OPTIMIZED SYSTEMS SOFTWARE

OSS BASIC A+

for the ATARI 800 (R)

MAY 1981

Version 3.1

NOTICE

Optimized Systems Software
10379 Lansdale Avenue
Cupertino, CA   95014
Telephone: 408-446-3099

NOTE: Sections Marked with an asterisk (*) are new or substantially changed from standard Atari Basic.

PREFACE

1 GENERAL INFORMATION

Terminology	1
Special Notations Used in This Manual	3
Abbreviations Used in This Manual	4
Operating Modes	5
Special Function Keys	5
*Arithmetic Operators	6
*Operator Precedence	7
Built-In Functions	7
Graphics	8
Sound and Games	8
Wraparound and Keyboard Rollover	8
Error Messages	8

2 PROGRAM DEVELOPMENT COMMANDS

BYE	9
CONT	9-A
END	9
LET	10-A
LIST	10
NEW	10
REM	10
RUN	11
STOP	11
*Advanced Program Development Commands	12-A
*TRACE/TRACEOFF	12-A
*LVAR	12-A
*LOMEM	12-A
*DEL	12-A

3 EDIT FEATURES

Screen Editing	13
Control (CTRL) Key	13
Shift Key	13
Double Key Functions	14
Cursor Control Keys	14

Keys Used with CTRL Key 14
Keys Used with Shift Key 14
Special Function Keys 14
Break Key 14
Escape Key 14

4 PROGRAM CONTROL STATEMENTS

FOR...TO...STEP / NEXT 15
GOSUB / RETURN 16
GOTO 17
IF...THEN 18
ON...GOSUB 20
ON...GOTO 20
RESTORE 21
*TRAP 22
*Advanced Program Control Statements 22-A
*IF...ELSE...ENDIF 22-A
*WHILE / ENDWHILE 22-B

5 INPUT/OUTPUT COMMANDS

Input/Output Devices 23
CLOAD 24
CSAVE 24
DOS and ~~ØP~~ OS 25
ENTER 25
INPUT 25
LOAD 26
LPRINT 26
NOTE 26
OPEN and CLOSE 26
POINT 28
PRINT 28
PUT and GET 28
READ and DATA 28
SAVE 29
*STATUS 29
XIO 30
Chaining Programs 30
*Advanced Input / Output Commands 32-A
*INPUT "..." 32-A
*DIR 32-A
*PROTECT and UNPROTECT 32-B
*ERASE 32-B
*RENAME 32-B
*PRINT USING 32-C
*TAB (as a statement) 32-G
*BPUT 32-H
*BGET 32-H
*RPUT 32-H
*RGET 32-I

6 FUNCTION LIBRARY

Arithmetic Functions	33
ABS	33
CLOG	33
EXP	33
INT	33
LOG	34
RND	34
SGN	34
SQR	34
Trigonometric Functions	34
ATN	34
COS	34
SIN	35
DEG / RAD	35
Special Purpose Functions	35
ADR	35
FRE	35
PEEK	35
POKE	35
USR	36
*Advanced Functions	36-A
*DPEEK / DPOKE	36-A
*ERR	36-A
*TAB	36-B

7 STRINGS

ASC	37
CHR$	37
LEN	38
STR$	38
VAL	38
String Manipulations	39
*Advanced Strings	40-A
*Substrings	40-A
*FIND	40-B

8 ARRAYS AND MATRICES

DIM	41
CLR	43

9 GRAPHICS MODES AND COMMANDS

GRAPHICS 45
Graphics Modes 45
Mode 0 46
Mode 1 and 2 46
Modes 3, 5, and 7 47
Modes 4 and 6 48
Mode 8 47
COLOR 48
DRAWTO 48
LOCATE 48
PLOT 49
POSITION 49
PUT / GET (as applied to graphics) 49
SETCOLOR 50
XIO (special FILL application) 54
Assigning Colors to Text Modes 54
Graphics Control Characters 56

10 SOUND AND GAME CONTROLLERS

SOUND 57
PADDLE 59
PTRIG 59
STICK 59
STRIG 60
*Advanced Game Control 60-A
*HSTICK 60-A
*VSTICK 60-A
*PEN 60-A

11 ADVANCED PROGRAMMING TECHNIQUES AND INFORMATION

Memory Conservation 61
Programming in Machine Language 63
*Numbers (BASIC A+ numeric representation) 68-A

12 ADVANCED SYSTEM FEATURES

*SET and SYS 69
*MOVE 71

13 PLAYER / MISSILE GRAPHICS

*An Overview	72
*Conventions	73
figures PMG-1 and PMG-2	74
*The PMG Statements	75
*PMGRAPHICS	75
*PMCLR	75
*PMCOLOR	76
*PMWIDTH	76
*PMMOVE	77
*MISSILE	78
*The PMG Functions	79
*PMADR	79
*BUMP	79
*PMG Related Statements	80
*POKE and PEEK	80
*MOVE	80
*BGET and BPUT	81
*USR	81
*Example Player/Missile Graphics PROGRAMS	82

APPENDIX A	Atari Basic Reserved Words
APPENDIX B	* Error Messages
APPENDIX C	ATASCII Character Set
APPENDIX D	* Atari 400/800 Memory Map
APPENDIX E	Derived Functions
APPENDIX F	Printed Versions of Control Characters
APPENDIX G	Glossary
APPENDIX H	User Programs
APPENDIX I	* Memory Locations
APPENDIX J	* Compatabilities
APPENDIX K	* SYNTAX SUMMARY AND KEYWORD INDEX
APPENDIX L	* BASIC A+ Memory Usage

ABOUT THIS MANUAL

This BASIC A+ manual is intended as an "add-on" or appendix to the "BASIC REFERENCE MANUAL" supplied by Atari, Inc. Make sure that your BASIC REFERENCE MANUAL is Atari part number C-015307, REV. 1 !!

GETTING STARTED

To use BASIC A+ with OS/A+:

Place the OS/A+ master disk in drive 1 and turn on the power in the same manner used to boot an Atari disk.

In response to the OS/A+ prompt "D1:", simply type in "BASIC [return]" and BASIC A+ will load and run.

If you exit from BASIC A+ to OS/A+ (via DOS or CP commands or via the RESET key), you may return to BASIC A+'s warmstart point by simply entering RUN to OS/A+. NOTE: see OS/A+ manual for circumstances under which this does not work. If necessary, you may use 'RUN addr' from OS/A+ to enter at BASIC A+'s coldstart or warmstart address. See table below for those addresses.

To use BASIC A+ with Atari's DOS:

Boot an Atari master diskette, and enter the Atari menu DOS.

Put the diskette with BASIC A+ in a disk drive and use the Atari LOAD BINARY FILE from the menu to load BASIC A+.

Use the Atari RUN AT ADDRESS menu command to do a "coldstart" of BASIC A+. The address to use depends upon the amount of free RAM in your system.

If you exit BASIC A+ (via the DOS or CP commands), you may return without losing any program currently in memory by using the Atari menu RUN AT ADDRESS command to do a "warmstart". Again, the warmstart address depends upon the amount of free RAM.

size of free RAM	32k	40k	48k
coldstart address	4400	6400	8400
warmstart address	4403	6403	8403

SPECIAL SUPPLEMENT TO

GETTING STARTED

for

OSS BASIC A+

USING BASIC A+ WITH ATARI DOS.

If You have purchased the OSS BASIC A+ package without CP/A. The diskette in your package contains the OSS file "BASIC.OBJ" as well as the ATARI DOS (version 2S, including DOS.SYS and DUP.SYS).

The first thing you should do is make a working copy of your disk. To do this, simply use the duplicate disk MENU command of Atari's DOS. You may follow these steps:

1. If you have 48K bytes of RAM (and we recommend this amount), remove all cartridges from your computer.
2. Turn disk drive(s) power on; insert your OSS diskette into drive 1.
3. Turn computer power on. After a few moments, the disk will boot and display the Atari DOS 2S "menu". Use the LIST DIRECTORY option to verify that the diskette contains the appropriate files.
4. Use the MENU to format a new disk. Then use DUPLICATE DISK to copy your OSS disk to this new (working) disk.
5. Use the MENU to create MEM.SAV on the working disk.

You now have a working disk set up and are ready to use BASIC A+.

BASIC.OBJ is a standard Atari-format load-and-go file, so you may load and run BASIC A+ from DOS 2S by simply using the LOAD BINARY FILE menu command (respond with BASIC.OBJ when asked for the load file name). BASIC A+ will take a few moments to load, after which an OSS message will be displayed, and you may start using the system.

