size of this area. The following `++PRAGMA` setting affect the size of this area:

```
++PRAGMA ADD_EXTRA_STRWRK constant_expression
++PRAGMA EXTRA_STRWRK constant_expression
++PRAGMA SET_STRWRK constant_expression
```

`constant_expression` is a compile-time expression that follows the same rules as values for a `DEFINED` constant. The values must be between -2MB and +2MB. See Appendix I for a discussion on the string work area and its required size.

5.5[∞]RUNNING A PROGRAM

Enter RUNP and the name of the program you want to execute. For example:

```
RUNP LOOP 
```



You can also pass command line arguments to your program. See the CMDLIN statement in Chapter 10.

You may enter a full file specification, including device name and account number. The monitor looks for the run-time package, RUNP.LIT, in memory; if it isn't found in system or user memory, it loads RUNP into memory from the disk. RUNP prepares an area in memory for your program to use, then searches for your program in memory. If it is not there, RUNP loads the specified .RP file from the disk. Then RUNP executes your program. Upon completion, or if you press / to stop the program, RUNP returns you to monitor level.



After making changes to your program, you must re-compile it prior to using RUNP, or you will run the old version of the program.

5.5.1[∞]Runtime Options

The runtime system has one optional switch controlling its operation, /R. You can use /R in two ways. First, it can display features and available options of RUNP. To do this, type:

```
RUNP/R 
```

Second, you can use /R to display features and required option features (set by a COMPLP /P switch or ++PRAGMA FEATURE list) of an object file. To do this, type:

```
RUNP/R filename 
```

filename is the name of the object file you want information about.



Do not place the /R after the filename or else RUNP will execute the file with a command line of "/R."

CHAPTER 6

VARIABLES

This chapter discusses the following topics:

- Variable Names
- Label Names
- User Defined Function Names
- External Assembly Language Subroutine (XCALL) Names
- Subprogram Names
- Numeric Variables
- IEEE Variables
- String Variables
- Array Variables

6.1 NAMES

Names in AlphaBASIC PLUS crop up in many places. Variables and labels, user defined function names, XCALL subroutine names, and so on. This chapter defines rules for the formation of valid names.

6.1.1 Variable, Label, and User Defined Function Names

These names obey a common set of rules, in addition to some specific rules for their type. The common set of rules is that the name:

- Must start with an alphabetic character.

- Can contain alphanumeric characters, plus apostrophes and underscores, both of which are significant in the name.
- Is case sensitive.
- Must be less than 500 characters long.

The case sensitivity rule and the rule concerning the apostrophe and underscore mean that AlphaBASIC PLUS treats the following names as referring to different items:

| | | |
|----------|----------|-----------|
| REC'SIZE | Rec'Size | Rec''Size |
| rec'size | rec_size | rec'size' |

6.1.1.1 Variable Names

In addition, variable names must follow these additional rules:

- A variable name cannot start with the characters FN (or FN' if the /F COMPLP option is used), otherwise it is interpreted as a user defined function.
- The entire variable name cannot be in the list of reserved words in Appendix B. This check is done without regard to case.
- Optionally, the last character of the variable name can be either a \$ or a % to designate the variable's data type.

The following variable names are legal:

| | | |
|----------------|-----------------|-----------------|
| CUSTOMERNUMBER | CUSTOMER'NUMBER | CUSTOMER_NUMBER |
| CUSNO | CUST1'NO | CUST2''NO_3 |
| CUS_NO' | CUSNO'' | CUSNO% |
| CUST'NAME\$ | | |

The following variable names are illegal:

| | |
|------------|--|
| 1ST'NAME | Does not start with an alphabetic character |
| _NAME | Does not start with an alphabetic character |
| FIRST NAME | Space terminates a name |
| FNAME | Legal only if the /F COMPLP switch is used |
| OUTPUT | Is a reserved word |
| Output | Is a reserved word (check is case insensitive) |

6.1.1.2 Label Names

Labels follow these rules in addition to the common rules listed above:

- A label must be the first item on a line, apart from an optional line number.
- A label must be terminated by a colon without any intervening spaces. The colon is used when defining the location of the label: it is not used when referring to the label.

6.1.1.3 User Defined Function Names

User defined functions are discussed in Chapter 20. A user defined function name must obey the following rule in addition to the common rules listed above:

- It must begin with the characters `FN` (or `FN'` if the program is compiled with the `/F` switch).
- Optionally, the last character of the user defined function name can be either a `$` or a `%` to designate the type of the return value. A dollar sign signifies a string return value, a percent sign signifies a four-byte true integer return value, and neither one being present signifies a floating point return type.

6.1.2 External Assembly Language Subroutine (XCALL) Names

External assembly language subroutines are discussed in Chapter 19. The name of the routine must obey these rules:

- The name must be one to six characters long
- The name can consist of any combination of letters A through Z and the numbers 0 through 9.
- Uppercase and lowercase letters are identical

6.1.3 Subprogram Names

Subprogram names follow the same rules as external assembly language subroutine names.

6.2 VARIABLE TYPES

AlphaBASIC PLUS supports three basic classes of variables: numeric, character, and unformatted. The first two classes are further subdivided.

- Numeric variables are variables whose contents are a number which can be used in mathematical calculations.
- Strings consist of zero or more consecutive characters.
- Unformatted variables are treated as containers of consecutive bytes without interpretation.

Each of these classes are described below.

6.2.1 Numeric Variables

The numeric variable class has two subclasses: whole numbers and floating point numbers:

- Whole numbers ("integers"), which cannot have a decimal fraction associated with their value.
- Floating point ("real") numbers, which have a decimal fraction associated with their value.

6.2.1.1 Integers

An integer variable can be one of two types:

- Unsigned integers, which can hold integer values greater or equal to zero.
- Signed integers, which can hold both positive and negative values as well as zero.

Thus 12 is an example of an integer, and you can use either a signed or unsigned integer to hold its value. The value -19 is another integer, but can only be held in a signed integer variable. The value 65.3 is a floating point number, and cannot be held exactly in an integer: the decimal fraction part (0.3) would be truncated and lost.

AlphaBASIC PLUS uses both types of integer variables:

6.2.1.1.1[∞]True Integers

True integers use the CPU's native signed integer instruction set for calculations, and are stored in native binary form in memory. Integers can be one, two or four bytes (the default) in size. Integer arithmetic is faster than real number arithmetic. This type of variable is defined by using a MAP statement for the variable name and specifying an **I** data type, or by appending a **%** to the variable name to specify a four-byte integer.

6.2.1.1.2[∞]Binary Variables

Binary variables come closest to the unsigned integer type. Binary variables can have different sizes, depending on the range of values they need to represent. Only 5-byte binary variables are signed variables, other (shorter) binary variables are unsigned. Although binary variables represent integers, they are always converted to AMOS floating point format (even if the **/I COMPLP** switch is used) before calculations are done. The results of the calculation is converted back to binary before being stored in the results variable. Binary variables can only be defined by using a MAP statement specifying a **B** data type. This conversion of binary variables has two important implications:

- 1.[∞]Binary arithmetic is slower than true integer arithmetic, due to the conversions to and from floating point format.
- 2.[∞]Floating point arithmetic is implicitly inexact due to rounding errors.



Note for assembler programmers. The four-byte binary (longword) variable does not have the same layout in memory as the four-byte true integer variable. In addition to the usual difference of the interpretation of the high-order bit as either a numeric value or a sign bit, the order of the two words comprising the longword value is swapped from native ordering. AlphaBASIC PLUS deals with this difference transparently. It is only evident when data is written to an external medium and read by a non-AlphaBASIC PLUS (or AlphaBASIC) program, or when data is passed to or from an assembly language subroutine.

6.2.1.1.3[∞]Other Integer Representations

There is also a special format for character constants and alternate radix constants that specify integer constants, which is detailed in a section below. These constants behave identically to true integers. The characteristics of the various integer data types are summarized below:

| Type | MAP | Signed? | Minimum Value | Maximum Value |
|---------|------|---------|------------------|-----------------|
| Binary | B, 1 | No | 0 | 255 |
| | B, 2 | No | 0 | 65,535 |
| | B, 3 | No | 0 | 16,777,215 |
| | B, 4 | No | 0 | 4,294,967,295 |
| | B, 5 | Yes | -549,755,813,888 | 549,755,813,887 |
| Integer | I, 1 | Yes | -128 | 127 |
| | I, 2 | Yes | -32,768 | 32,767 |
| | I, 4 | Yes | -2,147,483,648 | 2,147,483,647 |

6.2.1.2 Real Numbers

Real numbers (ones with a decimal fraction) come in two styles:

- 6-byte AMOS floating point numbers.
- 4-byte ("single precision") and 8-byte ("double precision") IEEE floating point numbers.

6.2.1.2.1 AMOS Floating Point Numbers

AMOS floating point numbers are manipulated by software, and are generally not comprehensible to non-AMOS systems without first being converting to another format. However, while on AMOS systems, they are very efficient.

6.2.1.2.2 IEEE Floating Point Numbers

IEEE floating point variables have their format and behavior strictly specified by an international standards body, the Institute of Electrical and Electronic Engineers. This rigorous specification ensures the transportability of floating point numbers between different hardware platforms, and the reproducibility of results of mathematical calculations, a notoriously difficult aim to achieve with inherently inexact representations.

IEEE variables may either be slower or faster in operation on your system than AMOS real numbers. If your system has an IEEE math chip incorporated (either as a separate chip or built into the CPU), IEEE variables may be faster than AMOS real numbers for identical CPU clock speeds. If not, AMOS real numbers are generally faster, but this may vary from system to system.

6.2.1.3°Real Number Characteristics

AlphaBASIC PLUS real numbers have the following characteristics:

| | AMOS Fpt | IEEE Single | IEEE Double |
|-------------------------------------|-----------------------|-----------------------|------------------------|
| MAP | F, 6 | F, 4 | F, 8 |
| Significant digits | 11 | 6 | 15 |
| Largest positive number (approx) | 1.7×10^{38} | 3.4×10^{38} | 1.8×10^{307} |
| Minimum positive number (approx) | 2.9×10^{-39} | 1.4×10^{-45} | 4.9×10^{-324} |

6.2.1.4°Real Number Mathematics

By definition, real numbers are approximations to absolute values. Real numbers can hold integers exactly up to a maximum value that depends on the floating point format, and certain values involving decimal fractions that are involved with powers of 1/2 (e.g, 1/2, 1/4, etc.) down to a certain value dependent on the format. Outside these values, only an approximate value is held. This is similar to trying to represent the number 1/3 as an exact decimal fraction: the number is approximately 0.333 - but not exactly: 0.33333 is a better approximation, and 0.3333333333 is even better but is still not exact. This is relevant when you do arithmetic on real numbers, because you can get some surprising results.

Example 1

If you evaluate the expression $10000/3 * 3$ by hand, you would get the answer 10000. But in real arithmetic, the floating point division yields 3333.33333333. which cannot be held exactly. Doing a floating point multiplication by 3 can give (say) 9999.9999876 as the answer. If this result was assigned to an integer, the result would be 9999, not the expected 10000, as integer conversion is done by truncation towards minus infinity (see Chapter 7).

Example 2

Adding small numbers to an already large real number can result in a loss of accuracy. For example, adding 1.0 to 1 hundred million million a total of one hundred times may result in a result of 1 hundred million million:

```

READY
10 SUM = 1E14
20 FOR I% = 1 TO 100
30   SUM = SUM + 1.0
40 NEXT
50 IF SUM = 1E14 PRINT "Loss of precision" ELSE PRINT "OK"
RUN
Loss of precision

```


The better way of totaling is to add all the smaller numbers together before adding that intermediate result to the larger number:

```

READY
10 SUM  = 1E14
20 SUM1 = 0
30 FOR I% = 1 TO 100
40   SUM1 = SUM1 + 1.0
50 NEXT
60 SUM = SUM + SUM1
70 IF SUM = SUM1 PRINT "Loss of precision" ELSE PRINT "OK"
80 PRINT SUM
RUN
OK

```

Note that if the smaller number was smaller than the example used, the second way of dealing with the additions may not even be sufficient: it all depends on the ratio of the larger to the smaller number, and the number of significant digits held in the binary floating point representation of the larger number.

Example 3

Even storing innocuous-looking decimal fractions with only a few decimal places (such as two decimal places for many currency values) can result in a very small loss of accuracy which is magnified as more calculations are done. This program has two subtle bugs even for small amounts:

```

MAP 1      FPT, F, 6
MAP 1      INT'VALUE, I, 4
INPUT "Enter monetary value:", FPT
INT'VALUE = FPT * 100

```

The conversion from ASCII (from the user's input) to floating point may not be exact. The conversion to four-byte integer is done by converting the result of a floating point multiply operation - which will exacerbate any loss of precision already encountered. For example, if the value input was 200.66, the value stored could be 20065! The result of the ASCII conversion and the floating point multiplication could be 20065.999987. When that is converted to an integer by truncation of the decimal fraction, the result is the unexpected and unwanted 20065. The programmer must defend against the possible loss of precision ("rounding error") inherent in floating point arithmetic by modifying the program to:

```

MAP 1      FPT, F, 6
MAP 1      INT'VALUE, I, 4
INPUT "Enter monetary value:", FPT
INT'VALUE = (FPT * 100) + 0.005

```

or

```

MAP 1      FPT, F, 6
MAP 1      INT'VALUE, I, 4
INPUT "Enter monetary value:", FPT
INT'VALUE = RNDN(FPT * 100)

```

The extra term added is one half of the smallest decimal fraction that can be entered. This would bump the intermediate result to 20066.444487, which correctly converts to 20066. The RNDN() function achieves the same result.

This is just a very brief overview on floating point arithmetic. Many books and papers have been written about floating point arithmetic and ways to reduce loss of precision and accuracy, and you are recommended to read one for further details.

6.2.2[∞]String Variables

A string is any character or group of consecutive characters. Strings can be from zero to 65,535 characters in length. The characters can be letters, numbers, typographical symbols, non-printing symbols, or any other character from the system's character set. Each character is stored in one byte in the computer. As a byte consists of 8 bits, there is a total of 256 values or unique characters that can be represented. If each byte is treated as an unsigned character, a character can have a value (or "code point") ranging from zero to 255 inclusive. The symbol (or "glyph") associated with each code point is locally determined by all the input and output devices used on the system. AMOS expects the devices to be consistent (i.e. be set to produce the same glyph for the same code point), defaulting to the ISO Latin-1 character set shown in Appendix F.

ISO Latin-1 is a 256-character set, and is a superset of the US ASCII character set, which itself consists 128 characters. A 128-character national replacement character set may be used instead. If the device settings are not consistent, printing, say, a glyph for code point 123 (decimal) may produce a { (left brace) on one device, but a Latin small letter E with an acute accent on another.

The only code point that cannot be stored in a string is zero, the null character, as this character is used internally by AlphaBASIC to signal the end of the string. Under AlphaBASIC PLUS, a string is defined to have a maximum size. Either the string fills up that size completely, or the string is terminated by a null character, and subsequent bytes (up to the size of the string) are undefined and inaccessible to programs. AlphaBASIC PLUS stops processing subsequent bytes after seeing the null character.

A string variable can be specified by appending a \$ to the variable's name, or by defining the variable in a MAP statement using the S data type (in which case the trailing dollar sign in the name is not required, but can be used if needed). If the dollar sign option is used without a MAP definition, the maximum size of the string is governed by the value associated with the STRSIZ statement last encountered by the compiler. If no STRSIZ statement has been encountered, the default size is 10 characters. A MAP statement definition also defines the size of the string.

String variables can be manipulated by string functions, such as LEFT\$, RIGHT\$, etc. as described in Chapter 11. Strings can be concatenated (joined together) by the + operator, and pulled apart by the functions shown in Chapter 11 and by the substring modifying features described in Chapter 8. String variables can be used in arithmetic expressions as described in Section 7.5 on mode independence.

6.2.3 Unformatted Variables

AlphaBASIC PLUS treats unformatted data as a set of consecutive bytes of a specified size. The maximum size of an unformatted variable is 65,535 bytes. Unformatted variables are defined by using a MAP statement with a data type of `x`, which automatically happens if an unformatted variable is MAPed with sub-variables at a lower MAP level. AlphaBASIC does not interpret the contents of an unformatted variable in any way. Usually unformatted data is only moved to another unformatted variable, or to or from an external device. The move is done by copying bytes from one variable to another. Any null bytes are transferred intact, and do not terminate a copy, unlike their behavior in a string operation. Assigning other data types to an unformatted data type and back can give strange results and the "round trip" is not guaranteed to produce an unchanged value, PRINTing unformatted variables to sequential files and subsequently INPUTing them also gives undefined results. All these operations are not recommended unless you understand all the ramifications and are willing to deal with undocumented behavior.

6.2.4 Literals and Constants

Literals and constants can have the following data types:

- Integer, also single character.
- Floating Point
- String

6.2.4.1 Integer Literals and Character Constants

Integer literals or constants are written as whole numbers without a decimal point or exponent symbol. Examples:

```
12      19347563      -72947756
```

Character constants can be used as expressions. They are distinguished by enclosing a value representing a single byte in single quote marks (apostrophes). They are equivalent to their corresponding code point ("ASCII value"), and are treated as true integers. Examples:

```
'a'    =    97
'A'    =    65
'Z'    =    90
```

A backslash can introduce a special value:

| | | |
|---------|-----|--|
| '\15' = | 13 | a slash followed by a number specifies an octal value |
| '\t' = | 9 | a tab character |
| '\n' = | 10 | a linefeed character |
| '\r' = | 13 | a carriage return character |
| '\f' = | 12 | a formfeed character |
| '\b' = | 127 | a delete character |
| '\'' = | 39 | an apostrophe |
| '\\' = | 92 | a backslash character itself |

Character constants are useful for clarifying code in many cases. For an example, it is clearer to write:

```
FOR I% = 'a' TO 'z'
```

than

```
FOR I% = 97 to 122
```

Note that the character constants are true integers. Therefore, you cannot set up a string containing carriage return and line feed by using:

```
READY
10 A$ = '\r' + '\n'
20 PRINT A$
RUN
1310
```

The mixing of data types within the same expression are outlined in a later section.

6.2.4.2[∞]Alternative Radix Constants

Alternative radix constants can also define true integer constants. As with character constants, they can often clarify your intentions in your source code. Only the compiler understands these symbols, so they cannot be used with DATA or INPUT statements. COMPLP understands the following alternative radix prefixes (which can be in either case):

| | |
|----|-------------------------------------|
| &h | hexadecimal format |
| &o | Octal format |
| &b | Binary format |
| &r | RAD50 format, three characters only |

The prefixes are placed immediately before the value they are acting upon, with no intervening spaces. For example:

```
EXTENSION% = &rRP
FLAG% = &h01 OR FLAG%
```

The *AMOS Monitor Calls Manual* contains conversion tables for these constants.

6.2.4.3 Floating Point Literals

Floating point numbers are defined by having either a decimal point or exponent marker (or both) embedded in the number.

Examples:

| | | |
|-------------|---------------|----------------|
| 12. | 12.0 | -1.05766445E24 |
| 1.92373D123 | 11.999275F-03 | 127125.234E29 |

A number with an exponent marker (D, E, or F) is interpreted as the decimal fraction multiplied by ten to the power of the number following the marker:

| | | |
|------------|---|----------------|
| 1E3 | = | 1 * 10^3 |
| -4.345E-05 | = | -4.345 * 10^-5 |

If an F marker is used, the type of the number is forced to be AMOS floating point. If a D marker is used, the number is forced to be an IEEE double-precision value. Types in either of the two previous cases are not affected by the /I COMPLP switch. If an E marker is used, the type of the value is dependent on the default floating point type: IEEE double precision if /I is set, otherwise AMOS floating point. As AlphaBASIC PLUS does not do any operations in IEEE single precision mode, there is no marker for single precision constants. Note that the F modifier is a compile time only feature that only works with literals or constant values. It will not work with DATA, READ, or INPUT statements.

6.2.4.4 String Literals

String constants (literals) are identified by being enclosed in quotation marks. For example:

```
NAME$ = "Mortimer T. Smith"
```

The value of NAME\$ (a string variable) is set to the string literal Mortimer T. Smith in the statement. String literals can include apostrophes:

```
NAME$ = "Mortimer T. Smith's"
```

Quotation marks can be included in the constant by writing two consecutive quotation marks:

```
READY
QUOTE$ = "Then he said, ""Hello, there""
PRINT QUOTE$
Then he said, "Hello there"
```

```

READY
QUOTE$ = "'Then he said, ""Hello, there""'"
PRINT QUOTE$
'Then he said, "Hello, there"

```

A string literal cannot be split over more than one line without using a runtime + (concatenation) operator.

6.2.5 Default Data Type

AMOS floating point format is the default format for all variables unless explicitly defined otherwise. This can be achieved by:

- Appending a \$ to a variable name to specify it is a string variable.
- Appending a % to a variable name to specify it is a four-byte integer variable.
- Using a MAP statement to set the data type and size of the variable. Note that if the MAP statement specifies the F data type without specifying its size, the format and size of the variable will change depending on whether or not the program was compiled with the /I switch. Compiling with the /I switch changes the default floating point format from AMOS to IEEE double precision.
- Compiling with the /I switch, changing the default type to IEEE double precision.

Note that numeric literals without a decimal point or exponent marker are always integer variables.

6.3 ARRAY VARIABLES

You can use both numeric and string variables in arrays. An array is a powerful way of storing and manipulating variable data. An array consists of a set of ordered variables, all under one variable name. Each array variable consists of a standard variable name: NUMBER plus a subscript: NUMBER(1). The subscript tells you which variable within the array is being worked on. For instance, when you enter:

```
NUMBER(3) = 5
```

AlphaBASIC PLUS assigns the value of 5 to the 3rd element in array NUMBER.



Do not use 0 as the subscript number in an array—it will cause an error.

The subscript number can also be a variable, allowing you a lot of flexibility in handling arrays. Here is a small program that fills an array with values:

```
For COUNT = 1 to 10
```

```

NUMBER(COUNT) = COUNT * COUNT
Next COUNT

```

Using the loop control variable `COUNT` (we'll explain FOR-NEXT loops later on) as the subscript variable allows us to fill the array with values. The first time through the loop, `COUNT` has the value of 1, so `NUMBER(1) = 1`. The second time, `COUNT = 2`, so `NUMBER(2) = 4` (2 times 2). And so on.

When you use an array variable, a certain amount of memory space is allocated for that variable and its subscripts. The default value is 10; that is, you can have up to 10 subscripts. If you want to have more than 10, or you don't need that many, you can change the size by MAPing the required array variable (see Chapter 14), or by using a DIM (dimension) statement. For example, to set the array `NUMBER` to hold 20 values:

```
MAP1 NUMBER(20),F,6
```

or:

```
DIM NUMBER(20)
```

One of these statements must precede the use of the array variable in the program. You can set each array variable to whatever size you need. It is a good idea to set up the array without much "overhead." Dimensioning an array for more subscripts than you need can use up valuable memory space. Once an array has been dimensioned by a MAP or a DIM statement, it may not be redimensioned in the same program.



You may not use DIM statements within user defined functions.

You can also make multi-dimension arrays. For instance, a double-subscripted array is referred to like this:

```
NUMBER(1,3)      or:      NUMBER(X,Y)
```

Arrays may be any number of levels deep (within memory limits) but practicality dictates a reasonable limit. The amount of memory you have dictates how much data you can store in an array.

At no time may the number of subscripts vary in any of the references to an array. For example, you cannot define an array as:

```
A$(10,20)
```

and then refer to it in this manner:

```
READ A$(1,2,5)
```

The number of subscripts in each element reference must also match the number of subscripts in the corresponding MAP or DIM statement that defined the array size. See Chapter 9 for more information on the DIM statement and Chapter 14 for more on MAP.

CHAPTER 7

EXPRESSIONS AND EVALUATION

This chapter discusses:

- What is an expression?
- How to use mathematical expressions
- The precedence of mathematical operators
- Conversions and mixed type expressions
- Type Independence

7.1 WHAT IS AN EXPRESSION?

An expression can contain variables, constant values, operator symbols, functions, or any combination of these. By expression, we mean a section of AlphaBASIC PLUS code treated as a unit within an AlphaBASIC PLUS statement. For example, in the statement:

```
IF (Inventory - Supplies'Used) < 100 THEN GOTO ORDER'NEW
```

(Inventory - Supplies'Used) < 100 is an expression. If the expression evaluates to true, then the instructions in the THEN clause are carried out. Even though the expression is made up of two variables, a number, and two operators, it can be seen as one logical unit in the way it functions within the IF - THEN statement.

7.2 MATHEMATICAL EXPRESSIONS

A mathematical expression is any expression containing numbers and mathematical operators. For example:

```
1 + (FIX(TOTAL'RECS * REC'SIZE)/512)
```


Parentheses can be used in most cases to tell AlphaBASIC PLUS which operation to perform first. If no parentheses are included, AlphaBASIC PLUS follows its rules of operator precedence (explained below) to determine which part of the expression is done first. AlphaBASIC PLUS recognizes the following mathematical operators:

| | | | |
|-------|-----------------------------|-----|-----------------------|
| + | addition, or unary plus | = | equal |
| - | subtraction, or unary minus | < | less than |
| * | multiplication | > | greater than |
| / | division | <> | unequal |
| ^ | raise to power | >< | unequal |
| ** | raise to power | # | unequal |
| " | string literal | <= | less than or equal |
| NOT | logical NOT | =< | less than or equal |
| AND | logical AND | >= | greater than or equal |
| OR | logical OR | => | greater than or equal |
| DIV | integer division | MOD | integer remainder |
| FMOD | floating point remainder | EQV | logical equivalence |
| USING | expression formatting | XOR | logical XOR |
| MIN | minimum value | MAX | maximum value |

The difference between / (division) and DIV (integer division), and between FMOD and MOD, are described in the section on Rules for Calculations.

AlphaBASIC PLUS cannot divide a number by zero. If a division by zero occurs, an error will occur. AlphaBASIC PLUS's handling of such a case can be controlled by the DIVIDE'BY'ZERO statement described in Chapter 9.

AlphaBASIC PLUS automatically evaluates expressions for you. For example, consider the statement:

```
PRINT (32 * 100/2 MAX 25 + 30/54)
```

First, AlphaBASIC PLUS computes the multiplication operation, determining 32*100 equals 3200. Then it computes the division operations, determining 3200/2 equals 1600, and 30/54 equals .5556. Next it adds 25 and .5556 to get 25.5556. Finally, it applies MAX to the parts of the expression (i.e., 1600 MAX 25.5556), and returns the greater or maximum value of the two sub-expressions, 1600. The order of evaluation of the terms in the expression, and the rules of mixing variable types, are described below.

7.3^oLOGICAL OPERATORS

The logical operators, AND, OR, XOR, EQV and NOT, are useful in conditional statements such as IF-THEN statements to evaluate and compare expressions. The AND statement compares two expressions. If both expressions are TRUE, the IF condition is also TRUE. If either or both statements are FALSE, the IF condition is FALSE. For example:

```
QUESTION:
  INPUT "Enter a number between 0 and 100: ", NUMBER
  IF NUMBER > 0 AND NUMBER < 100 THEN
    GOTO START
  ELSE
    PRINT "Number incorrect!"
    GOTO QUESTION
  ENDIF
START:
```

If either of the expressions, `NUMBER > 0` or `NUMBER < 100` are FALSE, the program above goes back and asks the question again. We can demonstrate OR with a similar section of code:

```
QUESTION:
  INPUT "Enter a number between 0 and 100: ", NUMBER
  IF NUMBER < 1 OR NUMBER > 99 THEN
    PRINT "Number incorrect!"
    GOTO QUESTION
  ENDIF
```

With OR, the IF statement is TRUE if any of the expressions are TRUE.

With XOR, which means Exclusive-OR, the IF statement is TRUE only if one of the expressions is TRUE. If more than one is TRUE, or all are FALSE, the IF statement is FALSE.

EQV, which means equivalence, causes the IF statement to be TRUE if both expressions evaluate to the same condition (TRUE or FALSE).

NOT negates the value of an expression. For example, these statements are equivalent:

```
IF (I = 1) THEN GOTO WRONG'ANSWER

IF NOT(I <> 1) THEN GOTO WRONG'ANSWER
```

7.4[∞]OPERATOR PRECEDENCE

The precedence of operators determines the sequence in which mathematical operations are performed when evaluating an expression not having overriding parentheses to define hierarchies. We showed you an example of this above in "Mathematical Expressions." Here is the order of precedence AlphaBASIC PLUS uses:

Exponentiation
 Unary plus and minus
 Multiplication and Division, DIV, MOD, and FMOD
 Addition and Subtraction
 Relational operations (comparisons)
 Logical NOT
 Logical AND, OR, XOR, EQV, MIN, MAX
 USING



The USING operator allows you to format numeric or string data using a format string. For information on USING, see Chapter 13.

In cases of equal precedence, such as between multiplication and division, operations are done according to rules of associativity. This means operations are done left to right (as in the example in section 7.2 above, where the multiplication on the left is done before the division). The only exception to this rule is exponentiation, which is done right to left. For example:

$$5^{2^4}$$

In this case, AlphaBASIC PLUS raises 2 to the fourth power first, and then raises 5 to the sixteenth power.

7.5°°RULES OF CALCULATION

After the precedence rules have been invoked to sort out the order of evaluation of the expression, AlphaBASIC PLUS follows these rules for calculations:

- 1.°°Binary variables are converted to real numbers before use and back after use.
- 2.°°Single precision IEEE variables are converted to double precision variable before use and back after use.
- 3.°°For a given operation, the operation is done in the larger of the two formats of the operands, and the result is left in that format unless the operation forces conversion to a specified format. mathematical functions will automatically (at compile time) select an AMOS or an IEEE floating point version to maintain precision.

Each mathematical function (described in Chapter 10) exist in both the AMOS floating point as well as IEEE double precision forms. The compiler will select the appropriate choice so as to achieve the required degree of precision.

- 4.°°Numeric literals in the source code are treated as integers unless they have a trailing decimal point or an exponent marker.
- 5.°°All logical operations (such as AND, OR, NOT, etc.) cause values to be converted to 5-byte binary variables before the logical operation is performed. The value -1 represents a 40-bit mask of all ones. Any relational comparison between two

expressions or variables returns -1 if true, or 0 if false.

6. If two strings are of equal length, they are compared on a character-by-character basis. If they are of different length, the shorter is padded with spaces (on the right) to the length of the longer, and then the comparison proceeds. Thus the string "PAST DUE" is equal to the string "PAST DUE ".
7. When conversions are done, for example from real to integer, any data outside the limits of the destination variable is merely lost without an error being raised. Range checks are the responsibility of the programmer. Converting from real numbers (of any format) to integers is done by truncating the value towards minus infinity. Therefore 12.9 will be converted to 12, -12.9 converted to -13, and -12.1 converted to -13. This is the behavior of the INT() function as well. The FIX() and RNDN() functions can be used to produce a slightly different results:

| N | -1.9 | -1.1 | 1.1 | 1.9 |
|---------|------|------|-----|-----|
| INT(N) | -2 | -2 | 1 | 1 |
| FIX(N) | -1 | -1 | 1 | 1 |
| RNDN(N) | -2 | -1 | 1 | 2 |

8. When converting from negative values to binary variables, AlphaBASIC PLUS stores the signed integer result in the binary variable by doing a simple byte move. As the sign bit of an integer value is the most significant bit in the value, this has the result of making the value of the binary number a very large positive value, as that same bit is treated as a positional bit in an unsigned value. For example:

```

SIGNIFICANCE 11
MAP 1 F, f, 6, -1.9
MAP 1 B, b, 4
B = F
? B
4294967294

```

The floating point value -1.9 is converted to an integer value (-2). The two's complement bit pattern representing -2 (11111.....110) is placed in B. The unsigned value of that bit pattern is B's value.

9. If strings and numeric variables and/or constants are used in the same expression, AlphaBASIC PLUS will convert one type to another automatically according to the rules for mixed type expressions described in the next section.

7.6[∞]MIXED TYPE RULES

Expressions may contain any mixture of variable types and constants in any arrangement. AlphaBASIC PLUS performs string and numeric conversions as necessary to make sure the result is in the proper format. For example, if two strings are multiplied together they are first converted to numeric format before the multiplication takes place. If the result is then to become a string, it is reconverted back to string format before the assignment is performed. In other words, the statement:

```
A$ = B$ * "345"
```

is perfectly legal and works correctly, as long as B\$ contains a number (and no alphabetic characters). This is a powerful feature which can save programming effort when used correctly.

There is a seemingly ambiguous situation which arises from this mode independence. The plus symbol (+) is used both as an addition operator for numeric operations and as a concatenation operator for string operations. The value of 34 + 5 is equal to 39 but the value of "34" + "5" is equal to the string "345." The operation of the plus symbol is unambiguous, but may take a little thought to figure out in a given situation. A few examples might help:

If the first operand is numeric and the second is string, the second is converted to numeric form and addition is done:

```
34 + "5" = 39
```

If the first operand is string and the second operand is numeric, the second is converted to string and concatenation is done:

```
"34" + 5 = "345"
```

The above two examples apply only when the compiler is not "expecting" a particular type of variable or term. This generally occurs only in a PRINT or IF expression, such as PRINT "34" + 5. At other times, the compiler expects a specific type of variable; the conversion of the first variable is then performed prior to inspecting the operator (plus sign). What the plus sign does then depends upon that first variable—if the variable is a numeric one, the plus sign does addition; if string, it concatenates (this is the one case where parentheses will not change the operation—the plus sign obeys the type of variable it is being assigned to regardless of any parentheses). In the following example:

```
NUM = 5 * "34" + 4
```

The multiplication operator (*) forces us to expect a numeric term to follow. The "34" string is therefore immediately converted to numeric 34 and multiplied by the 5. The plus sign then performs numeric addition instead of concatenation. The result is in numeric format. Here are a few examples:

| EXAMPLE | RESULT | EXAMPLE | RESULT |
|----------------|--------|------------------|--------|
| A = 34 + 5 | 39 | A\$ = 34 + 5 | 345 |
| B = 34 + "5" | 39 | B\$ = 34 + "5" | 345 |
| C = "34" + 5 | 39 | C\$ = "34" + 5 | 345 |
| D = "34" + "5" | 39 | D\$ = "34" + "5" | 345 |

You can see conversion is affected by the type of variable being used. You might like to try a few examples of your own on your system to see what the results are. Any potentially ambiguous expression may always be forced to one type or the other by use of the STR(X) and VAL(A\$) functions (see Chapter 10). For more examples of mode independence, see the sample programs in Chapter 8.

7.7[∞]DIVISION AND MODULUS OPERATIONS

AlphaBASIC PLUS differentiates between integer and real operations and results for two operations: the division and modulus (remainder) operations. Division can be done in two ways: either as converting each operand to integers and producing an integer result, or converting each operand to floating point and producing a floating point result. The former is specified by the DIV operator, and the latter by the / operator. The two modulus operators operate in a similar way: the integer version is MOD and the floating point version is FMOD. Correct use of these operators is essential for accurate calculations. For example:

```
MAP 1  I, I, 4
I = 14/5 + 14/5
? I
5
I = 14 DIV 5 + 14 DIV 5
? I
4
I = 14 DIV 5 + 14/5
? I
4
```

In the first case, the / operator is evaluated first: 14/5 uses the floating point operator, requiring conversion to floating point for 14 and 5, and then the division yields a floating point result (2.8). The two results are then added together as floating point number, yielding 5.6. The result of the expression is thus a floating point number of value 5.6. To store this in a four-byte integer, the value is truncated towards minus infinity, yielding the answer of 5 being stored in I.

In the second case, the DIV operator is evaluated first: 14 DIV 5 uses the integer operator, resulting in an integer result of 2. Integer 2 is then added to integer 2, yielding an integer result of 4, which is directly stored in I.

In the third case, the DIV operator is evaluated first, giving an integer result of 2. Then the / operator is evaluated, giving a floating point result of 2.8. To do the addition, the integer 2 is converted to the real number 2.0 (using the "convert to the wider operand" rule), and a floating point addition to 2.8 is done, yielding a floating point result of 4.8. This is converted (by truncation towards minus infinity) to integer 4 and is stored as the value in I.

7.8[∞]THE SIGNIFICANCE OF SIGNIFICANCE

The SIGNIFICANCE program statement is discussed in Chapter 9. In brief, it has nothing whatsoever to do with the precision or accuracy of mathematical calculations or data format conversions. Instead, it controls the maximum number of digits to be output when printing numeric values. The value to be printed is copied, and the copy is rounded to the specific number of digits prior to printing. The default value is six digits.

CHAPTER 8

WORKING WITH STRING VARIABLES

This chapter discusses:

- What a substring modifier is
- How to use a substring modifier
- Setting the size of the string work area

8.1 WHAT IS A SUBSTRING MODIFIER?

Chapter 6 defined string variables. These strings of characters can also be broken down into substrings. A substring is a portion of an existing string, and may be as small as a single character or as large as the entire string. AlphaBASIC PLUS supports a unique method of manipulating substrings using substring modifiers.

Substring modifiers allow the substring to be defined in terms of character positions within the string, relative to either the left or right end of the string. The length of the substring is defined either in terms of its beginning and ending positions or in terms of its beginning position and its length. You define a substring by using the string variable (for example, A\$) followed by the substring modifier. The substring modifier is two numeric arguments enclosed within square brackets (for example, A\$[1,3]).

8.2 HOW DO YOU USE A SUBSTRING MODIFIER?

The substring modifier has two distinct formats:

```
[ beginning-position , ending-position ]
```

```
[ beginning-position ; substring-length ]
```

The first format defines the substring in terms of its beginning and ending positions within the string and uses a comma to separate the two arguments. The second format defines the substring in terms of its beginning position within the string and its length, using a semicolon to separate the arguments.

The beginning and ending positions are defined as character positions within the string relative to either the left or right end. A positive value represents the character position relative to the left end of the string, with character position 1 representing the first (leftmost) position.

A negative value represents the character position relative to the right end of the string, with character position -1 representing the last (rightmost) position. For example, assume a string consists of the letters ABCDEF. The positions are defined in terms of positions 1 through 6 (left-relative) or positions -1 through -6 (right-relative).

| | | | | | | |
|----|----|----|----|----|----|---------------------------------|
| A | B | C | D | E | F | 6 characters within main string |
| 1 | 2 | 3 | 4 | 5 | 6 | left-relative position values |
| -6 | -5 | -4 | -3 | -2 | -1 | right-relative position values |

Allowing negative values for right-relative positions lets you find characters in a string without having to calculate the total size of the string and work from the left.

The substring-length argument used by the second format may also take on negative values for a more flexible format. When the length is a positive value, it represents the number of characters from the beginning position and counting to the right. A negative length causes the index to move to the left and returns a substring whose last character is the one marked by the beginning-position argument.

A few examples should show the use of substring modifiers. If A\$ = "ABCDEF", then:

| | | | | | |
|------------|---|-------|------------|---|------|
| A\$[2,4] | = | "BCD" | A\$[-3,-2] | = | "DE" |
| A\$[3,-2] | = | "CDE" | A\$[3;-2] | = | "BC" |
| A\$[-3;-2] | = | "CD" | A\$[4;1] | = | "D" |

For example, A\$[3,-2] tells AlphaBASIC PLUS to return the substring beginning at character position 3 (from the left) and ends with character position 2 (from the right); that is, to return all characters between C and E, inclusive.

A\$[3;-2] tells AlphaBASIC PLUS to return the substring beginning with character position 3 (from the left) and extends 2 character positions toward the left; that is, return all characters starting with C and work backward two positions to B, inclusive.

Any position values or length values which cause the substring to overflow either end of the main string are truncated at the string end. For example:

| | | | | | |
|-----------|---|--------|-------------|---|----------|
| A\$[3,10] | = | "CDEF" | A\$[-14,34] | = | "ABCDEF" |
|-----------|---|--------|-------------|---|----------|

The main string to which the substring modifier is applied is actually any expression and does not need to be a defined single string variable. For example:

```

STRSIZ 50 RETURN
A$ = "Form X-913 " RETURN
B$ = "Prototype Ledger Screen " RETURN
C$ = "Schedule A-40 Deductions" RETURN
ANSWER$ = (A$ + B$ + C$)[6;30] RETURN
PRINT ANSWER$ RETURN
X-913 Prototype Ledger Screen
ANSWER$ = ("ABLE" + A$ + "QQ34")[4,17] RETURN
PRINT ANSWER$ RETURN
EForm X-913 QQ

```

The mode independence feature allows substring modifiers to be applied to numeric expressions. See Chapter 7 for information on mode independence. When a string is returned and the destination is a numeric variable, another conversion is made on the substring to return a numeric value. For example:

```

INPUT "Enter two numbers: ",NUMBER,NUMBER2
SUM = NUMBER + NUMBER2
? NUMBER;" + ";NUMBER2;" = ";SUM      ! Test rightmost digit:
IF SUM[-1;1] = 0 THEN PRINT "Divisible by 5 and 2"
IF SUM[-1;1] = 5 THEN PRINT "Divisible by 5"

```

Be sure you understand the concept of mode independence before you use substring modifiers or you may get answers you don't expect. For example, the second and third lines in the small program below return different answers, even though the subscripting is performed exactly the same in both cases. This is because the mode independence feature examines the data type of the destination variable before doing the operations. When it scans the second line, AlphaBASIC PLUS knows a string result is expected (because STRING\$ is a string variable), and so reads the "+" symbol as a string concatenation operator. In the third line, BASIC knows a numeric result is expected (because NUMERIC is a numeric variable), and so reads the "+" symbol as an addition operator.

```

VALUE1$ = "123" : VALUE2$ = "456"
STRING$ = (VALUE1$ + VALUE2$)[1;3]
NUMERIC = (VALUE1$ + VALUE2$)[1;3]
PRINT "NUMERIC = ";NUMERIC,"STRING$ = ";STRING$

```

The program above prints:

```

NUMERIC = 579          STRING$ = 123

```

You may apply substring modifiers to subscripted variables or expressions containing subscripted variables. Be careful not to confuse substring modifiers with subscripted variables. For example:

| | |
|----------------------------|---|
| <code>A\$(2,3)</code> | designates a location in array A\$ |
| <code>A\$[2,3]</code> | designates a substring of string A\$ |
| <code>A\$(1,5)[2,3]</code> | designates a substring in location A\$(1,5) |

These are valid uses of the substring modifiers:

```
Q$ = A$(1,9)[2,5]           Q$ = (A$(1) + B$(3))[-5,3]
```

Substring modifiers return a string value. These may be used as part of string expressions. For example:

```
Q$ = A$ + B$[2;5] + (A$[2,2] + C$)[-5;-3]
```

You may apply substring modifiers to the left side of an assignment in order to alter a substring within a string variable. Only that portion of the string defined by the substring modifier is changed. The other characters in the string are not altered. For example:

```
A$ = "ABCDEF" RETURN
A$[2,4] = "###" RETURN
PRINT A$ RETURN
A###EF
```

This procedure may not be applied to numeric variables (for example, `A[3;2] = "23"` is not valid when A is a numeric variable).



When substring modifiers are used on the left side of an assignment, the values represent the character position within the variable. For all other uses, the values represent the character position within the string which is contained within the variable.

8.3[∞]SETTING THE SIZE OF THE STRING WORK AREA

When RUNP executes a compiled program, it may have to manipulate string expressions which the programmer coded. RUNP allocates an area of memory, the string work area (SWA), for this manipulation. The size of this area is decided by COMPLP, the compiler. In most cases, the compiler can accurately estimate the required size of the SWA, but in some cases the programmer may need to increase the size. It is also possible to decrease the size of the SWA. A manual intervention can be required if the `SPACE()` or `FILL()` functions are used where the size of the resultant string is specified as a run-time expression (e.g. `SPACE(len(A$) * 2)`), when certain subprograms are SUBCALLED, or certain user-defined functions are executed.

The size of the SWA can be altered from the compiler-set value by using `++PRAGMA` compilation statements. See the section of `++PRAGMA` in Chapter 5 for details. The size of the SWA in a .RP file can be determined by using the `/R` option of RUNP. See section 5.5.1 for details.

A fuller explanation of the SWA and its size can be found in Appendix I.

CHAPTER 9

PROGRAM STATEMENTS

An AlphaBASIC PLUS source program contains statements which are executed in sequence, one at a time, as AlphaBASIC PLUS encounters them. Each of these statements normally starts with a statement followed by optional variables or statement modifiers. Many of these statements can also be used in interactive mode as direct statements.

This chapter lists most of the program statements and gives some examples for clarity. The program statements not discussed in this chapter are the statements dealing with file input/output and ISAM files. See Chapters 15 and 19, respectively, for those statements. See also the *AlphaBASIC PLUS Quick Reference Card* for the formats of all AlphaBASIC PLUS statements and commands.

9.1[∞]AMOS

The format is:

```
AMOS {/switch} command-string
```

where `command-string` is a string expression that is a legal AMOS-level command. AMOS executes the specified command. There must be enough memory left in the user memory partition for the command to execute (see MINF in Appendix H).



It is not a good idea to use the compile or run programs for AlphaBASIC or AlphaBASIC PLUS in this command, since there may not be enough memory to execute them.

The `switches` are `/S` and `/T`. If `/S` is specified, no output from the AMOS command is displayed on the terminal. This works like the `:S` mode in command files. If `/T` is specified, the output of the command is displayed on the terminal, as if `:T` was used in a command file. If no `switch` is used, the result is in whatever mode the system is in.

If the AlphaBASIC PLUS program was called from a command file, the output display will be what is set by the command file. The `ERR(3)` function (see Chapter 17) reports the error status code—this can help you determine whether the AMOS command executed properly. Here are some examples of the AMOS statement:

```

AMOS "TIME"
AMOS/T "DIR/W DSK0:[1,4]*.LIT"
AMOS/S User'request

```



Be sure ECHO is set (see ECHO, below) before you use AMOS. In some cases your job can hang if NOECHO is set and an AMOS command is executed.



When it starts up, AlphaBASIC PLUS remembers and sets up some internal variables based on the system environment. It does not re-scan its environment after an AMOS statement. Thus, you can get unexpected or undesired results if you change the system environment (e.g. your terminal driver) out from under AlphaBASIC PLUS by using certain command strings without restoring the original environment before continuing.

9.2[∞]CALL

See GOSUB, below.

9.3[∞]CASE

See SWITCH/CASE, below.

9.4[∞]CHAIN

The format is:

```
CHAIN filespec
```

Where *filespec* is a string literal or a string variable. *filespec* may have the forms:

```

{devn:}RP-filespec{[p,pn]} {arguments}
{devn:}monitor-command{[p,pn]} {arguments}
{devn:}CMD-file-name{[p,pn]}
{devn:}DO-file-name{[p,pn]}{arguments}

```

CHAIN causes control to be passed to the specified AlphaBASIC PLUS program, command file, or monitor command program. The program name may be a full file specification, including device and account specifications. Certain AMOS commands and DO files can have various *arguments* specified with them—the way you enter the command at AMOS level is the way you enter it after the CHAIN statement. The RUNP command also allows *arguments*—see the CMDLIN statement in Chapter 10.

CHAIN causes the current program to be cleared from memory. The specified file is then located and executed from the beginning.

A chained AlphaBASIC PLUS program must be a fully compiled program with the extension .RP in order to be run by the CHAIN command. It may be in user memory

(having previously been loaded with the monitor LOAD command) or it may be in system memory (the System Operator may place a file in system memory by modifying the system initialization command file).

If it is not already in memory, it is loaded from the specified disk account into user memory and then executed. If it can't be found, you are returned to AMOS level. Here are some examples of CHAIN:

```
CHAIN "PAYROL"  
70 CHAIN "DSK1:ACTRUN.CMD[100,7]"
```

There is no way to start the chained file at any point other than the beginning. You may pass common variables between chained AlphaBASIC PLUS programs either by writing them out to a file and then having the chained program read them back in, or by using the COMMON assembly language subroutine. For information on COMMON, see your *AlphaBASIC XCALL Subroutine User's Manual*. Literal strings can be passed by CMDLIN. For more information on CHAIN, see Chapter 16.

You may not CHAIN from a subprogram. You must exit all subcalls and return to the main program first. Similarly, you may not chain out of a user defined function.

9.5 DATA

See READ, RESTORE, and DATA below.

9.6 DEFAULT

See SWITCH/CASE below.

9.7 DEFINE

Defines a numeric or string constant. A constant is a value that does not change during the program. The format is:

```
DEFINE constant-name = value
```

constant-name is a label for the constant to be defined, and value is an expression composed of one or more constant values. You cannot use forward references to other defined constants. A constant definition makes it easy to change values throughout a program. A constant-name can only be DEFINEd once during a program, and may not be changed. Some examples:

```
DEFINE Control'number = 5  
DEFINE Divisor = 3.5943  
DEFINE FACTOR = Control'number + (15/Divisor)  
DEFINE Product'Name = "ELAXOR"
```

Don't specify a % or \$ to indicate an integer or a string—the constant takes the type of the value given. If the constant is used in an equation or comparison with another variable type, AlphaBASIC PLUS does the necessary conversion (if possible).

Constants can also be defined from COMPLP's command line. See the section on compiler options in Chapter 5 for details.

9.8[∞]DIM

Defines an array which is allocated dynamically at execution time. The format is:

```
DIM variable1(expr1...,exprN){,variableN(expr1...,exprN)}
```

Once allocated, an array can't be redimensioned during the execution of the program. If you use DIM statements with mapped variable arrays, you get a run-time error.

There is no set limit to the number of subscripts that may be used to define the individual levels within the array—the size of your program and the amount of memory you have determine how many levels you can have. In most cases, you are able to define many more levels than you actually need.

The statement DIM A(20) defines an array with 20 elements, referenced as A(1) through A(20). Multiple arrays may be dimensioned by a single DIM statement by separating them with commas. For example:

```
DIM A(20),B(15),C(10)
```

Subscripts are evaluated at execution time and not at compile time, thereby allowing variables as well as numeric constants to be used as subscripts. The statement DIM A(B,C) allocates an array whose size depends on the actual values of B and C at the time the DIM statement is executed. The number of elements in the resulting array equals B times C.

If a reference to an array is made during program execution without a previous DIM statement to define the array, AlphaBASIC PLUS assigns a default array size of 10 elements for each subscript level.

String arrays may be allocated, such as DIM A\$(5). The size of the array depends on the current string size in effect as specified by the last STRSIZ command (or the default of 10) seen by the compiler, since each element in the array must be this number of bytes, or less. For instance, if the current STRSIZ is 10, the statement DIM A\$(5) allocates 5 elements multiplied by 10 bytes per element, or 50 bytes of memory for the array. Here are some examples of valid DIM statements:

```
DIM A(10)
DIM A(B(4))
```

```
DIM TEST(A,B*4)
DIM C(8,8), C$(10,4)
```

You may not use the DIM statement within a user-defined function. For more information on arrays, see Chapter 6.

9.9°DIVIDE'BY'0

Turns off the "Divide by zero" error. When this statement is executed, all attempts to divide by zero result in an answer of 0 instead of an error. NO'DIVIDE'BY'0 (see below) turns the error condition back on.

9.10°DO WHILE/UNTIL LOOP

Executes a series of statements while a condition is true, or until a condition is met. The format is:

```
DO {WHILE and/or UNTIL condition}
    .
    statement(s)
    .
LOOP {WHILE and/or UNTIL condition}
```

The DO reserved word marks the beginning of the loop. If WHILE is used with a condition, the statement(s) that make up the loop are executed if the condition is TRUE. If UNTIL is used, the statement(s) are executed if the condition is FALSE. If condition is not met, control passes to the first statement after LOOP. If condition is met, the statement(s) are executed. Then LOOP is executed, and its WHILE/UNTIL condition is evaluated (if it contains one). If that condition is met, control passes back to DO. The DO—LOOP cycle continues until a WHILE or UNTIL passes control out of the DO—LOOP, or until an EXIT occurs. For example:

```
DO WHILE Control'Number > Count
    PRINT "Item Number: ";Count
    Count = Count + 1
    READ #1,Next'Inventory'Item
    PRINT Next'Inventory'Item
LOOP
```

In the above program section, the DO—LOOP is executed as long as Control'Number is greater than Count.