You may return to DOS via the DOS comand (though you should have less reason to now, since the more commonly used DOS menu functions are available as BASIC A+ statements). In general, you will then have to reenter BASIC A+ via the "coldstart" entry point (see the BASIC A+ manual for the specific address) by using the RUN AT ADDRESS command from the MENU. Note, however, that the load-and-go startup process places a jump to BASIC A+'s warmstart in location 630 (hex), which may be useful under some circumstances.

If you purchase CP/A at a later date, simply copy the file BASIC.OBJ from you Atari DOS disk to a file called BASIC.COM on the CP/A disk. You may then simply type "BASIC" at the CP/A command level to use BASIC A+.

CAUTIONS

The BASIC A+ manual is intended as a set of changes and addenda to the Atari BASIC Reference Manual. PLEASE be sure to use them to update your manual as soon as possible...it will make looking up information much easier, since the changes fit into logical places in the Atari manual.

ALSO be sure to read the notes about bugs, errors, etc., below. Pay special attention to our Appendix J.

ABOUT BUGS AND ERRORS

Probably no software product will ever be released that is totally free of bugs, and BASIC A+ is no exception. However, we have what we hope is a relatively painless method of dealing with bugs. On your disk you will find a file named 'BASIC300.COM' (or 'BASIC300.OBJ' if you purchased BASIC A+ only). This file is actually our first, original released version of BASIC A+ (known inexplicably as version 3.00). For as long as practicable, we intend to distribute this original version; in this way, all our customers start from the same common base, and all fixes sent out apply to all customers.

HOWEVER, on your disk you will also find a file named 'BASIC.COM' (or 'BASIC.OBJ'). This file is the most recent version of BASIC A+; it is actually a patched copy of version 3.00, and the patch was made under the control of the program known as 'BPATCH.SAV' (which is also on your disk). So you can load and run BASIC from your disk knowing that you have the most recent, bug-free version. And then, when the bugs appear, we will send all registered software licensees a list of lines to be added to BPATCH.SAV; you enter a few simple lines of code, run a program, and you once again have a bug-free program (until the next time).

ABOUT VERSION 3.04

This is the version you will find on your disk. With this release we have tried to eliminate all known incompatibilities between Atari BASIC and BASIC A+. In earlier versions, we had offered some 'improvements' to the language which, unfortunately, had the effect of making some programs (particularly those dealing with RS-232 I/O) do strange and nasty things to the computer. We would still, however, like to call your attention to Appendix J.

PLEASE, PLEASE call us if you discover other incompatibilities and/or bugs that we ought to know about. We really do want to help you.

ABOUT BPATCH.SAV

On the following pages is the listing for BPATCH.SAV, our BASIC A+ patching program (mentioned above). The listing is included here in case you don't have a printer: when we send out future patches to be integrated into BASIC A+ via this program, it will be much easier on you if you can refer to a listing. In any case, if nothing else, this program shows that you CAN write serious utilities in BASIC A+.

OPTIONAL:

You will note that when you LIST a program with control structures (IF...ENDIF, WHILE, FOR), BASIC A+ automatically indents each level of control structure. If you would like to save room on a disk LISTing by eliminating the indentations, you may use the following patch. The patch to restore indenting is also given.

memsize	32k	40k	48k
no indentation	dpoke 25484,2768	dpoke 33676,2768	dpoke 41868,2768
indentation	dpoke 25484,8264	dpoke 33676,8264	dpoke 41868,8264

CUTE TRICK:

Find out how much total ram (in K bytes) you have by the following program segment:

```
PMG. 1 : PRINT 1 + PMADR(0)/1024 ; " K Bytes " : PMG. 0
```

DIFFERENT VERSIONS

Two versions of BASIC A+, version 3.00, have been distributed. For persons NOT also ordering CP/A, the file BASIC.OBJ is sent on an Atari DOS 2S diskette. With CP/A, the filename is BASIC.COM . BASIC.OBJ may be used on a CP/A system without difficulty by simply RENAMing it, but BASIC.COM is clumsy and difficult to use under Atari DOS. The patches listed below are for BASIC.COM, but the minor changes needed for BASIC.OBJ are given after the main listing.

```
10 REM BPATCH -- PROGRAM TO PATCH BASIC A+
20 REM
30 REM THIS LISTING CONVERTS REV 3.00 TO REV 3.04
40 REM
100 REM *** ESTABLISH MEMORY SIZE, ENSURE PATCH PROGRAM WILL WORK ***
110 PMGRAPHICS 1:ADDR=PMADR(0)/1024+1:PMGRAPHICS 0
120 ADDR=(ADDR-32)*1024:REM LINE 110 SET ADDR = NUMBER OF K BYTES
130 POKE 28343+ADDR,7:POKE 28350+ADDR,11:REM WORKS ON ALL SIZES
200 REM ***** FIRST, OPEN FILES, ETC. *****
210 DIM FIX$(2000)
220 OPEN #1,4,0,"D:BASIC300.COM":OPEN #2,8,0,"D:BASIC.COM"
300 REM ***** MOVE AND CHANGE RELOCATER AND BIT MAP *****
310 BGET #1,ADR(FIX$),267:BPUT #2,ADR(FIX$),267
320 CURRENT=0:GOSUB 700:REM THUS ACTING ON BIT MAP
330 BGET #1,ADR(FIX$),1701-CURRENT:BPUT #2,ADR(FIX$),1701-CURRENT
400 REM ***** MOVE ACTUAL PROGRAM, BASIC A+ *****
410 CURRENT=17408:GOSUB 700:REM THIS DOES THE REAL WORK
500 TRAP 530:PRINT "CLEAN UP"
510 WHILE 1:GET #1,BYTE:PUT #2,BYTE
520   ENDWHILE :REM LOOPS FOREVER...UNTIL ERROR OCCURS
530 IF ERR(0)<>136:PRINT "UNEXPECTED ERROR # ";ERR(0)
540   ELSE :CLOSE #1:CLOSE #2:PRINT "NORMAL END"
550   ENDIF :END
700 REM ******* ACTUAL READ AND CHANGE ROUTINE *******
710 READ ADDR:REM JUST TO GET THE LOOP STARTED
720 WHILE ADDR:READ MEMNEW,MEMOLD:SKIP=ADDR-CURRENT
730   PRINT "ADDR ";ADDR,"SKIP ";SKIP
740   WHILE SKIP>1000:SKIP=SKIP-1000
750     BGET #1,ADR(FIX$),1000:BPUT #2,ADR(FIX$),1000:ENDWHILE
800   WHILE SKIP:SKIP=SKIP-1
810     GET #1,BYTE:PUT #2,BYTE:ENDWHILE
820   GET #1,BYTE:REM BYTE TO BE PATCHED, SHOULD MATCH DATA
830   IF BYTE<>MEMOLD THEN PRINT "OUT OF SYNC":STOP
840   PUT #2,MEMNEW:CURRENT=ADDR+1:READ ADDR
850   ENDWHILE
890 RETURN
1000 REM ******** DATA FOR BIT MAP CHANGES *********
1020 DATA 1422,4,0
1030 DATA 1423,9,0
1040 DATA 1424,32,0
1050 DATA 1425,36,0
1490 DATA 0
1500 REM ******** DATA FOR PROGRAM CHANGES *********
1519 REM .             VERSION NUMBER
1520 DATA 17453,52,48
1540 DATA 17564,96,169
1549 REM .             PATCH 'LVAR'
1550 DATA 20974,112,98
1551 DATA 20975,135,91
1560 DATA 23273,204,193
1570 DATA 24034,232,45
1571 DATA 24035,93,116
1599 REM .             ERROR 137
1600 DATA 27764,140,113
1620 DATA 28077,234,232
1621 DATA 28078,234,232
1622 DATA 28103,75,76
1623 DATA 28107,189,157
```

```
1629 REM .              PATCH 'STATUS'
1630 DATA 28238,76,32
1631 DATA 28239,120,127
1632 DATA 28240,112,116
1640 DATA 28327,112,148
1641 DATA 28328,112,116
1660 DATA 28343,7,4
1661 DATA 28350,11,8
1670 DATA 28409,11,8
1680 DATA 28506,7,4
1689 REM .              PATCH 'MOVE'
1690 DATA 28645,165,56
1691 DATA 28646,212,165
1692 DATA 28647,133,212
1693 DATA 28648,162,233
1694 DATA 28649,73,1
1695 DATA 28650,255,133
1696 DATA 28651,168,162
1698 DATA 28654,234,233
1699 DATA 28655,234,0
1700 DATA 28686,136,73
1701 DATA 28687,152,255
1740 DATA 28784,32,0
1741 DATA 28785,81,0
1742 DATA 28786,218,0
1743 DATA 28787,76,0
1744 DATA 28788,148,0
1745 DATA 28789,116,0
1749 REM .              PATCH 'STATUS'
1750 DATA 28792,169,0
1751 DATA 28793,13,0
1752 DATA 28794,32,0
1753 DATA 28795,170,0
1754 DATA 28796,116,0
1755 DATA 28797,32,0
1756 DATA 28798,127,0
1757 DATA 28799,116,0
1758 DATA 28800,76,0
1759 DATA 28801,177,0
1760 DATA 28802,116,0
1769 REM .              PATCH 'LVAR'
1770 DATA 28808,32,0
1771 DATA 28809,92,0
1772 DATA 28810,98,0
1773 DATA 28811,76,0
1774 DATA 28812,213,0
1775 DATA 28813,102,0
1879 REM .              FOR GTIA CHIP
1880 DATA 29188,173,141
1899 REM .              PATCH 'BUMP'
1900 DATA 29705,234,141
1901 DATA 29706,234,30
1902 DATA 29707,234,208
1990 DATA 30287,223,221
1999 DATA 0
```