The advantage of the DO—LOOP structure is its flexibility—the "test" can be at either end (or both) of the loop, or in the middle of the statements. You may use an EXIT statement to make tests in the middle of the loop (see below). You can also easily have multiple "tests." Another example:


```
NUM'TEST = 0
DO WHILE NUM'TEST <= MAX'TEST
    NUM'TEST = NUM'TEST + 1
    IF X > 0 THEN EXIT
    INPUT #1 A'TEST(NUM'TEST)
LOOP UNTIL EOF(1)
NUM'TEST = NUM'TEST - 1
```

You may also use REPEAT statements (see below) to transfer control to the DO test.

9.11°ECHO

Echo means you see the characters you type on the keyboard on your terminal screen. This is the normal condition. You only need to use ECHO if you have turned off the echo by using NOECHO (see below). The format is:

```
ECHO
```

9.12°ELSE

See IF, THEN, and ELSE below.

9.13°END

The format is:

```
END
```

END causes the program to terminate execution. END doesn't terminate compilation of the program nor is it required at the end of the program. If other program statements follow the end of the program (e.g., subroutines), terminating the program with END prevents your program from incorrectly entering those statements and executing them. A program may have more than one END statement.

9.14°ENDSWITCH

See below SWITCH and CASE

9.15°EXIT

Used inside a loop structure (FOR/NEXT, DO/LOOP, SWITCH, etc.) or a function to exit from the loop or function regardless of what the condition evaluates to. Control passes to the statement after the loop terminator. For example:

```
FOR I = 3 TO 100
  Last = INT(SQR(I))
  FOR Factor = 2 to Last
    IF 0 =(I - Factor * INT(I/Factor)) THEN EXIT
    IF A < 0 OR A > 100000 THEN EXIT
  NEXT Factor
  IF Factor = Last + 1 THEN PRINT I
NEXT I
```

In the above case, EXIT transfers control to IF Factor = Last + 1 THEN PRINT I if the IF A < 0... statement tests TRUE. Using EXIT lets you test conditions other than those tested in the loop-control variables themselves. It also lets you have a test in the middle of the loop instead of at the beginning or end.

9.16°FOR, TO, NEXT AND STEP

The format is:

```
FOR control-variable = expr TO expr {STEP {-}expr}
.
{statements}
.
NEXT {control-variable}
```

The FOR and NEXT statements initialize and control program loops. A loop is a structure in which the same statement or statements can be performed a specific number of times. Whether or not a loop is executed depends upon the value of the control-variable. The control-variable may even be subscripted. It can be either a six-byte or an eight-byte floating point variable, a 4-byte integer, or a single alphabetic string character, such as 'a.'

The delimiters expr may be any valid expression. FOR initializes the control-variable to the first expression.

NEXT increments or decrements the value of the control-variable each subsequent loop. There must be **one and only one** NEXT statement for every FOR statement, and the NEXT statement cannot be part of an IF-THEN-ELSE or SWITCH expression, or included in a GOSUB or subprogram that doesn't have the FOR statement it applies to.

The control-variable may be omitted in the NEXT statement, in which case the control-variable of the previous FOR statement is the one incremented. control-variable is incremented or decremented in units indicated by the STEP

statement (if STEP is used). If no STEP modifier is used, the step value is assumed to be positive 1.



AlphaBASIC PLUS does an initial test of the control-variable at the FOR statement. This means that if the condition is true, the statements within the loop are not executed.

FOR and NEXT statements are illegal as direct statements in Interactive Mode except when put on the same multi-statement line. For example, this line is valid:

```
FOR I = 1 TO 10 : PRINT I : NEXT I
```

Here are some samples of the different forms FOR-NEXT loops may take:

```
FOR COUNTER = 1 TO 10
  PRINT "The COUNTER ";
  IF COUNTER/2 = INT(COUNTER/2) THEN
    PRINT COUNTER;" is even."
  ELSE
    PRINT COUNTER;" is odd."
  ENDIF
NEXT COUNTER
```

```
10 ? "Enter the date of the first Sunday in the month: ";
20 INPUT LINE DAY
30 PRINT "The Sundays this month are on these dates: ";DAY;
40 FOR A = DAY + 7 TO 31 STEP 7
50 PRINT A; : NEXT A
```

```
FOR I = 10 TO 1 STEP -1
  PRINT I
NEXT
```

Loops within loops are legal and are called "nested" loops. Loops may be nested to many levels. Each time the outermost loop is incremented (or decremented) once, the loop nested within it is executed from beginning to end. During the execution of the second loop, the third loop (if any) is fully executed each time the second variable is incremented. And so on, for each nested loop in the series. For example:

```
DIM MATRIX(5,5)
FOR I = 1 TO 5
  FOR J = 1 TO 5
    MATRIX(I,J) = I - J      ! Nested loops
    PRINT MATRIX(I,J);
  NEXT J
  PRINT
NEXT I
```

It is not recommended to branch out of a loop before its completion (for example, by using a GOTO statement). If a loop is not properly ended, AlphaBASIC PLUS continues keeping track of the loop control variables in memory—these values could cause errors in future calculations in the program. Instead, exit the loop by using EXIT, or by setting control-variable to the terminal value given in the FOR statement. For example:

```
START'LOOP:

FOR I = 1 TO 10
  INPUT "Enter number of pennies: ",Pennies
  IF Pennies < 0 &
    PRINT "You can't have negative pennies!" : EXIT
  PRINT "You have ";Pennies/100;" dollars."
NEXT I
```

You may use the REPEAT statement to cause the next iteration of the loop without going all the way to the NEXT statement.

9.17[∞]GOSUB (OR CALL) AND RETURN

The formats are:

```
GOSUB label or line number
CALL label or line number
RETURN
```

These commands call a subroutine which starts at the line number or label specified in the GOSUB or CALL statement. When RETURN is encountered, the subroutine returns to the statement immediately after the GOSUB or CALL. Executing RETURN without first executing GOSUB results in an error message.

Both GOSUB and RETURN are illegal as direct statements in Interactive Mode. Note the CALL statement is merely another way of specifying GOSUB for those programmers used to this syntax from other versions of BASIC or other programming languages.

It is often the case you want to perform the same operation or series of operations at various points within your program. A subroutine is a set of program statements you may execute more than once simply by including a call for that subroutine within your program at the point where you want to execute the routine. For example:

```
! This program has a subroutine that validates numeric
! entries to make sure they are > 0 and < 100.
```

```
PRINT "We're going to do some mathematical operations."
RE'START:
PRINT "Your entries must be > 0 and < 100."
PRINT : INPUT "Enter two numbers to be added: ",A,B
GOSUB VALIDATE      ! Check if numbers are valid.
IF FLAG = 1 THEN GOTO RE'START      ! Check error flag
PRINT A;" + ";B;" = ";A + B
PRINT : INPUT "Enter two numbers to be subtracted: ",A,B
GOSUB VALIDATE      ! Check if numbers are valid.
IF FLAG = 1 THEN GOTO RE'START
PRINT A;" - ";B;" = ";A - B
PRINT : INPUT "Enter two numbers to be divided: ",A,B
GOSUB VALIDATE      ! Check if numbers are valid.
IF FLAG = 1 THEN GOTO RE'START
PRINT A;" / ";B;" = ";A/B
PRINT : PRINT "Th-th-th-that's all, folks!"
END
VALIDATE:      ! Subroutine to validate the data
IF A <= 0 OR B <= 0 OR A >= 100 OR B >= 100 THEN
    PRINT "Error - incorrect number entered in input!"
    FLAG = 1
ELSE
    FLAG = 0
ENDIF
RETURN
```

Note an END statement separates the main program from the subroutine; otherwise, AlphaBASIC PLUS executes VALIDATE again after printing Th-th-th-that's all, folks! and you see a RETURN without GOSUB message.

Also note the use of GOSUBs helps to modularize your programs, and thus makes them easier to design and maintain. Even before you completely "flesh out" your programs, you can insert dummy routines to later contain complete code. For example:

```
! This program is an example of a dental package.
PRINT "Welcome to the Dr. Plak Rental Dental Package."
PRINT
GOSUB INIT      ! Perform initialization of data files
GOSUB MENU      ! Ask user to pick function from main menu.
GOSUB DAY'END    ! Do End-of-day Processing
GOSUB FINISH'UP ! Finish up, close files, and exit.
END

! The subroutines start here.
INIT:
PRINT "This section will initialize files."
```

```

RETURN

MENU:
    PRINT "This section will display the main menu and "
    PRINT "ask user for selections."
RETURN

DAY'END:
    PRINT "This section will do day-end processing."
RETURN

FINISH'UP:
    PRINT "This section will close files and " &
        "clean up final data."
RETURN

```

You can nest subroutines. For example:

```

PRINT "Main Program:"
GOSUB OUTER'MOST           ! OUTERMOST calls
PRINT "  Return from Outermost" ! NEXTMOST and INNERMOST
END

OUTER'MOST:
    PRINT "  Outermost subroutine"
    GOSUB NEXT'MOST
    PRINT "    Return from Nextmost"
RETURN

NEXT'MOST:
    PRINT "    Nextmost subroutine"
    GOSUB INNER'MOST
    PRINT "      Return from Innermost"
RETURN

INNER'MOST:
    PRINT "      Innermost subroutine"
RETURN

```

The program above prints:

```

Main Program:
  Outermost subroutine
    Nextmost subroutine
      Innermost subroutine
      Return from Innermost
    Return from Nextmost
  Return from Outermost

```



It is good programming form to exit a subroutine by using the RETURN statement for that subroutine rather than using a GOTO statement.

9.18[∞]GOTO

The format is:

```
GOTO label or line number
or:
GO TO label or line number
```

The GOTO statement transfers execution of the program to a new program line. This program line must be identified either by a line number or a label somewhere in the program. You may use GOTOs to transfer control to a program line either before or after the program line containing the GOTO statement itself. For example:

```
! Program to demonstrate use of GOTOs.

PRINT "This program computes your account balance. Use"
PRINT "^C to stop; enter deposits as negative amounts."
INPUT "Enter old account balance: ",BALANCE

CALCULATE'BALANCE:
  PRINT : INPUT "Enter debit amount: ",DEBIT
  BALANCE = BALANCE - DEBIT
  PRINT "Your debit was :";DEBIT
  PRINT "Your current balance is :";BALANCE
GOTO CALCULATE'BALANCE
```

You can see there is an endless loop in which control is eternally transferred between GOTO CALCULATE'BALANCE and CALCULATE'BALANCE. The program above runs until you press /.

If you use GOTOs on a multi-statement line, remember to place it last on the line; any statements after the GOTO are never executed. For example:

```
NET = GROSS - DEDUCTION : GOTO GET'TAX : PRINT DEDUCTION
```

the last statement, PRINT DEDUCTION, can never be executed.



Do not use a GOTO from a location outside a loop or function to a location inside a loop or function (a FOR-NEXT, DO-LOOP, etc.). This causes a compiler error.

9.19[∞]IF, THEN AND ELSE

The IF statement may be either in multi-line form:

```
IF expression {THEN}
    statement(s) or label(s)/line#(s)
{ELSE
    statement(s) or label(s)/line#(s)}
ENDIF
```

or, in single-line form:

```
IF x {THEN} y {ELSE z}
```

If there is no statement or label/line# after the IF expression on the first line, it defines a multi-line IF statement.

The IF statement evaluates *expression*, and conditionally performs the specified operation(s) in response to that evaluation.

The conditional processing features in AlphaBASIC PLUS give a wide variety of formats. Some of the acceptable format combinations are:

```
IF expression THEN line#
IF expression THEN GOTO line#/label
IF expression GOTO line#/label
IF expression THEN line# ELSE GOTO line#/label
IF expression THEN GOTO line#/label &
    ELSE GOTO line#/label
IF expression THEN statement
IF expression statement
IF expression statement ELSE statement
IF expression THEN statement ELSE statement
```

Notice from the examples above you may sometimes omit the GOTO keyword when transferring control to another program location. You may NOT omit the GOTO keyword when you are referring to a label or when you are in an ELSE clause and referring to a line number.

The above formats may be nested to a depth that depends on the amount of memory you have available in your memory partition.

You may often omit THEN also. You can't omit THEN if you are transferring control to another area (a line number or label) and you don't have a GOTO (A transfer of control must have either a THEN or a GOTO). Some examples:

```
IF A = 5 THEN GOTO PROGRAM'EXIT
IF A > 14 THEN 110 ELSE GOTO 220
IF B$ = "END" PRINT "END OF TEST"
IF TOTAL > 14.5 GOTO START
IF P = 5 AND Q = 6 IF R = 7 PRINT 567 &
    ELSE PRINT 56 ELSE PRINT "NONE"
IF A = 1 PRINT 1 ELSE IF B = 2 &
    THEN 335 ELSE GOTO 345
IF A AND B THEN PRINT "A and B are nonzero."
```



```

IF Revenue > Expenses THEN
    PRINT "We have made a profit!"
    GOSUB Print'Stockholders'List
    GOSUB Print'Caterers'Phone'Numbers
ELSE
    PRINT "We have experienced a loss."
    GOSUB Print'Company'Budget
    GOSUB Print'Employee'List
    GOSUB Print'Problem'Employee'List
ENDIF

```

Note the expression evaluated by the IF statement usually contains relative operators (e.g., IF A = B; IF A > 0; etc.). However, it may be any legal expression. For example:

```

A = 0 : B = 1
IF B THEN PRINT "B is not zero."
IF (B AND A) PRINT "nonzero numbers" &
    ELSE PRINT "at least one zero number."

```

9.20 INPUT

The format is:

```
INPUT {"prompt-string",}var1{,var2...,varN}
```

Allows data to be entered from your terminal and loaded into specific variables at run-time. The INPUT statement contains one or more variables separated by commas. If you omit the `prompt-string`, AlphaBASIC PLUS displays a question mark on the terminal display to signal a request for data entry. For example:

```

INPUT A 
?  5 
PRINT A 
5

```

If you use a `prompt-string`, AlphaBASIC PLUS displays it instead of a question mark to prompt the user of your program for data. If you wish to have no prompt at all, use a null `prompt-string`, as in:

```
INPUT "",A$,B$
```

`prompt-string` must be in the form of a string literal (enclosed in quotation marks). For example:

```
INPUT "Enter your account number: ",ACCOUNT'NUM
```

prints:

```
Enter your account number:
```

You may specify integer, floating point, and string variables in an INPUT statement. INPUT accepts the E and D exponent markers for AMOS or IEEE floating point variables. String variables should be in ASCII format.

If you use INPUT or INPUT LINE to read data into a variable that contains a null character, that variable is terminated by that null character (even if other data follows it). The only case where this is not true is with unformatted variables.

Some examples of valid INPUT statements are:

```
INPUT A
INPUT "Enter account #, name, and age: ",ACCOUNT,NAME$,AGE
INPUT " ",A%,B%,C%
INPUT "Enter a positive number:",NUMBER
INPUT Q(8)
```

If you specify multiple variables in the INPUT statement, you are expected to enter multiple items of data. If the data being entered is numeric, you may separate data items with either commas or spaces. If the data being entered is a string, you must separate data items with commas.

If you mix floating point and string input, you must use commas to separate the data being input. For example, if A, B, and C are numeric variables and D\$ and E\$ are string variables, consider the following legal examples:

```
INPUT A,B,CRETURN
? 1,2,3RETURN

INPUT A,B,D$,CRETURN
? 1 2 DAY 3RETURN

INPUT D$,E$RETURN
? DAY,MONTHRETURN
```



For information on the statement to use if you want to enter strings containing commas, quotes, and other special characters, see "INPUT LINE" below.

When you use INPUT, remember the default size of unmapped string variables is ten characters; if you want to use larger strings, use the STRSIZ statement to reset the default string size or use a MAP statement for the string variable, specifying the size you need. See below for information on STRSIZ.

If a user of your program does not enter as many items of data as are expected by the variables in the INPUT statement, AlphaBASIC PLUS displays a double question mark to ask for more. For example:

```

INPUT A,B,C 
? 1,2 
?? 3 

```

The direct statement asks for three items of numeric data. Because only two were entered, AlphaBASIC PLUS responds with ?? to ask for the third value.

Be careful to correctly enter the type of data the variables in the INPUT statement expect. If you enter a string for a numeric variable, AlphaBASIC PLUS sets that variable to zero. For example:

```

INPUT A1 
? ME 
PRINT A1 
0

```

If you enter a numeric value for a string variable, AlphaBASIC PLUS accepts it as a string. Therefore, your programs should make sure the correct data has been entered.

If a value has not been assigned to a variable, AlphaBASIC PLUS assumes the variable contains a zero (if a numeric variable) or a null (if a string variable). If you press in response to an INPUT statement request for data, AlphaBASIC PLUS leaves the variable being input set to a zero or null (if a value has not yet been assigned) or to the value previously assigned to the variable. The same is true for pressing / if Control-C or error-trapping is on. For example:

```

A = 3 
INPUT A 
? 
PRINT A 
3

```

If you press or / in response to a data request, and the INPUT statement contains several variables, AlphaBASIC PLUS skips over any variables remaining in the INPUT statement, leaving their values unchanged. An example might help to clarify. If you run the following program:

```

START:
  INPUT "Enter day, month, year: ",DAY,MONTH,YEAR
  PRINT "Day:";DAY,"Month:";MONTH,"Year:";YEAR
  PRINT : GOTO START

```

it looks like this:

```
Enter day, month, year:  21,4 RETURN
?? RETURN
Day: 21      Month: 4      Year: 0

Enter day, month, year:  8 RETURN
?? RETURN
Day: 8       Month: 4      Year: 0

Enter day, month, year:  31,12,1980 RETURN
Day: 31      Month: 12     Year: 1980

Enter day, month, year:  CTRL / C

Operator interrupt in DAY.RP
```

You may also use the INPUT statement to read data from sequential files. For more information on this use of the statement, see Chapter 15.

9.21 INPUT LINE

The format is:

```
INPUT LINE {"prompt-string",}variable
```

Although you may specify a numeric variable, the real purpose of INPUT LINE is to let you enter string data from your terminal including commas, quotation marks, blanks, and other special characters.

You usually want to use INPUT (see the section above) for inputting numeric data or multiple items of string data.

INPUT LINE loads into the specified string variable an entire line of data, up to but not including the first LINE FEED character it encounters. It removes a Carriage Return character if one immediately proceeds the LINE FEED character, but not if it is embedded in the line. Therefore, DO NOT specify more than one string variable in an INPUT LINE statement.

If you use INPUT or INPUT LINE to read data into a variable that contains a null character, that variable is terminated by that null character (even if other data follows it). The only case where this is not true is with unformatted variables.

AlphaBASIC PLUS never prints a question mark prompt for INPUT LINE as it does for INPUT, but you may include your own `prompt-string`, which AlphaBASIC PLUS displays as a request for data. It must be a string literal (enclosed in quotation marks).

Unlike INPUT, if you press RETURN in response to a data request, INPUT LINE sets the variable to zero (if numeric variable) or null (if string variable). Remember, in a like case, INPUT leaves the value of the variable unchanged.

When you use INPUT LINE, remember the default size of unmapped string variables is ten characters; if you want to use larger strings, use the STRSIZ statement to reset the default string size or use a MAP statement for the string variable, specifying the size you need. See below for information on STRSIZ.

Some examples of INPUT LINE:

```
INPUT LINE A$  
INPUT LINE "ENTER YOUR FULL NAME, PLEASE: ",NAME$
```

You may also use the INPUT LINE statement to read data from a sequential file. For more information on using INPUT LINE and files, see Chapter 15.

9.22 INPUT RAW

INPUT RAW allows you to input special characters to your program. The format is:

```
INPUT RAW {"prompt-string",}variable{,variable(s)}
```

The variable(s) MUST be unformatted. See Chapter 14 for information on unformatted data. INPUT RAW inputs in data mode. It accepts characters from the keyboard until all of the variables are filled. It accepts, but otherwise ignores, any special characters, such as carriage returns, linefeeds, ^C, etc. The characters entered are NOT echoed back to the terminal.

9.23 LET

The format is:

```
LET variable = expression
```

Assigns a value to a specific variable during execution of the program. You do not have to specify the LET keyword in an assignment statement. Some examples:

```
LET A5 = 12.4  
LET SUM(4,5) = A1 + SQR(B1)  
LET C$ = "JANUARY"  
A5 = 12.4  
SUM(4,5) = A1 + SQR(B1)  
C$ = "JANUARY"
```

9.24 LOOP

See DO WHILE/UNTIL LOOP, above.

9.25 NEXT

See FOR, TO, NEXT, and STEP, above.

9.26 NO'DIVIDE'BY'0

Turns on the "Divide by zero" error. When this statement is executed, all attempts to divide by zero result in an error. This is the default condition, unless changed by a DIVIDE'BY'0 command (see above).

9.27 NOECHO

Turns off the keyboard echo. The format is:

```
NOECHO
```

When the keyboard echo is off, no characters are displayed on the terminal as they are entered from the keyboard. This is useful for entering passwords or other data you want only the user to see. Use the ECHO statement to turn the echo on again.



Don't turn the ECHO off if you are going to use the AMOS statement to execute AMOS commands, because you could hang your job. NOECHO is not allowed as a direct statement in interactive mode.

9.28 ON CTRLC GOTO AND RESUME

Transfers control if a ^C is used during the program execution. The format is:

```
ON CTRLC GOTO line-number or label
```

You can change the line-number/label by specifying a new one by using the statement more than once in a program, or turn the GOTO off by specifying 0 as the line number. While you use ON CTRLC GOTO, Control-C is shut off until the RESUME statement is used (see the next section for more on RESUME). This statement is used in error recovery—for more information, see Chapter 17.

9.29[∞]ON ERROR GOTO AND RESUME

Turns error trapping ON and OFF. The formats are:

```
ON ERROR GOTO 500
ON ERROR GOTO traproutine
ON ERROR GOTO
```

In the first two cases, if an error occurs, the program goes to the specified area. In the third case, the program ends and the appropriate system error message is printed. After your error recovery procedure is done, use RESUME to continue the program execution:

```
RESUME {line number or label}
```

For more information on error trapping, see Chapter 17.

9.30[∞]ON GOSUB (CALL)

The ON GOSUB statement allows multi-path branching to one of several subroutines within the program based on the result of evaluating an expression. The formats are:

```
ON expression GOSUB label/line#1{,label(s)/line#(s)}
ON expression CALL label/line#1{,label(s)/line#(s)}
```

expression can be any valid expression which is evaluated and truncated (or rounded, if IEEE) to a positive integer result. The result of the expression evaluation is then tested. The subroutine at label/line#1 is executed if the result is 1, the subroutine at label/line#2 is executed if it is 2, etc. If the result is zero, negative or greater than N, the program goes on to the next statement.

As with GOSUB, the verb CALL may be used in place of the verb GOSUB, giving an ON CALL statement. Here is an animation program using ON and GOSUB:

```
START:
  I = 3 * RND(0) + 1           ! Random number 1 to 4
  ON I GOSUB UP,DOWN,STRAIGHT ! Go to 1 of 3 subroutines
  GOTO START
UP:
  PRINT "/" ; TAB(-1,3);
  RETURN                      ! Draw symbol, up 1 row
DOWN:
  PRINT TAB(-1,4) ; "\" ;
  RETURN                      ! Down 1 row, draw
STRAIGHT:
  PRINT "___" ;
  RETURN                      ! Draw symbol
```

9.31[∞]ON - GOTO

The format is:

```
ON expr GOTO label/line#1{,label/line#2,...label/line#N}
```

The ON GOTO statement allows multi-path GOTO branching to one of several points within the program based on the result of evaluating the `expr` expression.

`expr` can be any valid expression which is evaluated and truncated or rounded (if IEEE) to a positive integer result. The result is then tested to branch to `label/line#1` if 1, `label/line#2` if 2, etc. If the result is zero, negative or greater than N, the program falls through to the next statement. The following is a portion of a menu-selection program:

```
PRINT TAB(22);"Select One of the Following Operations:"
PRINT : PRINT TAB(25);"1.  Insert/Edit NAME Information."
PRINT TAB(25);"2.  Insert/Edit PHONE NUMBER Information."
PRINT TAB(25);"3.  Quit without insertion or editing."
PRINT : INPUT "Your choice (1, 2 or 3)? ",A
ON A GOTO NAME, PHONE, QUIT
NAME:  INPUT "Select a name: ",N
```

(The program continues with all three alternatives)

At the end of the program section called by the GOTO, an END or another GOTO may be used to keep the program from executing other portions of program code.

9.32[∞]PRINT

Evaluates and displays on your terminal the expressions you specify. The format is:

```
PRINT expression-list
```

or:

```
? expression-list
```

For example:

```
PRINT 3 + 4;" HAVE " + "A NICE " + "DAY" RETURN
7 HAVE A NICE DAY
```

AlphaBASIC PLUS prints a carriage return/linefeed after `expression list`, unless it ends in a comma or semicolon. Remember an expression may consist of a string or numeric variable, numeric constant, string literal or constant, function with arguments, operator symbols, or a combination of these elements. For example, the following is one string expression:


```
"STRING DATA" + NAME$ + MID$(A$,1,2).
```

AlphaBASIC PLUS displays numeric data with a trailing blank. It also prints one leading blank if the number is positive. It does not print a leading blank if the number is negative. AlphaBASIC PLUS displays string data with no leading or trailing blanks.

You may place more than one numeric expression after the PRINT keyword if you separate them with commas or semicolons. If you separate the expressions by semicolons, AlphaBASIC PLUS does not print extra spaces when it prints the evaluations of those expressions. For example:

```
? 12 + 12;-32;8/2
```

prints:

```
24 -32 4
```

There are no blanks between the numbers above except for the normal leading and trailing blanks displayed with numeric data.

String expressions may be separated by blanks, semi-colons, or commas. Semi-colons (blanks are the same as semi-colons) cause the elements to be printed with no spaces between them. For example:

```
DAY$ = "Monday" RETURN
PRINT "Status Report, ";DAY$ RETURN
Status Report, Monday
```

If you separate the expressions by commas, AlphaBASIC PLUS prints the data in "print zones." AlphaBASIC PLUS divides the area in which data is to be displayed into five zones of 14 spaces each. If an expression in a PRINT statement is followed by a comma, AlphaBASIC PLUS prints that expression in the next available print zone. For example, the AlphaBASIC PLUS statements:

```
PRINT 34,1024,-32,20,72
PRINT "AA","BB","C","DDD","EE","FFFF"
```

display:

| | | | | |
|------|------|-----|-----|----|
| 34 | 1024 | -32 | 20 | 72 |
| AA | BB | C | DDD | EE |
| FFFF | | | | |

When you look at the display above, remember AlphaBASIC PLUS prints numeric data with a leading and trailing blank if the number is positive, but just a trailing blank if the number is negative.

Note the strings in the second line were displayed on two different lines—when AlphaBASIC PLUS still has an expression to print after it has printed something in the fifth zone, it starts over again with the first zone on the next line.

If you end the PRINT statement expression list with a semicolon or comma, AlphaBASIC PLUS does not output a carriage return/linefeed when it finishes displaying that expression list. This makes the output resulting from the next PRINT or INPUT statement appear on the current display line.

The next output appears in the next print zone if the current PRINT statement ends with a comma; or, the next output appears immediately following the last character of the current PRINT statement if the PRINT statement ends with a semicolon.

Here are a few examples of the PRINT statement:

| | |
|-----------------------------|--------------------------------------|
| A\$ = "HERE" : A = 7 | |
| ! | ! What you see: |
| PRINT | ! A blank line |
| PRINT A | ! 7 |
| PRINT A\$ | ! HERE |
| PRINT 1 + 2 | ! 3 |
| PRINT "ANY TEXT" | ! ANY TEXT |
| PRINT "NOTE THE SPACE",A\$ | ! NOTE THE SPACE HERE |
| ? "YOU ARE NUMBER";A | ! YOU ARE NUMBER 7 |
| ? "YOU ARE #";A;"IN CLASS." | ! YOU ARE # 7 IN CLASS. |
| PRINT "THERE ARE"; | ! Semicolon suppresses CR/LF: |
| PRINT A;"DAYS LEFT." | ! THERE ARE 7 DAYS LEFT. |

You may also use PRINT for writing data to sequential files; see Chapter 15.

9.33 PRINT USING

The formats are:

```
variable = expression USING format-string
PRINT USING format-string, expression-list
PRINT expression USING format-string
```

PRINT USING is for formatting output and is described extensively in Chapter 12.

9.34 PROGRAM

Lets you specify a revision level for your program. The format is:

```
PROGRAM Name, A.B{C}{(D)}
```

where:

| | |
|------|--|
| Name | A reference name (non-functional) |
| A | The major revision number (from 0 to 255) |
| B | The minor revision number (from 0-15) |
| C | The revision subscript (optional, letter from A-O) |
| (D) | The edit number (optional, from 0 to 4095) |

The group of characters following the PROGRAM keyword are evaluated when the program is compiled, and therefore must be literal characters or numbers—not variables. AlphaBASIC PLUS checks each of the fields to determine if the proper range of number or letter is specified. There must be a least one space after the PROGRAM keyword. If part of your revision specification is incorrect, you see an error. Here is an example of a full PROGRAM statement:

```
PROGRAM Test3,1.0b(103)
```

If your AlphaBASIC PLUS program contains a PROGRAM statement, the compiled version of that program (the .RP file) contains the version information you specified.

The AMOS command DIR displays general information about specified disk files. If you use the /V option with the DIR command, you also see version information for any .RP files included in that directory display. For example:

```
DIR/V *.RP RETURN
```

```
MYPROG  RP  2      1.0B(103)  DSK2:[50,0]
RECEIV  RP  15     4.2E(50)
INVEN   RP  102    0.2
Total of 3 files in 119 blocks
```

If the original program file from which the .RP file was generated did not contain a PROGRAM statement, DIR displays the version number as 0.0.

9.35 RANDOMIZE

The format is:

```
RANDOMIZE
```

Resets the random number generator seed to begin a new random number sequence starting with the next RND(X) function call. See Chapter 10 for information on the random number generator.

9.36 READ, RESTORE, AND DATA

The formats are:

```
READ variable1{,variable2,...variablen}
RESTORE
DATA data1{,data2,...dataN}
```

These calls let data be an integral part of the source program with a method for getting this data into specific variables in an orderly fashion.

DATA statements are followed by one or more literal values (that is, not variables) separated by commas. String literals need not be in quotes unless the literal data contains a comma. All data statements are placed into a dedicated area in memory no matter where they appear in the source program.



Do not place comments on a DATA statement line—the comment may be interpreted as data!

READ statements are followed by one or more variables separated by commas. Each time a READ statement is executed, the next item of data is retrieved from the DATA statement pool and loaded into the variable named in the READ statement.

If there is no more data left in the data pool, the program can only continue to read data if a RESTORE statement is executed, which reinitializes the reading of the data pool from the beginning. Otherwise, you see an "out of data" message, and the program stops. READ accepts the E and D exponential markers for AMOS or IEEE floating point variables, but not alternate radix specifiers. Here are some forms READ and DATA may take:

```
DATA 1,2,3,4,5
DATA 2.3,0.555,ONE STRING,"4,4"
READ A,B,C
READ A$
READ C(2,3),B$(4)
```

Here is a program example using READ, RESTORE, and DATA:

```
START:
  INPUT LINE "How much did you pay for your car? $",WORTH
  ? "Based on national averages, your car " &
    "will depreciate: " : PRINT
  FOR I = 1 TO 5
    ? "After the "; : READ YEAR$ : ? YEAR$;" year, ";
    ? "your car will be worth about:"; : READ PERCENT
```

```

        WORTH = WORTH * PERCENT
        ? WORTH USING "$$#####.##" : ?
    NEXT I
DATA first,.77,second,.78,third,.79,fourth,.81,fifth,.84
RESTORE
INPUT LINE "Would you like to see another schedule? ",L$
IF UCS(L$[1,1]) = "Y" THEN
    GOTO START
ELSE
    PRINT "Goodbye."
ENDIF
END

```

9.37^{oo}RENAME

Renames a disk file. The format is:

```
RENAME old-filename,new-filename{,status}
```

or:

```
RENAME old-filename TO new-filename{,status}
```

The `filenames` can be any string expression. The `status` is a variable containing a value returned by `RENAME` based on the results of the renaming operation. If `status` = 0, the rename worked fine. If `status` = 1, an error occurred at AMOS level, and the file wasn't renamed. If you don't specify a status variable, a run time error is generated if an error occurs at AMOS level. For example, if you try to rename a file that doesn't exist, `status` equals 1, or, if you didn't specify `status`, your program ends and you see an AMOS error message explaining the file doesn't exist. Some examples:

```

RENAME "payrol.dat" TO "payrec.old",status
RENAME NEW'DATA'REPORTS,WAVE'LENGTHS' FILE
RENAME "Data.log" TO a$,status'of'rename

```

The standard AMOS rules apply—you can't rename files on different devices or in accounts with a different project number than the one the program is running in.

There is also a `RENAME` function—see Chapter 11.

9.38^{oo}REPEAT

Starts the next iteration of a loop structure. `REPEAT` can be used within `FOR-NEXT` or `DO-LOOP` structures to cause control to pass to the `NEXT` or `DO` statement, so the loop begins again. For example:

```
DO WHILE X < 100
  X = X + A
  IF X > 5 THEN
    X = X + B
  REPEAT
ENDIF
X = X + C
LOOP
```

In the above case, REPEAT causes the statement `X = X + C` to be skipped if `X > 5`.

9.39°°RESTORE

See READ, RESTORE AND DATA, above.

9.40°°RESUME

See ON ERROR AND RESUME, above.

9.41°°RETURN

See GOSUB (OR CALL) AND RETURN, above.

9.42°°SCALE

The format is:

```
SCALE value
```

The scaling factor represents the number of decimal places the digit window is effectively shifted to the right or left in any floating point number. SCALE is discussed in detail in Chapter 13.

9.43°°SIGNIFICANCE

The format is:

```
SIGNIFICANCE value
```

The significance statement lets you dynamically change the default `value` of the maximum number of digits to be printed for unformatted numbers. `value` can be any number from 1 to 11 for AMOS variables, and from 1 to 15 for IEEE variables. Rounding off to the specific number of digits is not done until just before printing the result.

value can be a literal number or a variable, and is truncated to an integer if expressed as a decimal (unless IEEE variables are being used, in which case it is rounded). For example, SIGNIFICANCE 3.6 is equal to SIGNIFICANCE 3 with AMOS variables, or SIGNIFICANCE 4 for IEEE variables.

The statement SIGNIFICANCE 8, for instance, sets the number of printable digits to 8. The current significance is ignored when PRINT USING is in effect.

Note the SIGNIFICANCE statement only affects the final printed result of all numeric calculations. The calculations themselves and the storage of intermediate results are always performed in full precision to reduce the number of errors.

The significance default is 6 digits. This is equivalent to standard single-precision formats used in most of the popular versions of BASIC. The significance is not reset by RUNP, and therefore may be set in interactive mode in a direct statement just prior to the actual running of a test program. Of course, any SIGNIFICANCE statements encountered during the execution of the program reset the value.

SIGNIFICANCE has no effect on the printing of integer values.

9.44[∞]SLEEP

Causes the job to sleep for a specified number of seconds. The format is:

```
SLEEP time-in-seconds
```

For example:

```
SLEEP 10
SLEEP wait'period
SLEEP 4.57
```

SLEEP truncates decimal values (or rounds if compiled with /I). SLEEP 4.57 causes a 4 second pause (or 5, if IEEE).

Sleeping for a negative amount of time will cause the program to sleep for a very long period.

9.45[∞]STEP

See FOR, TO, NEXT and STEP, above.

9.46[∞]STOP

The format is:

```
STOP
```

Causes the program to suspend execution. If you are in interactive mode, you may continue to the next statement in sequence by executing a CONT command or a single-step command. If you are running the program at AMOS command level, press **RETURN** to continue or **CTRL/C** to end the run.

9.47[∞]STRSIZ

Sets the default value for all unmapped string variables which are encountered for the first time after the STRSIZ statement when the program is compiled. Note that STRSIZ is a compile-time setting: all source code lines after the STRSIZ line are affected. Therefore you cannot "jump around" a STRSIZ statement at runtime by using GOTO commands. The format is:

```
STRSIZ value
```

The default string size is 10 characters. The statement:

```
STRSIZ 25
```

for example, causes all newly allocated strings which follow to have a maximum size of 25 characters, instead of 10. This includes the allocation of string arrays. The size value is evaluated at compilation time and so **must be** a single positive integer. The maximum size you can allocate depends on the amount of memory on your system, and on how many variables you use, but cannot be greater than 65,535 bytes.

9.48[∞]SWITCH/CASE

Selects sections of code based on a comparison of cases. The format is:

```
SWITCH expression
  CASE constant
    statement(s)
  CASE constant
    statements(s)
    .
    .
  {DEFAULT
    statement(s)}
ENDSWITCH
```


The constants in each CASE statement must be the same type as the expression in the SWITCH statement, and cannot be variables. The constant in each CASE statement is compared to the expression in the SWITCH statement. If they are equal, the statements following the CASE statement are executed.

Unless an explicit GOTO or EXIT statement is used, control passes to the next CASE whether that CASE matched or not. Once one match is found, the remaining statements in the SWITCH/CASE block are executed. Therefore, if you do not want subsequent CASEs to be executed, you must explicitly use a GOTO, EXIT or RESUME (if the SWITCH is inside an error-trapping routine) statement.

The optional DEFAULT clause is executed if no matches are found, and if no transfer of control is found in a previous matching case statement. If no DEFAULT clause is specified, and no matches occur, control passes to the statement after ENDSWITCH. Here is an example of SWITCH:

```

SWITCH Stock'Number
  CASE 00001
    GOSUB Print'Shipping'Label
    Inventory'00001 = Inventory'00001 - 1
    EXIT
  CASE 00002
    GOSUB Print'Mailing'Label
    Inventory'00002 = Inventory'00002 - 1
    EXIT
  CASE 00003
    GOSUB Print'Pickup'Order
    Inventory'00003 = Inventory'00003 - 1
    EXIT
  DEFAULT
    PRINT "There is no such Stock Number"
    GOTO Re'enter'Number
ENDSWITCH

```

Each constant can have a range of values. For example:

| | |
|----------------|---|
| CASE 12 | match only on 12 |
| CASE ...11 | match on 11 or less |
| CASE 13... | match on 13 or greater |
| CASE 2...14 | match from 2 to 14, inclusive |
| CASE 'a'...'g' | match from a to g, inclusive |
| CASE '\015' | match this octal value to ASCII decimal value 013 |

Values such as 'a' are character constants, which hold a decimal ASCII value. 'a' equals 97, for example. Values such as '\015' are in octal notation, and translate to an ASCII decimal value (013 in above example). See Chapter 7 for more on character constants. You may also use multiple CASE statements to select values not connected in any pattern. For example:

```
CASE 0
CASE 2
    [ statements ]
```

Matches on 0 or 2 only.

9.49[∞]THEN

See IF, THEN, and ELSE, above.

9.50[∞]TO

See FOR, TO, NEXT and STEP, above.

9.51[∞]UNTIL

See DO UNTIL, above.

9.52[∞]USING

See PRINT USING, above.

9.53[∞]WHILE

See DO WHILE, above.

9.54[∞]XCALL

The format is:

```
XCALL routine{,argument(s)}
```

Executes an external assembly language subroutine. Assembly language subroutines are discussed in detail in Chapter 18. For information on the assembly language subroutines available for use with AlphaBASIC PLUS programs, see your *AlphaBASIC XCALL Subroutine User's Manual*.

CHAPTER 10

FUNCTIONS

Functions compute and return a value. A function either operates on or is controlled by the argument, which is enclosed in parentheses. This chapter discusses these types of functions:

- Numeric
- Trigonometric
- String
- Miscellaneous

Functions are different from program statements in that they return a value. In order to see or use that value, you must include the function in a program statement that evaluates the expression the function call is a part of. For example:

```
SQR(16)
```

doesn't display a value. You must either assign the value returned by the function to a variable or display the value using a PRINT statement if you want to use or see the value returned. For example:

```
ROOT = SQR(16)
RESULT = ROOT * (SQR(NUMBER) + 24)
```

or:

```
PRINT "The answer is: "; SQR(16)
```

Functions either return a numeric or string value—the descriptions below describes the arguments each accepts and what type of value is returned.



You can also define your own functions—see Chapter 20.

Note the mode independence feature of the expression processor performs automatic conversions if a numeric argument is used where a string argument is expected, and vice versa.

10.1[∞]NUMERIC FUNCTIONS

Numeric functions accept a string or numeric argument and return a numeric value. The following numeric functions are supported:

10.1.1[∞]ABS(X)

Returns the absolute value (a positive number) of the argument X. For example:

```
READY
PRINT ABS(-32.4) RETURN
32.4
PRINT ABS(17.2) RETURN
17.2
```

10.1.2[∞]ASC(X)

Returns the ASCII decimal value of the first character of argument X. The argument may be any string expression. For example:

```
READY
First'Name$="Amy" RETURN
PRINT ASC(First'Name$) RETURN
65
```

You may also use string literals:

```
PRINT ASC("Z") RETURN
90
```

10.1.3[∞]EXP(X)

Returns the constant e (2.7182818285) raised to the power X. e is the base of the system of natural logarithms.

10.1.4[∞]FACT(X)

Returns the factorial of X. A factorial is the result of multiplying the number X by all integers less than X (i.e., if X = 7, the factorial equals 7*6*5*4*3*2*1). For example:

```
READY
PRINT FACT(4) RETURN
24
```



This function actually is the Gamma function, and can be used with all positive real numbers, and all negative non-integer real numbers.

10.1.5[∞]FIX(X)

Returns the integer part of X (fractional part truncated). For example:

```
READY
PRINT FIX(50.4685) RETURN
50
PRINT FIX(-1.9) RETURN
-1
```

10.1.6[∞]INT(X)

Returns the largest integer less than or equal to the argument X. The only time you see a difference between using INT and FIX is if you are working with negative numbers. For example, the largest integer less than or equal to 23.4 is 23. However, the largest integer less than or equal to -23.4 is -24 (FIX returns -23).

10.1.7[∞]LOG(X)

Returns the natural (base e) logarithm of the argument X. The logarithm is the number the constant e (2.7182818285) must be raised to the power of in order to get X.

10.1.8[∞]LOG10(X)

Returns the decimal (base 10) logarithm of the argument X. The logarithm is the number 10 must be raised to the power of in order to get X.

10.1.9[∞]RND(X)

Returns a random number between 0 and 1. The number returned is based on a previous value known as the "seed." The argument X controls the number to be returned. If X is negative, it is used as the seed to start a new sequence of numbers. If X is zero or positive, the next number in the sequence is returned, depending on the current value of the seed (this is the normal mode). The RANDOMIZE statement may be used to create a number which is truly random and not based on a fixed beginning value set by the system.



If you want to generate a random number greater than or equal to number A and less than number B, use the expression: $(B - A) * \text{RND}(0) + A$.

The INT function can be used when generating random integer numbers to give you an integer number. For example, to generate a random integer greater than or equal to 5 and less than 31, use the expression:

```
INT(26 * RND(0) + 5)      ! B - A = 26
```

10.1.10[∞]RNDN(X)

Returns the result of rounding the number to the nearest integer. Numbers with a decimal fraction of 0.5 or larger are rounded to the next higher integer (in absolute value); those with a smaller decimal fraction are rounded down. For example:

```
READY
PRINT RNDN(50.4685) (RETURN)
50
PRINT RNDN(50.876) (RETURN)
51
PRINT RNDN(-50.486) (RETURN)
-50
PRINT RNDN(-50.876) (RETURN)
-51
```

10.1.11[∞]SGN(X)

Tells you the sign of the number X. If X is a negative number, SGN(X) generates a -1. If X is 0, SGN(X) is 0. If X is a positive number, SGN(X) is 1.

10.1.12[∞]SQR(X)

Returns the square root of the argument X. The square root is the number when multiplied by itself gives you X. For example:

```
READY
PRINT SQR(25) (RETURN)
5
```

10.1.13[∞]VAL(A\$)

Returns the numeric value of the string variable or literal A\$ converted to floating point under normal AlphaBASIC PLUS format rules. For example, VAL("13") returns 13.

10.2[∞]TRIGONOMETRIC FUNCTIONS

The following trigonometric functions are implemented in full accuracy:

| | |
|------------|--------------------------|
| SIN(X) | Sine of X |
| COS(X) | Cosine of X |
| TAN(X) | Tangent of X |
| ATN(X) | Arctangent of X |
| ASN(X) | Arcsine of X |
| ACS(X) | Arccosine of X |
| DATN(X, Y) | Double arctangent of X,Y |

The arguments X and Y are assumed to be in radians. For computations on arguments in degrees, multiply the argument by pi divided by 180. For example:

```
PRINT TAN(20 * (3.1415926/180)) (RETURN)
.36397
```

10.3[∞]STRING FUNCTIONS

The following string functions accept numeric or string arguments, and perform operations on strings. You may specify these functions with a \$ after the name for clarity if you wish (i.e., CHR(90) or CHR\$(90)).

10.3.1[∞]CHR(X)

Returns a single character having the ASCII decimal value of X. Only one character is generated for each CHR function call. For example:

```
READY
PRINT CHR(90) (RETURN)
Z
```

X should be in the range from 0 to 255 inclusive. Values outside this range may produce strange result.

10.3.2[∞]EDIT\$(A\$,C)

The format is:

```
EDIT$(expression,edit-code)
```

EDIT\$ modifies the expression according to the edit-code. Numeric expressions are converted to string. The codes are:

- | | |
|----|--|
| 2 | Take out all spaces and tabs |
| 4 | Take out excess characters (CR, LF, FF, ESC, RUBOUT & NULLs) |
| 8 | Take out leading spaces and tabs |
| 16 | Reduce spaces and tabs to one space |
| 32 | Take out trailing spaces and tabs |

You may combine these editing options by specifying an additive of the numbers. For example, if you use B\$ = EDIT\$(A\$,6), it does both 2 and 4—take out spaces, tabs, and excess characters.

Note AlphaBASIC PLUS processes these **in order**—EDIT 2 first, then EDIT 4. Also, a string is not considered to have trailing spaces or tabs if a carriage return (CR) is at the end of the string. If you use an odd number, it is rounded down to the next even number, since there is no EDIT code 1. For example:

```
READY
STRSIZ 50 RETURN
Test'String$ = "This is a string with spaces" RETURN
PRINT EDIT(Test'String$,2) RETURN
Thisisastringwithspaces
```

Do not use values above 63—they won't cause any harm, but could make your program work incorrectly in the future if the list of EDIT codes is expanded.

10.3.3[∞]FILL\$(A\$,L)

Copies an expression (or part of an expression) to a certain length. If a numeric expression is given, it is first converted to a string. The format is:

```
FILL$(expression,length)
```

For example:

```
READY
SIGNIFICANCE 15 RETURN
A$ = "..XX" RETURN
PRINT FILL$(A$,14) RETURN
..XX..XX..XX..
```



You may need to adjust the String Work Area size if you use FILL\$(). See the section "Setting the String Work Area" in Chapter 8 for details.

10.3.4[∞]INSTR(X,A\$,B\$)

Performs a search for the substring B\$ within the string A\$, beginning at the Xth character position. It returns a numeric value of zero if B\$ is not in A\$ at or after the Xth position, or the character position if B\$ is found within A\$. The format is:

```
INSTR(position, expression, expression)
```

Character position is measured from the start of the string, with the first character position represented as one. Here are some direct statements to illustrate:


```
READY
A$ = "ELEPHANT" RETURN
B$ = "ANT" RETURN
PRINT INSTR(1,A$,B$) RETURN
6

PRINT INSTR(8,"MEADOWLARK","LARK") RETURN
0
```

In the second example, the specified string "LARK" is not found in the string "ARK", which is the substring starting at the 8th position, therefore, a zero is returned.

10.3.5[∞]LCS(A\$)

Returns a string which is similar to the argument string (A\$), but with all characters translated to lower case. For example:

```
READY
STRSIZ 20 RETURN
A$ = "A is for Alpha" RETURN
PRINT LCS(A$) RETURN
a is for alpha
```

LCS is useful for checking input when the case is not certain. For example:

```
IF LCS(VAR$) = "yes" THEN GOTO START
```

is easier than using:

```
IF VAR$ = "YES" OR VAR$ = "yes" THEN GOTO START
```

10.3.6[∞]LEFT(A\$,X)

Returns the leftmost X characters of a string expression. For example:

```
READY
A$ = "Now is the time" RETURN
PRINT LEFT(A$,8) RETURN
Now is t
```

10.3.7[∞]LEN(A\$)

Returns the number of characters in a string expression. For example:

```
READY
STRSIZ 50 RETURN
A$ = "Wherefore art thou, Romeo?" RETURN
PRINT LEN(A$) RETURN
26
```

This is also helpful for checking for null strings. For example:

```
IF LEN(VAR$) = 0 THEN GOTO RE'INPUT
```

10.3.8[∞]MID(A\$,X,Y)

Returns the substring composed of the characters of the string expression A\$ starting at the Xth character and extending for Y characters. A null string is returned if X is greater than the length of A\$. For example:

```
READY
STRSIZ 60 RETURN
A$ = "The quick brown fox jumped over the sleeping dog" RETURN
PRINT MID(A$,17,15) RETURN
fox jumped over
```

If X is negative, the starting point of the MID string is X number of characters from the end of the string. For example:

```
READY
STRSIZ 60 RETURN
A$ = "The quick brown fox jumped over the sleeping dog" RETURN
PRINT MID(A$,-12,8) RETURN
sleeping
```

If Y is negative, the MID string starts at a point Y distance before the X starting point. For example:

```
READY
STRSIZ 60 RETURN
A$ = "The quick brown fox jumped over the sleeping dog" RETURN
PRINT MID(A$,12,-8) RETURN
quick br
```

AlphaBASIC PLUS counts the positions even if the parameters go beyond the string's endpoints. For example:

```
READY
STRSIZ 60 RETURN
A$ = "Cost Accounting" RETURN
PRINT MID(A$,20,-15) RETURN
Accounting
```

10.3.9[∞]RIGHT(A\$,X)

Returns the rightmost X characters of the string expression A\$. For example:

```
READY
STRSIZ 60 RETURN
A$ = "I THINK, THEREFORE I AM" RETURN
PRINT RIGHT(A$,4) RETURN
I AM
PRINT RIGHT(1234,2) RETURN
34
```

Remember you can use numeric arguments for many string functions.

10.3.10[∞]SPACE(X)

Returns a string of X number of blank spaces. X must be a positive number. The statement:

```
READY
PRINT "COLUMN A"; : PRINT SPACE(10); : PRINT "COLUMN B" RETURN
COLUMN A          COLUMN B
```

where the 10 spaces between the first and second strings are the result of the SPACE(10) function. SPACE is especially handy for padding strings to a fixed length. For example:

```
STRSIZ 25
INPUT "Name?",NAME$
IF LEN(NAME$) < 25 &
    THEN NAME$ = NAME$ + SPACE(25 - LEN(NAME$))
```



You may need to adjust AlphaBASIC PLUS's string work area size if you use SPACE(). See the Section "Setting The String Work Area Size" in Chapter 8 for details.

10.3.11[∞]STR(X)

Returns a string which is the character representation of the numeric expression X. No leading space is returned for positive numbers. For example:

```
READY
A$ = STR(45) (RETURN)
PRINT A$ (RETURN)
45
```

10.3.12[∞]STRIP\$(A\$)

Returns a copy of an expression having no trailing spaces or tabs. The format is:

STRIP\$(expression)

If the expression is numeric, it is converted to a string.

10.3.13[∞]TIME

Returns the number of seconds since midnight.

10.3.14[∞]UCS(A\$)

Returns a string which is similar to the argument string (A\$), except all characters are translated to upper case. For example:

```
READY
STRSIZ 20 (RETURN)
A$ = "M is for Micro" (RETURN)
PRINT UCS(A$) (RETURN)
M IS FOR MICRO
```

This is useful for checking input when the case is not certain. For example:

```
IF UCS(VAR$) = "YES" THEN GOTO START
```

is easier than using:

```
IF VAR$ = "YES" OR VAR$ = "yes" THEN GOTO START
```

10.4 MISCELLANEOUS FUNCTIONS

10.4.1 CMDLIN

Returns a string containing the rest of the AMOS command line after **RUNP** [program-name]. For example, if you enter:

```
RUNP FIGURE This is a test. RETURN
```

when FIGURE.BAS contains this statement:

```
A$ = CMDLIN
```

A\$ contains "THIS IS A TEST." (CMDLIN translates all characters to upper case). CMDLIN thus lets you pass data into a program when the program is executed.

10.4.2 DITOS(X)

Returns the given date in separated format. See ODTIM below and DATE in Chapter 11 for more on date displays. The format is:

```
DITOS(date-in-internal-format)
```

DITOS will operate correctly for dates in the range March 1, 1900 to December 31, 2099.

10.4.3 DSTOI(X)

Returns the given date in internal format. See ODTIM below and DATE in Chapter 11 for more on date displays. The format is:

```
DSTOI(date-in-separated-format)
```

DSTOI will operate correctly for dates in the range March 1, 1900 to December 31, 2099.

10.4.4 ERRMSG(X)

Returns a string of the runtime error message corresponding to an error number returned by ERR(0). The format is:

```
ERRMSG(message-number)
```

where message-number is the error message number. This can be a literal, or an expression. For example:

```
READY
PRINT ERRMSG(10) RETURN
Divide by zero
```

or (not recommended):

```
IF ERRMSG(ERR(0)) <> "Divide by zero" THEN EXIT
```

Do not rely on the exact text of the error message. It could change from version to version.

10.4.5[∞]ERR(X)

Returns a status code referring to program status during error trapping. For a complete list of these codes and examples of using ERR(X) see Chapter 17. If X is 0, ERR returns the code of the error detected; if X is 1, ERR returns the number of the last program line encountered before the error occurred. If X is 2, ERR returns the file number of the last file accessed. If X is 3, ERR returns the error status code from the JOBERR field of the Job Control Block—this can be used after an AMOS statement to see how the command executed. See Appendix G for a list of AMOS error codes.

10.4.6[∞]FILEBLOCK(X)

Returns the absolute address of the Data-set Drive Block (DDB) corresponding to the OPENed channel specified by X. For example, FILEBLOCK(1) returns the DDB of the file opened on file-channel #1. This is useful to pass a DDB address to an XCALL assembly language subroutine. You get an error message if the file is not open or is open on channel #0.

10.4.7[∞]GETKEY(X)

Reads a character from the terminal. The format is:

```
GETKEY(X{,translation-table-filename})
```

If X = 0, GETKEY doesn't wait for input, and returns a -1 if there are no characters in the input buffer. If X = -1, GETKEY waits until a character is entered at the keyboard. If X = -2, GETKEY checks to see if there is input waiting. If there are not any characters in the input buffer, GETKEY will immediately return -1. If there is a character waiting, GETKEY will return its ASCII decimal value, but the character is not processed. Successive GETKEY(-2) will return the same value until the character is read with a GETKEY(0) or GETKEY(-1) statement, or another INPUT.

GETKEY returns the ASCII decimal value of the character. You can use CHR to translate the ASCII code to the corresponding character. Values for X other than 0, -1, and -2 are reserved for future use and produce undefined results. Here is an example:

```
User'Input = CHR(GETKEY(-1))  
IF User'Input = "A" THEN ...
```

The optional `translation-table-filename` specifies a memory-module (either in user or system memory) to be used as a translation table for the input characters. This table can translate any character into one or more other characters. It can be used to translate the character generated by special keys such as function keys into specific instructions. For example, you might want your program to display information when the HELP key is pressed. If you specify only an extension, AlphaBASIC PLUS uses the name of the terminal driver for the terminal being used as the translation table name. This allows you to make your translation tables terminal-specific.

The FIXTRN program creates these translation tables. Only modules created by FIXTRN are valid for use with GETKEY—if you use an invalid translation table, you may get erroneous results. See your *System Commands Reference Manual* for information on FIXTRN.

Let's look at an example of how a translation table works. After defining a translation table so the HELP key translates to "H" and the MENU key translates to "M," and placing this XT.XLT file in user memory, this program shows how to make use of them:

```
START:  
  a = getkey(-1,"xt.xlt")  
  print a;"          ";chr(a)  
    if chr(a) = "H" then goto HELP  
    if chr(a) = "M" then goto MENU  
    if chr(a) <> "E" then goto START  
HELP:  
  print "This is the HELP screen"  
  goto START  
MENU:  
  print "This is the MENU screen"  
  goto START
```

10.4.8[∞]ODTIM

Returns a formatted string of the specified time and date. The format is:

```
ODTIM (date,time{,flag-value})
```

The `flag-value` determines the format of the time and date. You determine the `flag-value` by first deciding which options you want from these lists:

ORDERING AND FORMATTING OPTIONS

- 0 Omit date from the output, and ignore all other date formatting flags.
- 9 Omit time from the output, and ignore all other time formatting flags.
- 14 Affects numerical output for day, month and year only. If on, leading zeros are suppressed on all elements. This produces output suitable for use in an isolated message. If off, leading zeros are added if appropriate, producing output of a constant width that is useful for printing tables. Note that full month and weekday texts are never columniated, and time formats always have leading zeros. Overridden by Option 18. See also Option 19.
- 26 Output the time before the date, else the time will follow the date.

DATE FORMATTING

Date Punctuation

- 7-8 Date punctuation control (can be overridden by Option 21):
 - Both off: Use dashes, as 20-Feb-82
 - 7 only: Use spaces, as in 20 Feb 82; if Option 6 is on and Option 3 is off, add a comma, as in February 20, 1990
 - 8 only: Use slashes, as 2/20/82.
 - Both on: Use the character defined as the date separator in the job's language definition file.
- 20 Do not output any commas in the date string. Overrides Option 2 and Option 6.
- 21 Do not output date punctuation. Overrides Options 7 and 8.

Date Ordering

- 6 Output the day of the month after the month, otherwise output it before. Ignored if Options 7 or 8 are set. A comma is appended if the day is immediately followed by the year and the month is not output as a number.
- 16 If set, the year precedes the day and month (which are printed in the order specified by Option 6).

Date Items Output

- 1 Output the day of the week.
- 23 Do not output the day. Overrides Options 6 and 19.
- 24 Do not output the month. Overrides Options 3 and 4.
- 25 Do not output the year. Overrides Options 5 and 16.

Weekday Options

- 2 Use the full text for the weekday, otherwise use a three-letter abbreviation. A comma will be added if alphabetic output is generated and the date follows the weekday. Requires Option 1. Option 22, if set, overrides this option.
- 22 Output the weekday as a number (0-6) as the first element of date output. Overrides Option 2, requires Option 1 to be set.

Day Options

- 19 Add English day suffix (st, nd, rd, th) to the numerical day output. Forces Option 14 on for day output.

Month Options

- 3 Output the month as a number 1-12; ignore Option 4.
- 4 Use the full text for the month, otherwise use a three-letter abbreviation.

Year Options

- 5 Output a four-digit year, otherwise use a two-digit year. Option 14 may disable leading zeros for two-digit years; Option 18 may force them on.
- 18 Force leading zeros in two-digit years for years in the range xx00-xx09. Overrides Option 14.

TIME FORMATTING

- 11 Use 12-hour and AM/PM, otherwise use 24-hour time.
- 15 The time value is supplied in separated format. Otherwise the time value is in internal format.
- 28 If 24-hour clock, output "hrs" after the time string. Otherwise ignored.

Hours Options

- 12 Do not output a separator between the hour and the minute values.
- 27 Affects hours output: if 12-hour clock, suppress any leading zero. If 24-hour clock, ignored.

Seconds Options

- 10 Omit seconds; otherwise include seconds preceded by the time separator.

Punctuation Options

- 13 Force a colon as the time separator, otherwise use the character defined as the time separator in the job's language definition file.
- 17 Do not output separator between the minute and the second.

To determine the flag-value, follow these steps:

1. Decide which options you want from the tables above, and note the number of each option.
2. For each number in step 1, total the corresponding value in the table below.
3. The grand-total value is flag-value.

Option Flag Values

| | | | |
|----|-------|----|-----------|
| 0 | 1 | 15 | 32768 |
| 1 | 2 | 16 | 65536 |
| 2 | 4 | 17 | 131072 |
| 3 | 8 | 18 | 262144 |
| 4 | 16 | 19 | 524288 |
| 5 | 32 | 20 | 1048576 |
| 6 | 64 | 21 | 2097152 |
| 7 | 128 | 22 | 4194304 |
| 8 | 256 | 23 | 8388608 |
| 9 | 512 | 24 | 16777216 |
| 10 | 1024 | 25 | 33554432 |
| 11 | 2048 | 26 | 67108864 |
| 12 | 4096 | 27 | 134217728 |
| 13 | 8192 | 28 | 268435456 |
| 14 | 16384 | | |

If date is zero, any value given for time is ignored, and the current time and date are used. If date is not zero, and time is zero, the time is returned as midnight. date is interpreted in separated format, and time is interpreted in internal format. These are the default formats for the DATE and TIME functions (see Chapter 11). For convenience, a flag-value of -1 returns an output in the following format: "Wednesday, April 1, 1995 04:53:23 PM". Here are two examples of ODTIM:

```
READY
PRINT ODTIM(0,0,2294) RETURN
Wednesday, January 15, 1994 11:00:00 AM

OPEN #1,"INVEN.DAT",INPUT
INPUT #1,Last'Inventory'Date,Time'of'day
Display'date = ODTIM(Last'Inventory'Date,Time'of'Day,694)
Display'time = ODTIM(Last'Inventory'Date,Time'of'Day,3073)
PRINT "Last inventory was ";Display'date;" at ";Display'time
```

ODTIM will work correctly with dates from March 1,1900 to December 31, 2099.

CHAPTER 11

SYSTEM AND FILE FUNCTIONS

AlphaBASIC has a unique group of operators called system functions, which allow you to:

- Access various system parameters
- Access the computer's input/output ports
- Access physical memory
- Work with external files

System functions use a format similar to that used by standard functions, with the reserved word representing the desired function followed by optional arguments enclosed within parentheses. The major difference is the reserved word of a system function may appear on the left side of an assignment statement. When a system function is used on the left side of the equal sign in a statement, it outputs the result of that function to a specified place.

System functions used within expressions on the right side of an assignment statement perform an input or read operation and deliver back a result to be used in the expression evaluation.

11.1 BYTE(X), WORD(X), AND LONG(X)

The BYTE, WORD and LONG functions let you inspect and alter any memory locations within the memory addressing range of the computer. BYTE and WORD are often called PEEK and POKE statements in other versions of BASIC.

The BYTE function deals with 8 bits of data in the range of 0 to 255, WORD deals with 16 bits of data on an even address boundary (which means the "X" you specify is rounded up to the next even number) in the range of 0 to 65535, inclusive, and LONG deals with 32 bits of data on an even address boundary (rounded up to the next-plus-one even number). Any unused bits are ignored, with no error message.



These commands are not protected; it is possible to cause severe damage to the operating system in memory if you use the commands improperly. If you don't understand the computer's internal memory addressing, we recommend you not use these commands.

| | |
|----------------|--|
| BYTE(X) = expr | Writes the low byte of <expr> to decimal memory location X |
| WORD(X) = expr | Writes the low word of <expr> to decimal memory location X |
| LONG(X) = expr | Writes the low longword of <expr> to decimal memory location X |
| A = BYTE(X) | Reads decimal memory location X and puts the byte in A |
| A = WORD(X) | Reads decimal memory location X and puts the word in A |
| A = LONG(X) | Reads decimal memory location X and puts the longword in A |

11.2[∞]DATE

The DATE system function returns the system date in a two-word format. You cannot set the system DATE using this function. The format is:

DATE{ (X) }

where x is:

| | |
|----|---|
| 1 | Internal format |
| 0 | Compatibility (same as AlphaBASIC—separated format) (default) |
| -1 | Full year x 10000 + month x 100 + day. For example, 19950422 |



In separated format, the year returned is actually "year - 1900." For example, 1997 is returned as 97; 2001 is returned as 101.

See your *Monitor Calls Manual* for information on separated format. DATE is mostly used for comparison purposes, to see how many days have passed between events, etc. Use ODTIM (see Chapter 10) if you want an understandable, formatted date display. Here is an example of a program to display the date in separated format:

```
MAP1  The'Date
      MAP2  Year,b,1
      MAP2  Day'of'the'Week,b,1
      MAP2  Month,b,1
      MAP2  Day,b,1

MAP1  Date'Format,I,4,@The'Date

Date'Format = DATE(0)
PRINT Day,Day'of'the'Week,Month,Year
```

You can use a similar program to compare days, days of the week, months, or years. See Chapter 14 for using the @ symbol within MAP statements.

11.3[∞]IO(X)

Allows the input/output ports to be selectively read from or written to. In both cases only one byte is considered, and a port number expression greater than 511 causes BASIC to ignore the unused bits. The range of ports available is 0 to 511.

IO(X) = expr

Writes low byte of expr to port X

A = IO(X)

Reads decimal port X into A



If (X) is 0-255, the function accesses the external I/O ports (the addresses are FFFFFFF00-FFFFFFF). If (X) is 256-511, it accesses the internal (on-board) I/O ports (addresses FFFFFFFE00-FFFFFFFEFF).

The user must have read system memory privileges to use the IO(x) function, or he/she will receive a protection violation error.

11.4[∞]MEM(X)

Returns a positive integer value which specifies the decimal number of bytes currently in use for various memory areas used by the compiler system.

The most common use of MEM(X) is to return the number of free bytes left in the user memory partition. The MEM(0) call duplicates the action performed by the FRE(X) function in some other versions of BASIC.

Other values of the argument X return memory allocations which pertain to various areas in use by the compiler, and may or may not be of use to you. The byte counts returned for the various values of X are:

| | |
|----|--|
| 0 | Free memory space remaining in current user partition |
| 1 | Total size of current user partition |
| 2 | Size of source code text area |
| 3 | Size of user label tree |
| 4 | Size of user symbol tree (variable/function names) |
| 5 | Size of compiled object code area |
| 6 | Size of data pool (compiled DATA statements) |
| 7 | Size of array index area (dynamic links to arrays) |
| 8 | Size of variable storage area (excluding arrays) |
| 9 | Size of dummy data termination field (always zero) |
| 10 | Size of file I/O linkage and buffer area |
| 11 | Size of variable array storage area (dynamically allocated at run-time) |
| 12 | Size of the user partition left free for use |
| 13 | Size of area between top of heap and top of stack less required gap size |
| 14 | Size of string work area |

The statement PRINT MEM is equivalent to PRINT MEM(0).

11.5 TIME

Retrieves the time of day from the system monitor communications area. You cannot set the system time from within AlphaBASIC. The format is:

```
TIME{(modifier)}
```

where modifier is:

| | |
|----|------------------------------------|
| 1 | Internal format |
| 0 | Same as AlphaBASIC—internal format |
| -1 | Separated format |

If modifier is not specified, or if it equals 0, the result is in internal format: a two-word integer representing the number of seconds since midnight. See your *Monitor Calls Manual* for more on separated format. Normally, TIME is used to measure time intervals. For example:

```
X = TIME
GOSUB SORT'FILE
Y = TIME
PRINT "It took ";Y - X;" seconds to sort the file."
```

You can use ODTIM (see Chapter 10) if you want a formatted time display. Here is an example of a program to display the time in separated format:

```
MAP1 The'Time
MAP2 Hours,b,1
MAP2 Minutes,b,1
MAP2 Seconds,b,1
MAP2 Excess,b,1

MAP1 Time'Format,I,4,@The'Time

Time'Format = TIME(-1)
PRINT Hours,Minutes,Seconds
```

You can use a similar program to time processes using seconds, minutes, or hours as a differential. See Chapter 14 for using the @ symbol within MAP statements.

11.6 LOOKUP

The LOOKUP function looks for a file on the disk and returns a value which tells you if the file is found and, if so, how many disk blocks it contains. The format is:

```
LOOKUP("filespec")
```

where filespec is any string expression which evaluates to a legal file description (the default extension is .DAT). The file specification can be up to 48 characters in length.

The result can be placed in any legal floating point variable. The returned value may be:

| | |
|--------------|--|
| 0.0 | File not found |
| 0.5 | File has zero blocks. Usually this signifies that the file has been opened for output by any user on the system. |
| $n \geq 1.0$ | File found. The file is a sequential file and contains n disk blocks |
| $n < 0.0$ | File found. The file is a contiguous (random) file and contains n disk blocks |

For example:

```
IF LOOKUP("TEST.DAT") = 0 THEN GOTO CREATE'FILE

RESULT = LOOKUP("TEST.DAT")
IF RESULT > 500 THEN GOTO OPEN'NEW'FILE
```

You may also use LOOKUP as a keyword—see Chapter 15.

11.7[∞]KILL

The KILL function erases a file from the disk and returns a value according to the result of the operation. The format is:

```
KILL(filespec)
```

For example:

```
if KILL("NEW.TXT") = 0 then goto FILE'NOT'FOUND
```

The `filespec` is any string expression which evaluates to a legal file description. The default extension is `.DAT`.



Do not try to KILL a file you currently have OPEN. This could cause corruption on the disk. If you try to KILL a file another program has OPEN, you see an error message.

If you try to delete a protected file, you see a `?Protection violation` error message.

You may also use KILL as a keyword—see Chapter 15.

11.8°RENAME

The RENAME function renames a file on the disk and returns a value according to the result of the operation. The format is:

```
RENAME( filespec ), new-name
```

For example:

```
RESULT = RENAME( "NEW.TXT" ), "OLD.TXT"
```

The **filespec** is any string expression which evaluates to a legal file description. The default extension is .DAT. The function returns a 1 if the file was found and renamed, or 0 if it was not found or if it was protected.



Do not try to RENAME a file you currently have OPEN. This could cause corruption on the disk.

You may also use RENAME as a keyword—see Chapter 9.

11.9°VER\$

The VER\$ function returns a string of the current version number of one of a choice of programs, etc. The format is:

```
[string-variable] = VER$(code)
```

code is a number representing the thing you want the version of, from these choices:

- | | |
|----|--|
| 0 | The version of the currently executing file |
| -1 | The version of RUNP.LIT that is executing this file |
| -2 | The version of AMOS in memory |
| -3 | The version of the compiler that compiled this file * |
| -4 | The date this program was compiled, in internal format * |
| -5 | The time when this program was compiled, in separated format * |

*: This information is available only if the file was compiled with the /R switch. If /R was not used, these values return a null string. Notice that the date and time are returned as string variables, not integers.

A positive number for the code represents the memory address of a PHDR of a program, and returns the version number of the program at that address. The version number is actually 2 bytes past the start of PHDR, but you need to specify PHDR for the address to be correctly returned. PHDR is set by the PROGRAM statement. The format of PHDR is described in Appendix A of the *AMOS Monitor Calls Manual*.

CHAPTER 12

FORMATTING OUTPUT

Most business applications programs spend a great deal of effort in generating reports and printouts in which data must be neatly and clearly presented. In other words, correctly formatting output is usually a major concern of the programmer. AlphaBASIC PLUS has several features that help you format data. This chapter discusses:

- How to use the USING modifier to format data
- The formatting characters you can use
- How to use tab functions to control the output of data to the terminal screen

12.1 THE USING MODIFIER

The USING modifier lets you format numeric or string data by specifying a format string (sometimes called an "editing mask"). Although you can use the USING modifier to store the formatted data in a string variable, you may also use it in combination with the PRINT statement to send the formatted data to a terminal display or to a file. For information on PRINT, see Chapter 9.

By "formatting" data, we mean the process of adjusting the appearance of data (that is, by inserting commas or spaces) so it fits the pattern of a specific format string. It might help to think of the format string as a template or pattern with which you are going to control the format of your data. The USING modifier allows you to apply the format string to your data. Using format strings and USING, you can do such things as:

- Line columns of numbers up by their decimal points
- Insert dollar signs and commas into numeric data to represent dollar amounts
- Line up numeric and string data within specified fields
- Generate and print leading zeros for numeric data
- Print asterisks instead of leading spaces

- Print numeric data in exponential form

and many other things. The sections below talk about the special formatting characters within the format string allowing you to make such adjustments. The statements in which you use the USING modifier take these forms:

```
variable = expression USING format-string

PRINT expression USING format-string

PRINT USING format-string, expression-list
```

where `expression` is usually a numeric or string constant, or a numeric or string variable. You can also use `?` instead of the word `PRINT`. As an example of `PRINT USING`, if you want to format the number 2345.678 with the format string "\$\$####.##", you could say:

```
NUMBER = 2345.678 USING "$$####.##"
```

or:

```
PRINT 2345.678 USING "$$####.##"
```

or:

```
PRINT USING "$$####.##", 2345.678
```

You may use the first and second formats only for numeric data; you may use the third format for string and numeric data. Also, remember `USING` has the lowest precedence of all operators. Therefore, all other operations in expressions surrounding the `USING` operator are performed before formatting is done. For example:

```
READY
PRINT 23 + 4 USING "###" + ".#" RETURN
27.0
```

The format string may be a string expression (for example, `MID$(A$,4,5)`), a string literal (for example, `"###.##"`), a string constant, or a string variable (for example, `MASK$`).

If you use the third `PRINT USING` variant above, you may supply a list of expressions to be formatted, separating the expressions with commas as with the regular `PRINT` statement (for example: `PRINT USING "#####.##",A,B,C,D,E`).

The mask can consist of multiple fields to be filled by multiple expressions. If you supply more expressions than the format string is meant to handle, AlphaBASIC PLUS re-uses the format string until each of the elements in the expression list has been formatted. If you supply fewer expressions than the format string is meant to handle, AlphaBASIC PLUS ignores the unused portion of the format string.

You may also send formatted data to a file by specifying a file-channel number after the PRINT keyword. For example:

```
PRINT #1, USING format-string, expression-list
```

For information on sending data to files, see Chapter 15.

12.2 HOW TO SPECIFY FORMAT STRINGS

The sections below discuss the special characters that make up a format string and control the output of your data. Characters other than these special formatting characters which appear in a format string are output literally as part of your data.

12.2.1 Formatting a Numeric Field (#)

The # symbol in a format string always indicates you want to format numeric data. Each # symbol in a format string represents one numeric digit. The simplest numeric format string consists of just # symbols. For example:

```
PRINT C USING "####"
```

The statement above tells AlphaBASIC PLUS to format the numeric variable C into a field of four digits, with no fractional part. If the format string causes AlphaBASIC PLUS to remove the fractional part of a number, AlphaBASIC PLUS rounds the number to the next integer, rather than truncating it. For example:

```
READY  
PRINT 2367.88 USING "####" RETURN  
2368
```

If the numeric field is too small to contain the specified number (for example, if we specify 650456.56 with the format string "####"), AlphaBASIC PLUS prints the number in standard format preceded by a % symbol, indicating overflow. For example:

```
READY  
PRINT 150450 USING "####" RETURN  
%150450
```

If the field is larger than the number, AlphaBASIC PLUS right justifies the number, inserting leading blanks into the digit positions not needed. For example:

```
READY  
PRINT USING "#####", 23 RETURN  
      23
```

Note that other formatting characters discussed below (for instance, the \$\$ and ** symbols) also define digit positions as well as perform special formatting functions.

You can't format string data with a numeric field format string. If you try, AlphaBASIC PLUS prints the format string, showing it was unable to format the data. For example:

```
READY
PRINT USING "#####", "Hi there" RETURN
#####
```

12.2.2^oFormatting a String Field (\)

You may specify fields for string data by using the backslash symbol (\). Two backslashes define a string field whose size equals the number of characters enclosed in the backslashes plus the backslashes themselves.

Although the usual practice is to enclose blanks in the string field (for example, "\oooo\"), AlphaBASIC PLUS permits the use of any characters. Since these characters are never printed, but simply define the size of the field by which a string is to be formatted, non-blank characters serve only as a comment.

However, when using several string fields within a single format string, it can be useful to visually separate them from the spaces between the fields by using non-blanks within the backslashes. For example:

```
"\---field1---\      \-----field2-----\  \-field3-\"
```

String fields allow you to define the placement and size of string data. For example:

```
READY
STRSIZ 20 RETURN
A$ = "Now is the time." RETURN
PRINT USING "As he once said, '\-----\'," A$ RETURN
As he once said, 'Now is the time.'
```

If the string to be formatted is larger than the string field, AlphaBASIC PLUS truncates the extra characters. If the string to be formatted is smaller than the string field, AlphaBASIC PLUS adds trailing blanks to the string to make it the same size as the field, and thus left justifies it in the field. You may combine string and numeric fields in a single format string. For example:

```
READY
STRSIZ 25 RETURN
MAP1 MASK,S,38,"\-10char-\ ####.## \---15 char---\" RETURN
C$ = "(in millions)" RETURN
PRINT USING MASK,"1979",34.556,C$,"1980",678.456,C$ RETURN
1979      34.56 (in millions)
1980      678.46 (in millions)
```



Remember the default string size is 10 characters, so explicitly define any strings over 10 characters by using MAP statements or by including a STRSIZ statement in your program to adjust the default string size.

12.2.3[∞]One-character String Fields (!)

The exclamation mark identifies a one-character string field. AlphaBASIC PLUS replaces the exclamation mark with a corresponding string. If the string constant or string variable contains more than one character, AlphaBASIC PLUS ignores any characters past the first. For example:

```
STRSIZ 40
MASK$ = "The temperature is:   ###!   =   ##!"
PRINT USING MASK$,50,"F",10,"C"
PRINT USING MASK$,68,"F",20,"C"
PRINT USING MASK$,86,"F",30,"C"
PRINT USING MASK$,104,"F",40,"C"
```

prints:

```
The temperature is:   50F   =   10C
The temperature is:   68F   =   20C
The temperature is:   86F   =   30C
The temperature is:  104F   =   40C
```

If no string is available to be substituted for the ! symbol, AlphaBASIC PLUS prints the ! symbol instead. For example, if we took our program above and removed the first "F" from the PRINT USING expression list, the first line of our display becomes:

```
The temperature is:   50!   =   10C
```

12.2.4[∞]Using Decimal Points in Numeric Fields (.)

You may include one period within a numeric field to specify where a decimal point is to appear in the formatted number. For example:

```
READY
PRINT USING "#####.##",2345.502,1100.657,200 RETURN
2345.50
1100.66
200.00
```

If the number specified contains more digits to the right of the decimal point than the format string, AlphaBASIC PLUS rounds the number so it contains the right number of digits in the fractional part.

If the format string has more digits to the right of the decimal point than the specified number, AlphaBASIC PLUS fills in the unused digit positions with zeros (as in the case of the number 200, above).

If the format string specifies any digits before the decimal point, AlphaBASIC PLUS prints at least one digit before the decimal point for each number, even if it is zero.

12.2.5^{oo}Dollar Signs and Numeric Fields (\$\$)

Two dollar signs (\$\$) at the front of a numeric field format string tell AlphaBASIC PLUS to put a dollar sign in front of the formatted number. \$\$ defines two digit positions, one of which is taken up by the dollar sign itself. For example:

```
READY
PRINT USING "$$#####.##",17500.66,100,345.2 (RETURN)
$17500.66
$100.00
$345.20
```

Notice the difference between using \$\$ and using the single non-formatting character "\$" in the format string:

```
READY
PRINT USING "$#####.##",17500.66,100,345.2 (RETURN)
$17500.66
$ 100.00
$ 345.20
```



Because you use \$\$ to format data representing money amounts, you may want to use the floating comma symbol in combination with \$. See the section below for information on this formatting character.

Remember you can include non-formatting characters in a format string. In the case above, a single dollar sign is **not** a formatting character, and so AlphaBASIC PLUS simply prints it as part of the formatted data. As another example:

```
READY
PRINT USING "###%",23.45,56.78,99.84 (RETURN)
23%
57%
100%
```

In the example above, the "%" symbol is NOT a special formatting character. Another example:

```
READY
PRINT USING "Telephone: (###) ### ####",714,555,1212 (RETURN)
Telephone: (714) 555 1212
```

12.2.6[∞]Putting a Comma Every Three Digits (,)

Include a comma to the left of the decimal point in your format string to tell AlphaBASIC PLUS to put a comma every three digits to the left of the decimal point. The comma defines one character position. For example:

```
READY
PRINT 6507501.89 USING "#####,.##" RETURN
6,507,501.89
```

12.2.7[∞]Fill Leading Blanks with Asterisks (**)

Include two asterisks at the front of your format string to tell AlphaBASIC PLUS to replace any leading blanks normally output in front of a number with asterisks. This is especially useful when printing checks. ** defines two digit positions. For example:

```
READY
PRINT 231.69 USING "***#####.##" RETURN
*****231.69
```

You may want to use asterisk-fill formatting when printing dollar amounts; remember you may include a \$\$ in the format string. For example:

```
READY
PRINT 231.69 USING "***$#####.##" RETURN
*****$231.69
```

12.2.8[∞]Fill Leading Blanks with Zeros (Z)

To generate leading zeros, include the Z symbol within your format string. The format string must begin with one # symbol followed by a series of Zs. The total size of the formatted string is the number of Zs plus the one # symbol. For example:

```
READY
PRINT 123 USING "#ZZZZZ" RETURN
000123
```

12.2.9[∞]Add a Trailing Minus Sign to a Number (-)

You may cause the sign of a number to be printed following the number by ending a numeric field in a format string with a minus sign. If the number is positive, AlphaBASIC PLUS prints a blank after the number; if it is negative, AlphaBASIC PLUS prints a minus sign after the number. For example:

```

READY
MAP1 MASK,S,26,"--7--\          $$#####.##-" (RETURN)
C$ = "Credit:" (RETURN)
D$ = "Debit:" (RETURN)
PRINT USING MASK,C$,35.67,D$,-57.89,C$,10.89,D$,-356.33 (RETURN)
Credit:          $35.67
Debit:           $57.89-
Credit:          $10.89
Debit:           $356.33-

```

12.2.10[∞]Printing Numbers as Exponents (^^^)

You may specify exponential format by following the numeric field in a format string with four carets (^^^). These define the spaces taken up by the "E nn" exponent characters. AlphaBASIC PLUS left justifies the significant digits, adjusting the exponent as necessary. As with other numeric formats, AlphaBASIC PLUS allows any decimal point arrangement. For example:

```

READY
PRINT USING ".#####^^^^",100,2345.66,5000,.0004 (RETURN)
.10000E+03
.23457E+04
.50000E+04
.40000E-03

```

12.3[∞]FORMATTING EXAMPLES AND HINTS

All of our examples above used the PRINT statement to print formatted data. Remember you may also format a numeric value without displaying it by using the USING modifier without the PRINT statement. For example:

```
A$ = B USING C$
```

The statement above formats the number in B using the format string in C\$, and leaves a string result in A\$.



This format of the USING modifier is **only** for formatting numeric data. Also note even though we are formatting numeric data, the result is always a string.

This type of format allows you to create headings and image lines you use more than once, and to inspect and manipulate formatted data before printing it.

You may not use the USING modifier recursively. That is, you may not use a format string that is itself the result of a USING modifier. For example, if you have specified:

```
C$ = "###.##"
```

you may not say:


```
PRINT C$ USING "#####.##"
```

When using the PRINT USING format, remember PRINT USING differs from the regular PRINT statement in that the use of semicolons to separate the elements of the print list has no effect on the spacing of those formatted elements.

12.4°TAB FUNCTIONS

The TAB function can be used to control various functions of your terminal. The TAB function is only used in a PRINT statement. It operates in the traditional manner when supplied with only a single numeric argument such as TAB(X). In this case the function causes the cursor to be positioned on the "X + 1" column on the current line. For instance, TAB(5) causes 5 spaces to be printed, and the following characters begin in column six.

When supplied with two arguments such as TAB(R,C), however, the TAB function performs special CRT operations.

If the value of R is positive, the R,C arguments are treated as row and column coordinates for positioning the cursor on the terminal screen. The specified characters are then printed beginning in that position. As in other functions, the R and C arguments may be expressions. Terminals are assumed to begin with row 1 (top of screen) and column 1 (left end of each row).

If the value of R is -1, the function is interpreted as a special terminal command and the C argument then specifies the code of the screen function you wish to use. The codes are transmitted to the terminal driver (.TDV file in DSK0:[1,6]), which does the actual interpretation and performs the special function for your terminal.

You may be able (depending on the capabilities of your terminal) to do such things as display in reverse video, draw lines on the screen, display special characters, etc.

If you have a color terminal, you may select colors by setting R to -2 for a foreground color, and to -3 for a background color. The colors are:

| | | | |
|---|---------|---|--------|
| 0 | Black | 4 | Red |
| 1 | White | 5 | Yellow |
| 2 | Blue | 6 | Green |
| 3 | Magenta | 7 | Cyan |

For information on using your terminal's display features from within a program, see your *AMOS Terminal System Programmer's Manual*. See Appendix D for the standard decimal codes in use for the terminal drivers supplied by Alpha Micro.

CHAPTER 13

SCALED ARITHMETIC

Whenever a mathematical operation with long decimal numbers is performed, the number of significant digits used becomes important. AlphaBASIC PLUS uses a floating point format which gives an accuracy of 11 significant digits (or 15, if using IEEE format). Unfortunately, this accuracy is absolute only when dealing with numbers that are total integers (with no numbers to the right of the decimal point). This is because of the conversions required from ASCII decimal input to the floating point format used to do arithmetic.

For most business people, the range of numbers usually important are those in "money" format—with two digits to the right of the decimal and up to nine digits to the left of the decimal point. When the fractional part of the number is converted between decimal and floating point formats, a small but significant error is sometimes introduced which may show up when you want to see accurate dollars-and-cents values. As an example, in the following statements:

```
SIGNIFICANCE 11  
PRINT 26.4 - INT(26.4)
```

Instead of the expected answer of .4, we see the answer:

```
.39999999999
```

This is NOT an error in AlphaBASIC PLUS, but simply represents the side effects of converting a decimal fraction to floating point representation and back again. Some decimal fractions cannot be exactly expressed as a floating point fraction in a finite number of digits, and so a round-off error occurs.

The error is only visible because our program set the number of significant digits to 11 (the usual number of significant digits is six). Such errors can accumulate and present themselves when you do a large number of multiplications and divisions using decimal fractions.

AlphaBASIC PLUS has a scaling feature which helps this problem by storing all floating point numbers with a scale offset. This offset tells AlphaBASIC PLUS where the 11 absolute accuracy digits are located in relation to the decimal point. AlphaBASIC PLUS does this by multiplying every input number by the scaling factor and then dividing it out again before printing.

This is a simplified explanation, and many other checks and conversions are done internally to scaled numbers. If you are interested in more detail on scaling numbers, see the section at the end of this chapter.

13.1 THE SCALE STATEMENT

Scaled arithmetic is normally entered at the start of a program and continues in effect throughout the program. The statement for setting the program into scaled mode is:

```
SCALE factor
```

`factor` must be an integer digit in the range of -35 to 35. It may not be a variable, since scaling is done at compile time for constant values as well as at run-time for input and output conversions.

Negative scaling moves the 11-digit window to the left. A negative scaling factor takes care of those cases where your numbers are too LARGE, rather than too small.

A few words of caution are in order here. Once AlphaBASIC PLUS detects the SCALE statement during compilation, AlphaBASIC PLUS scales all subsequent constant values by the scaling factor so they are stored properly. If you execute a computation between numbers with different scale factors, you get incorrect results. For this reason, it is a good idea to use the SCALE statement at the beginning of your program, before any variables are defined.

In addition, a run-time command is generated in the executable program which causes the actual scaling to be performed on INPUT and PRINT values when the program is running.



If two or more different SCALE statements are executed in the same program, some very strange results may come out unless you are completely familiar with the compile-time and run-time conversion processes.

Let's add a SCALE argument to our previous program:

```
SCALE 2
SIGNIFICANCE 11
PRINT 26.4 - INT(26.4)
```

now it gives us the proper answer— .4.

If you are using a positive scaling factor to adjust real numbers, note SCALE does nothing to prevent inaccuracies if the scale factor is not large enough to cause AlphaBASIC PLUS to handle your data as integers.

For example, if you want to handle numbers with three digits to the right of the decimal point, a scaling factor of 2 leaves one digit to the right of the decimal point, and a scaling error can still occur. So, if you are using numbers with a fractional part of two digits, use a scaling factor of 2; if the fractional part is three digits, use 3; etc.



Floating point numbers stored in files by the sequential output PRINT statement are unscaled and output in ASCII without problems. Floating point numbers written to random access files by using WRITE are not unscaled first; any program reading this file as input must either be operating in the same scaling mode in which the data was written, or else must apply the scale factor explicitly to all values from the file. Integer and string values are never modified, regardless of the scaling factor currently in use.

13.2[∞]HOW THE SCALING FACTOR WORKS



All of the conversions we discuss in this section are done automatically by AlphaBASIC PLUS. This section merely talks about the background theory of scaling numbers.

The scaling factor represents the number of decimal places the 11- or 15- digit "window" is effectively shifted to the right in any floating point number. For example, the most common application is in a business environment where the scaling factor of 2 is used to give absolute 11 or 15 place accuracy to numbers which extend 2 places to the right of the decimal point.

This means the value of 50.12 is multiplied by the scaling factor of 2 digits (100) and stored as the floating point value of 5012. Since this value is an integer, it has absolute accuracy. Just before printing, AlphaBASIC PLUS divides this number by the scaling factor to reduce it to its intended value of 50.12.

Other conversions have been included into the system to handle the subtle effects of storing scaled numbers. For example, when converting scaled numbers to integer or floating point format, AlphaBASIC PLUS must unscale the number first before converting it.

When AlphaBASIC PLUS multiplies two scaled numbers together, the result is a number which must be unscaled once, while division of two scaled numbers creates exactly the opposite problem. Dealing with scaled numbers for exponential, logarithmic and trigonometric functions creates even more exotic problems—but AlphaBASIC PLUS handles them all for you.

CHAPTER 14

MAPPING VARIABLES

This chapter discusses the following subjects:

- What variable mapping is
- The format of MAP statements
- How to use MAP statements
- How memory is allocated by MAP statements
- Locating variables during debugging

14.1 WHAT IS VARIABLE MAPPING?

Mapping variables is a method of grouping variables together. The ability to group variables together is useful when those variables contain related information. Variable mapping is a powerful tool similar to COBOL data description techniques, Pascal record definitions and C structures.

Why should you use mapped variables? Well, mapped variables offer the following advantages:

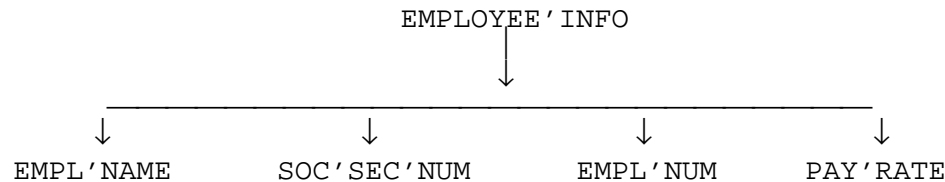
- They can save memory
- They make variables easier to refer to
- They can make it easier to access data in random data files
- They can improve the logic and readability of your program. If all variables are mapped at the start of the program, it is easy to see what variables are used and how they are related. Defining all variables used is good programming practice.
- They let you define binary variables (which can save memory when using flags)

Let's take a look at an example to show you how mapped variables can help you. Say you want to write a program that keeps a record for each of your employees. You want to know the employee's name, social security number, employee number, and rate of pay. In most cases where you are dealing with the variable containing the employee's name, you also want to work with one or more of the other variables too.

Using regular AlphaBASIC PLUS variables, you might have EMPL'NAME\$, SOC'SEC'NUM\$, EMPL'NUM, and PAY'RATE\$. If you want to read one employee's records in from a file, you need a statement like this:

```
READ #1, EMPL'NAME$, SOC'SEC'NUM$, EMPL'NUM, PAY'RATE$
```

If you use variable mapping, however, you could set up the variables so they all "connect" to a single variable (let's call it EMPLOYEE'INFO). Here is a visual representation of this idea:



Now, when you refer to the variable EMPLOYEE'INFO (for instance, if you say READ #1 EMPLOYEE'INFO), you access the other four variables so the total information about the employee is brought in from the data file.

You may have noticed none of the variable names in the above diagram have a \$ after it, which is AlphaBASIC PLUS's usual way of denoting string variables. This is because you specify what type of variable it is when you MAP a variable. AlphaBASIC PLUS knows EMPL'NAME is a string, and knows what maximum size it can be since you also specify that in setting up the variable.

You can form a variable "structure" like the one above using different types of variables also. In the above example, EMPL'NAME should be a string variable, but EMPL'NUM might be a numeric variable. The variables can also be different sizes. You can refer to a single element of the group or to the group as a whole.

14.2[∞]THE FORMAT OF MAP STATEMENTS

Now you have an idea what mapped variables are like, let us take a look at the statement that sets up mapped variables. The MAP statement has the form:

```
MAPn variable{(dim)}, {type}, {size}, {value}, {origin}
```

where *n* is a positive number defining the level of the MAP statement. The elements after the variable name are optional, depending on the kind of variable you are defining. For example, if you are defining an array variable, you include the optional number specifying the dimension in the MAP statement. We'll discuss each of these optional elements in detail in the following sections.

If you "skip" an element in the MAP statement (for example, you want to specify the "value" but not the "size"), you must keep the comma showing where the missing "size" should be. For example:

```
MAP1 NEW'VARIABLE,F,,23
```

The MAP statement may not end with a comma. If you don't specify any of the optional elements, do not use any commas.

The MAP statement above defines NEW'VARIABLE, assigns it the data type F (for floating point), does NOT specify a size (which becomes the default size—6 if an AMOS variable, 8 if the program is compiled with /I, selecting IEEE variables), and assigns it the initial value of 23. Without the extra comma, BASIC thinks you were trying to assign a size of 23 bytes to NEW'VARIABLE—an illegal operation for a floating point variable.

14.2.1[∞]MAP Level

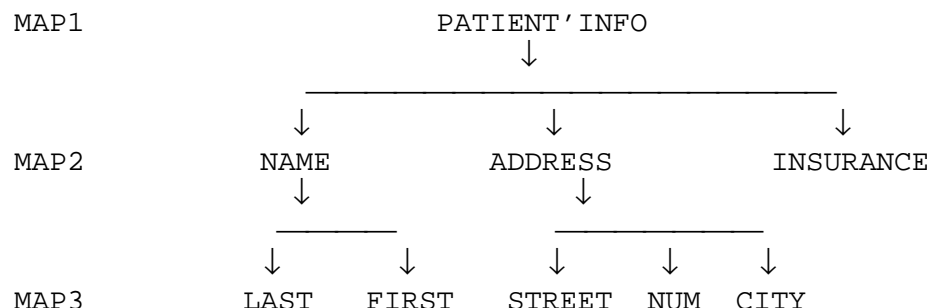
The number following the MAP statement (MAPn) represents the level of the mapped variable. It must be within the range of MAP1 through MAP16. MAP statements are hierarchical in nature. For example, a variable mapped with a MAP1 statement may consist of several sub-variables mapped by using MAP2 statements. Each of those variables may in turn consist of several variables mapped by using MAP3 statements. And so on, down to MAP16.



Nested numeric MAP statements cause an error. You cannot specify a numeric variable (such as a floating point), and then have another numeric variable defined below it with a lower MAP level. You can only nest a numeric variable within string or unformatted type variables.

MAP16 represents the lowest-level (or innermost) variable; MAP1 represents the highest level variable. You do not need to map levels in strict numeric sequence—for example, a MAP5 statement may follow a MAP3 statement without an intervening MAP4 statement.

You may refer to variables at any level. An example may help to clarify this idea:



The diagram above shows three levels of variables mapped with MAP1, MAP2, and MAP3 statements. You may refer to the level 1 variable PATIENT'INFO as a whole, or may refer to one of the variables on levels 2 and 3 representing sub-groups of the variable PATIENT'INFO, such as NAME, ADDRESS, or STREET.

When referring to any unformatted variable in the group, you are actually referring to the information in all of the variables below it in the hierarchy.

For example, when you refer to NAME you get the information in the variables LAST and FIRST. As AlphaBASIC PLUS allocates the variables NAME and ADDRESS, it automatically includes them (and their sub-variables) within the variable PATIENT'INFO.

The MAP statements for the variable group above might be:

```
MAP1  PATIENT'INFO
      MAP2  NAME                ! Patient's name
            MAP3  FIRST,S,15
            MAP3  LAST,S,20
      MAP2  ADDRESS              ! Patient address
            MAP3  STREET,S,30
            MAP3  NUM,S,10
            MAP3  CITY,S,30
      MAP2  INSURANCE,B,1       ! Flag if patient has insurance
```

14.2.2^oVariable Names in MAP Statements

The variable name is the name your program uses to refer to the mapped variable; it must follow the rules for AlphaBASIC PLUS variable names described in Chapter 6. However, since you may specify the type, you do not need to follow a string variable with a dollar sign, or a four-byte true integer with a percent sign.

If the variable name is followed by a set of subscripts within parentheses, the variable is assigned as an array with the dimensions specified by the subscripts, just as if a DIM statement had been used. For example, the statement:

```
MAP1  NUMBER,F,6
```

assigns a single floating point variable called "NUMBER," but the statement:

```
MAP1  NUMBER(5,10),F,6
```

assigns a floating point array with 50 elements in it (5 x 10), just as if the statement DIM NUMBER(5,10) had been executed.



Since these mapped arrays are assigned memory at compile time and not at run-time, the subscripts must be decimal numbers or constants instead of variables.

14.2.3[∞]Type

The type is a single character code which specifies the type of variable mapped:

| | |
|---|---|
| B | binary unsigned numeric variable |
| F | floating point variable |
| I | integer variable (true binary signed numeric) |
| S | string variable |
| X | unformatted absolute data variable |

If no type code is entered, AlphaBASIC PLUS assumes an unformatted variable. Variable types are described in Chapter 6.

14.2.3.1[∞]Unformatted Data

Unformatted variables are used to "group" together any variables mapped below them. It might help to think of an unformatted variable as the "file folder" holding all the information stored in the variables below it. You usually define an unformatted variable so you can refer to a group of other variables as one unit.

The contents of unformatted data variables should only be moved to other unformatted data variables.

In the example above, the variables PATIENT'INFO, NAME, and ADDRESS are unformatted variables (since nothing is specified, they default to unformatted). An unformatted variable is like a storage area for all the variables mapped below it (in the case of NAME, the variables LAST and FIRST). This allows these "groups" of information to be moved as a unit, simply by using the unformatted variable name. For example:

```
READ #1, PATIENT'INFO
WRITE #3, NAME
HOLD'IT = PATIENT'INFO
```

■ Unformatted data at the MAP1 level can have a maximum size of 65,535 bytes.

14.2.3.2[∞]String Data

If the string contained within a string variable is less than the allocated size, it is terminated by a null character. If you move a string into a shorter variable, any characters in excess of the length of the new variable are truncated.

■ String variables default to a length of 0, and can have a maximum size of 65,535 bytes.

14.2.3.3 Floating Point Data

Floating point numbers default to the AMOS standard of 6 bytes long. You may also specify either 4 or 8 bytes, which are IEEE format two-word and four-word floating point variables. Any other specification is illegal.

14.2.3.4 Binary Data

Binary variables may range in size from 1 to 5 bytes. Binary data is handy for the storage of small integers, or bit flags. All numbers are stored as two's-complement values.

Since AlphaBASIC PLUS converts all binary variables to floating point format before performing any arithmetic calculations, binary arithmetic is actually slower than normal floating point arithmetic.

Please note the use of binary numeric variables is not allowed in some instances. FOR-NEXT loops may not use a binary variable as the control variable, although they may be used in the expressions designating the initial and terminating values of the control variable, as well as in the STEP expression.

14.2.3.5 Integer Data

AlphaBASIC PLUS supports true integers of one, two, or four bytes. The default size is 4. These integers are stored in binary form. Integers offer increased speed of calculation over floating point variables.

14.2.4 Size

The size parameter in the MAP statement is optional but, if it is used, it must be a decimal number specifying the number of bytes to be used in the variable. If it is omitted, it defaults to 0 for unformatted and string types, 6 for floating point types, 4 for integer types, and 2 for binary types. The size parameter of floating point variables **MUST** be 4, 6, or 8. Integer variables must be 4, 2, or 1. Binary variables must be 1, 2, 3, 4, or 5. String and unformatted variables must be less than 65,535 bytes in size.

14.2.5 Value

An initial value may be given to any mapped variable (except an array variable) by including any valid expression in the value parameter. This value may be a numeric constant, a string literal or constant, or a complete expression including variables.

Remember, however, the expression is resolved when the MAP statement is executed at run-time, and the current value of any variable within the value expression is the one used to calculate the assignment result. You may reload the initial value by going back to the MAP statement and executing it again (remember you can't define the same variable twice in the same program).

Note if you omit the size parameter (such as for floating point variables), but you use the value parameter, there must be an extra comma to indicate the missing size parameter:

```
MAP1  PI,F,,3.1415926535
MAP1  HOLIDAY,S,, "CHRISTMAS"
```

The first example preloads the value 3.1415926535 into the floating point variable called PI. The second example preloads the letters CHRISTMAS into the string variable called HOLIDAY.

14.2.6[∞]Origin

In some instances, it may be desirable to redefine records or array areas of different formats so they occupy the same memory area. For instance, a file may contain several different record formats with the first byte of the record containing a type code for that record format.

The origin parameter allows you to redefine the record area in the different formats to be expected. When the record is read into the area, the type code in the first byte can be used to execute the proper routine for the record type.

Each different routine can access the record in a different format by the different variable names in that format. All record formats actually occupy the same area in memory. This feature is much like the REDEFINES verb in the COBOL language data division. Using the origin parameter can save large amounts of memory.

For instance, suppose you have three very large variables of 256 bytes each that define logical records, and you never use these variables at the same time. By defining the variables so they occupy the same area of memory, your program only uses 256 bytes for the variables instead of 768 bytes.



Origin overlays may not work properly if your mapped variable is an array. We recommend you don't overlay array variables.

Normally, a MAP statement causes allocation of memory to begin at the point where the last variable with the same level number left off. The origin parameter allows this to be modified so allocation begins back at the base of some previously defined variable, and therefore overlays the same memory area. If the new variable is smaller than the previous one (or the exact same size), the value of the new one is totally contained in the previous variable.

If it is larger than the previous one, it spills over into newly allocated memory or possibly into another variable area of the same level depending on whether there are more variables following.

The origin parameter must be the last parameter on the line. It takes this form: an @ symbol followed by the name of the previously mapped variable whose area you wish to overlay. This variable must be on the same level as the variable you are presently allocating. If size and value parameters are not included in this statement, you may omit them with no dummy commas. For example:

```

MAP1  CUSTOMER' ID
      MAP2  NAME, S, 13
      MAP2  ID' NUM, F
      MAP2  SEX, B, 1
MAP1  PRODUCT' INVENTORY, @CUSTOMER' ID
      MAP2  BRAND, S, 13
      MAP2  PARTNO, F, 6
      MAP2  RESALE, B, 1

```

The MAP statements above allocate the variable CUSTOMER'ID which uses a total of 20 bytes. Then it allocates the variable PRODUCT'INVENTORY (also using 20 bytes), and specifies, by using the @CUSTOMER'ID origin parameter, PRODUCT'INVENTORY occupies the same space in memory as CUSTOMER'ID.

The following statements define three areas which all occupy the same memory area, but which may be referred to in three different ways:

```

100  MAP1  INVENTORY
110      MAP2  INDEX, S, 28
120      MAP2  ITEM, S, 30
200  MAP1  ADDRESS, @INVENTORY
210      MAP2  STREET, S, 24
220      MAP2  CITY, S, 14
230      MAP2  STATE, S, 4
300  MAP1  PRICE' CODE, @INVENTORY
310      MAP2  UNIT
320      MAP3  CODE, B, 2
330      MAP3  PRICE, F, 6

```

Statements 100 to 120 define a map level with two string elements: a total of 58 bytes in memory. Statements 200-230 define an area with three string variables, for a total of 42 bytes. Normally, this area follows the INVENTORY area in memory, but the origin parameter in statement 200 causes it to overlay the first 42 bytes of the INVENTORY area instead.

Statements 300-330 define another map level of a different format: one 2-byte binary variable (CODE) and one floating point variable (PRICE). The origin parameter in statement 300 also causes this area to overlay the INVENTORY area exactly.



Overlaying allows variables to be referred to in a different format than when they were entered into memory. Make sure your program is reading the correct variables at the correct time.

14.3[∞]USING MAP STATEMENTS

MAP statements can be used as direct statements in interactive mode as a learning tool to see how variable mapping works. Interactive mode also gives you immediate feedback if you enter the MAP statement incorrectly, making it easier to learn the MAP syntax.

MAP statements are not designed to be practical in interactive mode, however, and are best used by putting them into a program file and compiling the program.

MAP statements should come at the beginning of the program, and must come before any references to the variables being mapped. If you refer to a variable before it is mapped (such as `A = 5.8`), the variable is set up according to AlphaBASIC PLUS defaults. When the MAP statement is reached, it then returns an error, since the variable is already defined.