CHANGES NEEDED FOR BASIC.OBJ

Only two lines differ from the program given above:

```
220 open #1,4,0,"D:BASIC.OBJ" : open #2,8,0,"D:BASIC303.OBJ"

310 bget #1,adr(fix$),281 : bput #2,adr(fix$),281
```

IF YOU DIDN'T BUY OS/A+

Are you serious about assembly language programming? Do you like the idea of having your software in RAM, where the bugs can be patched? Can you make use of a professional operating system? OS/A+ is a logical companion to BASIC A+, and we hope to be able to offer many more compatible products in the future. Ask your dealer for more information. Or ask us for a brochure and a list of OSS dealers.

SOFTWARE AUTHORS

We are interested in quality software which is compatible with OS/A+ and/or BASIC A+. We can offer customized versions of BASIC A+ which offer better program security. This service is available on a royalty basis to outside authors, or it can be included with programs which we distribute for you. Please call or write for more information.

CONT
(CON.)

```
Format:         CONT
Example:        CONT
                100 CONT
```

In direct mode, this command resumes a program after a STOP statement or BREAK key abort or any stop caused by an error.

Caution: Execution resumes on the line following the halt. Statements on the same line as and following a STOP or error will not be executed.

In deferred mode, CONT may be used for error trap handling.

```
Example:        10 TRAP 100
                20 OPEN #1,12,0,"D:X"
             ┌→ 30
             │  ..
             │  ..
             └─ 100 IF ERR(0)=170 THEN
                    OPEN #1,8,0,"D:X":CONT
```

In line 20 we attempt to open a file for updating. If the file does not exist, a trap to line 100 occurs. If the "FILE NOT FOUND" error occured, the file is opened for output (and thus created) and execution continues at line 30 via "CONT".

/

```
Format:             [LET] avar=aexp
                    [LET] svar=sexp[,sexp...]
Exapmle:            LET X=3.5
                    LET LETTER$="a"
                    A$="*",A$,A$,A$,A$,A$
```

Normally an optional keyword, LET must be used to assign a value to a variable name which starts with (or is identical to) a reserved name.

String concattenation may be accomplished via the for shown in the last example above . Note that a concatenation of the form

```
        A$=B$,C$
```

is exactly equivalent to

```
        A$=B$
        A$(LEN(A$)+1)=C$
```

```
Examples:           DIM A$(100),B$(100)
                    A$="123"
                    B$="ABC"
                    A$=A$,B$,A$

          (At this point, A$= "123ABC123ABC")

                    A$(4,9)="X",STR$(3*7),"X"

          (At this point, A$="123X21X23ABC")

                    A$(7)=A$(1,3)

          (Finally, A$="123X21123")
```

ADVANCED PROGRAM DEVELOPMENT COMMANDS

TRACE

TRACEOFF

Formats:	TRACE
	TRACEOFF
Examples:	100 TRACE
	TRACEOFF

These statements are used to enable or disable the line number trace facility of BASIC A+. When in TRACE mode, the line number of a line about to be executed is displayed on the screen surrounded by square brackets.

Exceptions: The first line of a program does not have its number traced. The object line of a GOTO or GOSUB and the looping line of FOR or WHILE may not be traced.

Note: A direct statement (e.g., RUN) is TRACED as having line number 32768.

LVAR

Format:	LVAR filename
Example:	LVAR "E:"

This statement will list (to any file) all variables currently in use. The example will list the variables to the screen. Strings are denoted by a trailing '$', arrays by a trailing '('.

LOMEM

Format:	LOMEM addr
Example:	LOMEM DPEEK(128)+1024

This command is used to reserve space below the user's program space. The user then might use the space for assembly language routines. The usefulness of this may be limited, though, since there are other more usable reserved areas available.

Caution: LOMEM wipes out any user program currently in memory.

DEL

```
Format:          DEL line[,line]
Example:         DEL 1000,1999
```

DEL deletes program lines currently in memory. If two line numbers are given (as in the example), all lines between the two numbers (inclusive) are deleted. A single line number deletes a single line.

Example:

```
        100  DEL 1000,1999
        110  SET 9,1:TRAP 1000
        120  ENTER "D:OVERLAY1"
        1000 REM THESE LINES ARE DELETED BY
        1010 REM LINE 100
        1020 REM
        1030 REM PRESUMABLY THEY WILL BE
        1040 REM OVERLAID BY THE ENTERED PROGRAM
        1990 REM SEE 'ENTER' AND 'SET' FOR
        1999 REM MORE INFO
```

ADVANCED PROGRAM CONTROL

BASIC A+ adds Structured Programming capability with two new Program Control Structures.

IF...ELSE...ENDIF

Format:

```
IF aexp: <statements>
[ELSE: <statements> ]
 ENDIF
```

Examples:

```
200 IF A>100:PRINT "TOO BIG"
210 A=100
220 ELSE:PRINT "A-OK"
230 ENDIF


1000 IF A>C : B=A : ELSE : B=C : ENDIF
```

BASIC A+ makes available an exceptionally powerful conditional capability via IF...ELSE...ENDIF

In the format given, if the expression evaluates non-zero then all statements between the following colon and the corresponding ELSE (if it exists) or ENDIF (if no ELSE exists) are executed; if ELSE exists, the statements between it and ENDIF are skipped.

If the aexp evaluates to zero, then the statements (if any) between the colon and ELSE are skipped and those between ELSE and ENDIF are executed. If no ELSE exists, all statements through the ENDIF are skipped.

The colon following the aexp IS REQUIRED and MUST be followed by a statement. The word THEN is NOT ALLOWED in this format

There may be any number (including zero) of statements and lines between the colon and the ELSE and between the ELSE and the ENDIF.

The second example above sets B to the larger of the values of A and C.

Note: IF structures may be nested.

Example:

```
100 if A>B : REM SO FAR A IS BIGGER
110   IF A>C : PRINT "A BIGGEST"
120   ELSE : PRINT "C BIGGEST"
130   ENDIF
140 ELSE
150   IF B>C : PRINT "B BIGGEST"
160   ELSE : PRINT "C BIGGEST"
170   ENDIF
180 ENDIF
```

WHILE

ENDWHILE

Format: WHILE aexp : <statements> : ENDWHILE

Example:

```
100 A=3
110 WHILE A: PRINT A
120   A=A-1 : ENDWHILE
```

With WHILE, the BASIC A+ user has yet another powerful control structure available. So long as the aexp of WHILE remains non-zero, all statements between WHILE and ENDWHILE are executed.

Example: WHILE 1 : ....
The loop executes forever

Example: WHILE 0 : ....
The loop will never execute

Caution: Do not GOTO out of a WHILE loop or a nesting error will likely result. (though POP can fix the stack in emergencies.)

Note: The aexp is only tested at the top of each passage through the loop.

Note: As with ALL BASIC A+ control structures, WHILEs may be nested as deep as memory space allows.

ADVANCED INPUT/OUTPUT

INPUT

```
Format:         INPUT string-literal,var[,var..]
Example:        INPUT "3 VALUES >>",V(1),V(2),V(3)
```

BASIC A+ allows the user to include a prompt with the INPUT statement to produce easier to write and read code. The literal prompt ALWAYS replaces the default ("?") prompt. The literal string may be nul for no prompt at all.

Note: No file number may be used when the literal prompt is present.

Note: In the example above, if the user typed in only a single value followed by RETURN, he would be reprompted by BASIC A+ with "??". But see chapter 12 for variations available via SET.

DIR

```
Format:         DIR filespec
Example:        DIR "D:*.COM"
```

List the contents of a directory to the screen. Action is similar to OS/A+ DIR command, but there are no default file specifications. The example above would list all COMmand files on drive 1.

PROTECT

UNPROTECT

```
Format:            PROTECT filespec
                   UNPROTECT filespec
Examples:          PROTECT "D:*.COM"
                   100 UNPROTECT "D2:JUNK.BAS
```

PROTECTing a file implies that the file cannot be erased or written to. UNPROTECT eliminates any existing protection. Similar to OS/A+ PROtect and UNProtect, but there are no default file specifications. In the examples, the first would protect all command files on drive 1 and the second would unprotect only the file shown.

ERASE

```
Format:            ERASE filespec
Example:           ERASE "D:*.BAK
```

Erase will erase any unprotected files which match the given filespec. The example would erase all .BAK (back-up) files on drive 1. Similar to OS/A+ ERAse, but there are no default file specifiers.

RENAME

```
Format:            RENAME <filespec,filename>
Example:           RENAME "D2:NEW.DAT,OLD.BAK"
```

Allows renaming file(s) from BASIC A+. Note that the comma shown MUST be imbedded in the string literal or variable used as the file parameter.

Caution: It is strongly suggested that wild cards (* and ?) NOT be used when RENAMing.

PRINT USING

```
Format:          PRINT   [#fn;]USING sexp,exp [,exp...]
Example:         (see below)
```

PRINT USING allows the user to specify a format for the output to the device or file associated with "fn" (or to the screen). The format string "sexp" contains one or more format fields. Each format field tells how an expression from the expression list is to be printed. Valid format field characters are:

```
# & * + - $ , . % ! /
```

Non-format characters terminate a format field and are printed as they appear.

```
Example 1) 100 PRINT USING "## ###X#",12,315,7

        2) 100 DIM A$(10) : A$="## ###X#"
           200 PRINT USING A$,12,315,7
```

Both 1) and 2) will print

```
12 315X7
```

Where a blank separates the first two numbers and an X separates the last two.

NUMERIC FORMATS:

The format characters for numeric format fields are:

```
# & * + - $ , .
```

DIGITS (# & *)

Digits are represented by:

```
# & *
```

# - Indicates fill with leading blanks
& - Indicates fill with leading zeroes
* - Indicated fill with leading asterisks

If the number of digits in the expression is less than the number of digits specified in the format then the digits are right justified in the field and preceded with the proper fill character.

NOTE: In all the following examples b is used to represent a blank.

Example:

```
Value          Format Field        Print Out
```

1	###	bb1
12	###	b12
123	###	123
1234	###	234
12	&&&	012
12	***	*12

DECIMAL POINT(.)

A decimal point in the format field indicates that a decimal point be printed at that location in the number. All digit positions that follow the decimal point are filled with digits. If the expression contains fewer fractional digits than are indicated in the format, then zeroes are printed in the extra positions. If the expression contains more fractional digits than indicated in the format, then the expression is rounded so that the number of fractional digits is equal to the number of format positions specified.

A second decimal point is treated as a non-format character.

Example:

Value	Format Field	Print Out
123.456	###.##	123.46
4.7	###.##	bb4.70
12.35	##.##.	12.35.

COMMA (,)

A comma in the format field indicates that a comma be printed at that location in the number. If the format specifies a comma be printed at a position that is preceeded only by fill characters (0 b *) then the appropriate fill character will be printed instead of the comma.

The comma is a valid format character only to the left of the decimal point. When a comma appears to the right of a decimal point, it becomes a non-format character. It terminates the format field and is printed like a non-format character.

Example:

Value	Format Field	Print Out
5216	##,###	b5,216
3	##,###	bbbbb3
4175	**,***	*4,175
3	&&,&&&	000003
42.71	##.##,	42.71,

SIGNS (+ -)

A plus sign in a format field indicates that the sign of the number is to be printed. A minus sign indicates that a minus sign is to be printed if the number is negative and a blank if the number is positive.

Signs may be either fixed, floating or trailing.

A fixed sign must appear as the first character of a format field.

Example:

Value	Format Field	Print Out
43.7	+###.#	+b43.7
-43.7	+***.*	-b43.7
23.58	-&&&.&&	b023.58
-23.58	-&&&.&&	-023.58

Floating signs must start in the first format position and occupy all positions up to the decimal point. This causes the sign to be printed immediately before the first digit rather than in a fixed location. Each sign after the first also represents one digit.

Example:

Value	Format Field	Print Out
3.75	++++.##	bb+3.75
3.75	----.##	bbb3.75
-3.75	----.##	bb-3.75

A trailing sign can appear only after a decimal point. It terminates the format and prints the appropriate sign (or blank).

Example:

Value	Format Field	Print Out
43.17	***.**+	*43.17+
43.17	&&&.&&-	043.17b
-43.17	###.##+	b43.17-

DOLLAR SIGN ($)

A dollar sign can be either fixed or floating, and indicates that a $ is to be printed.

A fixed dollar sign must be either the first or second character in the format field. If it is the second character then + or - must be the first.

Example:

Value	Format Field	Print Out
34.2	$##.##	$34.20
34.2	+$##.##	+$34.20
-34.2	+$###.##	-$ 34.20

Floating dollar signs must start as either the first or second character in the format field and continue to the decimal point. If the floating dollar signs start as the second character then + or - must be the first. Each dollar sign after the first also represents one digit.

Example:

Value	Format Field	Print Out
34.2	$$$$$.##	bb$34.20
34.2	+$$$$$.##	+bb$34.20
1572563.41	$$,$$$,$$$.##+	$1,572,563.41+

NOTE: There can only be one floating character per format field.

NOTE: +, - or $ in other than proper positions will give strange results.

STRING FORMATS:

The format characters for string format fields are:

% - Indicates the string is to be right justified.
! - indicates the string is to be left justified.

If there are more characters in the string than in the format field, than the string is truncated.

Example:

Value	Format Field	Print Out
ABC	%%%%	bABC
ABC	!!!!	ABCb
ABC	%%	AB
ABC	!!	AB

ESCAPE CHARACTER (/)

The escape character (/) does not terminate the format field but will cause the next character to be printed, thus allowing the user to insert a character in the middle of the printing of a number.

Example: PRINT USING "###/-####",2551472 prints

255-1472

Example:

```
100 AREA = 408
200 NUM = 2551472
300 PHONE = (AREA*1E+7)+NUM
400 DIM F$(20)
500 F$ = "(###/)###/-####"
600 PRINT USING F$,PHONE
700 END
```

This program will print

(408)255-1472

NOTE: Improperly specified format fields can give some very strange results.

NOTE: The function of "," and ";" in PRINT are overridden in the expression list of PRINT USING, but when file number "fn" is given then the following "," or ";" have the same meaning as in PRINT. So to avoid an initial tabbing, use a semicolon (;).

Example: PRINT #5; USING A$,B

Will print B in the format specified by A$ to the file or device associated with file number 5.

Example: PRINT USING "## /* #=###",12,5,5*12

```
12 * 5=60
```

Example: PRINT USING "TOTAL=##.#+",72.68

```
TOTAL=72.7+
```

Example:

```
100 DIM A$(10) : A$="TOTAL="
200 DIM F$(10) : F$="!!!!!!##.#+"
300 PRINT USING F$,A$,72.68
```

```
TOTAL=72.7+
```

NOTE: IF there are more expressions in the expression list than there are format fields, the format fields will be reused.

Example: PRINT USING "XX##",25,19,7 will print

```
XX25XX19XXb7
```

WARNING:

A format string must contain at least one format field. If the format string contains only non-format characters, those characters will be printed repeatedly in the search for a format field.

TAB

Format: TAB [#fn,] aexp
Example: TAB #PRINTER,20

TAB outputs spaces to the device or file specified by fn (or the screen) up to column number "aexp". The first column is column 0.

NOTE: The column count is kept for each device and is reset to zero each time a carriage return is output to that device. The count is kept in AUX2 of the IOCB. (See OS documemtation).

NOTE: If "aexp" is less than the current column count, a carriage return is output and then spaces are put out up to column "aexp".

BPUT

Format: BPUT #fn, aexp1, aexp2
Example: (see below)

BPUT outputs a block of data to the device or file specified by "fn". The block of data starts at address "aexp1" for a length of "aexp2".

NOTE: The address may be a memory address. For example, the whole screen might be saved. Or the address may be the address of a string obtained using the ADR function.

Example:

```
BPUT #5, ADR(A$), LEN(A$)
```

This statements writes the block of data contained in the string A$ to the file or device associated with file number 5.

BGET

Format: BGET #fn, aexp1, aexp2
Example: (see below)

BGET gets "aexp2" bytes from the device or file specified by "fn" and stores them at address "aexp1".

NOTE: The address may be a memory address. For example, a screen full of data could be displayed in this manner. Or the address may be the address of a string. In this case BGET does not change the length of the string. This is the user's responsibility.

Example:

```
10 DIM A$(1025)
20 BGET #5,ADR(A$),1024
30 A$(1025) = CHR$(0)
```

This program segment will get 1024 bytes from the file or device associated with file number 5 and store it in A$. Statement 30 sets the length of A$ to 1025.