Remember the COMPLP program offers a `/M` switch that reports unmapped variables in your program—useful if you want all your variables to be mapped. It is also helpful in preventing misspellings of variable names, which can be a hard-to-find program error.

14.3.1[∞]Examples

Let us take a mapped variable group:

```
MAP1  ADDRESS
      MAP2  STREET,S,20
      MAP2  CITY,S,20
      MAP2  STATE,S,20
      MAP2  ZIP'CODE,S,9
```

With this group, we can do a number of things. We could print the whole group to the terminal screen:

```
PRINT ADDRESS
```

or write it to a file with a simple statement:

```
WRITE #1,ADDRESS
```

And we could sort the information in useful ways by sorting the CITY, STATE, or ZIP'CODE variables.

Another use of MAP statements is to set up arrays. The following two statements produce identical arrays:

```
MAP1  ARRAY(10),F,6
DIM  ARRAY(10)
```

Both statements produce arrays containing ten floating point variables, referred to as ARRAY(1) through ARRAY(10). The first statement, however, defines its placement in memory in relation to other mapped variables. Similarly, the statements:

```
MAP1  ARRAY(5)
MAP2  AR'VAR(20),F,6
```

produce the same two-dimensional array as:

```
DIM  ARRAY(5,20)
```

As another example, these statements:

```
DIM  WIDTH(10)
DIM  HEIGHT(10)
```

produce two arrays, each with ten variables. But these:

```
MAP1  AREA(10)
MAP2  WIDTH,F,6
MAP2  HEIGHT,F,6
```

produce one array with twenty variables in it. The variables are still referred to as WIDTH(1) through WIDTH(10) and HEIGHT(1) through HEIGHT(10), but their placement in memory is quite different. The WIDTH variables are interlaced with the HEIGHT variables, giving WIDTH(1), HEIGHT(1), WIDTH(2), HEIGHT(2)... WIDTH(10), HEIGHT(10).

There are also ten unformatted variables AREA(1) through AREA(10), each containing the respective pairs of WIDTH-HEIGHT variables in tandem. Referencing one AREA variable references the unformatted item composed of the WIDTH-HEIGHT pair of the same subscript.

When using MAP variables, keep these points in mind:

- Use an unformatted variable to group multiple mapped variables together.
- If printing a string or unformatted variable, any floating point or binary variable mapped beneath it doesn't display properly.
- When using string variables, don't assign a size much larger than you need for the variable (this is a waste of memory space).

- Use only the variables you need. For example, if you want an address, do not break down the address into partial strings unless you need to refer to that part of the address. The more variables you define, the more memory you use.

14.4 USING MAPPED DATA WITH DATA FILES

You may often use MAP statements to define groups of information to be transferred in and out of random files. Because of the structure of sequential files, it is difficult to input and output mapped variables as "groups." The information only transfers correctly if all of the variable information is carefully "padded" to the proper lengths. Therefore, it is best to use random files when your program uses grouped mapped variables.

Let us take a look at an example defining a logical record. Our program probably uses a file containing a large number of logical records in this format, each record containing information about a single check.

In effect, MAP statements give us a way to form a template in memory into which you can read information from the random file and transfer information from the program to the file. This allows you to quickly and efficiently read in an entire group of information whose elements may be of different types and sizes, and to access information in that group simply. For example:

```
MAP1  CHECK' INFO
      MAP2  CHECK' NUMBER, F, 6
      MAP2  DATE, S, 6
      MAP2  AMOUNT, F, 6
      MAP2  TAX' DEDUCTIBLE, B, 1
      MAP2  PAYEE, S, 20
      MAP2  CATEGORY, S, 20
      MAP2  BANK' ACCOUNTS
            MAP3  SAVINGS, S, 20
            MAP3  CHECKING, S, 20
            MAP3  TERM, S, 20
      ! Define a file containing an account balance:
      MAP2  ACCOUNT' BALANCE, S, 22, "DSK1: BALANC.DAT[ 200, 1 ]"
```

Once these MAP statements have been executed, we can access the group of variables as a whole by specifying CHECK'INFO, or we can access specific sub-fields in the record (for example, BANK'ACCOUNTS or CHECKING). For example, once the program has filled the variables with the specific data for a record, we can write it to the file by saying:

```
WRITE #1, CHECK' INFO
```

Another section of the program might want to read in a record of data. By using a simple statement:

```
READ #1, CHECK' INFO
```

we can bring in all the information related to that check. This is much easier than using 11 statements to read in the 11 variables contained in CHECK'INFO! We can also use the individual items in AlphaBASIC PLUS program statements. For example:

```
IF AMOUNT > 100 GOTO VERIFY'TRANSACTION
```

14.5 HOW VARIABLES ARE ALLOCATED IN MEMORY



This section goes into the details of how memory is mapped. If you are not interested in this, you may want to skip to the next chapter.

During compilation, AlphaBASIC PLUS allocates memory storage for all defined variables in a contiguous and predictable area. The compiled program refers to all variables by an indexing scheme.

Each variable in the working storage area has a representative item in the index area which contains all the information needed to define and locate that variable. The working storage area therefore contains only the pure variables themselves without any associated or intervening descriptive information. The index area is a separate entity, physically located before the working storage area in memory.

AlphaBASIC PLUS normally allocates variable storage as it encounters each variable during compilation. A different method must be derived which can override normal allocation processes if you wish to have the variables allocated in a predetermined manner. Also, the disk I/O system requires variables used be in a specific relationship to each other when used in some of the more sophisticated programs.

The MAP statement has been included in AlphaBASIC PLUS for the purpose of allocating variables in a specific manner. MAP statements are non-executable at run-time, but merely direct the compiler in the definition and allocation of the referenced variables.

Each MAP statement contains a unique variable name to which the statement applies. When the compiler encounters this statement, it allocates the next contiguous space in working storage as required and assigns it to that variable. All variables not defined in a MAP statement are then automatically assigned storage in sequence, for total compatibility with existing standards.

To eliminate potential allocation problems, AlphaBASIC PLUS forces all MAP1 level variables to begin on an even memory address. This ensures certain binary and floating point variables begin on word boundaries for assembly language subroutine processing. The instruction set performs most efficiently when word data is aligned on word boundaries.

In interactive mode, if an error occurs in the syntax of the statement, the variable has already been added to the tree in memory. This is the reason MAP statements are not useful in interactive mode (except for learning and debugging).

14.6[∞]LOCATING VARIABLES DURING DEBUGGING

A command has been implemented which can assist you in locating the mapped variables and in understanding the allocation techniques used by the AlphaBASIC PLUS variable mapping system. The command has the general format of a commercial atsign (@) followed by a variable name.

The system searches for the requested variable and prints out all parameters about the variable for you on the terminal. This may actually be two definitions, since the variable "A" may actually be two different variables; one could be a single floating point number and the other could be a subscripted array.

The information returned about the variable is:

- [∞]The type of variable (string, binary, etc.)
- [∞]The dimensions of the array if the variable is an array
- [∞]The size of the variable in bytes
- [∞]The offset to the variable from the base of the memory area which is used to allocate all variables

If you enter a reserved word (such as @PRINT) the system tells you the name is a reserved word. If you enter an undefined variable, AlphaBASIC PLUS informs you.

The general format of the definition line returned by the system is:

```
memory-type var-type {dimensions},size n,located at x
```

For actual examples of the definition line, see the Examples section below. `memory-type` is the method of memory allocation used when defining the variable. `memory-type` may be MAPn (where n is a number from 1 to 16), FIXED or DYNAMIC. FIXED variables are not defined by a MAP statement and are allocated automatically when the compiler finds references to them in the program.

DYNAMIC variable arrays are allocated by a DIM statement or by a default reference to a subscripted variable.

If the array is dynamic and has not been allocated yet, the subscript values are replaced by the letter "X" to indicate they are not known at this point. Remember any variable defined in a MAP statement which is in a lower level relative to another variable inherits all subscripts from that higher level variable.

The size of the variable is given in decimal bytes. In the case of arrays, the size represents the size of each single element within the array.

The location of the variable is a little tricky to explain, since it is actually an offset to the base of a storage area set aside for the allocation of user variables.

As each new variable or array is allocated, it is assigned a location which is relative to the base of this storage area. The location information given here is an example to help you understand the relative placement of the variables in the mapping system, and does not represent the actual memory locations which they occupy.

There are two distinct areas in use for variables, and thus the offsets of the variables are to one of these two areas. All FIXED and MAPped variables are allocated in the fixed storage area, while all DYNAMIC arrays are allocated in the array storage area.

As dynamic arrays are dimensioned, their positions may shift relative to one another and relative to the dynamic storage area base. Variables in the fixed storage area never change position relative to each other or to the storage area base.

Array location information given is only pertinent to the base of the array itself, which is the location of the first element within the array. The actual range of locations used by the array may or may not be contiguous in memory depending on whether overlapped dimensioning techniques are being used in the MAP statements. Simple (non-array) variables are defined as a location range which tells exactly where the entire variable lies within the storage area.

Keep in mind this "@" command is to assist you in following the allocation of variables, particularly in more complex mapping schemes. A few minutes at the terminal with direct MAP statements followed by "@" commands can help you see how the mapping scheme works.

14.6.1 Examples

Given the sample MAP statements below:

```
MAP1  CUSTOMER' ID
MAP2  NAME
      MAP3  FIRST,S,15
      MAP3  LAST,S,15
MAP2  ADDRESS
      MAP3  STREET,S,15
      MAP3  CITY,S,10
      MAP3  STATE,S,2
MAP2  PHONE
      MAP3  HOME,B,3
      MAP3  BUSINESS,B,3
MAP2  TRANSACTIONS(12)
      MAP3  BALANCE,F,6
      MAP3  CREDIT,F,6
      MAP3  YTD,F
```

Here are the results of using the @ command in interactive mode to determine the locations of several of the variables above:

```
READY
```

@CUSTOMER' ID (RETURN)

MAP1 Unformatted, size 279, located at 0-278

@TRANSACTIONS (RETURN)

MAP2 Unformatted Array (12), size 18, base located at 63

@CITY (RETURN)

MAP3 String, size 10, located at 45-54

@HOME (RETURN)

MAP3 Binary, size 3, located at 57-59

We can also use the @ command to locate unmapped variables. For example:

READY

DIM A(2,3) (RETURN)

@A (RETURN)

Dynamic AMOS Floating point Array (2,3), size 6

A = 15 (RETURN)

@A (RETURN)

Fixed AMOS Floating point, size 6, located at 72-77

Dynamic AMOS Floating point Array (2,3), size 6

Note we allocated two different variables: a fixed floating point variable, A, and a dynamic floating point Array variable, A(2,3).

CHAPTER 15

THE FILE INPUT/OUTPUT SYSTEM

This chapter contains information on creating and using disk files from within your AlphaBASIC PLUS program. The following subjects are discussed:

- How to identify files
- Input/output statements
- Sequential files
- Random files
- How to lock files
- File statements

15.1 WHAT IS A DISK FILE?

A disk file is a group of data items stored on a hard disk device. AlphaBASIC PLUS programs are usually stored on the disk as a disk file. But AlphaBASIC PLUS can use other disk files to access and store data. In many applications, using disk files is much more efficient and versatile than using user input or DATA statements to obtain data. And it is the only efficient way to store data for later reference.

AlphaBASIC PLUS supports two kinds of data files—sequential access and random access disk files. You may write data either in ASCII characters, or in packed binary formats.

Files created by AlphaBASIC PLUS programs are compatible with all other system programs, and AlphaBASIC PLUS files may be interchanged with files from other languages. That is, AlphaBASIC PLUS data files can be read and manipulated by programs written in other languages. Conversely, files created by other languages and system utilities may be read and manipulated by programs written in AlphaBASIC PLUS.

Since the I/O processes differ somewhat between sequential and random data files, we discuss sequential and random files at a general level before getting into the specific commands you can use to work with these files.

15.2 FILE CHANNELS

All references to a file are made by specifying a file channel number, which may be any variable or literal that evaluates to a positive integer value from 1 to 65,535. You might think of the file channel number as a pipeline between your AlphaBASIC PLUS program and the data file, through which data can be transferred.

File channel #0 is defined as your terminal. You can use this in sequential file statements to write to your terminal at run-time if you wish. Random files cannot use file channel zero.

For example, when you first create a file, you assign it a file channel number (say, #1). Then, whenever you refer to that file, you use that number to tell AlphaBASIC PLUS which file you are talking about (say, READ #1). This allows you to have many files ready for use at the same time.

Once you close a file, the file channel is no longer associated with it, and you may open another file on that file channel. You may never have two files open at the same time having the same file channel number. The file channel always follows the verb (such as OPEN) in any file input/output statement, and may be any numeric expression which is preceded by a pound sign (#). For example:

```
OPEN #1, "PAYROL.DAT", INPUT
```

or:

```
OPEN #file'channel'num, "PAYROL.DAT", INPUT
```

or:

```
OPEN #(number'needed - num'in'stock), "INVEN.DAT", OUTPUT
```

Up to 256 files may be open at one time in a program.

15.3 INPUT/OUTPUT STATEMENTS

The following table shows the AlphaBASIC PLUS program statements you use to input and output data to and from files:

| ACTION | RANDOM | SEQUENTIAL |
|----------------|----------------------|---------------------------------|
| Read Data | READ, READ'READ'ONLY | INPUT, INPUT LINE, INPUT RAW |
| Read and Lock | READL | --- |
| Write Data | WRITE, WRITEN | PRINT |
| Write and Lock | WRITEL, WRITELN | --- |

15.4 SEQUENTIAL FILES

Sequential disk files are the easiest to understand and to implement in AlphaBASIC PLUS. AlphaBASIC PLUS writes data to a sequential file in ASCII format, and stores numeric data as ASCII string values, if possible. A sequential data file usually has the extension .DAT, and AlphaBASIC PLUS uses .DAT as the default extension.

Typical sequential data files are normal ASCII files in all respects, and you may edit them by using AlphaVUE or any of the system utilities. To open a sequential file, use the OPEN statement (see below), specifying INPUT, OUTPUT or APPEND mode.

Use the PRINT statement (followed by a non-zero file channel number) to write data to sequential files. The PRINT statement automatically appends a carriage return/linefeed to your data in the same manner it does when sending data to a terminal display (unless a semi-colon follows the data). PRINT also converts floating point and binary data to printable ASCII form.

Use INPUT, INPUT LINE or INPUT RAW (followed by a non-zero file channel number) to read data from a sequential file. See the examples below.

The following sections contain step by step instructions for using sequential files in the various modes (the commands used are discussed in detail later in this chapter):

USING SEQUENTIAL FILES FOR OUTPUT:

1. Use the LOOKUP command to see if the file already exists. When you output to a sequential file, you are creating a brand new file. If a file of the same name and extension already exists in the account you are writing to, AlphaBASIC PLUS automatically deletes the old file for you before it opens the new output file. Therefore, if you don't want AlphaBASIC PLUS to delete an existing file, be sure to use the LOOKUP command before you open a file for output to make sure such a file does not already exist.

If the file already exists, you can go ahead and open it (if you want AlphaBASIC PLUS to delete the existing file for you) or you can choose another file name and use the LOOKUP command again to see if **that** file already exists.

2. Use the OPEN statement to open the file for OUTPUT. Your file is then OPEN for exclusive use (no other program can access it while you have it open).
3. Use PRINT statements (specifying the file channel number associated with the file by the OPEN statement) to write data to the file.
4. When finished, use the CLOSE statement to close the file.

USING SEQUENTIAL FILES IN APPEND MODE:

1. Use the OPEN statement to open the file for APPEND. The file is open for exclusive use (no other program can write to it while you have it open). If the file does not exist, it is created.

2. Use the PRINT statement (specifying the file channel number associated with the file by the OPEN statement) to write data to the end of the file.
3. When finished, use the CLOSE statement to close the file.

USING SEQUENTIAL FILES FOR INPUT:

1. Use a file for input only if it already exists and contains data. You may wish to use a LOOKUP statement to make sure the file exists before trying to open it.
2. Use the OPEN statement to open the file for INPUT. The file is open for shared use (other programs can access it while you use it).
3. Use INPUT LINE, INPUT or INPUT RAW statements to read data from the file (specifying the file channel number associated with the file by the OPEN statement).
4. Check the EOF function after each input to make sure you haven't read beyond the end of the file.
5. When finished, use the CLOSE statement to close the file.

15.5 RANDOM FILES

Random access, or direct access, files are more complex than sequential files, but offer a more flexible method for storing and retrieving data in different formats. Random files are written in "unformatted" or packed data mode. Random file disk blocks are contiguously allocated on the disk. The major advantages of random files over sequential files are the speed and flexibility with which you may access data in a random file.

You may only open a sequential file for input OR output, but you may open a random file for input and output simultaneously. Accessing data in a sequential file requires your program to step through the file record by record.

In the case of a random file, however, you may access any record without referring to any other record in that file. In addition, random files can contain data in any format supported by AlphaBASIC PLUS (unlike sequential files, which only contain ASCII data). Random files, like sequential files, have a default extension of .DAT.

15.5.1[∞]Logical Records

All program accesses to random files are made by using the "logical record" approach. A logical record is defined as a fixed number of bytes whose format is explicitly under control of the program performing the access. Physical blocks on the disk are each 512 bytes long. AlphaBASIC PLUS allows you to choose whether you want to work with records that fit within this structure, or if you want to overlap your records over these boundaries.

If you want to "span" the 512 byte blocks, AlphaBASIC PLUS automatically computes the number of logical records that fit into one disk block, and performs the blocking and unblocking functions for you.

For example, if your logical record size is defined as 100 bytes, then each block on the disk contains 5 logical records with the last 12 bytes of each block being unused.



The most efficient use of random files comes when the logical record size is a number that divides evenly into 512 (256, 128, 64, etc.).

15.5.2[∞]Figuring Out How Many Blocks Your File Needs



This section assumes you are not using the SPAN'BLOCKS option to disregard physical block sizes.

Random access files are pre-allocated once, using the ALLOCATE statement, which gives the number of physical 512-byte blocks to allocate. It is up to you to calculate the maximum number of logical records required in the file, and then to calculate how many disk blocks are required to completely contain the number of logical records you desire.

For instance, assume the logical record size is 100 and you need a maximum of 252 logical records in the file. Each disk block is 512 bytes, and therefore contains 5 logical records. This number, 5, is called the "blocking factor," because it is the number of records that fit into each block of disk space. You need 252 logical records, so dividing 252 by 5 gives 50 full disk blocks plus 2 logical records remaining.

Since the file must be allocated in whole disk blocks, you need 51 blocks, which gives you a maximum of 255 logical records. These logical records are referred to in your program as records 0 through 254, since the first record of any random file is record 0, unless you have used FILEBASE. See "FILEBASE" below.

When figuring out how many blocks you need, use this formula:

$$\text{BLOCKS} = \text{RECORDS'NEEDED} / (\text{INT}(512 / \text{RECORD'SIZE}))$$

And round the result (BLOCKS) up to the next integer value. RECORDS'NEEDED is the number of records your program needs to handle, and RECORD'SIZE is the size (in bytes) of each individual record.



When your record size does not divide evenly into 512 bytes, it is a good idea to consider expanding it so it does. This leaves you room for future expansion of the data in the record. You use the same number of physical disk blocks whether or not you expand the record size, so you do not save anything by not doing so.

When you are opening a random file, you must specify the logical record size and the record number in the OPEN statement (also specifying RANDOM or RANDOM'FORCED mode); it is possible to get your data misinterpreted if you do not have the record size correct. No logical record size is maintained within the file structure itself.

This fact does make it nice in one respect—a file which is accessed by many programs can have its record size expanded without recompiling all the accessing programs. Here is how it works:

Assume (as an example) you have a file which is considered the parameter descriptor file for all other files in the entire system. This file gives the record size as 100 bytes for the vendor name and address file.

All programs which refer to the vendor file first read this parameter file to get the size of the vendor file logical record. The programs then set the size into a variable and use this variable in the OPEN statement for the record size.

Each READ, READL, WRITE or WRITEL statement then manipulates the 100 bytes of data by reading or writing to or from variables whose size totals 100 bytes. Let's say you now want to expand the file record size to 120 bytes and that most of the programs do not have to make use of the extra 20 bytes until some time in the future.

You write a program which copies the 100-byte record file into a new 120-byte record file and then you update the main parameter file to indicate the new record size for the vendor file is 120 bytes instead of 100.

Each program now opens the file using the new 120-byte record size (since it is read in from the parameter file at run-time), but only READs or WRITEs the first 100 bytes of each record due to the variables used by the READ, READL, WRITE and WRITEL calls.

15.5.3[∞]Accessing Random Files

The following sections show you the normal procedure for using Random files for each type of access:

USING RANDOM FILES IN RANDOM MODE:

- 1.[∞]Use the LOOKUP command to see if the file already exists. If it does, skip down to step #3.
- 2.[∞]If the file doesn't exist, create it. First, decide the size of the logical records (in decimal bytes). Then compute the blocking factor as discussed above. Use ALLOCATE to create the file with the number of disk blocks needed.

3. Use the OPEN statement to open the file for RANDOM processing. Specify the size of the logical records in the file, and the record-number variable holding the number of the logical record you are currently accessing.



The file is open for exclusive access. You may wish to use WAIT'FILE, in case the file is in use when you try to open it.

4. Use READ and WRITE statements (specifying the file channel number associated with the file by the OPEN statement) to read and write data in the file. Remember to change the record-number variable to the correct record number before performing each read or write operation so you access the logical record you want. Make sure the record-number variable contains a valid record number before performing the file I/O, or you get an error.
5. When you are finished reading and writing the file, use the CLOSE statement to close the file.

USING RANDOM FILES IN RANDOM'FORCED MODE:

1. When a random file is open in this mode, a READ operation forces a disk access, even if the requested block is already in memory. Likewise, if a WRITE is performed, the block is reread, modified and forced out to disk, even if the buffer is not yet full. This mode makes it possible for users to share random files.



RANDOM'FORCED causes the file to be OPENed for shared access. You may want to specify WAIT'FILE and/or WAIT'RECORD also, in case the file or record is in use when you try to access it.

2. If you want to only read the contents of a record, use the READ statement.
3. If you want to read in and then update, use the READL statement. Then use either a WRITE statement to write the record back to the file, or use an UNLOKR statement to unlock the record again.
4. If you want to write to a new or blank record, use the WRITEL statement.
5. Close the file.



You cannot sort a random file using XCALL BASORT if you opened the file in RANDOM'FORCED mode

ISAM FILES:

A special type of random file, opened in INDEXED and INDEXED'EXCLUSIVE modes. See Chapter 19 for information.

15.6 FILE LOCKING FOR SEQUENTIAL FILES

AMOS automatically locks files so simultaneous update problems do not occur. If you OPEN a sequential file for OUTPUT or APPEND, the monitor automatically OPENS the file for your exclusive use. Since you can modify the file in these modes, you must have exclusive use of the file. If you OPEN the file for INPUT, the monitor automatically OPENS the file for shared use, since you aren't going to be changing the file.

You can specify WAIT'FILE when you open sequential files. This causes your program to wait if the file you want to access is being used when you try to OPEN it. If you do not use this wait clause, when you try to access a file and the resource you need is already locked, AMOS prevents you from accessing it and returns an error code.

What happens then depends on your program—if your program has error trapping set, the actions specified by your error handling routine are performed; if your program does not have error trapping set, your program is interrupted and you are returned to monitor command level. For information on using error trapping to deal with system errors see Chapter 17.

If you specify WAIT'FILE, AMOS doesn't report an error if the program tries to access a locked file, but instead puts the job using the program to sleep until the file is available.

15.7 FILE LOCKING FOR RANDOM FILES

With Random files, you have a choice of how you want to OPEN the files. If you OPEN a file using RANDOM mode, the monitor OPENS the file for your exclusive use. If you specify RANDOM'FORCED mode, the file is OPENed for shared use. The RANDOM'FORCED mode is useful because it allows many users to access the same file at the same time without any updating problems.

When a file is OPENed for shared access, AMOS makes certain any records you are working with are locked before they are changed. AMOS recognizes if you are trying to WRITE to an unlocked record, and halts your program, displaying a "Record not locked" error message—unless you handle the error with an error trapping routine.

The WAIT'FILE option applies to Random files, also. Random files opened in RANDOM'FORCED and INDEXED modes have another option called WAIT'RECORD, which works just like WAIT'FILE, except it applies to records inside the file.

15.8 FILE LOCKING FOR ISAM PLUS FILES

You can also use ISAM (Indexed) files in the same manner. If you specify INDEXED when you OPEN the file, AlphaBASIC PLUS OPENS it for shared use. Or, you can specify INDEXED'EXCLUSIVE when you OPEN the file, which opens the file for your exclusive use. See Chapter 19 for information on ISAM files.

15.9 FILE STATEMENTS

AlphaBASIC PLUS automatically closes all open files when the program exits or when a CHAIN statement is executed, if the files have not already been explicitly closed by use of a CLOSE statement.



If your program should be terminated abnormally (for example, by a system crash or power failure), and a file was not closed with a CLOSE statement, AlphaBASIC PLUS may not have updated the last record. However, the file is unlocked if it was locked.

All file statements are valid as direct statements in Interactive Mode, but AlphaBASIC PLUS closes any open files before it executes a RUNP command. This prevents statements in an executing program from reading or writing to files which were opened by a direct statement. Each open file requires about 600 bytes of free memory for buffers and control blocks.

The following sections in this chapter show you the general format of each of the file statements and give detailed examples of their uses.

15.9.1 ALLOCATE

The ALLOCATE statement marks an area of the disk, which you may then open for random processing. This area may contain old data—you may want to write "blank" data to the area before using it. An attempt to allocate a file which already exists results in an error message.

A random file need only be allocated once and may then be opened for random read/write operations as many times as desired. The statement format is:

```
ALLOCATE filespec,number-of-blocks{,record-size}
```

As in the OPEN statement, *filespec* is any string expression which evaluates to a legal file description. *number-of-blocks* is a floating point expression which represents the number of physical 512-byte disk blocks to be allocated to the file (truncated for AMOS variables or rounded for IEEE variables).

The optional *record-size* sets the record size in the file directory entry of extended format disks. This lets you have records of any size you wish. If the file has a *record-size*, it must be opened with that same *record-size* whenever it is accessed.

A *record-size* of 0 is the default, and means standard record size (the same as not specifying *record-size*). For example:

```
ALLOCATE FILE$, BLOCKS  
ALLOCATE "NEW.DAT",20,50
```

Random files can also be created at command level using the command CREATE (see the CREATE reference sheet in your *System Commands Reference Manual*).

15.9.2°CLOSE

The CLOSE statement ends the transfer of data to or from a file. Once a file has been closed, no further references are allowed to it until another OPEN statement for it is executed. Any files still open when the program exits are closed automatically. Closing a file locked by AMOS unlocks it. The format of the CLOSE statement is:

```
CLOSE #file-channel
```

where #file-channel specifies the file channel number associated with the file you want to close. For example, if you have previously opened a file called VENDOR.DAT:

```
OPEN #3, "VENDOR.DAT[200,1]", INPUT
```

to close that file, use:

```
CLOSE #3
```

Once the file is CLOSED, the file-channel number can be re-used for another file.

15.9.3°CLOSEK

Allows you to close a file, but keep the system file lock in place so no other program can open the file. This allows you to maintain control of the file so you can open it from another program or process. The format is:

```
CLOSEK #file-channel
```

where #file-channel specifies the file channel number associated with the file you want to close. If the file was not OPENed with a lock, no lock is maintained. Be sure your program unlocks the file when done.

15.9.4°EOF(X)

The EOF function returns a value giving the status of a sequential file whose file channel number is X. The file is assumed to be open for INPUT. The values returned by the EOF function are:

- 1 if the file is not open or the file channel number X is zero. A file channel number of zero indicates the terminal is being used as the file.
- 0 if the file is not yet at end-of-file
- 1 if the file has reached the end-of-file condition

Due to the method used by the operating system for processing files, the end-of-file status is not achieved until after an INPUT statement has been executed.

Any INPUT statements which reach end-of-file return numeric zero or null string values for each subsequent INPUT statement. This means the normal sequence for processing sequential input files is to INPUT the data into the variables and then test the EOF(X) status before actually using the data in those variables, since if an end-of-file has been reached that data is invalid.

End-of-file should only be tested for sequential input files. Files open for output or for random processing always return a zero value.

15.9.5 FILEBASE

During normal operation, AlphaBASIC PLUS refers to the first record in a random file as record number zero (i.e., you set the record number variable to zero to access the first record in the file). In some applications you may want AlphaBASIC PLUS to refer to this first record by some number other than zero: for instance, to allow you to use zero to flag some special condition, such as a deleted record. The FILEBASE command allows you to set the number used to refer to the first record. For example:

```
FILEBASE 1
```

tells AlphaBASIC PLUS the first record in the file is record number one, not record number zero. You may use any numeric argument with FILEBASE, and the value may be a variable. Floating point numbers are converted to an integer value.

Note, FILEBASE does not associate its value with a single file (it affects all files in the program), but only takes effect when you execute the program it is in. If one program uses a FILEBASE command when referring to a file, all other programs which refer to that file should also use a FILEBASE command with the same value to be able to access the same record with the same record number.

If you are accessing multiple files within the same program, and these files have different bases, be sure to specify the accurate FILEBASE prior to accessing each file.

15.9.6 INPUT

Once a sequential file has been opened for input, you may use a special form of the standard AlphaBASIC PLUS INPUT statement to read data from the file. The INPUT statement uses a file channel number corresponding to the file channel assigned in the OPEN statement.

The variables in the list may be either numeric or string variables, but should follow the format of the data in the file being read. You can INPUT numeric data into string variables, but an attempt to INPUT a string variable into a numeric variable sets it to 0. The format is:

```
INPUT #file-channel, var1 {, var2, ... varN}
```

When input data is read into the variable list, all leading spaces are bypassed unless they are enclosed in quotes, as in the non-file form of the INPUT statement discussed in Chapter 9.

Also, all carriage returns and linefeeds are bypassed, allowing the file created by any PRINT statements to contain formatted line data if desired. Commas, spaces and end-of-line characters all terminate numeric data and then are bypassed. Null characters terminate all but unformatted variables.

As with the non-file version of the INPUT statement described in Chapter 9, the data being input must be in the proper format. In the case of the file version of the INPUT statement, you must be aware of the rules for properly separating data when you write the data out to a file using the PRINT statement. Here are the rules:

- Separate all floating point data with spaces or commas.
- Separate all string data with commas.
- If you have both floating point and string data in the same statement, separate them with commas.

Keep in mind the characteristics of the PRINT statement when writing data to a file, so you do not conflict with the rules above. See Chapter 9 for details on PRINT.

Using PRINT to send data to a file formats that data in exactly the same way it does if you use PRINT to send data to the terminal screen. Remember PRINT does not separate the data with commas for you. For example, the following statement:

```
PRINT #100, "HELLO", "AGE", "DATE"
```

sends this to the file:

```
HELLO      AGE      DATE
```

If you try to use INPUT to read that data in the file into three different string variables, the first variable contains:

```
HELLO
```

or:

```
HELLO      AGE      DATE
```

and the other two string variables contains null data. In the first case above, we are assuming no STRSIZ command is set, so the first string variable only picks up the first ten characters of the string. If STRSIZ is set to 40, you see the second display.

To read the data above as three separate pieces of string data, you must remember to separate the data by explicitly placing commas into the file. For example:

```
PRINT #100,"HELLO,AGE,DATE"
```

sends this to the file:

```
HELLO,AGE,DATE
```

which are input correctly by the following statement:

```
INPUT #100,A$,B$,C$
```

Note the statement:

```
PRINT #100,"HELLO", " ", "AGE", " ", "DATE"
```

improperly formats the data in the file because the unquoted commas above cause PRINT to separate the data with spaces as well as commas:

```
HELLO      ,      AGE      ,      DATE
```

Using semi-colons instead of the unquoted commas solves this problem. For example:

```
PRINT #100,"HELLO";" "; "AGE";" "; "DATE"
```

15.9.7 INPUT LINE

After a sequential file has been opened for input, the data can be read from the file by a special form of the INPUT LINE statement which uses a file channel number corresponding to the file channel assigned in the OPEN statement.

The variable may be either numeric or string, but should follow the format of the data in the file being read. String variables accept numeric data, but if you try to read string data into a numeric variable, the numeric variable equals 0. The general format is:

```
INPUT LINE #file-channel,variable
```

The INPUT LINE statement operation is identical to that of the INPUT statement with the exception that input into a string variable accepts the entire line up to but not including the carriage return and linefeed that ends the line. This allows commas, quotes, blanks and other special characters—except NULLs—to be input. Also, INPUT LINE accepts blank lines as input. INPUT LINE is usually used for string input rather than numeric.

15.9.8 INPUT RAW

Used to input raw data from a file. INPUT RAW does no processing of the data. The format is:

```
INPUT RAW #file-channel,variable{,variable(s)...}
```

where `variables` are unformatted variables of whatever length you specify when you define them. Each `variable` is filled to its length with characters from the file, including carriage-returns, non-printable characters, etc.

15.9.9 KILL

The KILL statement erases a file from the disk. It does not need a file channel number to KILL a file. The format is:

```
KILL filespec{,result-variable}
```

For example:

```
KILL "NEWDAT.DAT"
```

As in the OPEN statement, `filespec` is any string expression which evaluates to a legal file description. The default extension is `.DAT`. If you try to erase a non-existent file, you see a `?File not found` error message. To avoid this error, use the LOOKUP statement discussed below.



Do not try to KILL a file you currently have OPEN. This could cause corruption on the disk. If you try to KILL a file another program has OPEN, you see an error message.

You may not erase a file existing in an account outside of the project you are logged into. For example, if you are logged into account [110,2] and the program you are running tries to KILL a file in account [200,1], you see a `?Protection violation` error message.

If you try to delete a protected file, you see a `?Protection violationerror` message. If you did not use a `result-variable` in the KILL statement, this error message terminates your program. If you did specify a `result-variable`, a 0 is returned in it. The `result-variable` returns 1 on a successful deletion.

There is also a KILL function—see Chapter 11.

15.9.10 LOOKUP

The LOOKUP statement looks for a file on the disk and returns a value which tells you if the file is found and, if so, how many disk blocks it contains. The format for the statement is:

```
LOOKUP {/switch} filespec,result-variable
```

As in the OPEN statement, `filespec` is any string expression which evaluates to a legal file description (the default extension is `.DAT`). The file specification can be up to 48 characters in length. `result-variable` can be any legal floating point or binary or integer variable. However, a binary or integer cannot return all the possible values that a floating point variable can. A floating point variable may return:

| | |
|-----------------------|--|
| 0.0 | File not found |
| 0.5 | File has zero blocks. Usually this signifies that the file has been opened for output by any user on the system. |
| Positive $n \geq 1.0$ | File found. The file is a sequential file and contains n disk blocks. |
| $n \leq 0.0$ | File found. The file is a contiguous (random) file and contains n disk blocks |

A binary or integer variable cannot hold a value of 0.5. Therefore coding with a binary or integer variable will not allow distinguishing between a file that does not exist and a file that has zero blocks (which may be open for output). Further, using a binary variable will cause an unexpected result for contiguous files; binary variables are by definition unsigned, and so cannot hold negative values. If a negative value is stored in a binary value, it will be held as a very large positive value (the sign bit is ignored, and is presumed to be a high-order bit value). For this reason, it is recommended that a binary variable is not used to record the result of a LOOKUP statement. Therefore, an integer variable may return:

| | |
|---------------------|--|
| 0 | File not found, or file has zero blocks. |
| Positive $n \geq 1$ | File found. The file is a sequential file and contains n disk blocks. |
| $n \leq 1$ | File found. The file is a contiguous (random) file and contains n disk blocks. |

A binary variable may return:

| | |
|-----|---|
| 0 | File not found, or file has zero blocks. |
| n | File found. Impossible to distinguish between sequential file and a contiguous file in all cases. |

Remember, the number returned by LOOKUP is the physical disk blocks used by the file. Multiply this number of 512-byte blocks by the file's blocking factor to find out how many logical records the file contains. For example, after you do:

```
LOOKUP "PAYROL.DAT",BLOCKS
```

the variable BLOCKS contains the number of disk blocks in the file PAYROL.DAT, or a 0 if the file does not exist. For example, say that BLOCKS = -40, and the blocking factor for the file is 4. Multiplying 40 by 4 gives you 160 logical records.

There are two optional switches with LOOKUP. The default switch is /O. When this switch is in effect, LOOKUP searches for the file first in your current account, then in your [x,0] account. For example, if you are logged into account [100,3], LOOKUP looks for the file in [100,3], and, if it doesn't find it, looks in [100,0].

If /K is used, LOOKUP only searches for the file in the current account. This switch is provided to prevent a problem that can occur when using LOOKUP and KILL on the same file. Because of the AMOS file protection system, KILL can only erase a file from your current account. If the file exists only in your [x,0] account, you could find it with LOOKUP, but your program could not KILL the file, and you see an error. If you are using LOOKUP for the purpose of identifying a file to be erased, use /K. With /K, any file found by LOOKUP can be KILLED. The switches are only in effect if no account or disk is specified in the LOOKUP command.

There is also a LOOKUP function—see Chapter 11.

15.9.11[∞]OPEN

You must open a file before you can transfer data to or from it. The OPEN statement assigns a unique file channel number to a file and also specifies the name either to be given to an output file, or to be used in locating an input file. The format is:

```
OPEN #file-channel,filespec,mode{,record-size
    ,record#-variable}{,SPAN'BLOCKS}
    {,WAIT'FILE}{,WAIT'RECORD}{,READ'ONLY}
```

The elements of the OPEN command are:

| | |
|--------------|--|
| file-channel | Any numeric expression which evaluates to a positive integer. 0 is defined as the user terminal and treated as such. Random files cannot use #0 as the file channel. |
| filespec | Any string expression up to 48 characters long which evaluates to a legal file description. May be a string or literal. String literals must be in quotes. |
| mode | The mode for opening the file. See below. |

| | |
|------------------|--|
| Record-size | An expression which dynamically specifies at run-time the logical record size for read/write operations on the file. Not for Sequential files. Random files may have their size specified on creation. If you are accessing a file that already has a size, the record-size variable must specify the same size as the existing file. If the file does not have a specific size, you can OPEN it at whatever size you need. Records may be up to 65,535 characters long. If you specify a record-size, you also must use a record#-variable. |
| Record#-variable | A non-subscripted numeric variable which must contain the record number of the desired record for READ or WRITE statements when they are executed. It must be an integer, 6 or 8 byte floating point, or 1-4 byte binary variable—not a literal number. Not used in Sequential files. |
| SPAN'BLOCKS | Optional clause for Random files. Allows your records to disregard the 512-byte physical block limit. Thus, records may be up to 65,535 characters in length, and do not need to divide evenly into 512. Files written with SPAN'BLOCKS must be read with SPAN'BLOCKS, and cannot be read by earlier versions of AlphaBASIC that do not support SPAN'BLOCKS. |
| WAIT'FILE | Optional clause that causes your program to wait until the specified file is available for use. |
| WAIT'RECORD | Optional clause that causes your program to wait until the specified record is available for use. |
| | If your program tries to read or write to a file which has not been opened, you see an IO to unopened file in line nnn message and the program is aborted. If your program tries to read or write to a locked file, your program ends and a message is printed informing you the file is in use. Of course, if you use an error trapping routine, you can prevent your program from being aborted. |
| READ'ONLY | Allows you to OPEN a file for read-only operations. Does not allow any write operations to the file. If the file is already open for exclusive use, you cannot OPEN it with READ'ONLY. If you have a file open for READ'ONLY, another program can open it exclusively and "lock you out"—you lose your access to the file. |



Do not try to OPEN a sequential file for more than one type of operation at the same time. For example, don't OPEN a file for input and then OPEN the same file for output. This could cause data corruption.

filespec may be as brief as the name of the file, in which case it is assumed to have an extension of .DAT and to reside in the disk account you are logged into. Or it may be a complete file specification if you desire, giving the explicit location of the file which may be in another account or even on another disk drive. Some examples:

```
OPEN #1, "DATFIL", INPUT
OPEN #A, C$, OUTPUT
OPEN #3, "DSK1:OFILE.ASC[200,20]", OUTPUT
OPEN #1, "ACCTS:VENDOR.DAT", RANDOM, 100, RECNUM, WAIT' FILE
OPEN #1 + X, MID$(A$, 2, 3), OUTPUT
OPEN #25, "16942993-DSK13:MASTER", INDEXED, 80, RELKEY
```

The OPEN statement references the file by its actual ASCII filespec in the standard operating system format. Most of the other file commands refer to the file channel number which is assigned in the OPEN statement.

15.9.11.1[∞]OPEN Modes

The mode specifies how the file is to be OPENed:

| | |
|-------------------------|--|
| INPUT | Opens an existing sequential file for input operations. File Lock access shared. |
| OUTPUT | Opens an existing sequential file for output operations. File Lock access exclusive. Will create the file. |
| APPEND | Opens an existing sequential file so you can add data to the end of the file. Creates the file if it doesn't exist. Access is exclusive. |
| RANDOM | Opens an existing random file for random read/write. Access is exclusive. |
| RANDOM'FORCED | Opens an existing random file for random read/write. Access is shared. |
| INDEXED | Opens ISAM or ISAM PLUS data and primary index file. Access is shared. See note below. |
| INDEXED'EXCLUSIVE | Opens ISAM or ISAM PLUS data and primary index file. Access is exclusive. |
| ISAM'INDEXED | Same as INDEXED, for ISAM only. |
| ISAM'INDEXED'EXCLUSIVE | Same as INDEXED'EXCLUSIVE, for ISAM only. |
| ISAMP'INDEXED | Same as INDEXED, for ISAM PLUS only. |
| ISAMP'INDEXED'EXCLUSIVE | Same as INDEXED'EXCLUSIVE, for ISAM PLUS only. |



INDEXED and INDEXED'EXCLUSIVE normally open ISAM PLUS files. If you use the /X switch at compile time, they open ISAM files instead. ISAMP'INDEXED'EXCLUSIVE and ISAMP'INDEXED always open files as ISAM PLUS files, whether /X is used during compilation or not. Similarly, ISAM'INDEXED and ISAM'INDEXED'INCLUSIVE always open files as ISAM files, regardless of the /X switch.



Always use the correct mode when opening ISAM or ISAM PLUS files! Opening an ISAM PLUS file as an ISAM file, or vice-versa, could lead to unpredictable and destructive results.

15.9.12°PRINT

Once you open a sequential file for output, you use a special form of the PRINT statement (using the file channel assigned by the OPEN) to write data to the file.

All the techniques available to you when you use the normal form of the PRINT statement (which outputs to the terminal) are also available for sending data to a file, including PRINT USING for formatted data.

PRINT writes data to the file in the same format as it appears if you use PRINT to send the data to a terminal display (i.e., if you left off the file channel number). Here is the format for PRINT, and some examples:

```
PRINT #file-channel, expression-list

PRINT #1,A; B; C
PRINT #4,USING A$, A, SQR(A)
PRINT #Q1,USING "###.##", A1(10);
PRINT #1,"THIS IS A SINGLE LINE"
PRINT #file'number,"WRITE TO","PRINT ZONES",
```

See INPUT, above, for the required format for data when reading data from a file. If the file associated with the file-channel number is not OPEN, PRINTing to the file causes an error. For more information on PRINT, see Chapter 9.

15.9.13°READ

The READ statement reads a selected logical record from an open random file. The logical record which is transferred by the system I/O is the one whose record number corresponds to the record-number variable. The format of the READ statement is:

```
READ #file-channel,variable(s)
```

For example, if your OPEN statement is:

```
OPEN #1,"STOCK.DAT",RANDOM,rec'size,rec'number
```

and your READ statement is:

```
READ #1,PART'NUMBER
```

PART'NUMBER is filled with the data in whichever record number is currently defined in the variable `rec'number` from the file STOCK.DAT.

The `variable` in the READ statement may be any format, but should match the designated record format. The data is read into the variable(s) as unformatted bytes, without regard to variable type. Of course, if it is transferred into the wrong type of variable, you get unexpected results and/or errors when you use that variable. The data is transferred into the variable(s) until they are completely filled.

If you transfer less, no error occurs. If the record is longer than the variable specified, all excess data in the record is not transferred. If you try to transfer more data than is in the logical record size, you get an error message.

The most efficient use of random files comes when the variable or variables used are mapped by a MAP statement to the exact picture of the record format in use. See Chapter 14 for information on MAP statements.

15.9.14[°]READL

READL is a form of the READ statement that can be used in RANDOM'FORCED or INDEXED modes. It locks the record you are reading for your exclusive use. If you are in RANDOM or INDEXED'EXCLUSIVE mode, READL works the same as READ (that is, it does not lock the record, since the file is already locked). The format is:

```
READL #file-channel,var1
```

Once you lock the record with READL, use WRITE to rewrite the record to the file. WRITE automatically unlocks the record when it is done. If you use READL without later using WRITE, you must use the UNLOKR statement to unlock the record.

15.9.15[°]READ'READ'ONLY

Unlike READ, it allows you to examine data even if another program has the data locked. Where READ returns a ?File/Record in use message, READ'READ'ONLY lets you read the data.

READ'READ'ONLY is useful if you need the data immediately, and if you don't care if the data may be slightly old (the program that has the lock may be updating the data as you are reading it). You can not update the record and write it back to the file if you use READ'READ'ONLY

15.9.16°UNLOKR

UNLOKR allows you to unlock a record locked with a READL statement. The format is:

```
UNLOKR #file-channel
```

If you do not use a WRITE statement to unlock records after their use, you must use the UNLOKR statement, so other users can access those records.

15.9.17°WRITE

The WRITE statement is used to write a selected logical record to an open random file. The logical record which is transferred is the one whose record number corresponds to the current value of the record-number variable.

WRITE automatically unlocks the record after it is done if you are sharing the file (RANDOM'FORCED or INDEXED modes). The format of the WRITE statement is:

```
WRITE #file-channel,variable
```

For example, if your OPEN statement is:

```
OPEN #1,"STOCK.DAT",OUTPUT,rec'size,rec'num
```

and your WRITE statement is:

```
WRITE #1,PART'NUMBER
```

then the data in PART'NUMBER is written to the record that corresponds to the variable `rec'num` in the file STOCK.DAT.

The variable in the WRITE statement list may be in any format, but it should match the designated record format. The data is written into the logical record from the user variables as unformatted bytes, without regard to variable type.

If the record is longer than the variable list specifies, all excess data in the record is not modified. If you try to transfer more data than is in the logical record size, you get an error message.

The most efficient use of random files comes when the variable or variables used are mapped by a MAP statement to the exact picture of the record format in use.

15.9.18°WRITEL

WRITEL is a special form of the WRITE statement allowing you to write a new record if you are in RANDOM'FORCED or INDEXED modes. Normally, you use a combination of READL and WRITE to modify a record in the file, but if it is a new record, you can't use READL because the record does not yet exist.

Thus, to write to the record, you want to lock it for exclusive use for protection. If you are in RANDOM or INDEXED'EXCLUSIVE mode, WRITEL functions exactly like WRITE. The format is:

```
WRITEL #file-channel,variable
```

15.9.19°WRITEN

WRITEN is a special form of the WRITE statement allowing you to write to a record without releasing the lock after you finish. This lets you later write to that record without a wait. The format is:

```
WRITEN #file-channel,variable
```



Be careful your program writes to the record again, or otherwise releases the lock!

15.9.20°WRITELN

WRITELN is a special form of the WRITEL statement allowing you to write to a record without releasing the lock after you finish. This lets you later write to that record without a wait. The format is:

```
WRITELN #file-channel,variable
```



Be careful your program writes to the record again, or otherwise releases the lock!

CHAPTER 16

CHAINING AND SUBPROGRAMS

This chapter discusses:

- The CHAIN statement
- CHAINing to other programs
- CHAINing to System Functions
- What a Subprogram is
- Using a Subprogram
- Variables in Subprograms

16.1 THE CHAIN STATEMENT

The CHAIN statement stops the run of the current program and causes another program or system function to run. The new program to be executed must be named in the CHAIN statement itself; that name may be a full file specification.

The file named in the statement may be another AlphaBASIC PLUS program (compiled only), or it may be a system command or command file. This allows your program to execute a command file and invoke system commands as well as execute other AlphaBASIC PLUS programs.

16.1.1 Chaining to another AlphaBASIC PLUS Program

CHAIN assumes a default extension of .RP. If you specify another extension, AlphaBASIC PLUS assumes you have specified a command file or system command. Therefore, if you wish to use CHAIN to run another AlphaBASIC PLUS program, the file you specify must be compiled, and it must have an .RP extension.

If you do not specify a device and account, AlphaBASIC PLUS follows the search pattern outlined in Chapter 2 when looking for .RP files. If you do specify a device and account, AlphaBASIC PLUS looks in the specified area.



Note that CHAINing to another .RP file will bypass reading and processing of the target .BPR file and certain fields of the target's .RP files. See Appendix H for details.

All variables in the new program are first cleared to zero prior to the run. Some examples of legal CHAIN statements are:

```
CHAIN "PAYROL"  
CHAIN "PAYROL.RP"  
CHAIN "DSK1:PAYROL[101,13]"
```

Due to the fact programs are compiled and not interpreted, there is no way to execute a program at any entry point other than its physical beginning. You may, however, pass string literals into a program using the CMDLIN function (see Chapter 10), or other data by using the AlphaBASIC PLUS assembly language subroutine COMMON to store data in a common memory area.

COMMON lets you store information either in system memory (where programs run by all users on the system can access the information) or in your memory partition (where only programs run by you can access the information). For details on using COMMON, see your *AlphaBASIC XCALL Subroutine User's Manual*.

In addition to sharing information, you can use the common area to pass parameters to the chained program. For example, the current program can pass a parameter to the new program which it uses in an ON-GOTO or SWITCH statement to begin execution at some point in the new program based on the value passed in the parameter.

Another way to make sure chained programs can share information is the use of disk files. The current AlphaBASIC PLUS program can open a data file, write the variables it wants to share into it, and then close it. When the new file is chained in, it can open the file and read the necessary information.



Note that you may not CHAIN out of a user defined function or a subprogram. You must return back to the main program first.

16.1.2[∞]Chaining to System Functions

It is sometimes desirable to transfer execution to a system function or a command file from an AlphaBASIC PLUS program. If the extension of the file in the CHAIN statement is not .RP, AlphaBASIC PLUS assumes the file is a system command program or system command file (a .LIT, .DO or .CMD file). In this case, the AlphaBASIC PLUS run-time package creates a dummy command file at the top of the current user partition and transfers control to the monitor command processor.

The monitor then interprets this dummy command file as a direct command and executes it. Note the dummy command file created by the run-time package is merely the one line name specified in the CHAIN statement. It is not the command file being chained to itself, which is the target function desired. You can also include parameters for that program or command file. Some valid examples are:

```
CHAIN "TEST1.CMD"  
CHAIN "DSK0:BACKUP.CMD[2,2]"  
CHAIN "SEE.DO[110,0] MEMO.TXT"
```



Note that if you CHAIN to RUNP to execute another .RP file, the existing instance of RUNP will commit suicide by executing an EXIT monitor call after performing END processing. A new instance of RUNP will be initialized. This has the side effect of deleting all temporary memory modules in the user's position, and causing the target .RP and .BPR file to be read and processed.

If the device and account are not specified, the search path is:

1. System memory
2. User memory
3. The account and device you are logged into.
4. DO files look in [x,0], then in DSK0:[2,2]. CMD files look in [x,0].



To load a file into your user memory partition, use the monitor level LOAD command. To load the file into system memory (where it may be accessed by all users on the system), the System Operator must add the appropriate SYSTEM command line to your system initialization command file.

Note also when you chain to a monitor command, after the command has finished executing, it returns you to monitor level, rather than to your program. If you want to perform a system command without exiting your program, use the AMOS command (see Chapter 9).

16.2 WHAT IS A SUBPROGRAM?

A subprogram is a melding of a separate AlphaBASIC PLUS program and a more involved form of GOSUB. Subprograms are used for two primary purposes:

- As a way to structure a large program.
- As a way to share code between separate programs, thus simplifying coding and program maintenance.

A subprogram is a self-contained section of code, usually performing one or more defined business tasks. It operates in a separate, but linked, environment from the

program (or other subprogram) that called it which exists only for the duration of the subprogram's execution. This separate environment is a distinguishing feature of subprograms, and is a logical extension of the GOSUB/CALL and user defined function environment.