NOTE: No error checking is done on the address or length so care must be taken when using this statement.

RPUT

Format: RPUT #fn, exp [,exp...]
Example: (see below)

RPUT allows the user to output fixed length records to the device or file associated with "fn". Each "exp" creates an element in the record.

NOTE: A numeric element consists of one byte which indicates a numeric type element and 6 bytes of numeric data in floating point format.

A string element consists of one byte which indicates a string type element 2 bytes of string length, 2 bytes of DIMensioned length, and then X bytes where X is the DIMensioned length of the string.

Example:

```
100 DIM A$(6)
200 A$ = "XY"
300 RPUT #3,B,A$,10
```

Puts 3 elements to the device or file asscoiated with file number 3. The first element is numeric (the value of B). The second element is a string (A$) and the third is a numeric (10). The record will be 26 bytes long, (7 bytes for each numeric, 5 bytes for the string header and 6 bytes (the DIM length) of string data).

RGET

Format:

```
RGET    #fn, {svar} [,{svar}...]
             {avar} [,{avar}...]
```

Example: (see below)

RGET allows the user to retreive fixed length records from the device or file associated with file number "fn" and assign the values to string or numeric variables.

NOTE: The type of the element in the file must match the type of the variable (ie. they must both be strings or both be numeric).

Example:

```
1) RPUT #5,A
2) RGET #1,A$
```

If 1) is a statement in a program used to generate a file and 2) is a statement in another program used to read the same file, an error will result.

NOTE: When the type of element is string, then the DIMensioned length of the element in the file must be equal to the DIMensioned length of the string variable.

Example:

```
1) 100 DIM A$(100)
   .
   .
   800 RPUT #3,A$
   .
   .
```

```
2) 100 DIM X$(200)
       .
       .
       800 RGET #2, X$
       .
       .
```

If 1) is a section of a program used to write a file and 2) is a section of another program used to read the same file, then an error will occur as a result of the difference in DIM values.

NOTE: RGET sets the correct length for a string variable (the length of a string variable becomes the actual length of the string that was RPUT - not necessarily the DIM length).

Example:

```
1)100 DIM A$(10)
  200 A$ = "ABCDE"
   .
   .
  800 RPUT #4, A$

2)100 DIM X$(10)
  200 X$ = "HI"
   .
   .
  800 RGET #6, X$
  900 PRINT LEN(X$), X$
   .
   .
```

If 1) is a section of a program used to create a file and 2) is a section of another program used to read the file then it will print:

```
5         ABCDE
```

ADVANCED FUNCTIONS

DPEEK

DPOKE

Format:	DPEEK(addr)
	DPOKE addr,aexp
Examples:	PRINT "variable name table is at";DPEEK(130)
	DPOKE 741,DPEEK(741)-1024

The DPEEK function and DPOKE statement parallel PEEK and POKE. The difference is that, instead of working with single byte memory locations, DPEEK and DPOKE access or change Double byte locations (or "words"). Hence, DPEEK may return a value from 0 to 65535; and DPOKE's aexp may be any expression evaluating to a like range.

The primary advantage of DPEEK over PEEK is illustrated by the following two exactly equivalent program fragments:

```
100 A=PEEK(130)+256*PEEK(131)
100 A=DPEEK(130)
```

In the second example at the head of this section, the top of memory is lowered by 1k bytes in a single, easy-to-read statement.

ERR

Format:	ERR(aexp)
Example:	PRINT "ERROR";ERR(0); "OCCURRED AT LINE";ERR(1)

This function--in conjunction with TRAP, CONT, and GOTO allows the BASIC A+ programmer to effectively diagnose and dispatch virtually any run-time error.

```
ERR(0) returns the last run-time error number
ERR(1) returns the line number where the error occurred
```

Example:

```
100 TRAP 200
110 INPUT "A NUMBER, PLEASE >>",NUM
120 PRINT "A VALID NUMBER" : END
200 IF ERR(0)=8 THEN GOTO ERR(1)
210 PRINT "UNEXPECTED ERROR #";ERR(0)
```

TAB

```
Format:         TAB(aexp)
Example:        PRINT #3;"columns:";TAB(20);20;TAB(30);30
```

The TAB function's effect is identical with that of the TAB statement (page 32-A+). The difference is that, for PRINT statements, an imbedded TAB function simplifies the programmers task greatly (see the example).

TAB will output ATASCII space characters to the current PRINT file or device (#3 in our example). Sufficient spaces will be output so that the next item will print in the column specified (only if TAB is followed by a semi-colon, though). If the column specified is less than the current column, a RETURN will be output first.

Caution: The TAB function will output spaces on some device whenever it is used; therefore, it should be used ONLY in PRINT statements. It will NOT function properly in PRINT USING.

ADVANCED STRINGS

SUBSTRINGS:

A destination string is one that is being assigned to. Any other string is a source string. In

```
READ X$
INPUT X$
X$=Y$
```

X$ is the destination string, Y$ is the source string.

Substrings are defined as follows:

STRING	definition when destination string	definition when source string
S$	the entire string 1 thru DIM value	from 1st thru LEN character
S$(n)	from nth thru DIMth character	from nth thru LENgth character
S$(n,m)	from the nth thru the mth character	from the nth thru the mth character

It is an error if either the first or last specified character (n and m, above) is outside the DIMensioned size. It is an error if the last character position given (explicitly or implicitly) is less than the first character position.

```
Example:          Assume: DIM A$(10)
                          A$ = "VWXYZ"

                  1) PRINT A$(2)        prints:
                     WXYZ

                  2) PRINT A$(3,4)      prints:
                     XY

                  3) PRINT A$(5,5)      prints:
                     Z

                  4) PRINT A$(7)
                     is an error because A$ has a length of 5.
```

NOTE: Refer to the LET statement, page 10-a, for examples of BASIC A+ string concatenation.

FIND

Format: FIND(sexp1,sexp2,aexp)
Example: PRINT FIND ("ABCDXXXXABC","BC",N)

FIND is an efficient, speedy way of determining whether any given substring is contained in any given master string.

FIND will search sexp1, starting at position aexp, for sexp2. If sexp2 is found, the function returns the position where it was found, relative to the beginning of sexp1. If sexp2 is not found, a 0 is returned.

In the example above, the following values would be PRINTed:

```
2 if N=0 or N=1
9 if N>2 and N<10
0 if N>=10
```

More Examples:

```
10 DIM A$(1)
20 PRINT "INPUT A SINGLE LETTER:
30 PRINT "Change/Erase/List"
40 INPUT "CHOICE ?",A$
50 ON FIND("CEL",A$,0) GOTO 100,200,300
```

An easy way to have a vector from a menu choice

```
100 DIM A$(10): A$="ABCDEFGHIJ"
110 PRINT FIND (A$,"E",3)
120 PRINT FIND (A$(3),"E")
```

Line 110 will print "5" while 120 will print "3". Remember, the position returned is relative to the start of the specified string.

```
100 INPUT "20 CHARACTERS, PLEASE: ",A$
110 ST=0
120 F=FIND(A$,"A",ST):IF F=0 THEN STOP
130 IF A$(F+1,F+1)="B" OR A$(F+1,F+1)="C"
        THEN ST=F+1:GOTO 120
140 PRINT "FOUND 'AB' OR 'AC'"
```

This illustrates the importance of the aexp's use as a starting position.

ADVANCED GAME CONTROL

Note: See also chapter 13, PLAYER/MISSILE GRAPHICS.

HSTICK

VSTICK

```
Formats:          HSTICK(aexp)
                  VSTICK(aexp)
EXAMPLES:         IF HSTICK(0)>0 and VSTICK(0)<0
                     THEN PRINT "DOWN, TO THE RIGHT"
```

If the numbering scheme for STICK(0) positions dismayed you, take heart: HSTICK and VSTICK provide a simpler method of reading the joysticks.

```
VSTICK(n) reads joystick n and returns:
      +1 if the joystick is pushed up
      -1 if the joystick is pushed down
       0 if the joystick is vertically centered

HSTICK(n) reads joystick n and returns:
      +1 if the joystick is pushed right
      -1 if the joystick is pushed left
       0 if the joystick is horizontally centered
```

PEN

```
Format:           PEN(aexp)
Example:          PRINT "light pen at X=";pen(0)
```

The PEN function simply reads the ATARI light pen registers and returns their contents to the user.

```
PEN(0) reads the horizontal position register
PEN(1) reads the vertical position register
```

NUMBERS

All numbers in Basic are in BCD floating point.

RANGE:

Floating point numbers must be less than 10E+98 and greater than or equal to -10E-98.

INTERNAL FORMAT:

Numbers are represented internally in 6 bytes. There is a 5 byte mantissa containing 10 BCD digits and a one byte exponent.