16.2.1^o Subroutines, User Defined Functions, and Subprograms

When you call a subroutine by using a GOSUB/CALL statement, the subroutine has access to all the variables and environment settings of the calling program. If the subroutine alters the value of a variable, the calling program sees that change after RETURNing from the subroutine. Any changes in the environment, such as changing the ON ERROR destination or READING from the DATA pool, are also effective in the calling program. The statements making up the subroutine are placed in the same file as the statements for the calling program, and are compiled together into a single .RP file. Therefore the subroutine cannot be accessed from another .RP file: instead you have to copy the subroutine's statements into the second file (or use a ++INCLUDE statement), duplicating code and making the space taken up by the .RP files larger.

User defined functions put a thin wall up between the calling program and the statements inside the user defined function. User defined functions support the passing of parameters, as well as the use of local variables. Local variables are not visible from the calling program, and so give you a safety net: you can use them without affecting any variable of the same name in the calling program. The parameters allow you to pass variables to and from the function: if you alter the value of a parameter in the user defined function, the corresponding variable in the calling program is also changed. But in a user defined function you also have access to all the other variables and environment settings: changing any of them will be reflected in the calling program. Just like subroutines, user defined functions are part of the calling program's source code file, and are compiled together into a single .RP file. Other programs cannot access that copy of the user defined function at run time.

Subprograms go a step further. They support parameters and local variables. But they do not allow the changing of calling program variables unless they are passed as parameters. Neither are changes to the calling program's environment allowed, with a few minor exceptions. Any environment change will only be effective for the duration of the execution of the subprogram. The runtime system remembers the calling program's environment before executing the subprogram, and restores that saved copy after the subprogram returns to the calling program.

16.2.2°Types of Subprograms

Subprograms can exist in two forms. The first is in the main program file, in which case they are compiled together with the main program into one .RP file, and are not accessible outside that .RP file. There can be up to 50 subprograms in the main file, but each one must come after the main program at the end of the file. Such subprograms are termed "internal subprograms". The second form is in a completely separate source file (usually one with a .BP extension), with just one subprogram to a file. In this case, the file is compiled separately from other source files, producing a subprogram object code file with a .SPG extension. Such subprograms are termed "external subprograms".

16.2.2.1°Considerations for Internal Subprograms

Although the runtime environment of an internal subprogram is separate from the main program, the compile time environment is not. This has a number of consequences:

- °°Line numbers in internal subprograms must be unique in the entire source file.
- °°Any command line switches affect the subprograms as well as the main program.
- °°A ++PRAGMA setting takes effect immediately, and is in force for the rest of the source file, including across subprograms.
- °°Only SCALE and the string work area size settings are reset to their default on starting a subprogram compilation. Other settings, such as STRSIZ, are not changed, and so affect subsequent subprograms. Runtime settings, such as SCALE, are discussed in a later section.

The recommended practice is to explicitly set each compiler directive in each subprogram. This bulletproofs you from future changes in AlphaBASIC PLUS, and prevents an internal program executing differently if it is converted to an external program (and hence compiled separately) by a simple cut-and-paste editing operation.

16.2.3°Subprogram Structure

A subprogram is delineated by a pair of statements. The start of a subprogram is defined by a SUBPROGRAM statement, and the end marked by a SUBEND statement. A calling program uses the SUBCALL statement to execute a subprogram. It is the appearance of the SUBPROGRAM statement that makes the compiler produce a subprogram file with a .SPG extension. It is not a good idea to give a subprogram the same name as the main program file (even though they have different extensions), because COMPLP creates an .RP file, and then renames it to .SPG when it finishes if the file is a subprogram. This would erase your main program .RP file, and could cause confusion.

When a program SUBCALLs a subprogram, the SUBCALL specifies which subprogram to execute, as well as the parameters being passed. The runtime system will search for the subprogram using the following scheme:

1. Looking within the same .RP file for a subprogram with the same name.
2. Looking in user memory for a .SPG file with the same file name.
3. Looking in system memory for a .SPG file with the same file name.
4. Looking on disk for a .SPG file with the same name. The search path for disk searching is:
 - a. Your log-in device, drive, and PPN (dev#:[p,pn])
 - b. Your library account, that is your log-in device, drive, project number, and programmer number set to zero (dev#:[p,0])
 - c. The AlphaBASIC PLUS library account, DSK0:[7,35].

The ability to generate an external subprogram allows that subprogram to be shared amongst multiple programs, subprograms, and users while keeping just a single copy of the .SPG file and its source file. This can reduce memory overhead as well as program maintenance. Updating application suites can be easier if just a few .SPG files are updated, rather than having to change many, larger, .RP files.

An .SPG file cannot be executed directly by RUNP. It can only be called from a calling program by using the SUBCALL statement.

Just like subroutines and user defined functions, it is possible to call a subprogram from within a subprogram, just as you can call subroutines and user defined functions from within subprograms. The visibility rules still hold: an internal subprogram can only be SUBCALLED from the main program and other internal subprograms within the .RP file. An external subprogram can be SUBCALLED from the main program, internal subprograms within the main program, and other external subprograms.



Use a little caution when nesting SUBCALLs. It could cause an overflow of the stack area, resulting in an error and the termination of your program run. See Appendix H for details.

16.2.4 Subprogram Components

A subprogram is made up of a number of components as shown below:

```

SUBPROGRAM name
    {PARAMETER definition(s)}
    {local variable definitions(s)}
    program statement(s)
    .
    .
    {SUBEND {error-number}}

```

16.2.4.1[∞]Subprogram Name

For external subprograms, the SUBPROGRAM statement must be the first line of executable code in the file. Blank lines, comments and ++PRAGMAs can be placed before the SUBPROGRAM statement.

For internal subprograms, the SUBPROGRAM statement must be placed after the last executable statement of the main program. A SUBPROGRAM statement stops the compilation of the main program, and starts the compilation of the subprogram. Therefore, all the main program statements must have been compiled before a subprogram is started.

The name must be six alphanumeric characters or less. For internal subprograms, this name is the name used in SUBCALL statements to execute the subprogram. For external subprograms, this name is used for error reporting and documentary purposes only. It is not the name used in a SUBCALL statement: the name in the SUBCALL statement is the filename (without an extension) of the .SPG file that contains the subprogram.

16.2.4.2[∞]Parameters

You can declare formal parameters for a subprogram by using PARAMETER statements, which work just like MAP1 statements. A PARAMETER may have lower level MAP statements below it. For example:

```

SUBPROGRAM TaxAdd
    PARAMETER Base'Income
        MAP2 Deductions,F,6
        MAP2 Depreciation,F,6
    PARAMETER Tax'Rate
        MAP2 Base'Percentage,F,6
        MAP2 Clause_13_deduction,F,6

```

This defines a subprogram called TaxAdd with two parameters, Base'Income and Tax'Rate. Each of the parameters are unformatted variables, having two lower level MAP variables each. The calling program might call TaxAdd in the following manner:

```

map 1 CLIENT'BASE'INCOME

```



```

        map 2      CL'DEDUCTIONS,          f,  6
        map 2      CL'DEPRECIATION, f,  6
    ....
    map 1 CURRENT'TAX'RATE
        map 2      TR'BASE'PERCENTAGE,      f,  6
        map 2      TR'CL'13'DEDUCTION,      f,  6
    ....
    CL'DEDUCTIONS      = 1
    CL'DEPRECIATION    = 0.25
    TR'BASE'PERCENTAGE = 27.8
    TR'CL'13'DEDUCTION = 0.25

    SUBCALL "TAXADD", CLIENT'BASE'INCOME, CURRENT'TAX'RATE

```

The **PARAMETER** statements **MUST** appear immediately after the **SUBPROGRAM** name, except for blank and comment lines. You may also use **++INCLUDE** to include the parameter list (or part of it). AlphaBASIC PLUS allows you to add a **/P** (e.g., **++INCLUDE PARAM.BPI/P**) to the **++INCLUDE** statement so a **MAP1** statement in the included file is interpreted as a **PARAMETER** statement (allowing you to include the same file in a main program and a subprogram).

Each **PARAMETER** statement defines a "formal parameter", and defines the types and names of the formal parameters. When you **SUBCALL** a subprogram, you specify the name of the subprogram, and a list of variables (called "arguments") that you wish to pass over to the subprogram. The formal parameters define the format, type, and order of the arguments. The first argument in the **SUBCALL** statement corresponds to the first formal parameter, the second argument to the second formal parameter, and so on.

If the number of arguments in a **SUBCALL** exceeds the number of formal parameters, you get an error. However, if the number of arguments passed is less than the number of formal parameters, the remaining formal parameters will have null values.

Arguments can be passed in two ways:

1. **By value.** If you pass an argument by value, and then change the value of the corresponding parameter variable in the subprogram, the calling program will not see the change after the subprogram has finished. A copy of the argument is made, and the subprogram will operate on the copy. On return to the calling program, the copy is deleted, leaving the unchanged version for the calling program to use.
2. **By reference.** If you pass an argument by reference, any change you make to the value of the corresponding parameter variable in the subprogram will be visible to the calling program on return from the subprogram.

Expressions are always passed by value. Entire arrays can only be passed by reference, but individual elements can be passed by value.

You specify which argument passing method you want on an argument-by-argument basis for each **SUBCALL** statement. To specify passing by reference, prepend an **at**

sign to the argument's name. To specify passing by value, just used the unadorned variable name.

For example, the following SUBCALL passes the first argument by value, the second by reference. The third argument passes an entire array by reference, and the fourth passes a single array element by value:

```
SUBCALL "CALCUL", START'POS, @END'POS, @ARRAY(1), MATCH(3)
```

The argument passing method also has an effect on how values are passed. If you pass an argument by reference, you must be certain that the size of the argument and the size of the corresponding parameter are exactly the same, or you will get memory overwriting and/or undefined results. This applies particularly to passing arrays to subprograms: the receiving formal parameter must be of the same type and size as the array specified in the SUBCALL argument. No type conversion or any other operation is done if you pass arguments by reference. If you pass an argument by value, and the type of the argument is not the same as the type of the corresponding formal parameter, AlphaBASIC PLUS will convert the argument to the type of the formal parameter using its standard rules.

16.2.4.3 Subprogram Code

Subprogram code continues until the end of the file is reached or another SUBPROGRAM is defined.

16.2.4.4 SUBEND

A SUBEND program is used to delineate the end of the code for the subprogram, and instructs execution to return back to the calling program. If an END statement is encountered in the subprogram, the entire program will be terminated.

The SUBEND statement can take an optional parameter of an error code. If a non-zero number is specified, an AlphaBASIC PLUS error corresponding to that number is propagated back to the calling program. This can be used to control error handling, discussed below. The parameter can be either a literal or a variable (such as ERR(0)).

16.2.5 Error Handling in Subprograms

Each subprogram has its own environment, and so can have its own set of ON ERROR statements. If none are defined, and an error occurs, the program is terminated, even if the calling program has an ON ERROR set active. The terminating message will specify the error type, the line number of the error if set, and the name of the subprogram if it was an external subprogram, or the name of the .RP file if it was an internal one.

If a subprogram has an ON ERROR set, and an error occurs, control is passed to the line or label specified in the ON ERROR line. ERR(0) will hold the error code, and

ERR(1) the line number of the error in the subprogram if set. After executing some code, the subprogram can return to the calling program if it wishes by executing a SUBEND. If the SUBEND either has no parameter, or a parameter with a zero value is specified, the error setting is cleared, and control reverts to the calling program as if no error had occurred.

If the SUBEND statement specifies a parameter with a non-zero value, control is passed back to the calling program, and immediately an AlphaBASIC PLUS error with an ERR(0) of that number is triggered as if the error happened in the SUBCALL statement. The value of ERR(1) corresponds to the value of the line number of the SUBCALL line, if set. If the calling program has an ON ERROR set active, that set takes over control. If not, the program is terminated.

To propagate the same error code back up to the calling program, use SUBEND ERR(0). But note that there is no restriction to using the same SUBEND value as the original error code. However, for future compatibility, the value passed back must be a legal, defined, AlphaBASIC PLUS error code (see Chapter 17).

16.2.6^oUsing Subprograms

Because subprograms are compiled separately, they can only work with variables defined within them, or variables explicitly passed to them. For example:

```

MAIN' PROGRAM:
  A = 5
  B = 7
  SUBCALL "FIGURE", A
  PRINT "Program answer:";(A * B)
  END

SUBPROGRAM FIGURE
  PARAMETER A, F, 6
  MAP1 B, F, 6
  B = 10
  PRINT "Subprogram answer:";(A * B)
  SUBEND

```

The program above prints:

```

Subprogram answer: 50
Program answer: 35

```

As you see, the variable B is a different local variable in the main program and the subprogram. Only the variable A has the same value in both (since it was passed).

User defined functions and subroutines are private to the environment: a subprogram cannot call a user defined function in a calling program, for example.

The separate environment also means that you cannot GOSUB/CALL, RESUME, or GOTO into or out of a subprogram.

You must not CHAIN out of a subprogram. You must SUBEND back through all the calling subprograms to the main program (the .RP program) before CHAINing.

You may need to adjust the size of the String Work Area in certain cases. See Appendix I for details.

16.2.7[∞] Saved Runtime Environment

The saved calling program environment currently includes the following elements: ON ERROR, ON CTRLC, SCALE, SIGNIFICANCE, and data pool information.

Currently, FILEBASE, STRSIZ (a compile-time setting), and DIVIDE'BY'0 settings are global in scope: changes by the subprogram will be seen by the calling program. However, this may change in future versions of AlphaBASIC PLUS, so the recommended practice is to explicitly set them in each subprogram.

If you open files in the calling program, they will remain accessible in the subprogram. If you open files in a subprogram, they will remain accessible to the calling program after a SUBEND even after the subprogram is exited unless they are explicitly closed.

If you use the AMOS command to change system systems, or the ECHO or NOECHO commands, any changes you make to your job's (or the system's) environment will be visible to the calling program after the SUBEND.

CHAPTER 17

ERROR TRAPPING

This chapter discusses:

- What is error trapping?
- The ON ERROR GOTO statement
- The ERR(X) function
- The RESUME statement
- Control-C trapping

In addition, we include two sample programs illustrating how to use error trapping.

17.1 WHAT IS ERROR TRAPPING?

Normally, when an error occurs during the run of an AlphaBASIC PLUS program, an error message is printed, and the program run is halted. Using a technique called "error trapping," you can catch the error before the AlphaBASIC PLUS processor stops your program, giving you a chance to correct the error. This has the advantage of allowing your program to continue running, and gives you more control over what is happening in your program. Many of the errors you encounter can be handled this way.

17.2 THE ON ERROR GOTO STATEMENT

Error trapping is turned on and off by using the ON ERROR GOTO statement in one of two forms. The first form specifies a line number (or label) within the program. When the program encounters this ON ERROR statement, it stores the line number and turns error trapping on. If an error occurs any time after this, AlphaBASIC PLUS transfers control to the the line number or label you specified. Here is what such a statement looks like:

```
ON ERROR GOTO 500
ON ERROR GOTO 'TRAP' ROUTINE
```

The error routine must then take appropriate action based on the type of error.

The second form of the statement turns off error trapping by giving a line number of zero or leaving the line number off completely:

```
ON ERROR GOTO 0
ON ERROR GOTO
```

After executing a statement like the ones above, error trapping is off, and errors are handled in the normal way. We recommend all error trapping routines execute the ON ERROR GOTO 0 statement for all errors your program is not going to fix.



If an error occurs within the error trapping routine itself, it is processed and the error message `?Error in error trapping` is printed. There is no method to trap errors within an error recovery routine.

17.3[∞]ERR(X) FUNCTION

ERR(X) returns the following data based on conditions at the time of the error:

```
ERR(0)  = The number of the error detected
ERR(1)  = The number of the last line encountered prior to the error
ERR(2)  = The channel number of the last file accessed
ERR(3)  = Error status number from the JOBERR field of the Job Control Block.
```

Using any or all of these functions gives you the information you need to decide what your error recovery routine is going to do. ERR(3) returns standard monitor error codes. See Appendix G for a list.

17.3.1[∞]Error Codes Returned by ERR(0)

| Code | Meaning | Code | Meaning |
|------|------------------------|------|-----------------------------|
| 1 | Operator interrupt | 2 | System error |
| 3 | Out of memory | 4 | Out of data |
| 5 | NEXT without FOR | 6 | RETURN without GOSUB |
| 7 | RESUME without ERROR | 8 | Subscript out of range |
| 9 | Numeric point overflow | 10 | Divide by zero |
| 11 | Illegal function value | 12 | Subroutine not found |
| 13 | File already open | 14 | IO to unopened file channel |
| 15 | Record size overflow | 16 | File spec error |
| 17 | File not found | 18 | Device not ready |
| 19 | Device full | 20 | Device error |
| 21 | Device in use | 22 | PPN not found |

| Code | Meaning | Code | Meaning |
|------|--|------|--|
| 23 | Protection violation | 24 | Write protected |
| 25 | File type mismatch | 26 | Device does not exist |
| 27 | Bitmap kaput | 28 | Disk not mounted |
| 29 | File already exists | 30 | Redimensioned array |
| 31 | Illegal record number | 32 | Invalid filename |
| 33 | Stack overflow | 34 | Invalid syntax code |
| 35 | Unsupported function | 36 | Invalid subroutine version |
| 37 | File in use | 38 | Record in use |
| 39 | Deadly embrace possible | 40 | File cannot be DELETED |
| 41 | File cannot be RENAMED | 42 | Record not locked |
| 43 | System error | 44 | LOKSER queue is full |
| 45 | Device not file structured | 46 | Illegal ISAM sequence |
| 47 | #0 illegal for random files | 48 | File not open |
| 49 | Channel inappropriate file I/O | 50 | Remote not responding |
| 51 | First logical unit not mounted | 52 | Illegal key |
| 53 | Illegal key number | 54 | String overflow |
| 55 | Undefined array in XCALL | 56 | File is open exclusively |
| 57 | Reserved for expansion | 58 | Reserved for expansion |
| 59 | Reserved for expansion | 60 | Too many files open |
| 61 | Translation table not found in memory | 62 | Illegal record size |
| 63 | Too many parameters | 64 | Too many parameters to function |
| 65 | Channel number in use | 66 | Illegal channel number |
| 67 | String Work Area Exceeded | 68 | Significance out of range |
| 69 | Cannot find subprogram array | 70 | Cannot pass single value to subprogram array |
| 71 | Not enough memory to load XCALL routine | 72 | Bad symbolic key (not mapped) in ISAM code 7 |
| 73 | Cannot run .RP with DIM statements from RES: or MEM: | | |

17.4°°THE RESUME STATEMENT

Used to resume execution of the program after the error recovery procedure has been performed. It also turns Control-C detection back on (a Control-C interrupt is not allowed while AlphaBASIC PLUS processes the error trapping routine). It takes on two forms similar to the forms of the ON ERROR GOTO statement. The first form specifies a line number (or label) in the program where the execution is to be resumed:

```
RESUME 410
RESUME TRY' AGAIN
```

The second form specifies a line number of zero, or no line number at all, and causes the execution to be resumed at the statement which caused the error to occur:

```
RESUME 0  
RESUME
```

Both forms cause the error condition to be cleared and turns on error trapping again.



You should never use the GOTO statement to exit from an error trapping routine. Use RESUME. This is because RESUME clears the area of memory used by the error routine. GOTO does not, which could cause memory errors.

17.5[∞]CONTROL-C TRAPPING

When you press `CTRL/C` during the run of an AlphaBASIC PLUS program, the program stops at the next statement. What happens next depends on whether an ON CTRLC GOTO is defined, or whether error trapping is on or off.

If no ON CTRLC GOTO is defined, and if error trapping is off, the program stops and the appropriate message is printed on the terminal. If an ON CTRLC GOTO is defined, control passes to the routine that handles the Control-C.

If no ON CTRLC GOTO is defined, and error trapping is on, the error trapping routine is entered with the code in ERR(0) being set to 1. You can then decide what your program is going to do when a Control-C is encountered. If you simply let the program RESUME, you have effectively turned off Control-C interrupting (which may not be a good idea—if your program gets into an endless loop you couldn't stop the run, except by re-setting your computer).

Control-C trapping lets you prevent users of your program from exiting during a critical time, such as during a file update. The Control-C is detected immediately.

During an error recovery routine, no Control-C use is allowed, so the program cannot be stopped during this important phase.

17.6[∞]A SAMPLE ERROR RECOVERY ROUTINE

In this sample of an error recovery routine, we'll take a case where the user of your program is asked to input two numbers. The program then divides the first number by the second number. But, what if the user enters a zero as the second number? Normally, BASIC halts the run at that point and prints a `Divide by zero` error message. You can use the `DIVIDE'BY'ZERO` statement to turn off the error, but then the user gets an answer of zero if they enter a zero as the second number. By using an ON ERROR GOTO statement and an error recovery routine, the program can handle the error itself:


```
ON ERROR GOTO DIVIDE'BY'ZERO
INPUT "Enter two numbers: ",A,B
PRINT "A divided by B equals: "; A/B
END

DIVIDE'BY'ZERO:
  ! If error is not "divide by zero" exit the program:
  IF (ERR(0) <> 10) THEN &
    PRINT "System Error Detected.  Exiting..." : END
  PRINT
  PRINT "Division by zero does not work!"
  INPUT "Please re-enter the second number: ",B
  RESUME                ! Go back to the line where the
                        ! problem occurred.
```

Two sample runs of the program look like this:

```
Enter two numbers: 2,3 RETURN
A divided by B equals: .666667

Enter two numbers: 25,0 RETURN
A divided by B equals:
Division by zero does not work!
Please re-enter the second number: 5 RETURN
A divided by B equals: 5
```

Note a successful error trapping routine must either resolve the error or exit the program. For example, if the program above had merely printed an error message and then RESUMEd back to the line where the error occurred, the Divide by zero error still exists, AlphaBASIC PLUS again transfers control to the error trapping routine, and the program is stuck in an infinite loop. Instead, the program resolves the error by having the user change the value of the second variable, and then resuming program execution.

CHAPTER 18

EXTERNAL ASSEMBLY LANGUAGE SUBROUTINES

This chapter discusses:

- Why use external subroutines?
- Automatic subroutine loading

AlphaBASIC PLUS supports the use of external assembly language subroutine programs that can be called from your AlphaBASIC PLUS programs. We discuss the details of writing these subroutines in Appendix E. For information about assembly language, see your *Assembly Language Programmer's Manual* and your *AMOS Monitor Calls Manual*.

18.1 WHY USE ASSEMBLY LANGUAGE SUBROUTINES?

There are several good reasons why you might want to use an assembly language program to carry out a function rather than using another AlphaBASIC PLUS program or subroutine.

Assembly language programs are generally much smaller and faster than equivalent BASIC programs—when speed and size are important factors, you may want to code parts of your program into assembly language. Yet another reason for using assembly language programs is some tasks are too awkward (or even impossible) to do from within a higher level language.

Assembly language programs are uniquely suitable for applications requiring that you work more closely with the hardware or operating system than is convenient or possible in BASIC.

Although you may want to write your own assembly language subroutines (see Appendix E for further information), note we do provide a set of existing assembly language subroutines in the AlphaBASIC PLUS Library Account, BP:(DSK0:[7,35]). These files have an .XBR extension. For information on these subroutines, see your *AlphaBASIC XCALL Subroutine User's Manual*.

In addition, a set of business-oriented assembly language subroutines is available from your Alpha Micro dealer.

To call an assembly language subroutine from an AlphaBASIC PLUS program, use the XCALL statement. The syntax for this statement is:

```
XCALL routine{,argument1{,...argumentN}}
```

For example:

```
XCALL BASORT,3.0,3.0,80.0
```



XCALLs in AlphaBASIC PLUS assume an integer value unless a number is specified with a decimal point, as in the example above. If the statement is XCALL BASORT,3,3,80, BASORT uses the numbers as integers.

The routine to be called is an assembly language program which has been assembled using the machine language assembler.

When the XCALL statement is executed by the AlphaBASIC PLUS run-time system, the named subroutine is located in memory and then called.

18.2[∞]AUTOMATIC SUBROUTINE LOADING

When an AlphaBASIC PLUS program calls a subroutine by using an XCALL statement, AlphaBASIC PLUS attempts to locate the subroutine in user or system memory. If it cannot, it attempts to load the subroutine from the disk, following the search pattern outlined in Chapter 2.

If an AlphaBASIC PLUS program fetches a subroutine from the disk, AlphaBASIC PLUS loads the subroutine into memory only for the duration of its execution. The subroutine is loaded as a standard AMOS module in the area reserved by MINF at the top of your memory partition. See Appendix H for details. Once the subroutine has completed its execution, it is removed from memory.

Therefore, if a subroutine is to be called a large number of times, it is wise to load it into memory (using the monitor LOAD command) to avoid the overhead of fetching the subroutine from disk.



Subroutines loaded into memory by use of the monitor LOAD command remain in memory until you reset the system or until you use the monitor command DEL to delete them.

CHAPTER 19

USING ISAM PLUS FILES

This chapter is a quick reference guide to using ISAM PLUS from AlphaBASIC PLUS, and discusses:

- What ISAM PLUS is
- The ISAM PLUS file structure
- Updating ISAM PLUS files
- Creating an ISAM PLUS file
- Retrieving statistics of an ISAM PLUS file
- Error processing

19.1 WHAT IS ISAM PLUS?

The ISAM PLUS program is a tool for organizing and retrieving data. The name ISAM stands for:

Indexed
Sequential
Access
Method

and refers to the manner in which the data is organized. ISAM PLUS is an advanced version of ISAM.



Even though the word "Sequential" is part of the ISAM name, and they are referred to as "indexed sequential files", the **Access Method** is sequential—not the file. ISAM files are RANDOM files.

AlphaBASIC PLUS can process indexed sequential files by linking to the ISAM PLUS assembly language package. AlphaBASIC PLUS attempts to load ISAMP.SYS if it does not find it in memory. Therefore, if it is not in memory, there must be enough memory available in your partition to load it. ISAM PLUS supports multiple index files by using

some elementary ISAM PLUS statements allowing the direct control of index file and data file items.



We strongly recommend you use the AlphaBASIC PLUS command statements designed for file and record locking. ISAM PLUS files are particularly vulnerable to simultaneous update problems. Proper file and record locking keeps your data intact, helps clarify your code, and enhances compatibility with other systems and future Alpha Micro software.

It is important when reading the following sections you be familiar with opening and using random data files. If you are not, see Chapter 15 first. It is also a good idea to be familiar with error trapping, explained in Chapter 17, because error trapping is a necessity for ISAM PLUS programs.

This chapter also assumes you are familiar with the Alpha Micro ISAM PLUS system, which is fully explained in your *ISAM PLUS User's Guide*. This chapter is intended as a quick reference guide to the ISAM PLUS commands used within AlphaBASIC PLUS—more detailed information about using ISAM PLUS from AlphaBASIC is in your *ISAM PLUS User's Guide*.

19.2[∞]ISAM PLUS FILE STRUCTURE

ISAM PLUS files are more complex in structure than sequential or normal random files, allowing large data files that can be accessed faster and more precisely.



Both the INDEXED and INDEXED'EXCLUSIVE modes of ISAM PLUS require the ISAMP program be able to write to the disk containing the index files—even if you do not plan to do anything more than read from the disk. Therefore, make sure the disk containing the index files is not write-protected.

19.3[∞]UPDATING THE RECORDS OF AN ISAM PLUS FILE

This section describes the statements for positioning the file pointer, reading, writing, adding, deleting, locking, and unlocking the records of an ISAM PLUS file. Each description includes the FORMAT defining the syntax of a particular verb, and the COMMENTS explaining the parameters, variables, trapped errors and status values.

The GET, GET'NEXT and GET'PREV verbs with the qualifier 'LOCKED allow you to locate, read, and lock a record all in a single statement. True logical record locking is done. That is, when a logical record is locked, there is no impact on the ability of another user to access the file or lock different logical records, even in the same physical record (each physical record can contain more than one logical record).

Variables used to hold status conditions returned by ISAMP are defined in the file BP:ISAMP.BPI. Your programs should have a ++INCLUDE ISAMP statement to include this definition file.

19.3.1[∞]OPEN Statement for ISAM PLUS Files

The OPEN statement opens a pair of ISAM PLUS files for processing. The format is:

```
OPEN #file-channel,filespec,mode,relrecno,filstat{,END'FILE}
      {, WAIT'RECORD}{,WAIT'FILE}{,READ'ONLY}
```

where *filstat* is a floating point variable that holds the resulting status condition, and *END'FILE* opens the file with the file pointer positioned past the last record, based on the primary key. When used with *GET'PREV*, it allows reverse scanning of an indexed file. The other options are the same as explained in Chapter 15.



The three optional parameters may be specified in any order. The mode must be INDEXED, INDEXED'EXCLUSIVE, ISAM'INDEXED, ISAM'INDEXED'EXCLUSIVE, ISAMP'INDEXED, OR ISAMP'INDEXED'EXCLUSIVE. To use ISAM instead of ISAM PLUS, use ISAM'INDEXED or ISAM'INDEXED'EXCLUSIVE, or use the /X switch when you compile the program. To force use of ISAM PLUS, use ISAMP'INDEXED or ISAMP'INDEXED'EXCLUSIVE

19.3.2[∞]GET, GET'LOCKED, and GET'READ'ONLY

The GET statement permits random access by key value. This and other random access statements return status values and, if the access is successful, the associated relative record number. GET'LOCKED locks the record for update and has the same syntax as GET. GET'READ'ONLY gets the record for a read-only operation, and the format is also the same as GET. The format is:

```
GET channel#,{WAIT'RECORD,}ISAM'KEY(knum) rel kval,varlist
```

where:

| | |
|-------------|--|
| channel# | Channel number from the OPEN statement |
| WAIT'RECORD | Wait until record is available |
| knum | The key specifier assigned when you created the file |
| rel | Search key relational operator between key and record (=, >, etc.) |
| kval | Value of the key to search for |
| varlist | List of variables to store the record in |

The status values returned in the status variable (defined in the OPEN statement) are defined as:

| | | |
|---------|----|-------------------------------|
| ISAM'NF | -3 | Key not found |
| ISAM'NA | -2 | Record not available (locked) |
| ISAM'LT | -1 | Found key with lesser value |
| ISAM'EQ | 0 | Found key with equal value |
| ISAM'GT | 1 | Found key with greater value |

If the key is found, the relative record number variable is updated. For example:

```
GET #3, WAIT'RECORD, KEY(0) >= "ABCD", varlist
```

gets the record whose primary key is ABCD from the file opened on channel 3 and stores it in `varlist`. If no record in the file has the key ABCD, the GET operation returns the record with the next highest key.

The status variable specified in the OPEN is set to: `ISAM'NF` if no key is found greater than or equal, `ISAM'EQ` if a key "ABCD" is found, and `ISAM'GT` if key "ABCE" or greater is found.

19.3.3[∞]GET'NEXT, GET'NEXT'LOCKED, and GET'NEXT'READ'ONLY

GET'NEXT performs sequential access. It reads the next record, in key order, into the record buffer. GET'NEXT'LOCKED locks the record for update and has the same syntax as GET'NEXT. GET'NEXT'READ'ONLY gets the record for a read-only operation, and the format is also the same as GET'NEXT. The key number used is the last GET or FIND operation. The format is:

```
GET'NEXT #file-channel, {WAIT'RECORD,}varlist
```

GET'NEXT, GET'PREV, and their LOCKED and READ'ONLY variants return these status values in the status variable:

| | | |
|---------|----|-------------------------------|
| ISAM'NF | -3 | Key not found |
| ISAM'NA | -2 | Record not available (locked) |
| ISAM'EQ | 0 | Key found |

If the key is found, the relative record number variable is updated.

19.3.4[∞]GET'PREV, GET'PREV'LOCKED and GET'PREV'READ'ONLY

GET'PREV performs reverse sequential access. GET'PREV'LOCKED locks the record for update and has the same syntax as GET'PREV. GET'PREV'READ'ONLY gets the record for a read-only operation, and the format is also the same as GET'PREV. It reads the previous record, in key order, into the record buffer. The key number used is from the last GET or FIND. The format is:

```
GET'PREV #file-channel, {WAIT'RECORD,}varlist
```

19.3.5^{oo}FIND, FIND'NEXT, and FIND'PREV

These statements are similar to GET and GET'NEXT statements. The difference is FIND does not transfer data from the file to memory. It positions the current record pointer so a GET or GET'NEXT retrieves the record specified by the FIND, FIND'NEXT or FIND'PREV statement. The FIND statements may also be used to obtain the relative record number associated with a given key. The format is:

```
FIND #file-channel,ISAM'KEY(knum) rel kval, retkey
FIND'NEXT #file-channel,retkey
FIND'PREV #file-channel,retkey
```

See the GET section above for definitions of the elements of the statement.

The argument `retkey` is the variable which contains the key of the located record. It is assumed the `retkey` variable is of the correct type and size to receive the key. The status values assigned to the file's status variable are defined as:

| | | |
|---------|----|------------------------------|
| ISAM'NF | -3 | Key not found. |
| ISAM'LT | -1 | Found key with lesser value |
| ISAM'EQ | 0 | Found key with equal value |
| ISAM'GT | 1 | Found key with greater value |

If the key is found, the relative record number variable is updated. FIND'NEXT and FIND'PREV return these status values in the status variable:

| | | |
|---------|----|----------------|
| ISAM'NF | -3 | Key not found. |
| ISAM'EQ | 0 | Key found. |

Examples:

```
FIND #1, ISAM'KEY(1) >= "JONES", FOUND'KEY
FIND'NEXT #2, NEXT'KEY
FIND'PREV #3, PREV'KEY
```

19.3.6^{oo}UPDATE'RECORD

The UPDATE'RECORD statement is used to update the file. All of the indices are automatically updated, based on the data in the file as to where each key is located within the record. The format is:

```
UPDATE'RECORD #file-channel,expression-list
```

Because the UPDATE statement updates the last record successfully obtained by a GET or located by a FIND, and because failure of a GET or FIND operation doesn't include the updating of the current relative record number, you should rigorously check the status after these operations. Failure to do so may result in the update going to the wrong record.

Updated records must be currently locked by the program. An attempt to update an unlocked record results in a trapping error. UPDATE'RECORD does not affect the lock status of a record.

Any key, including the primary, can change during an update. The key must have been read by a GET'LOCKED, GET'NEXT'LOCKED or GET'PREV'LOCKED statement to lock it before the update.

An update failure causes an error trap. The system DDB error code giving the reason for the error is in the error status variable. You can get this code using the ERF function. See "Error Processing," below.

19.3.7[∞]CREATE'RECORD

This statement creates a new record. The format is:

```
CREATE'RECORD #file-channel,expression-list
```

The new key(s) are defined by the record content. If a record with the same key exists and duplicate keys are not allowed, the CREATE'RECORD operation fails. All indices are updated on a successful CREATE'RECORD operation.

All errors trap. The system DDB error code giving the reason for the error is in the error status variable. You can get this error code using the ERF function. See the section on "Error Processing" in this chapter.

19.3.8[∞]DELETE'RECORD

The record last accessed successfully by a GET or FIND is deleted from the file. Such a record must have been locked prior to a delete. The format is:

```
DELETE'RECORD #file-channel
```

All of the indices are updated upon a successful delete. All errors trap.



Starting in RUNP version 1.0(269)-2, the default error reporting behavior of DELETE'RECORD has changed significantly. You can control this behavior using a compiler option. See the description of compatibility option 1 in Appendix J for details.

19.3.9[∞]RELEASE'RECORD

This call is used to release the last record accessed or, to release a specific record using its relative record number. The format is:

```
RELEASE'RECORD #file-channel{,record-#}
```

The optional parameter `record-#` specifies the relative record number to be released.

The default record number is the record last accessed successfully from the file, using GET or FIND. Releasing an unlocked record is NOT an error. All errors are trapped.

19.3.10[∞]RELEASE'ALL

This call releases all the locked records. The format is:

```
RELEASE'ALL #file-channel
```

RELEASE'ALL normally does not produce errors. Any error status returned by ISAMP are trapped.

19.3.11[∞]CLOSE

You must close all ISAM PLUS files opened for processing. The format is:

```
CLOSE #file-channel
```

`file-channel` is the number assigned to the file by a previous OPEN.

19.3.12[∞]CLOSEK

Closes the ISAM PLUS file, but keeps it locked so the next program in the process can access it. The format is:

```
CLOSEK #file-channel
```

The `file-channel` is the number assigned to the file by a previous OPEN. If the program was not locked when OPENed, no lock is maintained. Be sure your program unlocks the file when done.

19.3.13[∞]UNLOKR

This statement is used to explicitly unlock the last successfully accessed record. This statement is functionally equivalent to a "RELEASE'RECORD #file-channel" statement and has been included for compatibility reasons. The format is:

```
UNLOKR #file-channel
```

`file-channel` is the number assigned to the file by a previous OPEN.

19.4[∞]CREATING AN ISAM PLUS FILE

This section shows how to create indexed files from within AlphaBASIC programs using the `ALLOCATE'INDEXED` statement and the `ALLOCATE'MAP` structure. It also shows how you construct the map structure using the templates provided in the file `BP:CREATE.BP`.

19.4.1[∞]ALLOCATE'INDEXED

This statement tells ISAM PLUS to create the named file using the parameters stored in the `ALLOCATE'MAP` structure you have prepared. The format is:

```
ALLOCATE'INDEXED filespec,ALLOCATE'MAP,{device}
```

`ALLOCATE'MAP` specifies the `MAP` structure which defines the file parameters. There is a file called `CREATE.BP` in `BP: (DSK0:[7,35])`. This file can be used to make a template for creating an ISAM PLUS file structure tailored to your needs. See your *ISAM PLUS User's Guide* for more information.

`Device` is the device on which where you want the data (`.IDA`) file created. If you do not include `device`, both the index (`.IDX`) and data (`.IDA`) files are created in the current account.

19.5[∞]RETRIEVING STATISTICAL INFORMATION

This section shows how to retrieve information about an ISAM PLUS file using the `INDEXED'STATS` statement and the `STATS'MAP` structure. It also shows how you construct the map structure using the templates provided in the file `BP:STATS.BP`.

19.5.1[∞]INDEXED'STATS

There are two different syntax notations for `INDEXED'STATS`. Each causes the statistical information to be gathered in a different way. They are:

```
INDEXED'STATS #file-channel,STATS'MAP
```

```
INDEXED'STATS filespec,STATS'MAP
```

The first syntax causes the file statistics to be received from `ISAMP.SYS`. It assumes an ISAM PLUS file has been successfully opened. The second syntax causes the file stats to be received from `ISMUTL.LIT`.

The `STATS'MAP` is a `MAP` structure to receive the ISAM PLUS file stats. See your *ISAM PLUS System User's Guide* for more information on the `STATS'MAP` variables. If the memory area defined by the `STATS'MAP` is insufficiently large to hold the stats information, the information is truncated to fit. Note it is not an error for `STATS'MAP` to be too small.

19.6[∞]ERROR PROCESSING

AlphaBASIC PLUS reports errors from ISAM PLUS and FILSER (the AMOS file service system). The reporting mechanisms should help you in turn to write routines for handling errors, for terminating abnormal conditions, or for re-trying an operation.

The ISAM PLUS error codes and status values are listed in your *ISAM PLUS User's Guide*. The FILSER error codes are listed in your *AMOS Monitor Calls Manual*.

CHAPTER 20

USER DEFINABLE FUNCTIONS

This chapter explains how to design your own function calls and use them. It discusses:

- The advantages of defined functions
- Single line functions
- Multiple line functions
- How to use defined functions

20.1 WHY USE DEFINED FUNCTIONS?

Defining your own functions is a feature of AlphaBASIC PLUS that can be very useful in your programs. It can save space and time by replacing repetitive sections of code, and it can help you organize and clarify your programs.

The object of a defined function is to write a section of program to do a task your program must do a number of times. Once defined, the instructions in that defined section of code can be done by just "plugging in" the function name. User definable functions allow you to return a value in an expression where a GOSUB routine does not.

20.2 SINGLE LINE FUNCTIONS

The format is:

```
DEF FN{' }name{%} {$} ( {formal-parameter-list} ) = expression
```

The name of the function must begin with FN or FN' and contain valid variable-name characters. The formal-parameter-list is a list of variables separated by commas which indicate the formal parameters to the function. The expression is what computes the value of the function. For example:

```

DEF FN'Multiply(a,b) = a*b

INPUT "Input a number:",Number'Input
INPUT "Input a number to multiply it by:",Multiplier
Answer = FN'Multiply(Number'Input,Multiplier)
PRINT "The answer is:",Answer

```

The above example defines a function to multiply two numbers together. When the function name is used, the variables (in this case, Number'Input and Multiplier are "plugged into" the equation defined in the function).

20.3[∞]MULTIPLE LINE FUNCTIONS

The format is:

```

DEF FN{' }name{%}{ $ } ( {parameter-list} ) { ,local-variable-list }
      line(s) of definition
ENDFN

```

The value of the function is determined by setting the function name in an assignment statement in the body of the function (for example, FN'Total = 2). The type of value returned by the function depends on the naming convention of the variable used:

| | |
|----------|---|
| FNname | A floating point variable returning a floating point number |
| FNname% | An integer variable returning an integer |
| FNname\$ | A string variable returning a string |



A function can have only one return type modifier. DO NOT use % and \$ in combination.

The function value defaults to zero or null (if it is a string) if no value is specified. If more than one value is specified, the last one encountered is returned. You may use EXIT statements within the function body to transfer control to the ENDFN statement.

Local variables defined within the function initially have a value of zero (if numeric) or are null (if string). As with standard variables, you may append a % sign to make the function value an integer, or a \$ to make it a string.

Variables defined in a function (in the local-variable-list) are local to the function only. All variables defined in the main program can be accessed by the function.

There are a few things you **cannot** have inside a function definition:

- [∞]GOSUB statements into or out of the function.

- DIM statements
- RESUME statements into or out of the function. RESUME statements within the body of the function are okay.
- Other function definitions.
- GOTO statements into or out of the function.
- CHAIN statements



You cannot pass arrays to a function, though you can pass an individual element of an array.

Here's an example of a multi-line function definition:

```
DEF FN'fact(n)
  IF n = 0 THEN
    FN'fact = 1
  ELSE
    FN'fact = n * FN'fact(n-1)
  ENDIF
ENDFN
```

Type conversions on the value are done as needed according to the function name. For example, if the function name is FN-Convert\$ and a numeric value is returned, it's converted to a string.

If you are using FNname, the floating point variable is either IEEE or AMOS format, depending on which is set.

If your function uses local variables and does not have any parameters, you must define and call it using empty parentheses. For example:

```
DEF FN'TOTAL'FUNCTION( ),VALUE1,VALUE2
Total = FN'Total'Function()
```

20.4 LOCAL VARIABLE AND PARAMETER TYPES

Local variable and parameter types are identified just as other AlphaBASIC PLUS variables. However, a curious situation can arise if you give your local variable or parameter the same name as a variable in the main program. The local variable or parameter assumes the **type** of the main program variable (though not its value—the value of the main variable is saved until the function is done, and then restored).

This could cause a problem in a program if you're not careful. For example, if you have a variable defined as a string in the main program called HOURS, and then define a function like this:

```

DEF FN'Wages (HOURS,OVERTIME)
    FN'Wages = (HOURS + OVERTIME)
ENDFN

```

You get an unexpected result from the function, because the function takes the type for `HOURS` as string, and concatenates the value passed to `HOURS` and to `OVERTIME` instead of adding them (+ concatenates strings but adds numbers). Thus, if you passed the values 3 and 5 to the function, the answer would be 35 instead of 8.

To avoid this problem, use unique local variable and parameter names, or MAP them at the start of your program.

20.5^{oo} USING DEFINED FUNCTIONS

Functions can be called recursively (that is, it can call itself within its own definition—see the `DEF FN'fact(n)` example above). How deep depends on many factors, including what is on the current runtime stack, the runtime stack size, the number of parameters and local variables used in the function, etc. The parameters are passed by default in a call-by-value manner. See Appendix H for details on how to adjust memory allocations if you want to execute deeply recursive functions. Functions can use all local and global variables, but cannot use variables defined by other functions.

If the parameters passed to the function by the function call are a different type than those defined in the formal parameter, they are converted before being worked on.

If you pass fewer variables to the function than are defined, the unused variables of the parameter list are assigned a 0 or null value. If the number of parameters passed to the function is greater than the number of formal parameters defined in the function, the program ends and displays an error message. For example, if your function definition is `FN'TEST(a,b)` and you call it with a statement like `Var = FN'TEST(x,y,z)` you get an error.



It is not a good idea to pass multi-level MAPped statements to a function, because there is no way to access the lower-level variables within the MAP structure.

If you re-define a function (using the same function name twice after a DEF statement), you get an error message.



Be sure you don't begin any array names or variable names with `FN`, or they will be interpreted as a function call. If the `/F` switch is being used with the compiler, function calls must begin with `FN'` (this is for compatibility with programs already containing variable names beginning with `FN`).

At run-time, a function definition is a non-operative statement—therefore, you can place your definition anywhere you like within your program file, and you don't have to use GOTOs to bypass the function definition.

User definable functions are illegal as interactive mode direct statements.



You may need to adjust the size of the String Work Area in certain cases. See Appendix I for details.

20.6[∞]EXAMPLES

Here are some examples of defined functions:

```
DEF FN'Dice'Roll() = INT(((7 - 1) * RND(0)) + 1)

DO
  Loop'counter = Loop'counter + 1
  RANDOMIZE
  First'die = FN'Dice'Roll()
  Second'die = FN'Dice'Roll()
  Roll = First'die + Second'die
  PRINT "The dice roll was:";Roll
LOOP UNTIL Roll = 12

PRINT "It took";Loop'counter;"rolls to roll a 12."
```

Here's a more complex, multi-line function:

```
MAP1 Employee'Name,S,25
Employee'Number = 4

DEF FN'Wages(Hours,Salary),Over'hours,Hold'Variable
  IF Hours > 40 THEN Over'hours = (Hours - 40) : Hours = 40
  Hold'Variable = Hours * Salary
  FN'Wages = Hold'Variable + (Over'hours * (Salary * 1.5))
ENDFN

FOR I = 1 TO Employee'Number
  READ Employee'Name,Salary
  PRINT "Hours for employee ";Employee'Name;
  INPUT ": ",Hours
  Wages = FN'Wages(Hours,Salary)
  PRINT "Employee ";Employee'Name;" to be payed: $";Wages
NEXT I

DATA "Ralph Habshatz",5.10
DATA "George Carver",4.65
DATA "Susan Winters",6.35
DATA "Winston T. Murgrove",13.76
```

CHAPTER 21

THE UNIFY RELATIONAL DATA BASE MANAGEMENT SYSTEM (RDBMS) INTERFACE

The Unify Relational Data Base Management System is explained in the *UNIFY Reference Manual*. This chapter discusses:

- What is UNIFY?
- The UNIFY XCALL (a quick reference)

21.1 WHAT IS UNIFY?

UNIFY is a relational data base management system. It allows you to create and modify application systems that store and retrieve data. Using AlphaBASIC PLUS, you can access and manipulate the data in a database, using the Host Language Interface functions UNIFY provides.

21.2 UNIFY XCALL FORMAT

The general format for the call is:

```
XCALL UNIFY,function,status-variable,parameters
```

where `function` is the UNIFY function you want to use, `status-variable` is a variable to contain information passed back from UNIFY, and `parameters` are whatever data or specifications are required for that function.

All the functions are defined in the file BP:UNIFY.BPI(DSK0:[7,35]).

CHAPTER 22

THE AlphaBASIC PLUS DEBUGGER

The AlphaBASIC PLUS debugger (DB.LIT) is an interactive program that lets you debug a BASIC PLUS program at the source language level. Functions, variables and statements are referred to by the names from the original source file. It lets you set breakpoints at statements, view and change variable values, and run the program on a statement-by-statement basis.

22.1[∞]SETTING UP YOUR PROGRAM FOR DEBUGGING

To use the debugger, you must compile the source program with /D. For example:

```
COMPLP test/D 
```

This produces the .RPD file used with the debugger.

22.2[∞]CALLING THE DEBUGGER

Enter DB and the name of your file (the default extension is .RPD). For example:

```
DB test 
```

22.3[∞]WHAT YOU SEE

When you enter the debugger, you will see your program, much like if you were editing it. Some things will be different, though. For example, in IF-THEN statements where no labels exist, you may see a GOTO statement followed by a number. This is the debugger's way of marking where the program goes next (the number is a byte offset).

22.4^{oo}CONTROLLING PROGRAM EXECUTION

Here are the key sequences you can use with DB and what they do:

| KEY | CONTROL | FUNCTION |
|-------------|---------|---|
| ↓ | CTRL/J | Cursor down one line. |
| ↑ | CTRL/K | Cursor up one line. |
| NEXT SCREEN | CTRL/T | Screen down one page. |
| PREV SCREEN | CTRL/R | Display initial screen. |
| | CTRL/D | Cursor to previous label (not on current screen). |
| | CTRL/F | Cursor to previous label. |
| | CTRL/G | Executes lines from home position to current cursor location. If the cursor is on a line that has already been executed, the whole program is executed, and a breakpoint is set on that line. |
| | CTRL/X | Proceed. |
| HOME | CTRL/A | Move cursor to current program pointer. |
| | CTRL/S | Redisplay screen. |
| RETURN | CTRL/M | Executes current source line. |
| TAB | CTRL/I | Executes a subroutine, subprogram, or function from call point. |
| | CTRL/P | Toggles break point. You may have 16 breakpoints. |
| | CTRL/E | Execute from cursor to end of subroutine. |
| ESC | CTRL/ | Takes you to command mode. |
| | CTRL/\ | Toggles subroutine mode. When active, single-stepping through a program executes a subroutine as if it were one line. |

22.5^{oo}COMMAND MODE

DB has two modes: *screen mode* and *command mode*. Screen mode shows you your source code and its execution displays. Command mode, like command mode in AlphaVUE or AlphaFIX, lets you enter commands. In command mode, you see the command mode prompt: >.

Commands mode commands are case insensitive, and echo in upper case. However, variables and other references within the program being debugged are case sensitive. Command mode knows all the legal commands and all your program's variables and labels, and won't let you enter anything that would not make up a legal command—the terminal beeps anytime you press an incorrect key.

Commands which display information fill one screen at a time. The down arrow key ↓ or **CTRL**/**T** scrolls you to the next screen. **CTRL**/**R** brings you back to the first screen.

Command mode lets you edit your command mode typing using the following keys/control sequences:

| SEQUENCE | FUNCTION |
|-----------------------------------|--|
| RETURN | Executes command. If command is unique but not full, fills in the rest of the command name and executes it. If command is not complete, waits for next word. If not unique, DB selects first possibility, filled out in reduced intensity, and waits. A second RETURN executes this command, or you can type over with correct command. CTRL / W and CTRL / R let you sequence through possibilities. |
| ESC | Returns you to screen mode. |
| TAB or SPACE BAR | Like RETURN , except won't execute partial commands. It fills in partial commands and waits. |
| RUBOUT or ← | Moves cursor back one space, deleting character it encounters. |
| CTRL / R | Backs up to last command completion. |
| CTRL / U | Deletes current word of command. |
| CTRL / W | Displays in reduced intensity the first possible completion for a partial command. Each use after displays the next possible completion. |
| CTRL / Y | Deletes entire current word of the command and backs you up to the last character of the previous word. |
| CTRL / Z | Deletes entire command line. |

The things you can do in Command Mode are:

22.5.1°Print Variable Value

If you want to see the value of a variable, enter ? or PRINT followed by the variable-name. For example:

```
>? inventory'number RETURN  
1098
```



If you have a local variable inside a function you will not be able to display the value. You will either see a null value, or, if you have a global variable with the same name, the value of the global variable.

22.5.2°See Variable Information

To see information about a variable, enter @ and the variable-name. For example:

```
>@ inventory'number RETURN  
inventory'number  
MAP1 AMOS Floating point, size 6, located at 68-73
```

22.5.3°Set a Breakpoint

Creates a breakpoint at a specified label. Any time a breakpoint is encountered, the program halts and waits for a new command. You may have up to 16 breakpoints. For example:

```
>break loop1 RETURN
```

sets a breakpoint at loop1. If you don't enter a label, it displays all breakpoints set.

22.5.4°Clear a Breakpoint

Clears a breakpoint. For example:

```
>clear loop1 RETURN
```

If you don't enter a label, all breakpoints are cleared.