The most significant bit of the exponent byte gives the sign of the mantissa (0 for postive, 1 for negative). The least significant 7 bits of the exponent byte gives the exponent in excess 64 notation. Internally, the exponent represents powers of 100 (not powers of 10).

```
        Example:         0.02 = 2 * 10^-2 = 2 * 100^-1

        exponent=        -1 + 40 = 3F

           0.02 =        3F 02 00 00 00 00
```

The implied decimal point is always to the right of the first byte. An exponent less than hex 40 indicates a number less than 1. An exponent greater than or equal to hex 40 represents a number greater than or equal to 1.

Zero is represented by a zero mantissa and a zero exponent.

In general, numbers have a 9 digit precision. For example, only the first 9 digits are significant when INPUTing a number. Internally the user can usually get 10 significant digits in the special case where there are an even number of digits to the right of the decimal point (0,2,4...).

ADVANCED SYSTEM FEATURES

SET and SYS

Formats:	SET aexp1,aexp2
	SYS(aexp)
Examples:	SET 1,5
	PRINT SYS(2)

SET is a statement which allows the user to exerices control over a varity of BASIC A+ system level functions. SYS is simply an arithmetic function used to check the SETtings of these functions. The table below summarizes the various SET table parameters. (Default values are given in parentheses.)

aexp1 PARAMETER #		aexp2 LEGAL VALUES	meaning
0,	(0)	0	-BREAK key functions normally
		1	-User hitting BREAK cause an error to occur (TRAPable)
		128	-BREAKs are ignored
1,	(10)	1 thru 127	-Tab "stop" setting fort the comma in PRINT statements.
2,	(63)	0 thru 255	-Prompt character for INPUT (default is "?").
3,	(0)	0	-FOR...NEXT loops always execute at least once (ala ATARI BASIC).
		1	-FOR loops may execute zero times (ANSI standard)
4,		0	-On a mutiple variable INPUT, if the user enters too few items, he is reprompted (e.g. with "??")
	(1)	1	-Instead of reprompting, a TRAPable error occurs.
5,		0	-Lower case and inverse video characters remain unchanged and can cause syntax errors.
	(1)	1	-For program entry ONLY, lower case letters are converted to upper case and inverse video characters are uninverted. Exception: characters between quotes remain unchanged.

6,	(0)	0	-Print error messages along with error numbers (for most errors)
		1	-Print only error numbers.
7,	(0)	0	-Missiles (in Player/Missile-Graphics), which move vertically to the edge of the screen, roll off the edge and are lost.
		1	-Missiles wraparound from top to bottom and vise versa.
8,		0	-Don't push (PHA) the number of parameters to a USR call on the stack [advantage: some assembly language subroutines not expecting parameters may be called by a simple USR(addr)].
	(1)	1	-DO push the count of parameters (ATARI BASIC standard).
9,	(0)	0	-ENTER statements return to the READY prompt level on completion
		1	-If a TRAP is properly set, ENTER will execute a GOTO the TRAP line on end-of-entered-file.

Note: The SET parameters are reset to the system defaults on execution of a NEW statement.

Note: System defaults may be changed either temporarily or permanently (by SAVEing a patched BASIC A+ via OS/A+) by POKEing the locations noted in the memory map.

Examples:

1)
```
SET 1,4 : PRINT 1,2,3,4
```

THe number will be printed every four columns

2)
```
SET 2,ASC(">")
```

Changes the INPUT prompt from "?" to ">"

3)
```
100 SET 9,1 : TRAP 120
110 ENTER "D:OVERLAY.LIS"
120 REM execution continues here after entry of
130 rem the overlay
```

4)
```
100 SET 0,1 : TRAP 200
110 PRINT "HIT BREAK TO CONTINUE"
120 GOTO 110
200 REM come here via BREAK KEY
```

5)
```
100 SET 3,1
110 FOR I = 1 TO 0
120 PRINT " THIS LINE WON'T BE EXECUTED"
130 NEXT I
```

MOVE

Format:	MOVE from-addr, to-addr, len
	[MOVE aexp, aexp, aexp]
Example:	MOVE 13*4096, 8*4096, 1024
Caution:	Be careful with this command.

MOVE is a general purpose byte move utility which will move any number of bytes from any address to any address at assembly language speed. NO ADDRESS CHECKS ARE MADE!!

The sign of the third aexp (the length) determines the order in which the bytes are moved.

```
     If the length is postive:
          (from) -> (to)
          (from+1) -> (to+1)
            ...         ...
          (from+len-1) -> (to +len-1)

     If the length is negative:
          (from+len-1) -> (to+len-1)
          (from+len-2) -> (to+len-2)
             ...            ...
          (from+1) -> (to +1)
          (from) -> (to)
```

The example above will move the character set map to BASIC A+'s reserved area in a 48K RAM system (it moves from $D000 to $8000).

PLAYER / MISSILE GRAPHICS

This section describes the BASIC A+ commands and functions used to access the Atari's Player-Missile Graphics. Player Missile Graphics (hereafter usually referred to as simply "PMG") represent a portion of the Atari hardware totally ignored by Atari Basic and Atari OS. Even the screen handler (the "S:" device) knows nothing about PMG. BASIC A+ goes a long way toward remedying these omissions by adding six (6) PMG commands (statements) and two (2) PMG functions to the already comprehensive Atari graphics. In addition, four other statements and two functions have significant uses in PMG and will be discussed in this section.

The PMG statements and functions:

PMGRAPHICS	PMCOLOR	PMCLR
PMMOVE	PMWIDTH	MISSILE
BUMP(...)	PMADR(...)	

The related function and statements:

MOVE	BGET	BPUT
POKE	USR(...)	PEEK(...)

AN OVERVIEW

For a complete technical discussion of PMG, and to learn of even more PMG "tricks" than are included in BASIC A+, read the Atari document entitled "Atari 400/800 Hardware Manual" (Atari part number CO16555, Rev. 1 or later).

It was stated above that the "S:" device driver knows nothing of PMG, and in a sense this is proper: the hardware mechanisms that implement PMG are, for virtually all purposes, completely separate and distinct from the "playfield" graphics supported by "S:". For example, the size, position, and color of players on the video screen are completely independent of the GRAPHICS mode currently selected and any COLOR or SETCOLOR commands currently active. In Atari (and now BASIC A+) parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical stripe on the screen. Sounds dull? Consider: each player (there are four) may be "painted" in any of the 128 colors available on the Atari (see Setcolor for specific colors). Within the vertical stripe, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to 0 paints no color at all! That is, any 0 bit in a player stripe has no effect on the underlying playfield display.

Why a vertical stripe? Refer to Figure PMG-1 for a rough idea of the player concept. If we define a shape within the bounds of this stripe (by changing some of the player's bits to 1's), we may then move the stripe anywhere horizontally by a simple register POKE (or via the PMMOVE command in BASIC A+). We may move the player vertically by simply doing a circular shift on the contiguous memory block representing the player (again, the PMMOVE command of BASIC A+ simplifies this process). To simplify:

A player is actually seen as a stripe on the screen 8 pixels wide by 128 (or 256, see below) pixels high. Within this stripe, the user may POKE or MOVE bytes to establish what is essentially a tall, skinny picture (though much of the picture may consist of 0 bits, in which case the background "shows through"). Using PMMOVE, the programmer may then move this player to any horizontal or vertical location on the screen. To complicate:

For each of the four players there is a corresponding "missile" available. Missiles are exactly like players except that (1) they are only 2 bits wide, and all four missiles share a single block of memory, (2) each 2 bit sub-stripe has an independent horizontal position, and (3) a missile always has the same color as its parent player. Again, by using the BASIC A+ commands (MISSILE and PMMOVE, for example), the programmer/user need not be too aware of the mechanisms of PMG.