22.5.5°Execute AMOS Command

Executes an AMOS command. For example:

```
>AMOS TIME RETURN  
11:10:15 AM
```



There are some AMOS commands you should not use, such as MONTST, DSKANA, etc.—any command which could cause data corruption to your currently open file, or inconvenience other users. Programs that change the job environment, such as LOG, should also be avoided. Also, DO NOT erase the file you are currently editing.

If you see a ?Memory allocation failed message, it means there wasn't enough memory to execute the AMOS command—you may want to increase your memory (see Appendix H).

22.5.6°Find a Specific Label

Locates a specified label. For example:

```
>SEARCH company RETURN
```

DB switches to the program display and positions the cursor on the label "company."

22.5.7°Trap a Verb

Defines an AlphaBASIC PLUS verb to be used as a tracepoint. Whenever the verb is executed, the debugging stops at that point. For example:

```
>TRAP if RETURN
```

You may TRAP up to 16 verbs at a time. If you don't enter a verb, it lists all TRAPs currently set.



IF, LOOP and DO loop statements are compiled into assembly code, which causes some loops to look the same to TRAP. If you TRAP an IF statement, execution stops at DO WHILE, DO UNTIL, LOOP WHILE and LOOP UNTIL. Trapping on DO also traps DO WHILE and DO UNTIL. Trapping LOOP also traps LOOP WHILE and LOOP UNTIL.

22.5.8°Turn Off TRAP

UNTRAP turns off the TRAP on the specified verb, or turns off all TRAPs if no verb is entered. For example:

```
>UNTRAP if RETURN
```

22.5.9°Send Program Display to Another Terminal

The ATTACH command re-directs the input/output display of the program to a specified terminal (if you leave off the name, your own terminal is used). This is useful so your program's screen displays don't mix with the debugger's output. For example:

```
>ATTACH term1 RETURN
```

ATTACH is not operative when you are using the AMOS command.

22.5.10°Display File Channel Information

The CHANNEL command displays information about a specified file channel number. For example:

```
>CHANNEL 2 RETURN  
Channel number: 2  
DDB display  
Filespec: INVENTORY.DAT  
D.ERR 0  
...
```

22.5.11°Print Error Value

ERR prints the value the ERR(#) function would return. For example:

```
>ERR 2 RETURN  
ERR(2) = 12
```

22.5.12°Print Information on All File Channels

FILE prints information on all currently open file channels. For example:

```
>FILE RETURN  
  
Channel number: 3      Filespec: ACCT.DAT      Open mode: OUTPUT  
Channel number: 2      Filespec: INVEN.DAT      Open mode: INPUT  
Channel number: 1      Filespec: EMPLOY.DAT      Open mode: APPEND
```

If no files are open, there is no display.

22.5.13^oDisplay Function Source Code

FUNCTION displays the source code starting at the specified function (or the first function if none is specified), and continuing until the end of the file. The format is:

```
FUNCTION [function-name]
```

22.5.14^oIndent Text

INDENT sets the number of spaces the debugger indents your program text. This is useful if your current program indentation is too close to read easily, or too deep to fit on the screen. The number can be from 1 to 10. For example:

```
>INDENT 5 RETURN
```

22.5.15^oDisplay List of Labels

LABEL displays a list of labels, starting from the specified label (or the start of the file if no label is specified) along with the label's object code offset (the number of bytes from the start of the file). Labels with a break-point set on them are displayed in reduced intensity (if your terminal supports this feature). The labels are listed in program-offset order. For example:

```
>LABEL First'List RETURN  
First'List          00001396  
Credit'Rating      00001412
```

22.5.16^oChange a Variable Value

LET sets a variable to the specified value. The format is:

```
LET variable value
```

For example:

```
>LET Credit'Balance 1034.92 RETURN
```

Do not put quotes around string values.

22.5.17°Re-Start Debugging

NEW re-initializes the debugger—the program is ready to begin again at the beginning. All of the variables are cleared, etc.

22.5.18°Run Program From Program Pointer

PROCEED or RUN runs the program starting at the current program pointer (the next step to be executed). This is useful if you have been single-stepping through the program and wish to begin conventional execution.

22.5.19°Exit Debugger

QUIT exits you out of the debugger to AMOS command level.

22.5.20°Clear Error

REMOVE clears an error condition. Execution of the program does not resume.

22.5.21°Clear Error and Resume Execution

RESUME clears an error condition and starts the execution again. You may specify a label to start the execution again. For example:

```
>RESUME FED'TAX RETURN
```

22.5.22°Set Scale Factor

SCALE sets the scale factor to the specified value, or prints the current value if no value is entered. For example:

```
>SCALE 7 RETURN
```

22.5.23°Set Significance Value

SIGNIFICANCE sets the significance to the value specified, or prints the current significance if no value is entered. For example:

```
>SIGNIFICANCE 10 RETURN
```

22.5.24°Break on Value Change

TRACE breaks execution whenever the variable specified is about to be written to. This lets you follow a variable through its changes. For example:

```
>TRACE Credit'Limit RETURN
```

TRACE may be made conditional also. That is, you can TRACE when the variable changes in a certain way. For example:

```
>TRACE Credit'Limit > 1000 RETURN
```

In this case, DB breaks the execution only if the variable Credit'Limit exceeds 1000. You can use all of the conditional expressions listed in Chapter 7.

22.5.25°Remove Tracepoint

UNTRACE turns off the TRACE on the specified AlphaBASIC PLUS variable, or turns off all TRACES if no variable is entered. For example:

```
>UNTRACE Credit'Limit RETURN
```

22.5.26°Display List of Variables

VARIABLE displays a list of variables in your program in definition order, from the specified variable. If no variable is entered, it lists all variables. TRACEd variables are displayed in reduced intensity (if your terminal supports this feature). MAPped variables are indicated, and their type, size, and current value (if not an array) are displayed.

22.5.27°Break at External Program

CHAIN causes the program to execute until the CHAIN call to the specified filename is encountered. This lets you debug the subprogram without having to step through the whole main program. Since DB executes all subprograms up to the specified filename, all external subprograms encountered must be compiled with /D. The format is:

```
CHAIN filename
```

22.5.28°Break at Subprogram

SUBPROGRAM causes the program to execute until the specified SUBPROGRAM call is found. This lets you debug a subprogram without having to step through the whole main program. Since DB executes all external subprograms up to the specified filename, all external subprograms encountered must be compiled with /D. The format is:

SUBPROGRAM filename

22.6[∞]DEBUGGING A SUBPROGRAM

The RUNP program gives you a way to run a program up to a CHAIN statement, and then debugs from that point. This means you don't have to compile all the preceeding subprograms, as with the CHAIN statement, above. The format is:

RUNP/D:subprogram-filename source-filename

DB runs the source file as RUNP would, until the subprogram is encountered, then goes into debug mode for the rest of the program. The subprogram file and any subsequent subprograms must be compiled with /D.

APPENDIX A

MESSAGES

Below is a complete list of all messages output by the AlphaBASIC PLUS system (BASICP, RUNP, and COMPLP), and a brief explanation of each message.

#####

Your program is using a PRINT USING statement, and it could not process the data given it. Check the syntax of the line against the rules for PRINT USING in Chapter 11.

?', ' or ';' expected

Adjust your code so the expected condition is fulfilled.

?']' expected

Adjust your code so the expected condition is fulfilled.

?', ' or ')' expected

You left off a comma or a parenthesis. Correct the line and try again.

?'= expected

You left out an equal sign. Correct the line and try again.

? '(' expected , ?')' expected

You left out a parenthesis. Correct the line and try again.

?Array expected

Adjust your code so the expected condition is fulfilled.

?ARRAYS cannot have initializers

Check your code and the format for the array and try again.

?Bad format in conditional compilation directive

A ++IF, ++IFDEF or ++IFNDEF line is not in the correct format. See section 5.3 for the correct format.

?Bad ++PRAGMA option

The word after ++PRAGMA is not a legal option. See section 5.4 for the valid options

?Bad ++PRAGMA option value

The value you were setting for the option is illegal. Each option has its own legal range of values. See section 5.4 for details

?Bad option set number 'n' in /PXn option

The value on 'n' you supplied is invalid. See Appendix I for details.

?Bad symbolic key variable (not mapped?) in ISAM code 7

The symbolic key you specified in an ISAM code 7 statement is not a simple variable. It must be mapped, non-subscripted, not a parameter, and not an expression (for example, a substring or concatenation).

?Bitmap kaput

Your program attempted a file operation (OPEN, ALLOCATE, etc.) on a device with a bad bitmap.

Break at line n

The program reached the breakpoint set at line n.

?/C definition is invalid. Use '/C:<name>=<value>'

You attempted to define a constant to the compiler on the command line, but the syntax is wrong. Check Section 5.3 for the correct syntax.

?Cannot change option - some code already compiled

You issued a ++PRAGMA too far down in the source code. Some ++PRAGMAs must appear at the top of the file before any object code has been output by the compiler. Move the ++PRAGMA to the top of the file, and check section 5.4 for more details.

?Cannot find [program-name]

Check your spelling, and/or check the directory to see if the file exists. Make sure the program specified is in the proper account.

? Cannot have DIM() within a function definition

You placed a DIM statement within a user function definition. This is illegal, as it will produce a memory leak and a runtime error 30, Redimensioned Array, if the user defined function is executed more than once.

?Cannot load ISAMP.SYS

See your System Operator about locating the ISAMP.SYS file.

?Cannot open input file

Make sure the file exists, and you have access to it, and try again.

?Cannot open output file

Make sure the file exists, and you have access to it, and try again.

?Cannot pass single value to subprogram array

You have tried to pass a non-array variable as a parameter to a subprogram, where in the subprogram that parameter is defined as an array.

?Cannot run .RP with DIM statements from RES: or MEM:

.RP or .SPG files produced by early versions of COMPLP did not allow you to load them into system or user memory if they contained DIM statements. Later versions of RUNP detect such .RP or .SPG files if they are loaded into system or user memory, and produce this runtime error. Recompile the source file under COMPLP version 1.0(26) or later, and execute it under RUNP version 1.0(267) or later.

?Cannot use CHAIN in a function

You put a CHAIN statement within a user defined function. This will produce a memory leak, and is forbidden. You must return out of all subprograms or user defined functions (via ENDFN statements) to the main program, and CHAIN from there.

?Cannot use CHAIN in a subprogram

You put a CHAIN statement within a subprogram. This will produce a memory leak, and is forbidden. You must return out of all subprograms (via SUBEND statements) to the main program, and CHAIN from there.

?Can't assign a value to a constant

Use a variable.

?Can't resolve constant in ++IF

A constant in the ++IF line has not been DEFINED or passed as a command line constant definition. You must DEFINE such a constant before the ++IF statement.

?Can't continue

You have attempted to continue a program which is not stopped at a breakpoint, or which has reached a point where it can go no further (for example, it has reached an END statement).

?Can't open file

BASICP couldn't find the file you tried to load. Check your syntax and/or your file directory.

?CASE, DEFAULT, or ENDSWITCH without SWITCH

Add a SWITCH statement.

?Channel expected

Adjust your code so the expected condition is fulfilled.

?Channel number in use

Use a different file channel number.

?Channel #0 is illegal for random files

You may not specify file channel #0 when accessing a random file. Change the file channel number to a positive integer, or use a sequential file (channel #0 equals your terminal if you are using sequential files).

?Character translation table not found

The file you specified in COMPLP's command line cannot be located.

?COMPLP does not support the requested /P option

You specified an option number that this version of COMPLP does not support. Use a later version of COMPLP.

?Constant definition buffer overflow: definition ignored

You have specified one or more constants on COMPLP's command line, but the length of COMPLP's internal buffer for storage is too small for all of the definition(s). Reduce the size and/or the number of definitions. A space takes up one byte of internal buffer storage.

?Constant expected

Adjust your code so the expected condition is fulfilled.

Copying from [program-name]

The program you are compiling contains an `++INCLUDE` command. This message is displayed as the file specified in the `++INCLUDE` command is copied into your program.

?Deadly embrace possible

Your program has opened files out of sequence with their order on the File Locking data base. This message only appears if the deadlock check indicators are set in the File Locking data base.

DELETE what?

You specified `DELETE` without specifying what line(s) are to be deleted.

?Device does not exist

The device you specified in a file operation (`OPEN`, `LOOKUP`, etc.) does not exist. Make sure your entry was typed correctly.

?Device error

An error has occurred on the referenced device. Check the documentation for that device to determine how to correct the error before rerunning the program.

?Device full

The specified device has run out of room during a `PRINT`, `CLOSE`, or `ALLOCATE` operation. Remember an `ALLOCATE` requires contiguous disk space, so this error may occur when there are still a number of non-contiguous blocks available.

?Device in use

The specified device is currently assigned to another user. Wait until that user is finished before running your program.

?Device is not file structured

The device you were trying to access is not an accessible device. Check your specification for spelling, or talk to your System Operator to find out what devices are accessible on your system.

?Device not ready

The specified disk is not ready for use. Check your device to see if the appropriate buttons are pushed/plugs are in, etc.

?Disk not mounted

The specified disk has not been mounted. Mount it by using the MOUNT monitor command or by using the XMOUNT subroutine.

?Division by zero

Your program attempted to perform a division by zero. Check the values that went into the equation. You may want to install a check for zero values, or use an error trapping routine to catch zero divisions.

?Duplicate label

Your program has defined the same label name more than once. Search the .BP file for the duplicate labels, and change one of them to a unique name.

?Duplicate line number

Change one of the duplicate line numbers.

?Duplicate SUBPROGRAM name

Change one of the SUBPROGRAM names.

***** End of Program *****

You have reached the end of the program during single-stepping in Interactive Mode. If you press the line feed key again, your program restarts at the beginning.

?ENDFN without DEF

Add an ENDFN statement to finish the function definition.

Enter <CR> to continue:

You have reached a STOP statement in your program. You may continue from the STOP statement by pressing **RETURN**, or you may abort the run by pressing **CTRL/C**.

?Error during error trapping

An error occurred while you were in the error trapping routine. Check your routine for incorrect statements or mis-spellings.

?Error writing to output file

Make sure the disk is not full. If it isn't, something went wrong when the compiler tried to create the output file on the disk. Report the problem and the circumstances to your System Operator or your Alpha Microsystems representative for help.

?File already exists

Your program tried to create a file which already exists. You may not need the section of code that creates the file, or you may need to install a check using LOOKUP to determine if the file exists before you try to create it.

?File already open

You have attempted to open a file on a channel number that is in use. You may not need that OPEN statement, or use a different channel number.

?File cannot be DELETED

Your program tried to delete a file that has File Locking DELETE protection. If you really do want to delete the file, change its status under File Locking first.

?File cannot be RENAMEd

Your program tried to rename a file that has File Locking RENAME protection. If you really do want to rename the file, change its status under File Locking first.

?File has been opened exclusively

A file that COMPLP wants to process, or RUNP wishes to execute, has been opened exclusively by another job on the system. Wait and try again later.

?File in use

The file your program wants to access is already locked by another user. You may want to use the WAIT'FILE option with the file access statement.

?File Locking queue is full

Your program attempted to lock a record or file, and there was no room in the File Locking queue to hold it. You may want to either delete some of the records/files in the File Locking queue, or increase the size of the queue. See your *File Locking System User's Manual*.

?File not found

Check your specification, and your directory. Make sure the account is correct.

?File not open

Your program attempted to access an unOPENed file. Have your program open the file before access is attempted.

?File specification error

The file specification you gave in a file operation (OPEN, LOOKUP, etc.) is in error. All file specifications must conform to the system standard (devn:filename.extension[p,pn]).

?File type mismatch

Your program tried to perform a sequential operation on a random file or vice-versa. Make sure you are using the correct files, and using the correct file statements.

?First logical unit not mounted

The first logical unit on the drive your program was trying to access was not mounted. Mount the unit and try your program again.

?Function definitions cannot be nested

Move the inner function definition out of the outer one.

?Function undefined

You referred to a function that doesn't exist. Check your spelling or define the function.

?GOTO expected

Adjust your code so the expected condition is fulfilled

?GOTO or GOSUB expected

Adjust your code so the expected condition is fulfilled.

?Illegal channel number

Channel numbers must be in the range 0-65535.

?Illegal command line

Check your syntax and try again.

?Illegal expression

The specified expression is not valid. Check your syntax and spelling.

?Illegal function name

Check your syntax and/or change the name.

?Illegal function value

The specified value is not valid for the particular function. See the instructions for that function to see what type and number of arguments it takes, and the valid values for its arguments.

?Illegal GOTO or GOSUB

The format of the GOTO or GOSUB statement is invalid. Check the label or line number it refers to.

?Illegal IF construct

Check the format of the IF statement and try again.

?Illegal in immediate mode

You cannot use whatever you entered in interactive BASIC, only in a program file.

?Illegal ISAM sequence

The sequence of commands used for the ISAM file were executed in the wrong order for the File Locking system.

Illegal key number

You have specified an illegal key number (one that does not exist) in an ISAM PLUS statement

?Illegal line number [number]

The specified line number is invalid. Line numbers must be positive integers less than 65,535.

?Illegal MAP level

Correct the MAP level to a legal number, 1-16.

?Illegal NEXT variable

The variable in the NEXT statement is invalid— it does not equal the variable in any previous FOR statement. Check your variable.

?Illegal or undefined variable in overlay

The variable specified in a MAP statement overlay (preceded by @) has not been previously defined, or is not a mapped variable. Check your spelling. You may want to define a MAP statement for the variable, or remove the @.

?Illegal PRINT USING format

The edit format used in a PRINT USING statement is invalid.

?Illegal PROGRAM argument

Make sure you entered the argument correctly, and the numbers are within the parameters allowed by the PROGRAM statement.

?Illegal record number

The relative record number specified in a random file processing statement (READ or WRITE) is either less than the current FILEBASE or outside of the file. Or, the record size of a contiguous file and the relative record number do not match the physical size of the file on disk.

?Illegal record size

The record size specified in a random file processing statement (READ or WRITE) is outside the range the file was OPENed for. Check your OPEN statement.

?Illegal REPEAT or EXIT usage

Check your syntax and take out or re-locate the statement.

?Illegal SCALE argument

The argument given in a SCALE statement is invalid. The argument must range between -35 and +35. Check your argument.

?Illegal size for variable type

The specified variable size is not valid for the particular variable type. AMOS floating point variables must be size 6, binary variables must have sizes 1 through 5, integers must have size 1, 2, or 4, and IEEE variables have size 4 or 8.

?Illegal STRSIZ argument

Check your syntax and the number you specified. The argument must be less than 65,535.

?Illegal subroutine name

The name specified as a subroutine is not valid. Names used must be legal filenames with a .SBR or.XBR extension.

?Illegal subscript

The subscript expression is not valid. Subscripts must be positive, whole numbers.

?Illegal TAB format

Your program incorrectly specified a TAB function. Check the syntax, and check to see there are not more than two numbers in the parenthesis.

?Illegal token

Contact Alpha Microsystems and report the problem and how it happened.

?Illegal type

The type of the variable you specified is inconsistent with the function you tried to use it with. Check the syntax of the function or statement and try again.

?Illegal type code

The variable type code used in a MAP statement is not valid. Check your number against the legal values allowed.

?Illegal user code

The specified PPN was not found on the specified device, or is not in a valid format. If you get this message while using File Locking, it may mean your program tried to access or lock a resource in a protected account.

?Illegally defined constant

Check the rules for constants, and re-define the constant.

?Inappropriate file IO

You have tried to access a random (contiguous) file by using sequential file commands, or vice versa. Check the file type and your OPEN statement for the file, and correct the file IO statements for that channel number.

Initializer in SUBPROGRAM parameter ignored

You specified an initializer value in a PARAMETER statement or in a MAP statement in a subprogram contained in a ++INCLUDE <filename>/P statement. Subprogram parameters cannot be initialized or have a default value assigned. They will receive their values from the SUBCALL in the calling program.

?Insufficient memory to load [program-name]

The RUNP program did not find enough free memory to be able to load the specified program. See your System Operator about increasing your memory, and/or see Appendix C for hints on making your programs more memory efficient.

?Integer expected

Adjust your code so the expected condition is fulfilled.

?Internal compiler error number [number]

Contact Alpha Microsystems and report the problem and how it happened.

?Internal ++PRAGMA error

An internal compiler error was generated when processing the current statement. Contact your dealer or Alpha Microsystem Technical Support, and have the offending source file and statement ready.

?Invalid filename

Make sure the filename contained six or less characters, and has a valid extension.

?Invalid number or illegal format for /P list

COMPLP cannot understand your option selection list for a /P or /PX switch.

?Invalid subroutine version

The subroutine specified in the XCALL statement is not in the correct format for the processor you are running on. You may have to rewrite or re-assemble the subroutine.

?Invalid syntax code

Check the format for the statement you used and try again.

?Invalid token

Contact Alpha Microsystems and report the problem and how it happened.

?IO to unopened file channel

The program tried to write to or from an unOPENed file. Check your code and make sure the file is OPEN before you try to use it, and check the file's channel number.

?ISAM'KEY expected

Adjust your code so the expected condition is fulfilled.

?Keyword used as variable

Change the keyword to a variable name.

?Label is out of scope

You tried to transfer to a label within a function or loop from outside. Revise the instruction.

?Label or line number out of scope

You tried to transfer to a label within a function or loop from outside. Revise the instruction.

?Label undefined

You referred to a label that doesn't exist in your program. Check your syntax or define the label.

?Line number is out of scope

Use a smaller line number.

?Line number must be 1 to 65535

Change the line number.

?Line [number] not found

The specified line was not found for a DELETE, LIST, etc., operation.

?Line number undefined

You referred to a line that doesn't exist in your program. Check your syntax or create the line.

LOKSER queue is full

A system resource, the LOKSER queue, is full. Contact your System Administrator and request the LOKSER queue be expanded.

?LOOP without DO

Add a DO statement to make a complete loop.

?MAP 1 S or X type max size of 64k exceeded

A MAP 1 variable cannot be greater than 65,535 bytes in size. Adjust your MAP statements to fulfill this condition.

?Mismatched FOR variable

The variable you used in FOR isn't the same type as what you are comparing it to. Change the statement so the variables have the same type.

?Missing colon after /C switch

You tried to specify a constant on COMPLP's command line, but did not put a colon after the /C and before the constant's name. Try again with the colon inserted.

?Missing colon after /P switch

You tried to specify a constant on COMPLP's command line, but did not put a colon after the /P and /PX switches, and before the constant's name. Try again with the colon inserted.

?Missing comma in /P switch

You tried to list on or more options for a /P switch, but left out a comma between option numbers.

?Missing [statement-type] statement - [text line]

Check the format of the statement and try again.

?Multiple PROGRAM statements

Only one PROGRAM statement is allowed per program. Remove any extra PROGRAM statements.

?Multiply defined function

You defined the same function twice. Change or remove one of the function definitions.

?Nested ++INCLUDE files are not permitted

Your program contains an ++INCLUDE command specifying a file which also contains an ++INCLUDE command. You either have to remove the ++INCLUDE command from the second file, or change the way you access that code.

?NEXT without FOR

A NEXT statement was encountered without a matching FOR statement. Check your program for the extra NEXT statement.

?No breakpoints set

This message just informs you there are currently no breakpoints set in your program.

?No source program in text buffer

You tried to compile when there was no program in memory. Load or create a program before trying to compile again.

?Not enough memory to initialize

See your System Operator about freeing up more of your user memory partition, or about increasing it.

?Not enough memory to load XCALL routine

RUNP could not locate an XCALL in system or user memory. It tried to dynamically load the XCALL from disk into the reserved free memory area at the top of the job's partition, but there was insufficient memory to do so. Either load the XCALL into system or user memory before execution, or adjust the memory allocation as described in Appendix H.

?Numeric overflow

A floating point overflow occurred during a calculation. This means the calculation went beyond the capabilities of the computer in some way (usually the number was too large). Check your calculation.

?Only one subprogram in a subprogram file

Place the extra subprogram(s) in separate files.

?Only one SUBPROGRAM of a main program allowed in BASICP

Use only one SUBPROGRAM.

?Operand expected

Adjust your code so the expected condition is fulfilled.

?Operator interrupt

This message appears when you use Control-C to interrupt a program run.

?Out of data

Your program tried to READ data after the data in all of the DATA statements had been used. You have to add more data to your DATA statements, execute a RESTORE statement, or examine your program to determine why it is doing too many READ operations (if the amount of your data is correct).

?Out of memory

See your System Operator about allocating more memory to your job, and/or see Appendix C about making your program more memory efficient. Appendix H contains details about altering certain RUNP memory allocations.

?Out of memory - Compilation aborted

COMPLP is telling you it does not have enough free memory to finish compiling your program. See your System Operator about allocating more memory to your job, and/or check your memory partition to see if you have any programs there that can be removed.

?PARAMETER statements can only follow SUBPROGRAM statements

Place the PARAMETER statement right below the SUBPROGRAM statement.

?PPN not found

Your program tried to access a file in a non-existent account. Check the syntax of your file specification. Make sure the correct disk and account have been given.

Program name:

You tried to SAVE or LOAD a program without providing a filename. Enter the filename at this point.

?Protection violation

Your program tried to write into another account where you do not have write privileges. Write to a different account, or talk to your System Operator about getting access to the account you want.

?Record in use

The random file record your program wants to access is already locked by another user. Only for File Locking systems. You may want to specify WAIT'RECORD in your file access statement.

?Record not locked

Your program tried to update a random file record it had not first locked with a READL instruction. Only for File Locking systems.

?Record size overflow

Your program tried to read a file record into a variable larger than the file record size. Re-adjust your variable size, or your record size.

?Redimensioned array

You tried to redimension an array. An array can only be dimensioned once in any one AlphaBASIC PLUS program. Remove any extra DIM statements for that array, and prevent each DIM statement from being executed more than once.

?Relational operator expected

Adjust your code so the expected condition is fulfilled.

?Remote not responding

Your program tried to access a file on a remote system, and that system was not open for access. Establish a connection with the remote system and try your program again.

?RESUME cannot be used within functions

Use a different method, such as a GOTO, to redirect.

?RESUME without error

A RESUME was encountered, but no error has occurred. Your program is probably running into your error trapping routine by accident. Make sure a GOTO or END or similar statement stops or redirects your program before the error routine.

?RETURN without GOSUB

A RETURN was encountered, but no corresponding GOSUB was executed. Your program is probably running into your subroutine by accident. Make sure a GOTO or END or similar statement stops or redirects your program before the subroutine.

?RUNP cannot execute this file

The object file needs more features than RUNP has. The object file was created with a set of options specified by /P in COMPLP. This version of RUNP does not support all the requested options. Use a later version of RUNP.

?RUNP file is in an incompatible format.

The program file you tried to RUNP is not a .RP file, is from a different processor type, or was created from a much earlier, incompatible version of COMPLP. You may have to rewrite or edit your source file and/or re-compile it.

?Significance out of range

The value for SIGNIFICANCE must be between 1 and 15.

?Source code without line numbers

Programs in interactive mode must have line numbers—add them or run the program at AMOS level with RUNP.

?Source line overflow

A line in the source program, including continuation lines, exceeds 500 characters. Check your code for the excessive line, and see if you can break it into multiple statements, or reduce its length.

?Stack overflow

This may be caused by nesting GOSUBs too deep, or branching out of FOR-NEXT loops. Check your code or increase the STACK value in your .BPR file. See Appendix H for details.

?String work area exceeded

RUNP requires more string work area than COMPLP assigned. You need to adjust the size of the string work area by using ++PRAGMA commands as explained in Appendix I.

?SUBPROGRAM name must be of string type

Change the name to a string.

?Subprogram name too large or null

Adjust the name.

?Subroutine not found

The specified subroutine could not be found. Check your spelling and your directory.

?Subscript out of range

The specified subscript is outside the range specified in the DIM or MAP statement for the subscripted variable. Check your subscript variable.

?Syntax error

The syntax of the specified line is invalid. Check your spelling.

?System error

This is used as a catch-all error message indicating AlphaBASIC PLUS can't identify the exact problem during the execution of the specified line. For example, if AlphaBASIC PLUS encounters a "Buffer not INITed" error message, it displays "System error," because it doesn't know how to handle this condition. Determine which line of your code is causing the problem, and correct it.

?TO expected**?TO or SUB expected**

Adjust your code so the expected condition is fulfilled.

?Too many arguments to function

Check the function format, and reduce the number of arguments.

?Too many errors

The compiler reported its limit of errors. When you have fixed the ones displayed, re-compile for possible further messages.

?Too many files open

There are too many files open at this point. The program needs to be modified.

?Too many parameters

You have passed too many parameters to a system-defined or user-defined function.

?Too many parameters to function

You have passed too many parameters to a system-defined or user-defined function.

?Transfer address out of range

Try again, using the /A switch with COMPLP.

?Translation table not found in memory

The required translation table was not found in memory. Locate it, load it into system or user memory, and rerun the program.

?Unable to find ++INCLUDE file [filespec]

Make sure you used the correct name, or make sure the proper file exists or is created.

?Unbalanced parenthesis

You are missing a (or a). Add the appropriate parenthesis.

?Undefined array in XCALL

You have specified an array to an XCALL which has not yet been allocated in memory. This means that either (1) the array was not MAPed, (2) the array has not been DIMed, or (3) the array has not been referenced before the XCALL, which would cause an automatic allocation of an array of ten elements.

?Undefined line number or label

The line number or label specified in a GOTO, GOSUB or ON ERROR GOTO is not defined in the program. Correct the reference, or add the proper line number or label.

?Unknown switch

See the list of available switches, and try again.

?Unsupported feature**?Unsupported function**

Your program attempted to use a feature not supported by AlphaBASIC PLUS. Check this manual for the features AlphaBASIC PLUS supports.

?Unmapped variable

You compiled a program using the /M option, and COMPLP encountered an unmapped variable. MAP the variable, or do not use the /M option.

?Unterminated string

Add a closing quote mark to the string definition.

?Variable defined more than once

Re-name or remove one of the duplicate definitions.

?Variable expected

Adjust your code so the expected condition is fulfilled.

?Variable must be on an even byte boundary

A variable is being used in a way that demands that it is located on an even byte boundary, but instead it is located on an odd byte boundary in memory. Adjust your MAP statements to force placement on an even byte boundary, or define the variable as a MAP1 variable, which forces even boundary alignment.

?WHILE or UNTIL expected

Adjust your code so the expected condition is fulfilled.

?Write protected

Your program tried to write to a write-protected device. Before running the program again, turn off the device's write-protection.

?Wrong number of subscripts

The number of subscripts specified is not the same as the number defined for the subscripted variable. Check your code.

?Setting size of string work area below calculated

For setting size of string work area below calculated, COMPLP calculated the required size of the string work area based on worst case conditions. You have used a ++PRAGMA to set the size below this calculated value. COMPLP is just warning you of this.

?[pragma option] on line [number] overridden by compiler switch

[pragma option] on line [number] overridden by compiler switch You have specified a COMPLP command line switch which matches a ++PRAGMA option in the source code. The command line switch takes precedence and overrides the ++PRAGMA setting. COMPLP is warning you of this.

APPENDIX B

RESERVED WORDS

Below is a list of the reserved words used by the BASIC compiler. Some of these reserved words may designate routines that have not been implemented at this time. However, you must not use ANY of these reserved words as variable names. This restriction applies to string variables as well as numeric variables (i.e., END\$ and END are both illegal variable names).

| | | | |
|-------------------------|----------------|------------------------|--------------------|
| ABS | ACS | ALLOCATE | ALLOCATE'INDEXED |
| AMOS | AND | APPEND | ASC |
| ASN | ATN | BASICP | BREAK |
| BYE | BYTE | CALL | CASE |
| CHAIN | CHR | CLOSE | CLOSEK |
| CMDLIN | COMPILE | COMPLP | CONT |
| COS | CTRLC | CREATE'RECORD | DATA |
| DATE | DATN | DEF | DEFAULT |
| DEFINE | DELETE | DELETE'RECORD | DIM |
| DITOS | DIV | DIVIDE'BY'0 | DO |
| DSTOI | ECHO | EDIT | ELSE |
| END | ENDIF | END'FILE | ENDFN |
| ENDSELECT | ENDSWITCH | EOF | EQV |
| ERF | ERR | ERROR | ERRMSG |
| EXIT | EXP | EXPAND | FACT |
| FILEBASE | FILEBLOCK | FILL | FIND |
| FIND'NEXT | FIND'PREV | FIX | FMOD |
| FN | FN' | FOR | GET |
| GETKEY | GETLOCK | GET'NEXT | GET'NEXT'LOCKED |
| GET'NEXT'READ'ONLY | GET'PREV | GET'PREV'LOCKED | GET'PREV'READ'ONLY |
| GET'READ'ONLY | GO | GOSUB | GOTO |
| IF | INDEXED | INDEXED'EXCLUSIVE | INDEXED'STATS |
| INPUT | INSTR | INT | IO |
| ISAM | ISAM'INDEXED | ISAM'INDEXED'EXCLUSIVE | ISAMP'INDEXED |
| ISAMP'INDEXED'EXCLUSIVE | KILL | LCS | LEFT |
| LEN | LET | LINE | LIST |
| LOAD | LOCK | LOG | LOG10 |
| LONG | LOOKUP | LOOP | MAP |
| MAX | MEM | MID | MIN |
| MOD | NEW | NEXT | NO'DIVIDE'BY'ZERO |
| NOECHO | NOEXPAND | NOT | ODTIM |
| ON | OPEN | OR | OUTPUT |
| PARAMETER | PRINT | PROGRAM | RANDOM'FORCED |
| RANDOM | RANDOMIZE | RAW | READ |
| READ'ONLY | READ'READ'ONLY | READL | RELEASE'RECORD |
| RELEASE'ALL | REM | RENAME | REPEAT |
| RESTORE | RESUME | RETURN | RIGHT |
| RND | RNDN | RUN | RUNP |
| SAVE | SCALE | SELECT | SGN |
| SIGNIFICANCE | SIN | SLEEP | |

SPACE
STEP
STRSIZ
SUBPROGRAM
TAN
UNLOKR
VAL
WHILE
WRITEL

SPAN'BLOCKS
STOP
SUB
SWITCH
THEN
UNTIL
VER
WORD
WRITEN

SQR
STR
SUBCALL
SYSCALL
TIME
UPDATE'RECORD
WAIT'FILE
WRITE
XCALL

STATS'MAP
STRIP
SUBEND
TAB
TO
USING
WAIT'RECORD
WRITEL
XOR

APPENDIX C

PROGRAMMING HINTS

In this appendix, we explore a few ways to make your programs more time and memory efficient. We discuss:

- Managing memory
- Increasing execution speed
- Increasing "readability" of program source files

C.1 MANAGING MEMORY

AlphaBASIC PLUS uses memory to store the variables you use within your program. It also uses memory space for calculations. Here are some hints for reducing your program's use of memory:

- When you allocate arrays and/or strings, give them only as much memory they need, and no more. Allocating an array to 50 when you only need 10 subscripts wastes a lot of memory.
- Use MAP statements if you have a large number of strings varying in size. By doing this, you can allocate each string individually, rather than setting STRSIZ to the size of the longest variable and wasting space for all the shorter variables. MAP statements generally help reduce memory use.
- Use as few variables as possible. Using one "scratch" variable that can be reset as needed instead of a number of individual variables can help.
- Do not rewrite sections of code that are used more than once. If you have ten lines of code used in more than one place, make those lines a subroutine, or user defined function. The more lines in the program, the more memory used. An additional benefit is there is only one routine to modify if changes become necessary.
- Where possible, MAP variables into the same storage area using the origin parameter (see Chapter 14). However, be careful not to forget about the multiple use of the same area.

- If you are using line numbers, do not assign a line number to lines containing MAP statements or lines having only comments or labels on them.
- Saving memory is generally good, but sometimes using a little extra memory gives you a simpler and easier to maintain program.

C.2 INCREASING EXECUTION SPEED

Many of the suggestions in the section above also improve execution speed as well as saving memory. Some other hints for improving program execution:

- Use integer variables in arithmetic operations, if possible. They are faster than floating point variables. Binary variables are slower than floating point, so, if you can't use integers, use floating point.
- Use the XCALL subroutines if you can, rather than writing program code. Assembly language routines are faster.

C.3 MAKING YOUR SOURCE PROGRAMS EASIER TO READ

One of the considerations you should take into account when you write a program is—what happens if somebody else has to update the program? Someday, someone may have to go into your source file (the .BP file) and make corrections and/or additions to it. If you are not around to explain your code, can that person understand it? Can that person find the section he or she needs to work on?

These questions are becoming increasingly important in the computer world. A few minutes spent making your source file clear and easy to read may save many hours of work for someone at a later time, and helps to minimize errors. It is also easier for you, because the program is easier to debug, and easier to modify later.

You should practice writing your programs in good program form, so your programs are always easy to understand and follow. It may take a few extra minutes when writing, but it saves much more time over the lifetime of the program!

Let's look at some examples. Which of these program lists is easier to read?

```

11 INPUT "INPUT THREE NUMBERS: ",A,B,C
13 D=(A+B+C)/3
20 ?"ANSWER: ",D
33 ? :INPUT "Do you want to try again?",E$
40 If E$="Y" then goto 11
43 END

```

or:

```
! This program takes three numbers as input and
! averages them:

MAP1  ANSWER,F
MAP1  FIRST,F
MAP1  SECOND,F
MAP1  THIRD,F
MAP1  QUERY,S,1

INPUT'NUMBERS:

      INPUT "Input three numbers: ", FIRST, SECOND, THIRD

CALCULATE'ANSWER:

      ANSWER = (FIRST + SECOND + THIRD)/3
      PRINT
      PRINT "The average is: ", ANSWER
      PRINT

DO'IT'AGAIN:

      PRINT "Would you like to average ";
      INPUT "more numbers? (Y or N): ", QUERY
      IF UCS(QUERY) = "Y" THEN GOTO INPUT'NUMBERS

END
```

Here are some tips, things the second program used the first did not:

- If possible, do not use line numbers. They make your program harder to read and update. If you want to insert lines, you may have to renumber the rest of the program. Without line numbers, you can also more clearly indent the program lines to show levels and structures.
- Use descriptive variable names (ANSWER instead of D, for example) and labels. This makes the program easier to understand.
- Do what you can to make the program easy for the user. The second program used UCS on QUERY before comparing it to "Y". In the first program, if the user answers the question with a lower case y, the program ends! Good programs don't allow a user to enter incorrect responses.
- Use spaces to make lines easier to read. In most cases, AlphaBASIC PLUS ignores blank spaces, so you can use them to make text more readable. For example:

```
ANSWER = (FIRST + SECOND + THIRD) / 3
```

instead of:

```
13 D = (A+B+C) / 3
```

- Use MAP statements so all variables are defined at the front of the program. This is especially useful in large programs.

The programs above were too simple to show the use of some ideas. We have tried to use examples throughout this book written in structured, clear code, both to make our examples easier to read and to encourage good program style. Here are a few more things you may want to keep in mind:

- Whenever possible, group code into sections by using subprograms, functions, and subroutines. Your program then becomes a series of sub-programs, each accomplishing a task. Organizing your program in this manner also makes it easier to design and write your program.
- Use tabs or regular spacing to indent sections of program code so a reader can see the structure. For example:

```
FOR I = 1 TO 10
  FOR J = 1 TO 20
    K = J + 1
  NEXT J
  K = I + J
NEXT I
```

- Don't use more than one statement per line.
- Use comments to explain lines that may be unclear, and especially to explain what each routine, subprogram, function, etc. do.
- Use blank lines and comments to visually set off independent sections of code (subroutines, functions, loops, etc.).
- Establish programming standards within your group or company, so all programs follow the same conventions.

APPENDIX D

SCREEN HANDLING CODES

Below is a list of the terminal handling codes (TCRT codes) established at the time of the printing of this manual. For the most up to date list, see your *AMOS Terminal System Programmer's Manual*. Chapter 12 explains how these codes can be used by AlphaBASIC PLUS programs.



All TCRT codes are reserved for future expansion by Alpha Micro. If you have a specific need for a new TCRT function, you may contact the Advanced Product Development group at Alpha Micro to reserve a TCRT code number. By using this reservation process, problems of conflicting and incompatible TCRT functions are avoided.

Calls marked with an asterisk are either obsolete calls or calls for special purposes. We recommend they not be used, as they may change in the future.

| TCRT CODE | FUNCTION |
|-----------|---|
| 0 | Clear Screen and set normal intensity |
| 1 | Cursor Home (move to 1,1) |
| 2 | Cursor Return (move to column 1 without linefeed) |
| 3 | Cursor Up one row |
| 4 | Cursor Down one row |
| 5 | Cursor Left one column |
| 6 | Cursor Right one column |
| 7 | Lock Keyboard |
| 8 | Unlock Keyboard |
| 9 | Erase to End of Line |
| 10 | Erase to End of Screen |
| 11 | Enter Background Display Mode (reduced intensity) |
| 12 | Enter Foreground Display Mode (normal intensity) |
| 13 | Enable Protected Fields |
| 14 | Disable Protected Fields |
| 15 | Delete Line |
| 16 | Insert Line |
| 17 | Delete Character |
| 18 | Insert Character |
| 19 | Read Cursor Address |
| 20 | Read Character at Current Cursor Address |
| 21 | Start Blinking Field |

| TCRT CODE | FUNCTION |
|-----------|--|
| 22 | End Blinking Field |
| 23 | Start Line Drawing Mode (enable alternate character set) |
| 24 | End Line Drawing Mode (disable alternate character set) |
| 25 * | Set Horizontal Position |
| 26 * | Set Vertical Position |
| 27 | Set Terminal Attributes |
| 28 | Cursor on |
| 29 | Cursor off |
| 30 | Start Underscore |
| 31 | End Underscore |
| 32 | Start Reverse Video |
| 33 | End Reverse Video |
| 34 | Start Reverse Blink |
| 35 | End Reverse Blink |
| 36 | Turn Off Screen Display |
| 37 | Turn On Screen Display |
| 38 | Top Left Corner |
| 39 | Top Right Corner |
| 40 | Bottom Left Corner |
| 41 | Bottom Right Corner |
| 42 | Top Intersect |
| 43 | Right Intersect |
| 44 | Left Intersect |
| 45 | Bottom Intersect |
| 46 | Horizontal Line |
| 47 | Vertical Line |
| 48 | Intersection |
| 49 | Solid Block |
| 50 | Slant Block |
| 51 | Cross-Hatch Block |
| 52 | Double Line Horizontal |
| 53 | Double Line Vertical |
| 54 | Send Message to Function Key Line |
| 55 | Send Message to Shifted Function Key Line |
| 56 | Set Normal Display Format |
| 57 | Set Horizontal Split (Follow with Row Code) |
| 58 | Set Vertical Split (39 Character Columns) |
| 59 * | Set Vertical Split (40 Character Columns) |
| 60 * | Set Vertical Split Column to Next Character |
| 61 | Activate Split Segment 0 |
| 62 | Activate Split Segment 1 |
| 63 | Send Message to Host Message Field |
| 64 | Up-Arrow |
| 65 | Down-Arrow |
| 66 | Raised Dot |
| 67 | End of Line Marker |
| 68 | Horizontal Tab Symbol |
| 69 | Paragraph |

| TCRT CODE | FUNCTION |
|-----------|---|
| 70 | Dagger |
| 71 | Section |
| 72 | Cent Sign |
| 73 | One-Quarter |
| 74 | One-Half |
| 75 | Degree |
| 76 | Trademark |
| 77 | Copyright |
| 78 | Registered |
| 79 | Print screen |
| 80 | Set to wide (132 column) mode |
| 81 | Set to normal (80 column) mode |
| 82 | Enter transparent print mode |
| 83 | Exit transparent print mode |
| 84 | Begin writing to alternate page |
| 85 | End writing to alternate page |
| 86 | Toggle page |
| 87 | Copy to alternate page |
| 88 | Insert column |
| 89 | Delete column |
| 90 | Block fill with attribute |
| 91 | Block fill with character |
| 92 | Draw a box |
| 93 | Scroll box up one line |
| 94 | Scroll box down one line |
| 95 | Select jump scroll |
| 96 | Select fast smooth scroll |
| 97 | Select medium-fast smooth scroll |
| 98 | Select medium-slow smooth scroll |
| 99 | Select slow smooth scroll |
| 100 | Start underscored, blinking field |
| 101 | End underscored, blinking field |
| 102 | Start underscored, reverse field |
| 103 | End underscored, reverse field |
| 104 | Start underscored, reverse, blinking field |
| 105 | End underscored, reverse, blinking field |
| 106 | Start underscored text without space |
| 107 | End underscored text without space |
| 108 | Start reverse text without space |
| 109 | End reverse text without space |
| 110 | Start reverse blinking text without space |
| 111 | End reverse blinking text without space |
| 112 | Start underscored blinking text without space |
| 113 | End underscored blinking text without space |
| 114 | Start underscored reverse text without space |
| 115 | End underscored reverse text without space |

| TCRT CODE | FUNCTION |
|-----------|---|
| 116 | Start underscored reverse blinking text without space |
| 117 | End underscored reverse blinking text without space |
| 118 | Start blink without space |
| 119 | End blink without space |
| 120 | Set cursor to blinking block |
| 121 | Set cursor to steady block |
| 122 | Set cursor to blinking underline |
| 123 | Set cursor to steady underline |
| 124 | Reserved |
| 125 | Reserved |
| 126 | Reserved |
| 127 | Reserved |
| 128 | Select top status line without address |
| 129 | End status line (all kinds) |
| 130 | Select unshifted status line without address |
| 131 | Select shifted status line without address |
| 132 * | Select black text |
| 133 * | Select white text |
| 134 * | Select blue text |
| 135 * | Select magenta text |
| 136 * | Select red text |
| 137 * | Select yellow text |
| 138 * | Select green text |
| 139 * | Select cyan text |
| 140 * | Select black reverse text |
| 141 * | Select white reverse text |
| 142 * | Select blue reverse text |
| 143 * | Select magenta reverse text |
| 144 * | Select red reverse text |
| 145 * | Select yellow reverse text |
| 146 * | Select green reverse text |
| 147 * | Select cyan reverse text |
| 148 | Save a rectangular area |
| 149 | Restore a rectangular screen area |
| 150 | Enter full graphics mode |
| 151 | Exit full graphics mode |
| 152 | Draw a box with rounded corners |
| 153 | Draw a window style box |
| 154 | Draw a box with double lines |
| 155 | Enable proportionally spaced text |
| 156 | Disable proportionally spaced text |
| 157 | Select color palette by RGB value |
| 158 | Enable graphics cursor |
| 159 | Disable graphics cursor |
| 160 | Select graphics cursor shape |
| 161 | Include graphics cursor location |

| TCRT CODE | FUNCTION |
|-----------|--|
| 162 | Define graphics cursor regions |
| 163 | Output form feed character |
| 164 | Output line feed character |
| 165 | Output new line character |
| 166 | Output vertical tab character |
| 167 | Output plus-or-minus character |
| 168 | Output greater-than-or-equal character |
| 169 | Output less-than-or-equal character |
| 170 | Output not-equal character |
| 171 | Output British pound character |
| 172 | Output Pi character |
| 173 | Enter bidirectional print mode |
| 174 | Exit bidirectional print mode |
| 175 | Set terminal time |
| 176 | Set terminal date |
| 177 | Select color palette by HLS value |
| 178 | Select PC character set |
| 179 | Select default character set |
| 180 | Select PC terminal emulation |
| 181 | Select ASCII terminal operation |
| 182 | Select AUX port host |
| 183 | Select main port host |
| 184 | Select toggle host ports |
| 185 | Select 8-bit character display |
| 186 | Select 7-bit character display |
| 187 | Select 8-bit keyboard mode |
| 188 | Select 7-bit keyboard mode |
| 189 | Select primary printer port |
| 190 | Select secondary printer port |
| 191 | Select 161 column display mode |

The actual routines that perform the screen controls are in the specific terminal drivers and not in AlphaBASIC PLUS itself. Not all terminal drivers have all of the functions above simply because not all terminals are able to perform all of these functions. The TRMCHR XCALL can be used to check if your terminal implements some of these features. See Chapter 9 of the AMOS AlphaBASIC XCALL Subroutine User's Manual for further details.

APPENDIX E

WRITING ASSEMBLY LANGUAGE SUBROUTINES

This appendix provides details on assembly language programming for those who want to write assembly language subroutines for AlphaBASIC PLUS programs. For further information about assembly language, see your *Assembly Language Programmer's manual* and your *AMOS Monitor Calls Manual*.



AlphaBASIC PLUS programs are different from AlphaBASIC programs. In assembly language routines, you can test for AlphaBASIC PLUS by examining the value of the longword at @A0. If it equals -2, AlphaBASIC PLUS is operating.

You write assembly language subroutines in the same manner you write assembly language programs. Use the OBJNAM statement to give it an .XBR extension, indicating it is a subroutine.

When an XCALL statement is executed by AlphaBASIC PLUS, the named subroutine is located in memory and then executed.

AlphaBASIC PLUS first saves all registers, then sets certain parameters into those registers for use by the external subroutine. The addresses of the arguments are calculated and entered into an argument list in memory along with their sizes and type codes. The base address of this list is then passed to the user routine in register A3.

The arguments may be one of two basic forms:

- [∞]A variable name, in which case the argument entry in the list references the selected variable within the user impure area. This variable is available to the called subroutine for both inspection and modification.
- [∞]An expression (numeric or string), in which case the expression is evaluated and the result is placed on the arithmetic stack.

This result, instead of a single variable, is then referenced in the argument list entry. It is only available for inspection, since the stack is cleared when the subroutine exits.

You may not use a non-MAPed array or array element as an argument if the array has not been either DIMed or previously referenced during program execution. The XCALL statement cannot allocate such an array at run-time.

You are free to write a routine using and modifying all of the general work registers (A0-A6 and D0-D7), and your program may use the stack for work space as required. When your subroutine has completed its execution, it must return to the run-time system by executing a RTN subroutine return instruction.

E.1 REGISTER PARAMETERS

The following registers are set up by the run-time system to be used as required by the external subroutine. They may be modified, if desired, since they have been saved before the subroutine was called:

- A0 Indexes the user impure variable area. A0 is used throughout the run-time system to reference all user variables. If the longword at @A0 = -2, AlphaBASIC PLUS is implied. A file, DSK0:[7,35]SBRSYM.M68, contains the offsets of A0. A0 may be used as a work register.
- A3 Points to the base of the argument list. A3 may be used to scan the argument list for retrieval of the argument parameters.
- A4 Points to the base of the free memory area that may be used by the external subroutine as work space. This is actually the address of the first word following the argument list in memory, and, if desired, may be used to store a terminator word to stop the scanning of the argument list.
- A5 This is the C stack index used by the run-time system. The stack is built at the top of the memory module allocated for use by RUNP in the user partition and grows downward as items are added to it. When the external subroutine is called, A5 points to the current stack base. Since the arithmetic stack may contain valid data, the external subroutine must not use the word indexed by A5 or any words above it.

E.2 ARGUMENT LIST FORMAT

The list of arguments specified in the XCALL statement may range from no arguments at all to a number limited only by the space on the command line. To pass these arguments to the external subroutine, an argument list is built in memory which describes each variable named in the list and tells where it can be located in the user impure area.

The variables themselves are not actually passed to the subroutine, but rather their absolute locations in memory are. In this way, the subroutine may inspect them and modify them directly in their respective locations. This does not apply to expressions which are built on the stack as described previously.

A3 points to the first word of the argument list, which is a binary count of how many arguments were contained in the XCALL statement. Following this count word comes one 5-word descriptor block for each argument specified. If there are no arguments in the XCALL statement, the argument list consists only of the single count word containing the value of zero.

The format of each 10-byte block describing one argument is:

Entry 1 One word containing a variable type code. Bits 0-3 contain the type code for the specific variable:

0 = unformatted
2 = string
4 = AMOS floating point
6 = binary
8 = integer
10 = IEEE floating point

All other values are currently unassigned. Other bits in the type code word are meaningless.

Entry 2 One longword containing an absolute address of a variable in a user impure area. This address is the first byte of the variable no matter its type or size.

Entry 3 One longword containing the size of the variable in bytes.

Note the above descriptions also apply to the expression arguments, except that the results are located above the address specified by A5 instead of below it.

The argument list is built in free memory directly above the currently allocated user impure area. A4 points to the word immediately following the last word in the argument list. You may scan the argument list and determine its end either by decrementing the count word at the base of the list or by scanning until the scan index reaches the address in A4.