CONVENTIONS

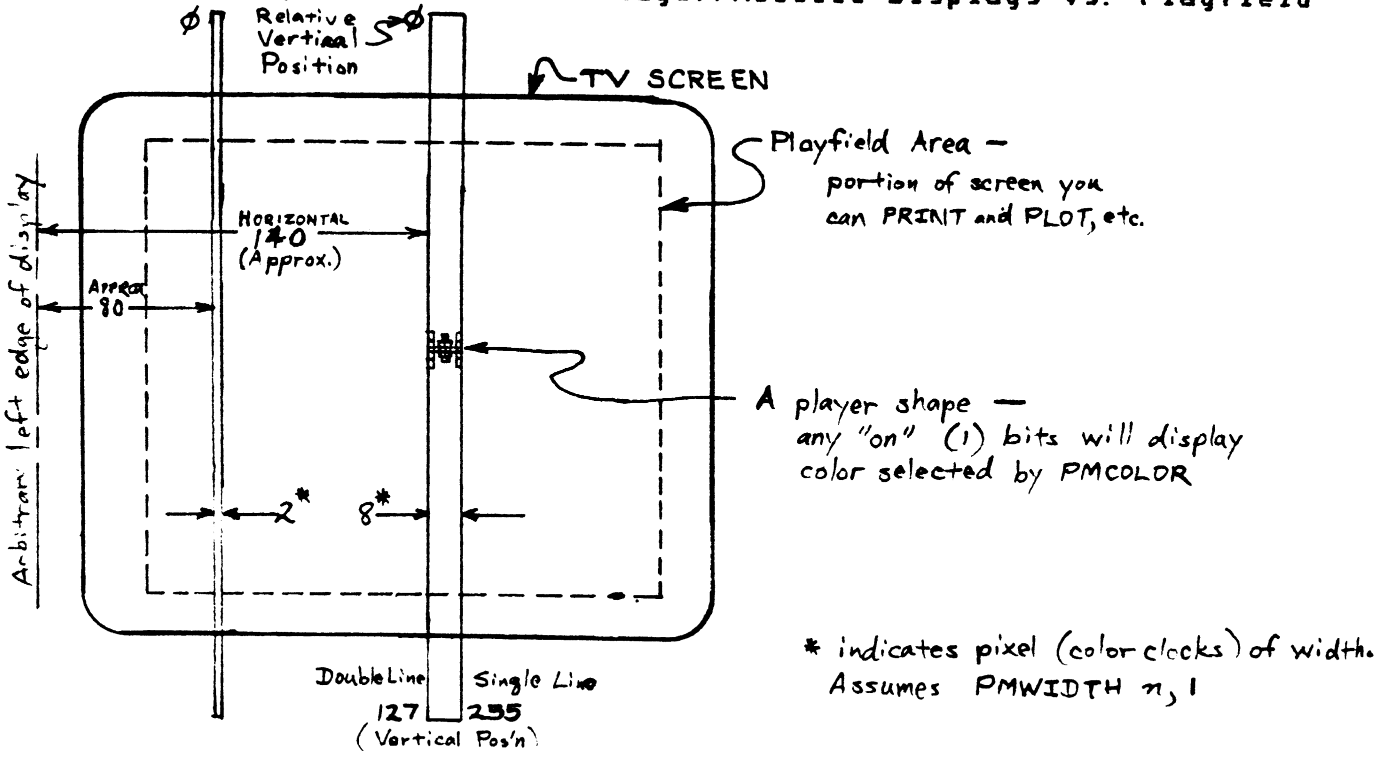
1. Players are numbered from 0 through 3. Each player has a corresponding missile whose number is 4 greater then that of its parent player, thus missiles are numbered 4 through 7. In the BUMP function, the "playfields" are numbered from 8 through 11, corresponding to actual playfields 0 through 3. (Note: playfields are actually COLORs on the main GRaphics screen, and can be PLOTted, PRINTed, etc).

2. There is some inconsistency in which way is "UP". PLOT, DRAWTO, POKE, MOVE, etc are aware that 0,0 is the top left of the screen and that vertical position numbering increases as you go down the screen. PMMOVE and VSTICK, however, do only relative screen positioning, and define "+" to be UP and "-" to be DOWN. [If this really bothers you please let us know!].

3. "pmnum" is an abbreviation for Player-Missile NUMber and must be a number from 0 to 3 (for players) or 4 to 7 (for missiles).

FIGURE PMG-1

Graphic Representation of Player/Missile Displays vs. Playfield



ϕ
Relative Vertical Position
ϕ
TV SCREEN
Playfield Area — portion of screen you can PRINT and PLOT, etc.
HORIZONTAL 140 (Approx.)
APPROX 80
Arbitrary left edge of display
2*
8*
A player shape — any "on" (1) bits will display color selected by PMCOLOR
* indicates pixel (color clocks) of width. Assumes PMWIDTH n, 1
Double Line 127
Single Line 255
(Vertical Pos'n)

FIGURE PMG-2

Memory Usage in Player/Missile Graphics

NOTE: assumes 48K system. Adjust addresses downward 8K or 16K for 40k or 32K systems.

Resolution:	single line	double line
Top of RAM	$C000	$C000
Player 3	$BFFF	$BFFF
	$BF00	$BF80
Player 2	$BEFF	$BF7F
	$BE00	$BF00
Player 1	$BDFF	$BEFF
	$BD00	$BE80
Player 0	$BCFF	$BE7F
	$BC00	$BE00
Missiles (all)	$BBFF	$BDFF
	$BB00	$BD80

PMGRAPHICS
(PMG.)

Format:	PMGRAPHICS aexp
Example:	PMG. 2

This statement is used to enable or disable the Player-Missile Graphics system. The aexp should evaluate to 0, 1, or 2:

PMG.0	Turn off PMG
PMG.1	Enable PMG, single line resolution
PMG.2	Enable PMG, double line resolution

Single and Double line resolution (hereafter refered to as "PMG Modes") refer to the height which a byte in the player "stripe" occupies - either one or two television scan lines. (A scan line height is the pixel height in GRaphics mode 8. GRaphics 7 has pixels 2 scan lines high, similar to PMG.2)

The secondary implication of single line versus double line resolution is that single line resolution requires twice as much memory as double line, 256 bytes per player versus 128 bytes. Figure PMG-2 shows PMG memory usage in BASIC A+, but the user really need not be aware of the mechanics if the PMADR function is used.

PMCLR

Format:	PMCLR pmnum
Example:	PMCLR 4

This statement "clears" a player or missile area to all zero bytes, thus "erasing" the player/missile. PMCLR is aware of what PMG mode is active and clears only the appropriate amounts of memory. CAUTION: PMCLR 4 through PMCLR 7 all produce the same action -- ALL missiles are cleared, not just the one specified. To clear a single missile, try the following:

```
SET 7,0 : PMMOVE 4;255
```

PMCOLOR

(PMCO.)

Format:	PMCOLOR pmnum,aexp,aexp
Example:	PMCOLOR 2,13,8

PMCOLORs are identical in usage to those of the SETCOLOR statement except that a player/missile set has its color chosen. Note there is no correspondence in PMG to the COLOR statement of playfield GRaphics: none is necessary since each player has its own color.

The example above would set player 2 and missile 6 to a medium (luminace 8) green (hue 13).

NOTE: PMG has NO default colors set on power-up or SYSTEM RESET.

PMWIDTH

(PMW.)

Format:	PMWIDTH pmnum,aexp
Example:	PMWIDTH 1,2

Just as PMGRAPHICs can select single or double pixel heights, PMWIDTH allows the user to specify the screen width of players and missiles. But where PMGRAPHICs selects resolution mode for all players and missiles, PMWIDTH allows each player AND missile to be separately specified. The aexp used for the width should have values of 1,2, or 4 -- representing the number of color clocks (equivalent to a pixel width in GRaphics mode 7) which each bit in a player definition will occupy.

NOTE: PMG.2 and PMWIDTH 1 combine to allow each bit of a player definition to be equivalent to a GRaphics mode 7 pixel -- a not altogether accidental occurence.

NOTE: Although players may be made wider with PMWIDTH, the resolution then suffers. Wider "players" made be made by placing two or more separate players side-by-side.

PMMOVE

Format:	PMMOVE pmnum[,aexp][;aexp]
Example:	PMMOVE 0,120;1
	PMMOVE 1,80
	PMMOVE 4;-3

Once a player or missile has been "defined" (via POKE, MOVE, GET, or MISSILE), the truly unique features of PMG under BASIC A+ may be utilized. With PMMOVE, the user may position the player/missile shape anywhere on the screen almost instantly.

BASIC A+ allows the user to position each player and missile independently. Because of the hardware implementation, though, there is a difference in how horizonal and vertical positioning are specified.

The parameter following the comma in PMMOVE is taken to be the ABSOLUTE position of the left edge of the "stripe" to be displayed. This position ranges from 0 to 255, though the lowest and highest positions in this range are beyond the edges of the display screen. Note the specification of the LEFT edge: changing a player's width (see PMWIDTH) will not change the position of its left edge, but will expand the player to the right.

The parameter following the semicolon in PMMOVE is a RELATIVE vertical movement specifier. Recall that a "stripe" of player is 128 or 256 bytes of memory. Vertical movement must be accomplished by actual movement of the bytes within the stripe - either towards higher memory (down the screen) or lower memory (up the screen). BASIC A+ allows the user to specify a vertical movement of from -255 (down 255 pixels) to +255 (up 255 pixels).

NOTE: The +/- convention on vertical movement conforms to the value returned by VSTICK.

Example: PMMOVE N;VSTICK(N)

Will move player N up or down (or not move him) in accordance with the joystick position.