E.3[∞]CONVERTING ARGUMENTS TO BINARY FORMAT

A standard subroutine, \$GTARG, is provided in the system subroutine library SYSLIB.LIB to assist in converting floating point and string arguments to binary format for processing within an AlphaBASIC PLUS subroutine.

For further information, see your *AMOS Monitor Calls Manual*.

E.4°FREE MEMORY USAGE

When the subroutine is called, indexes A4 and A5 mark the beginning and end of the free memory currently available for use as work space. This area is not preserved by the run-time system, and the subroutine must not count on its security between XCALL statements.

Note the word at @A4 may be used as the first word, but the word at @A5 is the base of the arithmetic stack and must not be destroyed. The last word of actually free memory is at -2(A5).

The run-time system has its own internal memory management system and does not conform to the operating system memory management method.

Registers A4 and A5 point to the free area between the heap and C stack as explained in Appendix H. This is the same as for AlphaBASIC. In AlphaBASIC PLUS only, you may be able to allocate standard AMOS memory modules via the GETMEM and GETIMP monitor calls if sufficient free memory is available at the top of your memory position. This area can be set aside by using the MINF setting in a .BPR file, as discussed in Appendix H.

E.5°LOCATING OPEN FILES

A standard library routine, \$FLSET, is provided in the system subroutine library, SYSLIB.LIB, to locate the DDB associated with a file opened by an AlphaBASIC PLUS program. For more information, see your *AMOS Monitor Calls Manual*.

E.6°PROGRAM HEADERS

All AlphaBASIC PLUS subroutines **must** contain a program header at the start of the subroutine. Program headers are defined by using the PHDR macro, discussed in your *AMOS Monitor Calls Manual*.

APPENDIX F

CHARACTER SETS

AMOS uses a single-byte character set. Such a character set can represent 256 different characters. The character set is aligned with a number of international standards.

F.1 A SHORT HISTORY OF CHARACTER SETS

The first international standard was set in 1965 by ECMA (European Computer Manufacturer's Association) and was known as ECMA-6. The character set was adopted by other standards bodies, and is also known as US-ASCII, DIN 66003, and ISO 646. The standard only defined a basic alphabet, and did not allow for national characters in use in many European countries. Such characters were incorporated by specifying twelve code points (see Note 1 in the table below) as being places where replacement characters could be defined. For example, Germany defined the letter Ä at code point 91, where the [character was located. These character sets were called the "national ISO 646 variants". Portability of files containing such characters were low.

In 1981, the IBM PC introduced an 8-bit character set with Code Page 437, a character set with many special characters. In 1982 DEC MCS (Multi Language Character Set) was released. This character set was very similar to ISO 6937/2, which in turn is almost identical to the modern standard for 8-bit character sets, ISO 8859. In 1985 ECMA standardized ECMA-94, which dealt with almost all European languages. ECMA-94 was taken up by ISO, as ISO 8859-1 through 8859-4, and standardized in 1987.

Microsoft released MS-DOS 3.3 in 1987, which used Code Page 850. This code page uses all the characters from ISO 8859-1, plus a few extra at code points representing the non-printing characters. A second code page, Code Page 819, is fully ISO 8859-1 compliant.

F.1.1 The ISO 8859 Family of Standards and AMOS

The ISO 8859-x character sets are designed for maximum interoperability and portability. All of them are a superset of US-ASCII and will render English text properly. The code points 0xA0 through 0xFF are used to represent national characters, while the characters in the range 0x20 through 0x7F are the same as in the ISO 646 (US-ASCII) character set. Thus ASCII text is a subset of all ISO 8859 character sets, and will be rendered properly by them. The code points 0x80 through 0x9F are earmarked as extended control characters and are not used for encoding characters.

The ISO 8859 family of standards consists of:

| | |
|---------|---|
| 8859-1 | For Europe, Latin America, the Caribbean, Canada, and Africa |
| 8859-2 | For Eastern Europe |
| 8859-3 | For SE Europe, and a miscellany of alphabets, such as Esperanto, and Maltese |
| 8859-4 | For Scandinavia, and the Baltic states (mostly covered by 8859-1 also) |
| 8859-5 | For languages using the Cyrillic alphabet |
| 8859-6 | For languages using Arabic |
| 8859-7 | For modern Greek |
| 8859-8 | For Hebrew |
| 8859-9 | Known as Latin-5. The same as 8859-1 except for Turkish instead of Icelandic characters |
| 8859-10 | Known as Latin-6, for Lappish, Nordic, and Eskimo languages |

ISO 8859-1 (also known as ISO Latin-1) has the required characters to display most Western European languages. It supports Afrikaans, Basque, Catalan, Danish, Dutch, English, Faeroese, Finnish, French, Galician, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish and Swedish. It cannot support Welsh, due to two missing characters (Latin Letter W with circumflex and Latin Letter Y with circumflex). It is the preferred encoding for the Internet.

AMOS follows this lead, and expects 8-bit aware software to use these ISO standards.

In passing, the ISO 8859-1 standard is a subset of the Unicode 1.x and 2.0 standards, which use 16-bit character sets to encode most of the world's alphabets. Unicode has aligned itself with a further ISO standard for 32-bit character sets, ISO 10646-1:1993. There are several mappings available (such as UTF-8) which can map Unicode characters to a variable length 8-bit based encoding.

| Char- acter | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|----------------|----------------|----------------|------------------|--------------|---|-----------------------------------|------|-------------|
| NULL | | 0 | 0 | 0 | | Null | Cc | |
| SOH | | 1 | 1 | 1 | | Start of Heading | Cc | |
| STX | | 2 | 2 | 2 | | Start of Text | Cc | |
| ETX | | 3 | 3 | 3 | | End of Text | Cc | |
| EOT | | 4 | 4 | 4 | | End of Transmission | Cc | |
| ENQ | | 5 | 5 | 5 | | Enquiry | Cc | |
| ACK | | 6 | 6 | 6 | | Acknowledge | Cc | |
| BEL | | 7 | 7 | 7 | | Bell | Cc | |
| BS | | 10 | 8 | 8 | | Backspace | Cc | |
| HT | | 11 | 9 | 9 | | Character Tabulation (Tab) | Cc | |
| LF | | 12 | 10 | A | | Line Feed | Cc | |
| VT | | 13 | 11 | B | | Line Tabulation (Vertical Tab) | Cc | |
| FF | | 14 | 12 | C | | Form Feed | Cc | |
| CR | | 15 | 13 | D | | Carriage Return | Cc | |
| SO | | 16 | 14 | E | | Shift Out | Cc | |
| SI | | 17 | 15 | F | | Shift In | Cc | |
| DLE | | 20 | 16 | 10 | | Data Link Escape | Cc | |
| DC1 | | 21 | 17 | 11 | | Device Control One | Cc | |

| Char- acter | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|----------------|----------------|----------------|------------------|--------------|---|------------------------------|------|-------------|
| DC2 | | 22 | 18 | 12 | | Device Control Two | Cc | |
| DC3 | | 23 | 19 | 13 | | Device Control Three | Cc | |
| DC4 | | 24 | 20 | 14 | | Device Control Four | Cc | |
| NAK | | 25 | 21 | 15 | | Negative Acknowledge | Cc | |
| SYN | | 26 | 22 | 16 | | Synchronous Idle | Cc | |
| ETB | | 27 | 23 | 17 | | End of Transmission Block | Cc | |
| CAN | | 30 | 24 | 18 | | Cancel | Cc | |
| EM | | 31 | 25 | 19 | | End of Medium | Cc | |
| SUB | | 32 | 26 | 1A | | Substitute | Cc | |
| ESC | | 33 | 27 | 1B | | Escape | Cc | |
| FS | IS4 | 34 | 28 | 1C | | File Separator | Cc | |
| GS | IS3 | 35 | 29 | 1D | | Group Separator | Cc | |
| RS | IS2 | 36 | 30 | 1E | | Record Separator | Cc | |
| US | IS1 | 37 | 31 | 1F | | Unit Separator | Cc | |
| SP | | 40 | 32 | 20 | Space | | Zs | |
| ! | | 41 | 33 | 21 | Exclamation Mark | | Po | |
| " | | 42 | 34 | 22 | Quotation Mark | | Po | |
| # | | 43 | 35 | 23 | Number Sign | (Hash) | So | 1 |
| \$ | | 44 | 36 | 24 | Dollar Sign | | Sc | 1 |
| % | | 45 | 37 | 25 | Percent Sign | | Po | |
| & | | 46 | 38 | 26 | Ampersand | | So | |
| ' | | 47 | 39 | 27 | Apostrophe | Apostrophe-Quote | Po | |
| (| | 50 | 40 | 28 | Left Parenthesis | Opening Parenthesis | Ps | |
|) | | 51 | 41 | 29 | Right Parenthesis | Closing Parenthesis | Pe | |
| * | | 52 | 42 | 2A | Asterisk | | So | |
| + | | 53 | 43 | 2B | Plus Sign | | Sm | |
| , | | 54 | 44 | 2C | Comma | | Po | |
| - | | 55 | 45 | 2D | Hyphen-Minus | Minus Sign | Pd | |
| . | | 56 | 46 | 2E | Full Stop | Period | Po | |
| / | | 57 | 47 | 2F | Solidus | Slash | Po | |
| 0 | | 60 | 48 | 30 | Digit Zero | | Nd | |
| 1 | | 61 | 49 | 31 | Digit One | | Nd | |
| 2 | | 62 | 50 | 32 | Digit Two | | Nd | |
| 3 | | 63 | 51 | 33 | Digit Three | | Nd | |
| 4 | | 64 | 52 | 34 | Digit Four | | Nd | |
| 5 | | 65 | 53 | 35 | Digit Five | | Nd | |
| 6 | | 66 | 54 | 36 | Digit Six | | Nd | |
| 7 | | 67 | 55 | 37 | Digit Seven | | Nd | |
| 8 | | 70 | 56 | 38 | Digit Eight | | Nd | |
| 9 | | 71 | 57 | 39 | Digit Nine | | Nd | |
| : | | 72 | 58 | 3A | Colon | | Po | |
| ; | | 73 | 59 | 3B | Semicolon | | Po | |
| < | | 74 | 60 | 3C | Less-Than Sign | | Sm | |
| = | | 75 | 61 | 3D | Equals Sign | | Sm | |
| > | | 76 | 62 | 3E | Greater-Than Sign | | Sm | |
| ? | | 77 | 63 | 3F | Question Mark | | Po | |
| @ | | 100 | 64 | 40 | Commercial At | | Po | 1 |
| A | | 101 | 65 | 41 | Latin Capital Letter A | | Lu | |
| B | | 102 | 66 | 42 | Latin Capital Letter B | | Lu | |
| C | | 103 | 67 | 43 | Latin Capital Letter C | | Lu | |
| D | | 104 | 68 | 44 | Latin Capital Letter D | | Lu | |
| E | | 105 | 69 | 45 | Latin Capital Letter E | | Lu | |
| F | | 106 | 70 | 46 | Latin Capital Letter F | | Lu | |
| G | | 107 | 71 | 47 | Latin Capital Letter G | | Lu | |
| H | | 110 | 72 | 48 | Latin Capital Letter H | | Lu | |

| Char- acter | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|----------------|----------------|----------------|------------------|--------------|---|-----------------------------------|------|-------------|
| I | | 111 | 73 | 49 | Latin Capital Letter I | | Lu | |
| J | | 112 | 74 | 4A | Latin Capital Letter J | | Lu | |
| K | | 113 | 75 | 4B | Latin Capital Letter K | | Lu | |
| L | | 114 | 76 | 4C | Latin Capital Letter L | | Lu | |
| M | | 115 | 77 | 4D | Latin Capital Letter M | | Lu | |
| N | | 116 | 78 | 4E | Latin Capital Letter N | | Lu | |
| O | | 117 | 79 | 4F | Latin Capital Letter O | | Lu | |
| P | | 120 | 80 | 50 | Latin Capital Letter P | | Lu | |
| Q | | 121 | 81 | 51 | Latin Capital Letter Q | | Lu | |
| R | | 122 | 82 | 52 | Latin Capital Letter R | | Lu | |
| S | | 123 | 83 | 53 | Latin Capital Letter S | | Lu | |
| T | | 124 | 84 | 54 | Latin Capital Letter T | | Lu | |
| U | | 125 | 85 | 55 | Latin Capital Letter U | | Lu | |
| V | | 126 | 86 | 56 | Latin Capital Letter V | | Lu | |
| W | | 127 | 87 | 57 | Latin Capital Letter W | | Lu | |
| X | | 130 | 88 | 58 | Latin Capital Letter X | | Lu | |
| Y | | 131 | 89 | 59 | Latin Capital Letter Y | | Lu | |
| Z | | 132 | 90 | 5A | Latin Capital Letter Z | | Lu | |
| [| | 133 | 91 | 5B | Left Square Bracket | Opening Square Bracket | Ps | 1 |
| \ | | 134 | 92 | 5C | Reverse Solidus | Backslash | Po | 1 |
|] | | 135 | 93 | 5D | Right Square Bracket | Closing Square Bracket | Pe | 1 |
| ^ | | 136 | 94 | 5E | Circumflex Accent | Spacing Circumflex; Caret | Lm | 1 |
| _ | | 137 | 95 | 5F | Low Line | Spacing Underscore; Underscore | So | |
| ` | | 140 | 96 | 60 | Grave Accent | Spacing Grave | Lm | 1 |
| a | | 141 | 97 | 61 | Latin Small Letter A | | Li | |
| b | | 142 | 98 | 62 | Latin Small Letter B | | Li | |
| c | | 143 | 99 | 63 | Latin Small Letter C | | Li | |
| d | | 144 | 100 | 64 | Latin Small Letter D | | Li | |
| e | | 145 | 101 | 65 | Latin Small Letter E | | Li | |
| f | | 146 | 102 | 66 | Latin Small Letter F | | Li | |
| g | | 147 | 103 | 67 | Latin Small Letter G | | Li | |
| h | | 150 | 104 | 68 | Latin Small Letter H | | Li | |
| i | | 151 | 105 | 69 | Latin Small Letter I | | Li | |
| j | | 152 | 106 | 6A | Latin Small Letter J | | Li | |
| k | | 153 | 107 | 6B | Latin Small Letter K | | Li | |
| l | | 154 | 108 | 6C | Latin Small Letter L | | Li | |
| m | | 155 | 109 | 6D | Latin Small Letter M | | Li | |
| n | | 156 | 110 | 6E | Latin Small Letter N | | Li | |
| o | | 157 | 111 | 6F | Latin Small Letter O | | Li | |
| p | | 160 | 112 | 70 | Latin Small Letter P | | Li | |
| q | | 161 | 113 | 71 | Latin Small Letter Q | | Li | |
| r | | 162 | 114 | 72 | Latin Small Letter R | | Li | |
| s | | 163 | 115 | 73 | Latin Small Letter S | | Li | |
| t | | 164 | 116 | 74 | Latin Small Letter T | | Li | |
| u | | 165 | 117 | 75 | Latin Small Letter U | | Li | |
| v | | 166 | 118 | 76 | Latin Small Letter V | | Li | |
| w | | 167 | 119 | 77 | Latin Small Letter W | | Li | |
| x | | 170 | 120 | 78 | Latin Small Letter X | | Li | |
| y | | 171 | 121 | 79 | Latin Small Letter Y | | Li | |
| z | | 172 | 122 | 7A | Latin Small Letter Z | | Li | |
| { | | 173 | 123 | 7B | Left Curly Bracket | Opening Curly Bracket | Ps | 1 |
| | | 174 | 124 | 7C | Vertical Line | Vertical Bar | So | 1 |
| } | | 175 | 125 | 7D | Right Curly Bracket | Closing Curly Bracket | Pe | 1 |
| ~ | | 176 | 126 | 7E | Tilde | | So | 1 |
| DEL | | 177 | 127 | 7F | | Delete | Cc | |

| Character | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|-----------|-------------|-------------|---------------|-----------|--|---|------|----------|
| PAD | | 200 | 128 | 80 | | Padding Character | Cc | |
| HOP | | 201 | 129 | 81 | | High Octet Preset | Cc | |
| BPH | | 202 | 130 | 82 | | Break Permitted Here | Cc | |
| NBH | | 203 | 131 | 83 | | No Break Here | Cc | |
| IND | | 204 | 132 | 84 | | Index | Cc | |
| NEL | | 205 | 133 | 85 | | Next Line | Cc | |
| SSA | | 206 | 134 | 86 | | Start of Selected Area | Cc | |
| ESA | | 207 | 135 | 87 | | End of Selected Area | Cc | |
| HTS | | 210 | 136 | 88 | | Character Tabulation Set | Cc | |
| HTJ | | 211 | 137 | 89 | | Character Tabulation with Justification | Cc | |
| VT | | 212 | 138 | 8A | | Line Tabulation Set | Cc | |
| PLD | | 213 | 139 | 8B | | Partial Line Forward | Cc | |
| PLU | | 214 | 140 | 8C | | Partial Line Backward | Cc | |
| RI | | 215 | 141 | 8D | | Reverse Line Feed | Cc | |
| SS2 | | 216 | 142 | 8E | | Single-Shift Two | Cc | |
| SS3 | | 217 | 143 | 8F | | Single-Shift Three | Cc | |
| DCS | | 220 | 144 | 90 | | Device Control String | Cc | |
| PU1 | | 221 | 145 | 91 | | Private Use One | Cc | |
| PU2 | | 222 | 146 | 92 | | Private Use Two | Cc | |
| STS | | 223 | 147 | 93 | | Set Transmit State | Cc | |
| CCH | | 224 | 148 | 94 | | Cancel Character | Cc | |
| MW | | 225 | 149 | 95 | | Message Waiting | Cc | |
| SPA | | 226 | 150 | 96 | | Start of Guarded Area | Cc | |
| EPA | | 227 | 151 | 97 | | End of Guarded Area | Cc | |
| SOS | | 230 | 152 | 98 | | Start of String | Cc | |
| SGCI | | 231 | 153 | 99 | | Single Graphic Character Introducer | Cc | |
| SCI | | 232 | 154 | 9A | | Single Character Introducer | Cc | |
| CSI | | 233 | 155 | 9B | | Control Sequence Introducer | Cc | |
| ST | | 234 | 156 | 9C | | String Terminator | Cc | |
| OSC | | 235 | 157 | 9D | | Operating System Command | Cc | |
| PM | | 236 | 158 | 9E | | Privacy Message | Cc | |
| APC | | 237 | 159 | 9F | | Application Program Command | Cc | |
| NBSP | | 240 | 160 | A0 | No-Break Space | | Zs | |
| ¡ | | 241 | 161 | A1 | Inverted Exclamation Mark | | Po | |
| ¢ | | 242 | 162 | A2 | Cent Sign | | Sc | |
| £ | | 243 | 163 | A3 | Pound Sign | | Sc | |
| ¤ | | 244 | 164 | A4 | Currency Sign | | Sc | |
| ¥ | | 245 | 165 | A5 | Yen Sign | | Sc | |
| ¦ | | 246 | 166 | A6 | Broken Bar | | So | |
| § | | 247 | 167 | A7 | Section Sign | | So | |
| ¨ | | 250 | 168 | A8 | Diaeresis | | Lm | |
| © | | 251 | 169 | A9 | Copyright Sign | | So | |
| ª | | 252 | 170 | AA | Feminine Ordinal Indicator | | So | |
| « | | 253 | 171 | AB | Left-Pointing Double Angle Quotation Mark | | Ps | |
| ¬ | | 254 | 172 | AC | Not Sign | | Sm | |
| - | | 255 | 173 | AD | Soft Hyphen | | Po | |
| ® | | 256 | 174 | AE | Registered Sign | | So | |
| ˆ | | 257 | 175 | AF | Macron | | Lm | |
| ° | | 260 | 176 | B0 | Degree Sign | | So | |

| Character | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|-----------|-------------|-------------|---------------|-----------|--|---------------|------|----------|
| ± | | 261 | 177 | B1 | Plus-Minus Sign | | Sm | |
| ² | | 262 | 178 | B2 | Superscript Two | | So | |
| ³ | | 263 | 179 | B3 | Superscript Three | | So | |
| ´ | | 264 | 180 | B4 | Acute Accent | | Lm | |
| µ | | 265 | 181 | B5 | Micro Sign | | So | |
| ¶ | | 266 | 182 | B6 | Pilcrow Sign | | So | |
| · | | 267 | 183 | B7 | Middle Dot | | Po | |
| ¸ | | 270 | 184 | B8 | Cedilla | | Lm | |
| ¹ | | 271 | 185 | B9 | Superscript One | | So | |
| º | | 272 | 186 | BA | Masculine Ordinal Indicator | | So | |
| » | | 273 | 187 | BB | Right-Pointing Double Angle Quotation Mark | | Pe | |
| ¼ | | 274 | 188 | BC | Vulgar Fraction One Quarter | | So | |
| ½ | | 275 | 189 | BD | Vulgar Fraction One Half | | So | |
| ¾ | | 276 | 190 | BE | Vulgar Fraction Three Quarters | | So | |
| ¿ | | 277 | 191 | BF | Inverted Question Mark | | Po | |
| À | | 300 | 192 | C0 | Latin Capital Letter A With Grave | | Lu | |
| Á | | 301 | 193 | C1 | Latin Capital Letter A With Acute | | Lu | |
| Â | | 302 | 194 | C2 | Latin Capital Letter A With Circumflex | | Lu | |
| Ã | | 303 | 195 | C3 | Latin Capital Letter A With Tilde | | Lu | |
| Ä | | 304 | 196 | C4 | Latin Capital Letter A With Diaeresis | | Lu | |
| Å | | 305 | 197 | C5 | Latin Capital Letter A With Ring Above | | Lu | |
| Æ | | 306 | 198 | C6 | Latin Capital Ligature AE | | Lu | 2 |
| Ç | | 307 | 199 | C7 | Latin Capital Letter C With Cedilla | | Lu | |
| È | | 310 | 200 | C8 | Latin Capital Letter E With Grave | | Lu | |
| É | | 311 | 201 | C9 | Latin Capital Letter E With Acute | | Lu | |
| Ê | | 312 | 202 | CA | Latin Capital Letter E With Circumflex | | Lu | |
| Ë | | 313 | 203 | CB | Latin Capital Letter E With Diaeresis | | Lu | |
| Ì | | 314 | 204 | CC | Latin Capital Letter I With Grave | | Lu | |
| Í | | 315 | 205 | CD | Latin Capital Letter I With Acute | | Lu | |
| Î | | 316 | 206 | CE | Latin Capital Letter I With Circumflex | | Lu | |
| Ï | | 317 | 207 | CF | Latin Capital Letter I With Diaeresis | | Lu | |
| Ð | | 320 | 208 | D0 | Latin Capital Letter Eth | | Lu | |
| Ñ | | 321 | 209 | D1 | Latin Capital Letter N With Tilde | | Lu | |
| Ò | | 322 | 210 | D2 | Latin Capital Letter O With Grave | | Lu | |
| Ó | | 323 | 211 | D3 | Latin Capital Letter O With Acute | | Lu | |

| Char- acter | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|----------------|----------------|----------------|------------------|--------------|---|---------------|------|-------------|
| Ô | | 324 | 212 | D4 | Latin Capital Letter O With Circumflex | | Lu | |
| Õ | | 325 | 213 | D5 | Latin Capital Letter O With Tilde | | Lu | |
| Ö | | 326 | 214 | D6 | Latin Capital Letter O With Diaeresis | | Lu | |
| × | | 327 | 215 | D7 | Multiplication Sign | | Sm | |
| Ø | | 330 | 216 | D8 | Latin Capital Letter O With Stroke | | Lu | |
| Ù | | 331 | 217 | D9 | Latin Capital Letter U With Grave | | Lu | |
| Ú | | 332 | 218 | DA | Latin Capital Letter U With Acute | | Lu | |
| Û | | 333 | 219 | DB | Latin Capital Letter U With Circumflex | | Lu | |
| Ü | | 334 | 220 | DC | Latin Capital Letter U With Diaeresis | | Lu | |
| Ý | | 335 | 221 | DD | Latin Capital Letter Y With Acute | | Lu | |
| Þ | | 336 | 222 | DE | Latin Capital Letter Thorn | | Lu | |
| ß | | 337 | 223 | DF | Latin Small Letter Sharp S | | LI | |
| à | | 340 | 224 | E0 | Latin Small Letter A With Grave | | LI | |
| á | | 341 | 225 | E1 | Latin Small Letter A With Acute | | LI | |
| â | | 342 | 226 | E2 | Latin Small Letter A With Circumflex | | LI | |
| ã | | 343 | 227 | E3 | Latin Small Letter A With Tilde | | LI | |
| ä | | 344 | 228 | E4 | Latin Small Letter A With Diaeresis | | LI | |
| å | | 345 | 229 | E5 | Latin Small Letter A With Ring Above | | LI | |
| æ | | 346 | 230 | E6 | Latin Small Ligature AE | | LI | 2 |
| ç | | 347 | 231 | E7 | Latin Small Letter C With Cedilla | | LI | |
| è | | 350 | 232 | E8 | Latin Small Letter E With Grave | | LI | |
| é | | 351 | 233 | E9 | Latin Small Letter E With Acute | | LI | |
| ê | | 352 | 234 | EA | Latin Small Letter E With Circumflex | | LI | |
| ë | | 353 | 235 | EB | Latin Small Letter E With Diaeresis | | LI | |
| ì | | 354 | 236 | EC | Latin Small Letter I With Grave | | LI | |
| í | | 355 | 237 | ED | Latin Small Letter I With Acute | | LI | |
| î | | 356 | 238 | EE | Latin Small Letter I With Circumflex | | LI | |
| ï | | 357 | 239 | EF | Latin Small Letter I With Diaeresis | | LI | |
| ð | | 360 | 240 | F0 | Latin Small Letter Eth | | LI | |
| ñ | | 361 | 241 | F1 | Latin Small Letter N With Tilde | | LI | |

| Char- acter | Also Called | Octal Value | Decimal Value | Hex Value | ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name | Also Known As | Type | See Note |
|----------------|----------------|----------------|------------------|--------------|---|---------------|------|-------------|
| ò | | 362 | 242 | F2 | Latin Small Letter O With Grave | | LI | |
| ó | | 363 | 243 | F3 | Latin Small Letter O With Acute | | LI | |
| ô | | 364 | 244 | F4 | Latin Small Letter O With Circumflex | | LI | |
| õ | | 365 | 245 | F5 | Latin Small Letter O With Tilde | | LI | |
| ö | | 366 | 246 | F6 | Latin Small Letter O With Diaeresis | | LI | |
| ÷ | | 367 | 247 | F7 | Division Sign | | Sm | |
| ø | | 370 | 248 | F8 | Latin Small Letter O With Stroke | | LI | |
| ù | | 371 | 249 | F9 | Latin Small Letter U With Grave | | LI | |
| ú | | 372 | 250 | FA | Latin Small Letter U With Acute | | LI | |
| û | | 373 | 251 | FB | Latin Small Letter U With Circumflex | | LI | |
| ü | | 374 | 252 | FC | Latin Small Letter U With Diaeresis | | LI | |
| ý | | 375 | 253 | FD | Latin Small Letter Y With Acute | | LI | |
| þ | | 376 | 254 | FE | Latin Small Letter Thorn | | LI | |
| ÿ | | 377 | 255 | FF | Latin Small Letter Y with Diaeresis | | LI | |

Notes:

1. This code point is used by National Replacement Character Sets (7-bit character sets). Devices using such an NRC will not print the glyph shown, neither will it print glyphs for code points above 127.
2. ISO may be reclassifying these code points as “Latin Letter”, as certain Scandinavian languages use these characters as a complete letter, not as a ligature.

Type: The characters are broken down into “character types” by Unicode:

| | |
|----|-----------------------------|
| Cc | Control or Format Character |
| LI | Lowercase Letter |
| Lm | Modifier Letter |
| Lu | Uppercase Letter |
| Nd | Decimal Number |
| Pd | Dash Punctuation |
| Pe | Close Punctuation |
| Po | Other Punctuation |
| Ps | Open Punctuation |
| Sc | Currency Symbol |
| Sm | Math Symbol |
| So | Other Symbol |
| Zs | Space Separator |

APPENDIX G

ERR(3) ERROR CODES

These are the error codes returned by ERR(3), with their meanings. See Chapter 17 for information on trapping and correcting these errors. ERR(3) returns a 16-bit number from the JOBERR monitor call. See your *Monitor Calls Manual* for more information. The first three bits represent the severity of the error:

| CODE | SEVERITY |
|------|--|
| 0 | Operation completed - no errors/warnings (if error code is zero) |
| 0 | Operation aborted due to fatal error (if error code is non-zero) |
| 2 | Operation aborted due to error or ^C |
| 4 | Operation completed with errors |
| 6 | Operation completed with warnings only |

The next three bits are reserved for future use. The final ten bits identify the error:

| ERROR CODE | ERROR |
|------------|--------------------------|
| 0 | No error detected |
| 1 | File specification error |
| 2 | Insufficient free memory |
| 3 | File not found |
| 4 | File already exists |
| 5 | Device not ready |
| 6 | Device full |
| 7 | Device error |
| 8 | Device in use |
| 9 | Illegal user code |
| 10 | Protection violation |
| 11 | Write protected |
| 12 | File type mismatch |
| 13 | Device does not exist |
| 14 | Illegal Block number |
| 15 | Buffer not INITed |
| 16 | File not open |
| 17 | File already open |
| 18 | Bitmap kaput |
| 19 | Device not mounted |

| ERROR CODE | ERROR |
|------------|--|
| 20 | Invalid filename |
| 21 | BADBLK.SYS has a bad hash total |
| 22 | BADBLK.SYS is in unsupported format |
| 23 | BADBLK.SYS not found |
| 24 | Insufficient queue blocks |
| 25 | MFD is damaged |
| 26 | First logical unit is not mounted |
| 27 | Remote is not responding |
| 28 | File in use |
| 29 | Record in use |
| 30 | Deadly embrace possible |
| 31 | File cannot be deleted |
| 32 | File cannot be renamed |
| 33 | Record not locked |
| 34 | Record not locked for output |
| 35 | LOKSER queue is full |
| 36 | Device is not file structured |
| 37 | Illegal record size |
| 38 | Block allocate/deallocate error |
| 256 | Miscellaneous error |
| 257 | Memory map destroyed |
| 258 | Insufficient privileges to run program |
| 259 | Must be logged into [1,2] |
| 260 | Must be logged into DSK0:[1,2] |
| 261 | Program requires 68020 processor |
| 262 | Must be logged in to run program |
| 263 | Bus error |
| 264 | Memory parity error |
| 265 | Address error |
| 266 | Illegal instruction |
| 267 | Divide by zero error |
| 268 | CHK instruction trap |
| 269 | TRAPV instruction trap |
| 270 | Privilege violation |
| 271 | Trace trap return |
| 272 | EM1111 instruction trap |
| 273 | Miscellaneous exceptions |
| 274 | Illegal interrupt on level 0 |
| 275 | Illegal interrupt on level 1 |
| 276 | Illegal interrupt on level 2 |
| 277 | Illegal interrupt on level 3 |
| 278 | Illegal interrupt on level 4 |
| 279 | Illegal interrupt on level 5 |
| 280 | Illegal interrupt on level 6 |
| 281 | Illegal interrupt on level 7 |
| 282 | Bus time-out error |
| 283 | MMU error |

| ERROR CODE | ERROR |
|-------------------|---|
| 284 | Co-processor protocol violation |
| 285 | FPCP branch or set on unordered condition |
| 286 | FPCP inexact result |
| 287 | FPCP divide by zero |
| 288 | FPCP underflow |
| 289 | FPCP operand error |
| 290 | FPCP overflow |
| 291 | FPCP signaling NAN |
| 292 | MMU co-figuration error |
| 293 | MMU illegal operation |
| 294 | MMU access level violation |
| 384 | Language processor aborted |
| 385 | Language processor aborted with errors |
| 386 | Language processor completed with errors |
| 387 | Undefined identifier in input |
| 388 | Runtime interpreter error |
| 389 | Syntax error in input |
| 390 | Assembly error in linkage |
| 512 | Process aborted by operator (^C) |
| 513 | Command line format error |
| 514 | Command line switch error |
| 515 | Bad SSD |
| 516 | Overlay not found |

APPENDIX H

MEMORY

This appendix discusses AlphaBASIC PLUS memory requirements, and how you can control how memory is used in your user partition.

H.1 MEMORY REQUIREMENTS

Several factors determine the amount of memory you need to run any AlphaBASIC PLUS component:

- The size of the program itself. However, if the program is loaded into system memory, you don't need to load it into your partition, and you can ignore this size.
- Each program needs a certain amount of additional memory to operate; this is called the DSECT size.
- Finally, AlphaBASIC PLUS needs to create a "stack"—an area of memory in which to store variables and do computations. You can control the size of this stack area with the MINS and MINF definitions in a .BPR file. (.BPR files are explained later in this appendix)

The table below shows the sizes of these factors for each AlphaBASIC PLUS component. By adding the appropriate numbers together, you can see how large a partition you need.

| PROGRAM | SIZE | DSECT SIZE | MINF (default) | MINS (default) | TOTAL (default) |
|------------|------|---------------|-------------------|-------------------|--------------------|
| BASICP.LIT | 150K | 32K | 8K | 64K | 254K |
| COMPLP.LIT | 90K | 19K | 4K | 64K | 177K |
| RUNP.LIT | 80K | 15K | 16K | 64K | 175K |

Remember, the size of the program doesn't need to be considered if it is loaded into system memory in the system initialization command file.

Note that the table reflects version 1.0(267). Other versions may have different values for the factors.

H.2°MEMORY MANAGEMENT

You can control AlphaBASIC PLUS's allocation of memory by using a .BPR file.

H.2.1°The .BPR file

The name of a .BPR file is the same as the .RP file to which it applies. A .BPR file is read only when RUNP.LIT initializes itself. Therefore, typing RUNP MYFILE from the command line will cause RUNP to look for and process MYFILE.BPR to configure memory. However, doing a CHAIN "MYFILE" or a SUBCALL "MYFILE" will not cause a new .BPR file to be processed: the .BPR file settings of the existing .RP file will still be in use. To force CHAIN to read the target .RP file's .BPR file, use CHAIN "RUNP MYFILE".

A .BPR file is a standard ASCII file which you can create through a text editor such as AlphaXED or AlphaVUE. It consists of one or more lines, with each setting defined on a separate line. The .BPR file must be located on the following search path:

```
User memory
.RP file's device, drive and PPN
.RP file's device, drive, [p,0]
RP: ersatz (usually DSK0:[7,35])
```

If the .RP file was loaded into MEM:, then the user's current login details are used instead of the .RP file's details.

The contents of the .BPR file are in the following format:

```
setting = value
```

H.2.2°BPR File Settings

The context of each setting is explained in the next section. The valid settings in the .BPR file are:

| | |
|-------|---|
| DELTA | The minimum number of bytes that must exist between the top of the heap and the bottom of the stack at all times. If the gap is any smaller, RUNP will abort with a non-trappable runtime error STACK/HEAP COLLISION - FATAL. |
| MAXS | The maximum size of the stack/heap area that RUNP allocates. |

| | |
|-------|--|
| MINF | The minimum size of the user memory partition not used by RUNP. This area is left untouched by RUNP. The free area can be allocated by GETMEM and GETIMP calls. If MINF bytes cannot be left free, then RUNP aborts with the error ?STACK+HEAP NO MEM. |
| MINS | The minimum size of the stack/heap area that RUNP allocates. If RUNP cannot allocate this amount, RUNP aborts with the error ?STACK+HEAP NO MEM. |
| STACK | The size of the AlphaBASIC PLUS Arithmetic Stack. This is the stack pointed at by Register A5 when an XCALL executes. |

The value item is a decimal number, representing the number of bytes for the appropriate setting. You can add a "K" suffix to indicate "kilobytes", or "M" to indicate "Megabytes".

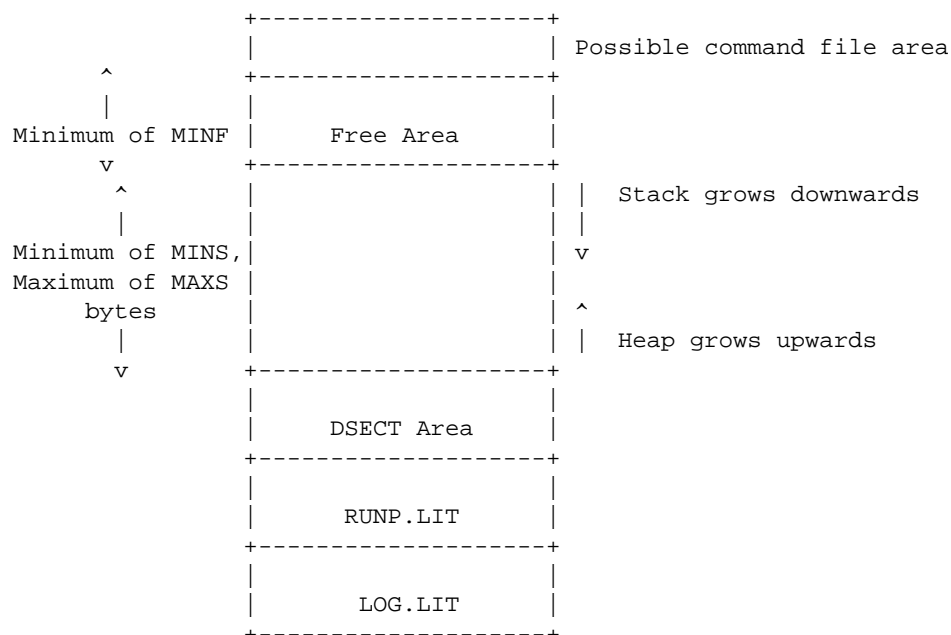
H.2.3[∞]RUNP Memory Partition Layout

As RUNP.LIT is a program written in C, the memory architecture of the runtime environment is heavily influenced by that required to execute any C program. There are three crucial memory areas:

- [∞]The DSECT area. This area is RUNP's impure area for some of its internal variables. The size of this area is decided by the programmer who wrote RUNP, and cannot be altered. The DSECT area is allocated when RUNP initializes.
- [∞]The Heap. The Heap is formed at the bottom end of an AMOS memory module. As RUNP does its work, it often needs memory that's persistent for some time. An example of this is the DDB and buffer required to access a file OPENed in AlphaBASIC. Another example is the area needed to hold the .RP file if it is not located in system or user memory. When a particular memory requirement is no longer needed, RUNP will return it back to the heap to be reused later. The Heap acts as a pool or reservoir of free memory. As more memory is used, the Heap grows upwards in memory towards the C stack at the top of the memory module.
- [∞]The C Stack. The C stack grows downwards from the top end of the memory module that also holds the Heap. The C Stack should not be confused with AlphaBASIC PLUS's Arithmetic Stack. The Arithmetic Stack is in fact a persistent memory allocation made from the Heap. The C stack is used by RUNP as it executes. The C Stack grows downwards towards the Heap as RUNP internally calls its functions as it executes. The requirement for the C stack is determined by the programmer who wrote RUNP and the complexity of your AlphaBASIC PLUS program.

There must always be a small amount of free space between the top of the Heap and the bottom of the C stack. If the size of this area drops below a minimum (the DELTA setting in the .BPR file), RUNP aborts with a non-trappable runtime error.

When RUNP is executing, the memory map in the partition looks like the following:



The size of the memory module that contains the C Stack and Heap is controlled by the MINS and MAXS settings in the .BPR file. The size of the area left free at the top of memory is controlled by the MINF setting in the .BPR file.

When RUNP initializes, it allocates a memory module for its DSECT area. It then allocates a memory module for the C Stack/Heap area. The size of the module is determined in the following way.

- Look at the size of free memory in the partition. If less than MINS + MINF + DELTA bytes are left, RUNP aborts with a ?STACK+HEAP NO MEM error.
- If sufficient bytes are left, provisionally subtract MINF bytes from the size of the free area. If the result is more than MAXS, allocate the Stack/Heap module with a size of MAXS bytes, thereby increasing the size of the free area above MINF bytes. If the result is less than MAXS bytes, allocate the Stack/Heap module with the size of the result, leaving MINF bytes free.

H.2.4 RUNP'S Use of the Stacks, Heap and Free Area

The Heap is used for allocating memory for such things:

- AlphaBASIC PLUS's Arithmetic Stack. The size of the Arithmetic Stack is controlled by the STACK setting in the .BPR file.

- The string work area. The size of this area is set by the compiler as described in Appendix I.
- DDBs, buffers and internal variables for file IO within AlphaBASIC PLUS.
- Holding .RP and .SPG files if they are not loaded into system or user memory.

If RUNP tries to allocate memory in the heap, and there's insufficient contiguous free memory in the Stack/Heap module, RUNP will abort with a trappable runtime error 3, Out of memory.

The C Stack is used internally by RUNP to carry out its own execution. As such, much of the size requirement is beyond an AlphaBASIC PLUS programmer's reach. The complexity of a given AlphaBASIC statement may have a small impact on the Stack's size. The biggest contributor to large C Stack sizes is having nested user defined functions and subprograms.

The size of the Heap and C Stack are controlled by the MINS, MAXS and MINF settings in the .BPR file, as outlined above.

AlphaBASIC PLUS's Arithmetic Stack is used for:

- General execution of each AlphaBASIC PLUS statement. Loops (such as FOR loops or DO WHILE loops) push control variables onto the Arithmetic Stack. The more nesting of these structures, the more Arithmetic Stack space is required.
- Holding actual parameters for user defined functions. If you call a particular function recursively, each level of recursion will push its own set of parameters on the Arithmetic Stack.

If there is insufficient room on the Arithmetic Stack, RUNP throws a trappable runtime error 33, Stack overflow. You can prevent this situation from reoccurring by increasing the STACK setting in the .BPR file.

RUNP can use the free area at the top of the memory partition (reserved by the MINF setting in the .BPR file) for:

- Executing the built-in AMOS function.
- Loading any XCALL subroutines not already loaded into system or user memory.
- Loading any device drivers that AMOS requires - this is more properly thought of as a function of AMOS rather than a function of RUNP.

You can increase the size of this reserved area by increasing the MINF setting in the .BPR file.

H.2.5[∞]Calculating Runtime Requirements

It is hard to predict the minimum requirements for a program. Execution order can affect the results. It is rare to set DELTA to anything but the 300 byte default. Decreasing this value is very risky, and increasing it is only indicated in very rare situations where RUNP suddenly had to do a large number of nested calls internally without having to do a memory allocation in the Heap. Symptoms of stack/heap collision problems would be obtaining erratic results after executing a particularly complex single AlphaBASIC PLUS statement at a deeply nested level, when you can guarantee that an XCALL has not run wild.

The MINF requirement is reasonably easy to predict for a given RES: and MEM: configuration. As you know which modules are already loaded, and which AMOS calls are made, setting the size of this area is straightforward.

The STACK requirement is most heavily affected by the level of nesting of control structures. The default is a reasonable number. Increase it if you have deeply nested structures.

The MINS and MAXS factors are the hardest to predict, especially as it is not possible to do memory compaction within the Heap of any C program. The Heap can therefore get fragmented, and the degree of fragmentation can be heavily dependent on the order of statement execution. You can use the MEM() function to probe some of the Heap and Stack allocations, but it is more of an art than a science.

H.2.6[∞]COMPLP and RUNP C Settings

You can also set the same parameters for interactive AlphaBASIC PLUS by including them in a file called BASICP.PRM (in the account you want to run your program in).

You can create a file (called COMPLP.PRM) for the COMPLP program itself, that defines the MAXS, MINS, and MINF parameters. The default for MINF with COMPLP is 4096, all the other defaults are the same as above.

APPENDIX I

THE STRING WORK AREA

AlphaBASIC PLUS allocates memory in the Heap (see Appendix H) for a work area in which to do string operations, such as concatenation, substring operations, and printing variables. This area is known as the String Work Area, or SWA. The required size of the SWA for a given .RP file is determined by the compiler (COMPLP) and passed to RUNP in the .RP file. The size of the SWA is reported by COMPLP's compilation statistics and the MEM(14) function at runtime.

COMPLP can make a reasonably accurate estimate of the size of the SWA needed to successfully execute the .RP file. But there are two cases where COMPLP does not have sufficient information to make an accurate guess:

- Where SPACE() or FILL() are used with a runtime expression to define the size of the resultant string. In such cases, COMPLP will use a set value (132 bytes in version 1.0(269)) in its calculations.
- If a user-defined function or subprogram is called. COMPLP cannot look inside a user defined function to see the maximum SWA size required for executing the call. Such information may not be available, especially if recursive calls are used. Similarly, subprograms are compiled separately, but will share the same SWA as the calling .RP file. COMPLP will not attempt to guess a reasonable value for the SWA requirements for this statement.

If you find that you need to adjust the size of the SWA from COMPLP's estimate, you can use ++PRAGMA compiler statements. Three ++PRAGMA statements are available:

```
++PRAGMA SET_STRWRK constant_expression
++PRAGMA EXTRA_STRWRK constant_expression
++PRAGMA ADD_EXTRA_STRWRK constant_expression
```

constant_expression is a compile-time expression that must evaluate to a value between -2MB(ytes) and +2MB(ytes). The syntax of the expression must follow that for the DEFINE statement in Chapter 9.

As COMPLP processes each source line, it estimates the size of the SWA required to execute the statement. It also remembers the largest value encountered so far. It is this value that COMPLP would use to set the size of the SWA if no ++PRAGMAs were encountered. Assume that there are four variables, COMPLP'SWA'SIZE, STMT'SWA'SIZE, SET'SWA'SIZE, and EXTRA'SWA'SIZE, maintained internally by COMPLP. COMPLP manipulates them as follows:

```
COMPLP'SWA'SIZE = 0
EXTRA'SWA'SIZE  = 0
SET_SWA_SIZE    = 0
while (there's a line to compile)
    STMT'SWA'SIZE = SWA requirement for this line
    COMPLP'SWA'SIZE = COMPLP'SWA'SIZE max STMT'SWA'SIZE
end while
```

SET'SWA'SIZE and EXTRA'SWA'SIZE can be changed by ++PRAGMA lines as follows:

```
SET_STRWRK:      SWA'SIZE = SET_STRWRK value
                  EXTRA'SWA'SIZE = 0

EXTRA_STRWRK:    EXTRA'SWA'SIZE = EXTRA_STRWRK value

ADD_EXTRA_STRWRK:
EXTRA'SWA'SIZE = EXTRA'SWA'SIZE + ADD_EXTRA_STRWRK value
```

At the end of the compilation, the SWA size is set:

```
if (SET'SWA'SIZE >= 0)
    COMPLP'SWA'SIZE = SET'SWA'SIZE
else
    COMPLP'SWA'SIZE = COMPLP'SWA'SIZE + margin()
end if
if (EXTRA'SWA'SIZE != 0)
    COMPLP'SWA'SIZE = COMPLP'SWA'SIZE + EXTRA'SWA'SIZE
end if
COMPLP'SWA'SIZE = COMPLP'SWA'SIZE rounded up to even number
write COMPLP'SWA'SIZE to .RP or .SPG file
```

margin() is calculated in version 1.0(269) as:

```
if COMPLP'SWA'SIZE <= 256
    COMPLP'SWA'SIZE = COMPLP'SWA'SIZE + 50
else
    COMPLP'SWA'SIZE = COMPLP'SWA'SIZE * 1.1
end if
```

You can have multiple ++PRAGMAs altering the SWA size in a source file. The ++PRAGMAs will interact as defined above. In conjunction with the conditional compilation ++PRAGMA directives (such as ++IF), you can tune the .RP file's SWA requirement.



If you CHAIN from one .RP file to another with CHAIN "MYFILE", RUNP will not delete its existing SWA and reallocate the SWA with the size specified in the incoming .RP file. The second .RP file will use the freshly-initialized SWA of the first .RP file. Therefore, the first .RP file must specify the maximum size of the SWA for its own requirements and for all .RP files that it CHAINS to. If you need to CHAIN to another .RP file and use its SWA setting, you must cause RUNP to reinitialize itself by using CHAIN "RUNP MYFILE".



The SWA setting in a .SPG file is not used by the calling .RP file in any way. Therefore the calling .RP's SWA setting must be sufficient for its own requirements and the requirements of any SUBPROGRAMS it executed. In general, nesting subprograms requires that the SWA size is set to the maximum requirement of any one subprogram (and calling .RP file). The individual requirements for subprograms at each level of nesting are not totalled to give the overall requirement.



Both nested user defined functions and multiple user defined functions in one AlphaBASIC PLUS statement require that the SWA size is the sum of the requirements for each user defined function active during the execution of that statement. Therefore if one function recurses to five levels, and there are two other user defined functions in that statement, the SWA requirement is at least six times the requirement for the recursive function, plus the individual requirements for the other two functions.

APPENDIX J

COMPILER AND RUNTIME OPTIONS

During the evolution of AlphaBASIC PLUS, its designers made a large number of decisions that defined how the compiler and runtime systems worked. In many cases, the decisions were influenced by existing practices and backwards compatibility with AlphaBASIC 1.3 and AlphaBASIC 1.4. As always, some decisions that were right at the time seem a little less correct now. But changing those decisions involves changing the way the compiler and runtime systems work - and that may break some existing programs that depend on the current behavior of the products. So no changes can be made - or so it seems.

J.1°RESOLVING THE DILEMMA

COMPLP and RUNP version 1.0(267) and later have a feature that can resolve the dilemma. At compile time, the program developer can specify certain options that cause the compiler and/or the runtime system to change their way of operation for that particular file. The list of options is processed and acted upon by the compiler, and then passed to the runtime system by storing the list inside the object file. When RUNP loads the file, it reads the option list, and alters its actions appropriately.

Over time, the list of options is expected to grow. But a given version of COMPLP and RUNP will only understand those options that were defined when they were constructed. Later versions may understand more options. Part of RUNP's action on loading the file is to check that it can understand and comply with all the requested options. If it cannot do so, RUNP will abort with a non-trappable error.

J.2°OPTION SPECIFICATION

Options are specified by a number, ranging from 1 upwards. To request a particular option at compile time, type:

```
COMPLP filename/P:option1{,option2...}
```

For example, to select options 1, 3 and 6 for MYFILE.BP, use:

```
COMPLP MYFILE/P:1,3,6 RETURN
```

You can also embed the option list inside the source file by using a `++PRAGMA` command. This saves you from having to remember to use the right option switches on the right files at every compilation. The `++PRAGMA` statement (see Chapter 5 for more details) corresponding to the command line is:

```
++PRAGMA FEATURE 1,3,6
```

COMPLP will check that it understands and can act on each of those options in turn, aborting if it cannot do so. It will compile the file, taking the appropriate option actions, and produce the output object file, MYFILE.RP. The compilation statistics will list the options selected. In addition, COMPLP will encode and write the option list (1,3,6) into MYFILE.RP for RUNP to take action on. This allows MYFILE to be invoked without any additional command line switches, and is a safety check in case MYFILE.RP is transferred to a system that has a version of RUNP that cannot understand one or more of the options.

When RUNP executes the file, it will check to see if options have been selected. If so, it will check to see if it understands and can take the appropriate action. If it cannot, RUNP will fail with a non-trappable runtime error.

J.3[∞]CHECKING AVAILABILITY OF OPTIONS

You can see which options your version of COMPLP understands by using the `/R` switch:

```
COMPLP /R RETURN
```

This lists all the options COMPLP understands. RUNP offers the same option:

```
RUNP /R RETURN
```

You can see which options have been set into an object file by using the `/R` switch with a filename:

```
RUNP /R MYFILE RETURN
```

This produces a list of statistics and option settings for MYFILE, and confirms whether or not the version of RUNP can be used with this file.

J.4[∞]ALTERNATIVE METHOD OF OPTION SPECIFICATION

Currently, COMPLP and RUNP have a maximum of 64 possible options, only a few of which are defined. So entering a small number of options on a list is simple. But as the possible list grows, it becomes impractical to type in the list on the command line with the `/P` switch. Either `++PRAGMA` settings can be used, or use can be made of the `/PX` switch. The `/PX` switch lets you enter the required list in a shorthand way.

`/PX1:number` is used to enter any and all options from 1 through 32 inclusive.
`/PX2:number` is used for options 33 through 64 inclusive.

To calculate the correct `number` to enter, you need to understand binary arithmetic. Each option number sets a bit in a 32-bit number, and all the unrequested options clear a bit in the same number. The decimal value of the resulting 32-bit number is the value to enter for `number`. In the example below, the options are in the range 1..32, so only `/PX1:` is specified. Option 1 sets the lowest-order bit (bit number zero, value of 1). Option 3 sets bit number two, value of 4. Option 6 sets bit number 5, value of 32:

Group the options into two groups, 1 through 32 inclusive, and 33 through 64 inclusive. For each group:

1. Start with a 32-bit number:

```
00000000000000000000000000000000
```

2. Number the bits from right to left (starting with the low-order bit), starting with 1 for the 1-32 group, or 33 for the 33-64 group.

3. For each option, set the option number's bit to a 1. Thus for options 1,3, and 6, set:

```
0000000000000000000000000000100101
```

4. Read that back as a decimal value:

```
1 + 4 + 32 = 37
```

5. Use that value in the `/PX:` command. For the first group, use `/PX1:.` For the second, use `/PX2:.` For options 1,3,6:

```
COMPLP MYFILE/PX1:37 RETURN
```

There is no corresponding shorthand for `++PRAGMA FEATURE`, as there can be multiple `++PRAGMA` lines in the file. However, all the `++PRAGMA FEATURE` lines must be at the top of the source file, before `COMPLP` generates any object code (i.e. prior to any `VERSION` statements, `++INCLUDE` statements or `MAP` statements).



Note that versions of `RUNP` earlier than 1.0(267) do not have any option facilities, and they are effectively blind to such requests. Files compiled under `COMPLP` 1.0(267) and later with option settings will be erroneously executed under `RUNP` version 1.0(266) and earlier without any indication of an error.

J.5 EFFECTS OF USING OPTIONS

The use of compatibility options requires the correct versions of COMPLP and RUNP to fit the situation. This section explains the implications of mixing COMPLP and RUNP versions. As an example, take two hypothetical versions of COMPLP and RUNP: version 10.0(600) which implements the ABC compatibility option, and version 10.0(606) which also implements the XYZ option.

- Rule 1—You cannot specify an option in an .RP or .SPG file unless the version of COMPLP you are using supports that option. In our example, you cannot specify the XYZ option to COMPLP 10.0(600): it will throw an error. If you use a version of COMPLP prior to 1.0(267), the /P and /PX command line switches are not recognized, nor is the ++PRAGMA FEATURE command, so the appropriate information cannot be put into the .RP or .SPG file.
- Rule 2—Unspecified options are not set. In the example, if an .RP is produced with the ABC option by COMPLP 10.0(600), and run by RUNP 10.0(606), the ABC option is set, but the XYZ option is cleared by definition.
- Rule 3—If you select an option, and later run the file under a version of RUNP that does not have the option defined, RUNP will throw an untrappable error when loading the file and will not execute it. In the example, if an .RP is produced by COMPLP 10.0(606) with the XYZ option set, and run by RUNP 10.0(600), that version of RUNP will see an option that it knows nothing about, display an error message, and abort to AMOS command level without executing any part of the file. Rule 4 is an exception to this rule.
- Rule 4—The exception to Rule 3: If an executable file is created with any option selected, and then run under a version of RUNP that does not have compatibility option functionality (that is, a version earlier than 1.0(267)), the file will be executed without complaint. All selected options are irrelevant to such a version of RUNP. Therefore, if you run a program compiled under COMPLP 10.0(600) with the ABC option set, RUNP 1.0(250) will execute it, ignoring the option setting.
- Rule 5—If you CHAIN to another program, that program is independent of the exiting program. The chaining program does not hand on any compatibility option requirements to the "chaine". RUNP will check the chained-to program's header and determine if it can execute the chained-to program as it loads the file. If it cannot execute the file, RUNP will fail with a non-trappable error. If you build PROG1.RP with COMPLP 10.0(606) with the ABC option selected, and that CHAINs to PROG2.RP built with COMPLP 10.0(606) with the ABC and XYZ options selected, all is well if PROG1.RP is run under RUNP 10.0(606). If you use RUNP 10.0(600), PROG1.RP will execute and start CHAINing to PROG2.RP. But PROG2.RP will not execute, as option XYZ is not understood by RUNP 10.0(600), and it will abort with a non-trappable error.
- Rule 6—If you make a SUBCALL to a subprogram that's internal to your main .RP file, that subprogram will execute using the selected options. However, a separately compiled subprogram is an entity completely isolated from the calling .RP (or .SPG) program. The calling program does not hand on its compatibility

option requirements to its callee. As it loads the called program, RUNP will check the callee's header, and determine if it can execute it. If not, it will fail with a non-trappable error.

Always minimize the number of compatibility options you specify at compile time, because that allows the executable file to run under more versions of RUNP. To assist with this, COMPLP 1.0(269)-2 and later will make reasonable efforts to tell you if a compatibility option was invoked but never triggered in the source lines. For example, assume the ABC option was something to do with the SIGNIFICANCE statement. If you specified the ABC option, but never wrote a SIGNIFICANCE statement in the file, you have probably specified an option in error, and COMPLP will warn you about it.

J.6[∞]DEFINED OPTIONS

| Number | First Added | Description |
|--------|-------------|---|
| 1 | 1.0(269)-2 | If set, DELETE'RECORD maintains the functionality of versions before 1.0(269)-2. The only error reported is Record Not Locked, and for that error you have to inspect the file's status variable to see if the error occurred. If this option is not set, DELETE'RECORD can report the full set of errors, setting the file status variable and throwing a standard AlphaBASIC PLUS trappable error. This new behavior (the default from version 1.0(269)-2 onwards) may break programs that do not expect DELETE'RECORD to throw errors. |

GLOSSARY

| | |
|------------------|--|
| ACTUAL PARAMETER | An expression, constant, or variable passed to a function in a function call. The function defines a FORMAL PARAMETER (see below) at compile time, and the actual parameter is substituted into the formula at run time. |
| ALLOCATE | The process of setting aside an amount of disk memory for a specific purpose. |
| ARRAY | A structure of data such that each piece of data (each variable) follows each other in a logical, accessible order. The data is ordered sequentially, i.e., one, two, three. |
| BASIC | B eginners A ll-purpose S ymbolic Instruction C ode. |
| BINARY | Data in the form of a series of 0s and 1s. This is how data is represented at the lowest level of the computer. |
| BINDING | An association of a variable with a value. Whenever you use an assignment statement, such as: |

A = 5

The value (in this case, 5) is **bound** to the variable (in this case, A). That variable "contains" that value until it is changed by another assignment statement.

| | |
|-------------------|---|
| BOOLEAN | An operation resulting in a TRUE or FALSE condition. Thus, a Boolean operator compares two expressions and returns a value of TRUE or FALSE depending on how they compared. |
| CALL BY REFERENCE | Is a way of associating an ACTUAL PARAMETER with a FORMAL PARAMETER. Subprograms use call by reference if the actual parameter is a variable preceded by the atsign, @. The purpose of call by reference is to have a way of letting subprograms pass values back to the calling program. Expressions cannot be passed by reference, and default to call by value. For example: |

```
A = 1
SUBCALL "DEMO" ,@A
PRINT A
END

SUBPROGRAM DEMO
  PARAMETER X,F,6
  X = X + 4
SUBEND
```

The SUBCALL above references A, rather than passing the value of A, so A is used in place of X in the SUBPROGRAM formula. The value of A is actually **changed** in the subprogram, so PRINT A displays 5.

CALL BY VALUE

A way of associating an ACTUAL PARAMETER with a FORMAL PARAMETER. Functions use call by value to pass the value of a variable without affecting it. For example:

```
Y = 20
DEF FNA(X) = X + 1
PRINT FNA(Y)
PRINT Y
```

prints 21, then 20. The value of Y is passed to the function, but the global value of Y is unchanged.

CHAINING

Moving from an AlphaBASIC PLUS program to another program or to a command file.

COMPILE

The process of changing an AlphaBASIC PLUS source file into an executable RUN file.

CONVERSION

The process of changing one type of data into another. For example, if you have a string expression, "45," you can convert it to a numeric value by using the VAL function.

DEBUGGING

The process of removing the errors, or "bugs," from a program. Most programs, when first written, do not work quite the way they are supposed to, and so a period of debugging is needed to fix the problems.

DELIMITER

An alphanumeric character marking a division between two pieces of data. For example, in the date display 12/04/86, the slash "/" is a delimiter. A dash "-" is also a common delimiter.

DISK FILE

An area of memory on a hard disk containing a program, text, or data file. Unless such a file is erased, it resides permanently on the disk.

DYNAMIC SCOPING

Scoping refers to the range over which a particular variable binding is accessible. "Dynamically scoped" means a binding is accessible based on how the program is run (hence, dynamic), as opposed to how a program is written (which is statically scoped). For example:

```
DEF FNA1(X,Y),Z
  Z = 1
  FNA1 = FNB(1)
ENDFN

DEF FNA2(X,Y),Z
  Z = 2
  FNA2 = FNB(1)
ENDFN

DEF FNB(Y) = Y + Z

Z = 100
A = FNA1(10,20)
B = FNA2(3,4)
```

The question is: What binding of Z is in effect when FNB is executed? Z is given the global value of 100. However, when FNA1 is called, the value of 100 is shadowed since Z is declared local to the function. Z is given the value of 1. When FNB is executed from FNA1, the binding of Z with 1 is in effect. This is the value FNB adds to Y, returning 2. When FNA1 returns, the global binding is restored—Z is once again 100. When FNA2 is called, the global binding of Z gets shadowed again, and the local value becomes 2. This time, when FNB is called, the value added to Y is 2, so FNB returns 3. The value of bindings are determined dynamically in AlphaBASIC PLUS, based on how the program runs, rather than how it might look on paper.

ECHO

A response by the computer to what you press on the keyboard. If you press a key (say), and you see a response on your terminal screen (an "a" appears), then your computer is in ECHO mode (the normal mode). Sometimes, for instance—when you are entering a password, the computer echoing can be turned off, so what you type goes to the computer, but isn't displayed on the screen.

| | |
|-----------------------|--|
| EXPONENT | If a number has more digits than AlphaBASIC PLUS usually displays, the number is displayed in scientific notation (a number expressed as a factor of ten). For example, 5,000,000 is expressed as 5E10^6. The E indicates the number is in scientific notation, and the 10^6 indicates how many zeroes must follow the 5. The value 6 is the exponent. |
| EXPRESSIONS | <p>A portion of AlphaBASIC PLUS syntax that can contain variables, constant values, operator symbols, functions, or combinations of these things. Expressions are not complete AlphaBASIC PLUS statements, but make up parts of AlphaBASIC PLUS statements. For example:</p> <pre>X = SIN(SQR(Number))</pre> <p>Everything to the right of the equal sign is an expression, and the expression combines with the X = to form an AlphaBASIC PLUS statement.</p> |
| FILE LOCKING | A process by which a file being used by one program is "locked," so any other program trying to access it is told the file is in use. This prevents problems occurring from two programs working on the same file at the same time. |
| FLOATING POINT NUMBER | A "real" number—a number that may contain a decimal part. For example, 4.175. Called "floating point" because the number of digits both right and left of the decimal point may vary, so the decimal point has no fixed position. |
| FORMAL PARAMETER | <p>A place holder in a definition to be filled at run time with an ACTUAL PARAMETER (see above). For example, in:</p> <pre>DEF FN'TEST (A,B) = A + B + 7</pre> <p>The variables A and B are formal parameters. Whatever actual values are specified when the function is called are used to compute A + B + 7.</p> |
| GLOBAL VARIABLE | A variable whose value can be accessed anywhere in the program (except when shadowed by a local variable—see SHADOWING, below). See also LOCAL VARIABLE. |

| | |
|----------------|--|
| INCREMENT | <p>The process of increasing the value of a variable by a constant amount. For example, the statement:</p> $\text{NUMBER} = \text{NUMBER} + 1$ <p>Increments the variable NUMBER by one.</p> |
| INDEXED FILE | An ISAM file (see ISAM, below). |
| INPUT | Any data coming into the program or computer. Input may come from the user at the keyboard, from a data file, or from some other external device. |
| INTEGER NUMBER | A "whole" number. That is, a number with no decimal part. For example, 3. |
| INTERACTIVE | Mode of AlphaBASIC PLUS operating in which the computer and the person writing or running a program interact. The computer responds directly to every input by the operator. |
| ISAM | Indexed Sequential Access Method. |
| LABEL | A name marking a place in a program, so other lines in the program can transfer control to the spot. For example, if you had a subroutine that adds three numbers, you might label it ADD'THREE'NUMS:. The label may not contain spaces, and must have a colon ":" after it (though you do not use the colon when referring to the label). |
| LIBRARY | In AlphaBASIC PLUS, a group of subroutines/programs in a special account. These subroutines/programs can be called by or incorporated in your AlphaBASIC PLUS programs. |
| LIST | The process of displaying the contents of an AlphaBASIC PLUS program, so you can see the lines of code making up the program. |
| LITERAL | <p>An actual value, rather than a variable. For example, in:</p> $A\$ = \text{"This is a literal string."}$ <p>the quoted material to the right of the equal sign is a literal string, whereas A\$ is a string variable.</p> |

| | |
|-------------------|---|
| LOCAL VARIABLE | A variable whose value can be accessed only from a limited part of a program. In AlphaBASIC PLUS, this occurs within function definitions and inside subprograms. In function definitions, the FORMAL PARAMETERS are local to the function definition. |
| LOGICAL OPERATORS | <p>AlphaBASIC PLUS words that perform tests of logic on expressions. Often used in IF-THEN statements, so the program can do different things depending on how the "logic test" turns out. For example, the statement:</p> <pre>IF NUMBER < 1 OR NUMBER > 100 & THEN GOTO ILLEGAL'NUMBER</pre> <p>Performs two tests on the variable NUMBER, to see if it is less than 1 or greater than 100. The OR is the logical operator, and instructs AlphaBASIC PLUS to GOTO ILLEGAL'NUMBER if either of the two tests are true.</p> |
| LOOP | <p>A repetition of the same section of code. For example:</p> <pre>FOR I = 1 TO 10 PRINT I NEXT I</pre> <p>causes PRINT I to be executed 10 straight times, and each time, the variable I is incremented by 1.</p> |
| MAPPED VARIABLE | A variable defined at the start of the program by a MAP statement. Mapped variables can be grouped into data structures. |
| MOUNT | The process of preparing a disk device for use. |
| NULL VALUE | A variable expression with no value. For a number, the value is zero (0). For a string, it's empty quotation marks (""). |
| OUTPUT | Any data coming from an AlphaBASIC PLUS program. Output is displayed on the terminal or on paper, or is placed in a data file. |

| | |
|------------|---|
| PRECEDENCE | <p>What comes first. In AlphaBASIC PLUS, certain expressions could be evaluated in more than one way, so a hierarchy of precedence is built into AlphaBASIC PLUS to determine which operations are done first. For example:</p> $\text{NUMBER} = 1 + 4 * 6$ <p>makes NUMBER equal to 25, because multiplication has precedence over addition (4 * 6 is done before 1 is added). If addition was higher in precedence, then NUMBER would equal 30 (1 + 4 would be done, then * 6).</p> |
| QUEUE | <p>A "waiting list". For example, when printing files, a queue is used so the files sent to the printer do not print all at the same time, but rather in an orderly way— first come, first served.</p> |
| RANDOM | <p>A method of storing data where the data is placed at random in memory, and pointers keep track of where the data record is. This type of data storage makes data retrieval faster.</p> |
| RECORD | <p>A piece of data. For instance, if you think of a data file as a bookcase, each "record" is like a book— a separate unit of data contained within the file.</p> |
| SAVE | <p>The process of transferring a file from user memory (which is temporary) to a disk (which is permanent).</p> |
| SCALING | <p>Storing variables by multiplying them by a power of ten so the decimal fraction isn't stored. This is done so mathematical operations are more accurate. Useful if you use large or long numbers which may cause "round off" errors.</p> |
| SEQUENTIAL | <p>A method of file storage in which each item of data follows the previous item of data in order. Sequential data files are usually slower for data retrieval than random files, but they can be edited using AlphaVUE.</p> |
| SHADOWING | <p>When a local variable hides the value of a global variable. This can occur in AlphaBASIC PLUS inside a function definition. If you have a FORMAL PARAMETER with the same variable name as a variable in your main program, the value of that variable is different within the function than in the main program. While in the function definition, the formal parameter variable has the value assigned to it in the definition. When the definition is finished, the variable again has the value assigned to it in the main program.</p> |

| | |
|----------------|--|
| SIGNIFICANCE | How many digits are used to represent a number. Normally, numbers are given in 6 digit significance, but can be set from 1 to 15 by the SIGNIFICANCE statement. |
| SOURCE | A file containing the original text or code. In AlphaBASIC PLUS, the source file is the file containing the AlphaBASIC PLUS code (the file usually has a .BP extension). |
| SPOOLING | Sending a file to a printer queue, so it can await printing. |
| STRING | An ASCII character or a group of ASCII characters. ASCII characters include letters, numbers, punctuation marks, and special symbols (see Appendix C for a complete list). |
| STRING LITERAL | <p>A string enclosed in quotation marks to indicate it is as it appears (as opposed to being a variable). For example:</p> <pre>"This is a literal string of characters."</pre> |
| SUBROUTINE | A section of program code operating as a unit to do a task. The main program "calls" the subroutine, and, when it has done its work, it returns control back to the next line of the main program. |
| SUBSCRIPTS | The numbers representing the elements of an array. |
| SUBSTRING | <p>A part of a string. A substring is produced by use of substring modifiers or string functions. For example, if you had a string, and you only wanted to print the first five characters rather than the whole string, you could say:</p> <pre>PRINT INPUT'String\$[1,5]</pre> <p>The first five characters printed are a substring of INPUT'String\$.</p> |
| SWITCH | An option that can be added to a command. |
| SYSTEM MEMORY | An area of memory available to all users of the computer. Any file loaded into this area (by a SYSTEM command in your system INI file) can be used by any person or program. The advantage of using system memory is, if a file is used regularly by many users, each user doesn't have to load a copy of it into their user memory, saving time and memory. |

| | |
|---------------------------|--|
| TRANCENDENTAL FUNCTION | An AlphaBASIC PLUS function that often results in an irrational number (for example, a number that repeats endlessly, or a number like pi, which never ends). The SQR (square root) function and the SIN (sine) functions are good examples of this. |
| UNARY (Plus/Minus) | An operative changing the value of a number or variable to positive or negative. For example, if NUM = -2, then +NUM changes the value of NUM to 2. And if you add a minus sign to a positive number, it changes the value to negative. |
| USER MEMORY | An area of memory available to one user. |
| VARIABLE | A name representing a number or character string that can change during the run of the program. |
| WILDCARD | A symbol that can represent a range of other characters. You might want to think of it as a joker in a deck of cards. For example, if you wanted to erase three files: FILE01.DAT, FILE02.DAT, and FILE03.DAT, you could type all this: |

```
ERASE FILE01.DAT RETURN
ERASE FILE02.DAT RETURN
ERASE FILE03.DAT RETURN
```

or you could type:

```
ERASE FILE0?.DAT RETURN
```

where the question mark symbol "?" represents all characters, or:

```
ERASE *.DAT RETURN
```

where * represents all combinations of characters (in this case, all filenames). Of course, you wouldn't use the second command if you had .DAT files you wanted to keep.

| | |
|-------|---|
| XCALL | An AlphaBASIC PLUS command that runs an external (outside the current AlphaBASIC PLUS program) subroutine. The subroutines XCALL can call are useful programs many people use, and thereby save you the time and trouble of writing such a subroutine yourself. |
|-------|---|

AlphaBASIC PLUS

Document History

Revision 00 - (Printed September, 1989)—New Document

Adapted from the *AlphaBASIC User's Manual*. Revised to include the new features of AlphaBASIC PLUS.

Revision 01 - (Printed April, 1991)

Revised Chapters 2, 5, 7, 8, 12, 16, and 20 to add new features and clarify text. Added Chapter 23 to document the new Debugger.

Revision 02 - (Printed September, 1996)

New compiler and runtime switches, added ++PRAGMA, rewrote memory handling, data types, and subprogram sections, removed old Chapter 9; added appendices I and J, added RNDN function, added string work area discussion. Other editing changes throughout.

Revision 03 - (Printed May, 1997)

Added notes to /PX and /R switches in Chapter 5; changed ODTIM description in Chapter 10. Chapter 11, updated VER\$; Chapter 19, changed DELETE'RECORD. In Appendix J, added rules about option use and description of option number 1.

Revision 04 - (Printed November, 1997)

Small changes to ODTIM in Chapter 10 and DATE in Chapter 11.

INDEX

| | |
|----------------|-------------------|
| ! | 2-2, 12-5 |
| # | 12-3, 15-2 |
| \$\$ | 12-6 |
| \$FLSET | E-4 |
| \$GTARG | E-3 |
| & | 2-3, 5-2 |
| ' | 2-2 |
| ** | 12-7 |
| , | 8-1, 9-22, 12-7 |
| - | 12-7 |
| . (period) | 12-5 |
| .BP extension | 3-5, 5-3 |
| .BPI | 1-5 |
| .BPR | 18-1 |
| .BPR file | H-2 |
| .DAT | 15-3 |
| .LIT extension | 5-3 |
| .RP | 1-5 |
| .SPG extension | 16-6 |
| .XBR | 1-5 |
| : | 2-2, 3-3 |
| ; | 8-1, 9-22 |
| ? | 22-4 |
| @ | 14-8, 14-13, 22-4 |
| A0 | E-2 |
| A3 | E-2 |

| | |
|---------------------------------|------------------|
| A4 | E-2 |
| A5 | E-2 |
| ABS(X) | 10-2 |
| Absolute value | 10-2 |
| Accessing random files | 15-6 |
| Account specification | 5-11 |
| ACS(X) | 10-4 |
| ALLOCATE | 15-5, 15-9 |
| ALLOCATE'INDEXED | 19-8 |
| AlphaBASIC PLUS | 1-2 |
| AlphaBASIC search path | 2-4 |
| Alphabetic characters | 1-4 |
| Alphanumeric characters | 1-4 |
| AlphaXED | 1-3, 5-1 |
| Alternative radix constants | 6-11 |
| AMOS | 22-5 |
| AMOS statement | 9-1 |
| Switches | 9-1 |
| Ampersand symbol (&) | 2-3, 5-2 |
| AND | 7-2 |
| Apostrophe | 6-1 |
| APPEND mode | 15-3 |
| Argument list format | E-2 |
| Arguments | 10-1, 10-7 |
| Pass by reference | 16-8 |
| Pass by value | 16-8 |
| Arithmetic stack | E-1 |
| Array variables | 6-13 |
| Default size | 6-14 |
| Numeric | 6-13 |
| String | 6-13 |
| ASC(X) | 10-2 |
| ASCII | 3-5, 10-2, 10-5 |
| Character set | F-1 |
| Decimal value of a character | 10-2 |
| ASN(X) | 10-4 |
| Assembly language subroutines | 18-1 |
| Assigning a value | 9-18 |
| ATN(X) | 10-4 |
| Atsign (@) | 14-8, 14-13, E-4 |
| ATTACH | 22-6 |
| Automatic subroutine loading | 18-2 |
| | |
| B.A.S.I.C. - what it stands for | 1-2 |
| Base of natural logarithms | 10-2 |
| BASIC | 1-2 |
| Compiler | 1-5 |
| BASIC search path | 2-4 |
| BASICP.LIT | 1-4, 3-7, 3-9 |
| Binary data | 14-6 |

| | |
|---|-----------------------|
| Binary format conversion | E-3 |
| Binary variables | 14-5 |
| Blank spaces | 10-9 |
| Blocking factor | 15-5 |
| BP: | 18-1 |
| Branching | 9-21 |
| BREAK | 4-1, 22-4 |
| Breakpoint | 22-4 |
| Breakpoints | 3-1, 4-1, 4-5 |
| BYE | 3-2, 3-9 |
| BYTE(X) | 11-1 |
| | |
| CALL | 9-2, 9-9, 9-20 |
| Carriage return/linefeed | 9-23 |
| CASE | 9-2 |
| CHAIN | 9-2, 15-9, 16-1, 22-9 |
| Chaining to system functions | 16-2 |
| CHANNEL | 22-6 |
| Character constants | 6-10 |
| CHR(X) | 10-5 |
| Clear | 22-4 |
| Clearing interactive memory | 3-3 |
| Clock ticks | 11-4 |
| CLOSE | 15-10, 19-7 |
| CLOSEK | 15-10, 19-7 |
| Closing an ISAM PLUS file | 19-7 |
| CMDLIN | 10-11 |
| Colon symbol (:) | 3-3 |
| Comma symbol (,) | 8-1, 9-22 |
| Command files | 16-2 |
| Comments | 2-2 |
| COMMON | 9-3, 16-2 |
| Compilation control | 5-5 |
| Compiler options | 5-3 |
| Compiler options - interactive mode | 3-8 |
| Compiling a program | 3-7, 5-3 |
| COMPLP | 1-5, 2-4, 5-3 |
| Options | 5-3, 6-12 |
| COMPLP.LIT | 1-4 |
| Conditional statement | 9-12 |
| Constants | 6-10, 7-4, 9-3 |
| Alternative radix | 6-11 |
| Character | 6-10 |
| Defining | 9-3 |
| Floating point | 6-12 |
| String | 6-12 |
| CONT | 4-2 |
| Continuation lines | 2-3, 5-2 |
| Continuing an interrupted program | 4-2 |
| Control-C | 4-3, 4-5, 5-11 |

| | |
|--|-------------|
| Control-C trap | 9-19, 17-4 |
| Control-characters | |
| CTRL/ C | 1-7 |
| Control-variables | 9-7 |
| Controlling compile | 5-5 |
| Converting arguments to binary | E-3 |
| Converting numbers to string | 10-10 |
| Converting to integer | 10-3 |
| Converting to lower case | 10-7 |
| Converting to upper case | 10-10 |
| Copying a string | 10-6 |
| COS(X) | 10-4 |
| CREATE'RECORD | 19-6 |
| Creating a program with AlphaXED | 5-1 |
| Creating a record | 19-6 |
| Creating an ISAM PLUS file | 19-8 |
| Creating interactive programs | 3-4 |
| DAFAULT | 9-3 |
| DATA | 9-3, 9-25 |
| Data Base Mangagement | 21-1 |
| Data files | 19-2 |
| Data formats | 1-4 |
| Array structures | 1-4 |
| Binary variables | 1-4, 14-6 |
| Floating point variables | 1-4, 14-6 |
| Integer variables | 1-4 |
| Simple variables | 1-4 |
| String variables | 1-4, 14-5 |
| Unformatted variables | 1-4, 14-5 |
| Data types | 8-3 |
| Default | 6-13 |
| Date | 10-13, 11-2 |
| DATN(X,Y) | 10-4 |
| DB | 22-1 |
| Command mode | 22-3 |
| Key sequences | 22-2 |
| Screen mode | 22-3 |
| DB commands | |
| ? | 22-4 |
| @ | 22-4 |
| AMOS | 22-5 |
| ATTACH | 22-6 |
| BREAK | 22-4 |
| CHAIN | 22-9 |
| CHANNEL | 22-6 |
| Clear | 22-4 |
| ERR | 22-6 |
| FILE | 22-6 |
| FUNCTION | 22-7 |

| | |
|---------------------|------|
| INDENT | 22-7 |
| LABEL | 22-7 |
| LET | 22-7 |
| NEW | 22-8 |
| PROCEED | 22-8 |
| QUIT | 22-8 |
| REMOVE | 22-8 |
| RESUME | 22-8 |
| RUN | 22-8 |
| SCALE | 22-8 |
| SEARCH | 22-5 |
| SIGNIFICANCE | 22-8 |
| SUBPROGRAM | 22-9 |
| TRACE | 22-9 |
| TRAP | 22-5 |
| UNTRACE | 22-9 |
| UNTRAP | 22-5 |
| VARIABLE | 22-9 |
| Debugger | 22-1 |
| Command mode | 22-3 |
| Key sequences | 22-2 |
| Screen mode | 22-3 |
| Debugger commands | |
| ? | 22-4 |
| @ | 22-4 |
| AMOS | 22-5 |
| ATTACH | 22-6 |
| BREAK | 22-4 |
| CHAIN | 22-9 |
| CHANNEL | 22-6 |
| Clear | 22-4 |
| ERR | 22-6 |
| FILE | 22-6 |
| FUNCTION | 22-7 |
| INDENT | 22-7 |
| LABEL | 22-7 |
| LET | 22-7 |
| NEW | 22-8 |
| PROCEED | 22-8 |
| QUIT | 22-8 |
| REMOVE | 22-8 |
| RESUME | 22-8 |
| RUN | 22-8 |
| SCALE | 22-8 |
| SEARCH | 22-5 |
| SIGNIFICANCE | 22-8 |
| SUBPROGRAM | 22-9 |
| TRACE | 22-9 |
| TRAP | 22-5 |
| UNTRACE | 22-9 |

| | |
|---|-------------|
| UNTRAP | 22-5 |
| VARIABLE | 22-9 |
| Debugging MAP statements | 14-13 |
| Decimal Logarithm | 10-3 |
| Decimal values | F-1 |
| DEF FN | 20-1 |
| DEFINE | 9-3 |
| Defined function examples | 20-5 |
| DEL | 5-3 |
| DELETE | 3-4 |
| DELETE'RECORD | 19-6 |
| Deleting a record | 19-6 |
| Deleting program lines | 3-4 |
| Device specification | 5-11 |
| Devn: | 1-6 |
| DIM | 6-14, 9-4 |
| Dimensioned arrays | 6-14, 9-4 |
| Direct access files | 15-4 |
| Direct statements | 3-3, 9-1 |
| Direct terminal input | 10-12 |
| Disk files | 1-3, 15-1 |
| Definition | 15-1 |
| DITOS(X) | 10-11 |
| DIV | 7-2 |
| DIVIDE'BY'0 | 9-5 |
| DO WHILE/UNTIL LOOP | 9-5 |
| DSTOI(X) | 10-11 |
| Duplicate line numbers | 4-3 |
| ECHO | 9-6 |
| Edit number | 9-24 |
| EDIT\$ | 10-5 |
| Editing a string value | 10-5 |
| Editing lines in interactive mode | 3-4 |
| Editing mask | 12-1 |
| Efficiency | C-1 |
| ELSE | 9-6, 9-12 |
| END | 9-6, 16-9 |
| END'FILE | 19-3 |
| End-of-file | 15-10 |
| ENDSWITCH | 9-6 |
| Entry 1 | E-3 |
| Entry 2 | E-3 |
| Entry 3 | E-3 |
| EOF(X) | 15-10 |
| EQV | 7-2 |
| Erasing disk files | 11-5, 15-14 |
| ERR | 22-6 |
| ERR(0) | 17-2 |
| ERR(1) | 17-2 |

| | |
|---|---------------------------|
| ERR(2) | 17-2 |
| ERR(3) | 17-2 |
| ERR(3) Error Codes | G-1 |
| ERR(X) | 10-12, 17-2 |
| ERRMSG(X) | 10-11 |
| Error codes | 10-12 |
| Error codes returned by ERR(0) | 17-2 |
| Error message | 10-11 |
| Error processing | 19-9 |
| Error trapping | 9-20, 16-9, 17-1 |
| Control-C | 17-4 |
| Sample programs | 17-4 |
| Even addresses | 14-12 |
| Examples of MAP statements | 14-9 |
| Exclamation symbol (!) | 2-3 |
| Executing command files from BASIC | 9-2 |
| EXIT | 9-7 |
| Exiting interactive mode | 3-9 |
| EXP(X) | 10-2 |
| Expression list | 9-22 |
| Expressions | 7-1, 7-4, 8-3, 9-22, 10-1 |
| Definition | 7-1 |
| Functions with arguments | 9-22 |
| Numeric constants | 9-22 |
| Numeric variables | 9-22 |
| Operator symbols | 9-22 |
| String literals | 9-22 |
| String variables | 9-22 |
| Extensions | 1-5 |
| .BP | 3-6, 5-3 |
| .DAT | 15-18 |
| .LIT | 5-3 |
| .RP | 3-6, 5-11, 16-1 |
| External assembly language subroutines .. | 9-31 |
| FACT(X) | 10-2 |
| Factorial | 10-2 |
| FILE | 22-6 |
| File channels | 15-2 |
| File functions | 11-1 |
| KILL | 11-5 |
| LOOKUP | 11-4 |
| RENAME | 11-6 |
| File input/output statements | 15-2 |
| File locking | 15-8 |
| ISAM PLUS files | 15-8 |
| Random files | 15-8 |
| Sequential files | 15-8 |
| File specifications | 5-11 |
| File statements | 15-9 |

| | |
|---|--------------|
| ALLOCATE | 15-9 |
| CLOSE | 15-10 |
| CLOSEK | 15-10 |
| FILEBASE | 15-11 |
| INPUT | 15-11 |
| INPUT LINE | 15-13 |
| KILL | 15-14 |
| LOOKUP | 15-15 |
| OPEN | 15-16 |
| PRINT | 15-19 |
| READ | 15-19 |
| UNLOKR | 15-21 |
| WRITE | 15-21 |
| File-channel numbers | 15-2, 15-10 |
| FILEBASE | 15-11 |
| FILEBLOCK | 10-12 |
| Filespec | 1-6 |
| FILL\$ | 10-6 |
| FIND | 19-5 |
| FIND'NEXT | 19-5 |
| FIND'PREV | 19-5 |
| Finding a numeric value of a string | 10-4 |
| Finding next/previous key | 19-5 |
| Finding string length | 10-8 |
| Finding the free bytes in memory | 11-3 |
| FIX(X) | 10-3 |
| FIXTRN | 10-13 |
| Floating point variables | 14-4 to 14-6 |
| Arrays | 14-4 |
| Format | 13-1 |
| FMOD | 7-2 |
| FOR | 9-7 |
| Formal parameter | 16-8 |
| Formal-parameter-list | 20-1 |
| Format of argument lists | E-2 |
| Formatted output | 12-1 |
| Formatted time/date | 10-13 |
| Formatting characters | 12-3 |
| Formatting examples | 12-8 |
| Formatting symbols | |
| ! | 12-5 |
| # | 12-3 |
| \$\$ | 12-6 |
| ** | 12-7 |
| , | 12-7 |
| - | 12-7 |
| . (period) | 12-5 |
| Z | 12-7 |
| \ | 12-4 |
| ^^^ | 12-8 |

| | |
|---------------------------------------|----------------------------|
| FRE(X) | 11-3 |
| Free memory usage | E-4 |
| FUNCTION | 22-7 |
| Functions | 10-1 |
| Numeric | 10-1 |
| Trigonometric | 10-4 |
| GET | 19-3 |
| GET'LOCKED | 19-3 |
| GET'NEXT | 19-4 |
| GET'NEXT'LOCKED | 19-4 |
| GET'NEXT'READ'ONLY | 19-4 |
| GET'PREV | 19-4 |
| GET'PREV'LOCKED | 19-4 |
| GET'PREV'READ'ONLY | 19-4 |
| GET'READ'ONLY | 19-3 |
| GETKEY(X) | 10-12 |
| Getting a record by key | 19-3 |
| Getting key input | 10-12 |
| Getting next/previous record | 19-4 |
| Getting statistical information | 19-8 |
| Going to another program | 9-2 |
| GOSUB | 1-4, 9-9 |
| GOTO | 1-4, 9-12 |
| Graphics conventions | 1-6 |
| Hexadecimal values | F-1 |
| Higher-level languages | 1-2 |
| I/O ports | 11-1 to 11-2 |
| IEEE | 5-4 |
| IF | 9-12 |
| INCLUDE statement | 19-2 |
| Increasing execution speed | C-2 |
| INDENT | 22-7 |
| INDEXED mode | 15-16 |
| Indexed sequential files | 19-1 |
| INDEXED'EXCLUSIVE mode | 15-16 |
| INDEXED'STATS | 19-8 |
| INPUT | 9-14, 15-2, 15-10 to 15-11 |
| Input from terminal | 10-12 |
| INPUT LINE | 9-17, 15-2, 15-13 |
| INPUT mode | 15-16 |
| INPUT RAW | 9-18, 15-2, 15-14 |
| Input/Output | 11-2 |
| Input/output statements | 15-2 |
| INSTR(X,A\$,B\$) | 10-6 |
| INT(X) | 10-3 |
| Integer data | 14-6 |
| Integer part of a number | 10-3 |

| | |
|---|---------------|
| Interactive BASIC | 1-3, 1-5, 3-1 |
| Interactive compiler options | 3-8 |
| Interactive mode | |
| Breakpoints | 4-1 |
| Compiling | 3-7 to 3-8 |
| Continuing the program | 4-2 |
| Deleting lines | 3-4 |
| Direct statements | 3-3 |
| Editing | 3-4 |
| Exiting | 3-9 |
| Listing a program | 3-7 |
| Loading programs | 4-3 |
| Prompt | 3-2 |
| Running a program | 3-8 |
| Saving programs | 3-5 |
| Single step | 4-4 |
| Stopping a program run | 4-3 |
| Interactive prompt | 3-2 |
| Interrupting programs | 4-3, 5-11 |
| IO(X) | 11-2 |
| ISAM files | 15-7 |
| ISAM PLUS | 19-1 |
| ALLOCATE'INDEXED | 19-8 |
| CLOSE | 19-7 |
| CLOSEK | 19-7 |
| CREATE'RECORD | 19-6 |
| DELETE'RECORD | 19-6 |
| Error processing | 19-9 |
| File structure | 19-2 |
| FIND | 19-5 |
| FIND'NEXT | 19-5 |
| FIND'PREV | 19-5 |
| GET | 19-3 |
| GET'LOCKED | 19-3 |
| GET'NEXT | 19-4 |
| GET'NEXT'LOCKED | 19-4 |
| GET'NEXT'READ'ONLY | 19-4 |
| GET'PREV | 19-4 |
| GET'PREV'LOCKED | 19-4 |
| GET'PREV'READ'ONLY | 19-4 |
| GET'READ'ONLY | 19-3 |
| INDEXED'STATS | 19-8 |
| OPEN | 19-3 |
| RELEASE'ALL | 19-7 |
| RELEASE'RECORD | 19-6 |
| UNLOKR | 19-7 |
| UPDATE'RECORD | 19-5 |
| ISAM PLUS statistical information | 19-8 |
| ISAM'KEY | 19-3 |
| ISAMP.BPI include file | 19-2 |

| | |
|--|---------------|
| ISO Latin-1 | F-1 |
| Key input | 10-12 |
| Keyboard echo | 9-6 |
| KILL | 11-5, 15-14 |
| LABEL | 22-7 |
| Labels | 1-4, 2-2, 5-2 |
| Largest integer <= a number | 10-3 |
| LCS(A\$) | 10-7 |
| Leading blanks | 9-22 |
| Leaving a loop | 9-7 |
| LEFT(A\$,X) | 10-7 |
| Left-relative position | 8-2 |
| LEN(A\$) | 10-8 |
| Length of strings | 10-8 |
| LET | 9-18, 22-7 |
| Library account | 18-1 |
| Library Searching | 2-4 |
| Line editing | 3-4 |
| Line numbers | 2-1, 3-4, 5-2 |
| Linefeed key | 4-4 |
| LIST | 3-7 |
| Listing an interactive program | 3-7 |
| Literals | 6-10 |
| Alternative radix | 6-11 |
| Character | 6-10 |
| Floating point | 6-12 |
| Integer | 6-10 |
| String | 6-12 |
| LOAD | 4-3, 5-3 |
| Loading a program into memory | 3-4, 4-3 |
| Loading files into system memory | 1-3 |
| Local-variable-list | 20-2 |
| Locating open subroutine files | E-4 |
| Locking a file | 19-6 |
| LOG(X) | 10-3 |
| LOG10(X) | 10-3 |
| Logarithm | 10-3 |
| Logarithm, decimal | 10-3 |
| Logical operators | 7-2 |
| AND | 7-2 |
| EQV | 7-2 |
| NOT | 7-2 |
| OR | 7-2 |
| XOR | 7-2 |
| Logical records | 15-5 |
| LONG(X) | 11-1 |
| LOOKUP | 11-4, 15-15 |
| Loop | 9-5, 9-7 |

| | |
|--------------------------------------|------------|
| Loop delimiters | 9-7 |
| Major revision number | 9-24 |
| Making programs more efficient | C-1 |
| Managing Memory | C-1 |
| MAP | 14-9 |
| Dim | 14-2 |
| Examples | 14-9 |
| Level | 14-3 |
| Origin | 14-2, 14-7 |
| Size | 14-2, 14-6 |
| Statement format | 14-2 |
| Type | 14-2, 14-5 |
| Value | 14-2, 14-6 |
| Variable name | 14-4 |
| MAP statement format | 14-2 |
| Matching strings | 10-6 |
| Mathematical expressions | 7-1 |
| Mathematical operators | 7-1 |
| AND | 7-2 |
| DIV | 7-2 |
| EQV | 7-2 |
| FMOD | 7-2 |
| MAX | 7-2 |
| MIN | 7-2 |
| MOD | 7-2 |
| NOT | 7-2 |
| OR | 7-2 |
| USING | 7-2 |
| XOR | 7-2 |
| MAX | 7-2 |
| Maximum line length | 2-4, 5-2 |
| MEM(X) | 11-3 |
| Values | 11-3 |
| Memory | 2-4 |
| Allocation | 2-4 |
| Efficiency | 3-9, C-1 |
| Memory Management | H-2 |
| Memory mapping | H-2 |
| Memory overlays | 14-7 |
| Memory Requirements | H-1 |
| MID(A\$,X,Y) | 10-8 |
| MIN | 7-2 |
| Minor revision number | 9-24 |
| MOD | 7-2 |
| Mode independence | 8-3, 10-1 |
| Monitor command level | 1-5, 3-2 |
| Multiple line functions | 20-2 |
| Multiple statement lines | 2-3, 3-3 |

| | |
|--------------------------------------|---|
| Nested loops | 9-8 |
| NEW | 3-3, 22-8 |
| NEXT | 9-7, 9-19 |
| NO'DIVIDE'BY'0 | 9-19 |
| NOECHO | 9-19 |
| NOT | 7-2 |
| Null strings | 9-14, 10-8, 15-11 |
| Numeric functions | 10-1 |
| Numeric variables | |
| Arguments | 10-1 |
| Binary | 14-5 |
| Conversion | 7-4 |
| Floating point | 14-5 |
| Functions | 10-2 |
| Value of a string | 10-4 |
| Object code | 1-3 |
| Object programs | 1-3, 3-6, 3-8, 5-3 |
| Octal values | F-1 |
| ODTIM | 10-13 |
| ON - GOTO | 9-21 |
| ON CALL | 9-20 |
| ON CTRLC GOTO | 9-19 |
| ON ERROR GOTO | 9-20, 17-1 |
| ON GOSUB | 9-20 |
| OPEN | 15-6, 15-16, 15-19, 15-21 |
| Modes | 15-18 |
| OPEN statement | 19-3 |
| Opening an ISAM PLUS file | 19-3 |
| Operator | 7-6 |
| Operator precedence | 7-3 |
| OR | 7-2 |
| OUTPUT mode | 15-16 |
| Overlaying variables in memory | 14-7 |
| PARAMETER | 16-8 |
| Pass by reference | 16-8 |
| Pass by value | 16-8 |
| Parentheses | 7-1 |
| Pause in a program | 9-28 to 9-29 |
| PEEK | 11-1 |
| PHDR macro | E-4 |
| Physical blocks | 15-5 |
| Physical memory | 11-1 |
| POKE | 11-1 |
| Ppn | 1-6 |
| PRINT | 7-6, 9-21, 12-1, 12-9, 15-2, 15-12, 15-19 |
| PRINT USING | 9-23, 9-28, 12-1, 15-19 |
| Print zones | 9-22 |
| PROCEED | 22-8 |

| | |
|--------------------------------|--------------------------------|
| Program | 9-24 |
| Compilation | 3-8 |
| Debugging | 3-1, 4-4 |
| Edit number | 9-24 |
| Execution | 3-8 |
| Form | 5-1 |
| Indentation | 2-3, 5-2 |
| Major revision number | 9-24 |
| Minor revision number | 9-24 |
| Pausing | 9-29 |
| Revision subscript | 9-24 |
| Run | 9-3 to 9-4, 9-9 |
| Statements | 9-1 |
| Version numbers | 11-6 |
| Program headers, subroutines | E-4 |
| Program wait | 9-28 |
| Prompt symbol | 3-2 |
| QUIT | 22-8 |
| Random files | 15-4 to 15-5 |
| Random mode | 15-6 |
| Random'Forced mode | 15-7 |
| RANDOM mode | 15-16 |
| Random numbers | 9-24, 10-3 |
| RANDOM'FORCED mode | 15-7, 15-16 |
| RANDOMIZE | 9-24 |
| Range checking | 14-6 |
| Raw data | 9-18, 15-14 |
| RDBMS | 21-1 |
| Re-entrant code | 1-3 |
| READ | 9-25, 15-2, 15-6, 15-17, 15-19 |
| READ'ONLY | 15-17 |
| READ'READ'ONLY | 15-20 |
| Readability of source programs | C-2 |
| READL | 15-2, 15-20 |
| READY | 3-2 |
| Record size | 15-5 |
| Record-number-variable | 15-17 |
| Register Parameters | E-2 |
| RELEASE'ALL | 19-7 |
| RELEASE'RECORD | 19-6 |
| REM | 2-2 |
| Remarks | 2-2 |
| REMOVE | 22-8 |
| Removing trailing spaces | 10-10 |
| RENAME | 9-26, 11-6 |
| Renaming disk files | 11-6 |
| REPEAT | 9-26 |
| Reserved words | 14-13, B-1 |

| | |
|--|--------------------------------|
| RESTORE | 9-25, 9-27 |
| RESUME | 9-20, 9-27, 17-3 to 17-4, 22-8 |
| RETURN | 9-9, 9-27 |
| Revision level | 9-24 |
| Revision subscript | 9-24 |
| RIGHT(A\$,X) | 10-9 |
| Right-relative position | 8-2 |
| RND(X) | 9-24, 10-3 |
| RNDN(X) | 10-4 |
| Rounding | 10-4 |
| RTN instruction | E-2 |
| RUN | 3-8, 22-8 |
| Run-time package | 1-3, 1-5, 5-3 |
| Running a program | 5-11 |
| Running an interactive program | 3-8 |
| RUNP | 1-5, 5-11, 9-28, 15-9, 22-10 |
| RUNP.LIT | 1-4 |
| Runtime options | 5-11 |
| SAVE | 3-5 |
| Saving an .RP file | 3-6 |
| Saving an interactive program | 3-5 |
| SCALE | 9-27, 13-2, 22-8 |
| Scaling factor | 13-2 |
| Screen handling codes | 12-9, D-1 |
| SEARCH | 22-5 |
| Search path | 2-4 |
| Searching for a Library | 2-4 |
| Searching for disk files | 11-4, 15-15 |
| Semicolon symbol (;) | 8-1, 9-22 |
| Sequential files | 15-3 |
| Append | 15-3 |
| Input | 15-4 |
| Output | 15-3 |
| SGN(X) | 10-4 |
| Sign | 10-4 |
| SIGNIFICANCE | 9-27, 22-8 |
| SIN(X) | 10-4 |
| Single line user-definable functions | 20-1 |
| Single-step | 3-1, 4-4 |
| SLEEP | 9-28 |
| Source programs | 3-6, 4-3, 9-1 |
| Readability | C-2 |
| SPACE(X) | 10-9 |
| SPAN'BLOCKS | 15-5, 15-17 |
| Spanning blocks | 15-5 |
| Speed | C-2 |
| SQR(X) | 10-4 |
| Square root | 10-4 |
| Statement modifiers | 9-1 |

| | |
|---------------------------------|------------------------|
| Status variable | 19-3 to 19-5 |
| STEP | 9-7, 9-28 |
| STOP | 4-5, 9-29 |
| Stopping a program | 9-6 |
| STR(X) | 7-7, 10-10 |
| String arguments | 10-1 |
| String conversion | 7-4 |
| String data | 14-5 |
| String functions | 10-5 |
| String variables | 14-5 |
| Arrays | 9-4 |
| Strings | 6-12 |
| Including quotation marks | 6-12 |
| STRIP\$ | 10-10 |
| STRSIZ | 9-4, 9-29 |
| Structured program | 5-2 |
| SUBCALL | 16-5 |
| Argument passing | 16-8 |
| Arguments | 16-8 |
| SUBEND | 16-5, 16-9 |
| Subprogram | 16-1, 16-3, 16-5, 22-9 |
| Arguments | 16-7 |
| Name | 16-7 |
| Parameters | 16-7 |
| Saved environment | 16-11 |
| Subprogram name | 16-7 |
| Subprograms | |
| .SPG extension | 16-5 |
| Calling | 16-5 |
| CHAIN | 16-11 |
| Defining | 16-5 |
| END | 16-9 |
| Error trapping | 16-9 |
| External | 16-5 |
| Internal | 16-5 |
| Search path | 16-6 |
| Subroutine | 9-6, 9-9 |
| Library | E-4 |
| Linking | 1-2 |
| Loading | 18-2 |
| Program headers | E-4 |
| Subscripts | 8-3, 9-4 |
| Substrings | 8-1 to 8-2, 10-6 |
| Modifiers | 8-1, 8-3 |
| Overflow | 8-2 |
| Truncation | 8-2, 14-5 |
| SWITCH | 9-29 |
| Syntax | 11-1 |
| Syntax errors | 3-4 |
| SYSLIB.LIB | E-3 to E-4 |

| | |
|---------------------------------------|--------------------|
| System functions | 11-1, 16-2 |
| System memory | 1-3 |
| System subroutine library | E-3 |
| TAB | 12-9 |
| TAB functions | 12-9 |
| TAN(X) | 10-4 |
| TCRT codes | 12-9, D-1 |
| Terminal input | 10-12 |
| THEN | 9-12, 9-31 |
| TIME | 10-10, 10-13, 11-4 |
| Timesharing | 1-3 |
| Timing processes | 11-4 |
| TO | 9-31 |
| TRACE | 22-9 |
| Trailing blanks | 9-22 |
| Trailing spaces | 10-10 |
| Transferring to another program | 9-2 |
| Translation tables | 10-13 |
| TRAP | 22-5 |
| Trapping errors with Control-C | 17-4 |
| Trigonometric functions | 10-4 |
| ACS(X) | 10-4 |
| ASN(X) | 10-4 |
| ATN(X) | 10-4 |
| COS(X) | 10-4 |
| DATN(X,Y) | 10-4 |
| SIN(X) | 10-4 |
| TAN(X) | 10-4 |
| Truncation | 10-3 |
| Turning off keyboard echo | 9-19 |
| Turning on the keyboard | 9-6 |
| Type codes | E-1 |
| Type independence | 7-4 |
| UCS(A\$) | 10-10 |
| Underscore | 6-1 |
| Unformatted variables | 14-5 |
| UNIFY | 21-1 |
| Unlocking | |
| All records | 19-7 |
| Files | 15-21 |
| Last accessed record | 19-6 to 19-7 |
| Record | 19-6 |
| Record by number | 19-6 |
| UNLOKR | 15-21, 19-7 |
| UNTRACE | 22-9 |
| UNTRAP | 22-5 |
| UPDATE'RECORD | 19-5 |
| Updating a record | 19-5 |

| | |
|--------------------------------------|--------------------------|
| Use of free memory | E-4 |
| User definable functions | 20-1 |
| Examples | 20-5 |
| User impure area | E-2 |
| User memory partition | 1-3 |
| USING | 7-2, 9-31, 12-1 |
| Using command files from BASIC | 16-2 |
| Using defined functions | 20-4 |
| Using subprograms | 16-10 |
| | |
| VAL(A\$) | 7-7, 10-4 |
| Value | 9-18 |
| VARIABLE | 22-9 |
| Variables | 6-1, 7-4 |
| Length | 1-4 |
| Mapping | 14-1 |
| Names | 1-4, 6-1 |
| Trees | 14-12 |
| Type code | E-3 |
| VER\$ | 11-6 |
| Version numbers | 11-6 |
| | |
| WAIT'FILE | 15-8, 15-17 |
| Word boundaries | 14-12 |
| WORD(X) | 11-1 |
| WRITE | 15-2, 15-6, 15-17, 15-21 |
| WRITEL | 15-2, 15-22 |
| WRITELN | 15-22 |
| WRITEN | 15-22 |
| | |
| XCALL | 9-31, 18-2 |
| XCALL loading | 18-2 |
| XOR | 7-2 |
| | |
| Z | 12-7 |
| | |
| \ | 12-4 |
| | |
| ^ | 3-4 |
| ^^^ | 12-8 |