NOTE: SET may be used to tell PMMOVE whether an object should "wraparound" (from bottom of screen to top of screen or vice versa) or should disappear as it scrolls too far up or down. SET 7,1 specifies wrap-around. SET 7,0 disables wraparound.

MISSILE
(MIS.)

Format: MISSILE pmnum,aexp,aexp
Example: MISSILE 4,48,3

The MISSILE statement allows an easy way for a parent player to "shoot" a missile. The first aexp specifies the absolute vertical position of the beginning of the missile (0 is the top of screen), and the second aexp specifies the vertical height of the missile.

Example: MISSILE 4,64,3

Would place a missile 3 or 6 scan lines high (depends on PMG. mode) at pixel 64 from the top.

NOTE: MISSILE does NOT simply turn on the bits corresponding to the position specified. Instead, the bits specified are exclusive-or'ed with the current missile memory. This can allow the user to erase existing missiles while creating others.

Example:
```
MISSILE 5,40,4
MISSILE 5,40,8
```

The first statement creates a 4 pixel missile at vertical position 20. The second statement erases the first missile and creates a 4 pixel missile at vertical position 24.

PMG FUNCTIONS

PMADR

Format:	PMADR(aexp)
Example:	PO=PMADR(0)

This function may be used in any arithmetic expression and is used to obtain the memory address of any player or missile. It is useful when the programmer wishes to MOVE, POKE, BGET, etc. data to (or from) a player area. See next section on "PMG RELATED STATEMENTS" for examples and hints.

NOTE: PMADR(m) -- where m is a missile number (4 through 7) returns the same address for all missiles.

BUMP

Format:	BUMP(pmnum, aexp)
Examples:	IF BUMP(4, 1) THEN ...
	B=BUMP(0, 8)

BUMP is a function which can be used in any arithmetic expression. BUMP accesses the collision registers of the ATARI and returns a 1 (collision occured) or 0 (no collision occured) as appropriate for the pair of objects specified. Note that the second parameter (the aexp) may be either a player number or playfield number (8 through 11).

Valid BUMPs:	PLAYER to PLAYER (0-3 to 0-3)
	MISSILE to PLAYER (4-7 to 0-3)
	PLAYER to PLAYFIELD (0-3 to 8-11)
	MISSILE to PLAYFIELD (4-7 to 8-11)

NOTE: BUMP (p, p), where the p's are 0 through 3 and identical, always returns 0.

NOTE: It is advisable to reset the collision registers if a relatively long time has occurred since they were last checked.

YOU MUST CLEAR THE COLLISION REGISTERS VIA
POKE 53278,0

PMG RELATED STATEMENTS

NOTE: See also decriptions of these statements in preceding sections. The discussions here pertain only to their usage with PMG.

POKE and PEEK

One of the most common ways to put player data into a player stripe may well be to use POKE. In conjunction with PMADR, it is easy to write understandable player loading routines.

Example:

```
100 FOR LOC=48 TO 52
110 READ N: POKE LOC+PMADR(0),N
120 NEXT LOC
...
900 DATA 255,129,255,129,255
```

PEEK might be used to find out what data is in a particular player location.

MOVE

MOVE is an efficient way to load a large player and/or move a player vertically by a large amount. With its ability to MOVE data in upwards or downwards movement, interesting overlap possibilities occur. Also, it would be easy to have several player shapes contained in stripes and then MOVEd into place at will.

Examples:

```
MOVE ADR(A$),PMADR(2),128
```

could move an entire double line resolution player from A$ to player stripe number 2.

```
POKE PMADR(1),255
MOVE PMADR(1),PMADR(1)+1,127
```

would fill player 1's stripe with all "on" bits, creating a solid stripe on the screen.

BGET and BPUT

As with MOVE, BGET may be used to fill a player memory quickly with a player shape. The difference is that BGET may obtain a player directly from the disk!

Example: `BGET #3,PMADR(0),128`

Would get a PMG.2 mode player from the file opened in slot #3.

Example: `BGET #4,PMADR(4),256*5`

Would fill all the missiles AND players in PMG.1 mode -- with a single statement!

BPUT would probably be most commonly used during program development to SAVE a player shape (or shapes) to a file for later retrieval by BGET.

USR

Because of USR's ability to pass parameters to an assembly language routine, complex PMG functions (written in assembly language) can be easly interfaced to BASIC A+.

Example: `A=USR(PMBLINK,PMADR(2),128)`

Might call an assembly language program (at address PMBLINK) to BLINK player 2, whose size is 128 bytes.

EXAMPLE PMG PROGRAMS

1. A very simple program with one player and its missile

```
100 setcolor 2,0,0       : rem note we leave ourselves in GR.0
110 PMGRAPHICS 2         : rem double line resolution
120 let width=1 : y=48 : rem just initializing
130 PMCLR 0 : PMCLR 4  : rem clear player 0 and missile 0
135 PMCOLOR 0,13,8       : rem a nice green player
140 p=PMADR(0)           : rem gets address of player
150 for i=p+y to p+y+4 : rem a 5 element player to be defined
160   read val           : rem see below for DATA scheme
170   poke i,val         : rem actually setting up player shape
180   next i
200 for x=1 to 120       : rem player movement loop
210   PMMOVE 0,x         : rem moves player horizontally
220   sound 0,x+x,0,15 : rem just to make some noise
230   next x
240 MISSILE 0,y,1        : rem a one-high missile at top of player
250 MISSILE 0,y+2,1      : rem another, in middle of player
260 MISSILE 0,y+4,1      : rem and again at top of player
300 for x=127 to 255     : rem the missile movement loop
310   PMMOVE 4,x         : rem moves missile 0
320   sound 0,255-x,10,15
330   IF (x & 7) = 7     : rem every eighth horizontal position
340     MISSILE 0,y,5    : rem you have to see this to believe it
350     ENDIF            : rem could have had an ELSE, of course
360   next x
370 PMMOVE 0,0           : rem so width doesn't change on screen
400 width=width*2        : rem we will make the player wider
410 if width > 4 then width = 1 : rem until it gets too wide
420 PMWIDTH 0,width      : rem the new width
430 PMCLR 4              : rem no more missile
440 goto 200             : rem and do all this again
500 rem THE DATA FOR PLAYER SHAPE
510 data  153            : rem $99          *  **  *
520 data  189            : rem $BD          * **** *
530 data  255            : rem $FF          ********
540 data  189            : rem $BD          * **** *
550 data  153            : rem $99          *  **  *
```

CAUTION : do NOT put the REMarks on lines 510 thru 550 !!!!!!!
(DATA must be last statement on a line !)

Notice how the data for the player shape is built up...
draw a picture on an 8-wide by n-high piece of grid paper, filling in whole cells. Call a filled in cell a '1' bit, empty cells are '0'. Convert the 1's and 0's to hex notation and thence to decimal.

This program will run noticably faster if you use multiple statements per line. It was written as above for clarity, only.

2. A more complicated program, sparsely commented.

```
100 dim hex$(15),t$(4) : hex$="123456789ABCDEF"
110 graphics 0          : rem not necessary, just prettier
120 PMGRAPHICS 2 : PMCLR 0 : PMCLR 1
130 setcolor 2,0,0 : PMCOLOR 0,12,8 : PMCOLOR 1,12,8
140 p0 = PMADR(0) : p1 = PMADR(1) : rem addr's for 2 players
150 v0 = 60 : vold = v0  :rem starting vertical position
160 h0 = 110            : rem  starting horizontal position
200 for loc =v0-8 to v0+7 : rem a 16-high double player
210  read t$            : rem a hex string to t$
220  poke p0+loc,16*FIND(hex$,t$(1,1),0) + FIND(hex$,t$(2,2),0)
230  poke p1+loc,16*FIND(hex$,t$(3,3),0) + FIND(hex$,t$(4,4),0)
      : rem we find a hex digit in the hex string; its decimal
            value is its position (becuz if digit is zero it is
            not found so FIND returns 0 ! )
240  next loc
300 rem  ANIMATE IT
310 let radius=40 : deg  : rem 'let' required, RAD is keyword
320 WHILE 1             : rem forever !!!
330   c=int(16*rnd(0)) : pmcolor 0,C,8 : pmcolor 1,C,8
340   for angle = 0 to 355 step 5  : rem in degrees, remember
350     vnew = int( v0 + radius * sin(angle) )
360     vchange = vnew - vold  : rem change in vertical position
370     hnew = h0 + radius * cos(angle)
380     PMMOVE 0,hnew;vchange : PMMOVE 1,hnew+8;vchange
           : rem move two players together
390     vold = vnew
400     sound 0,hnew,10,12 : sound 1,vnew,10,12
410     next angle
420   rem just did a full circle
430 ENDWHILE
440 rem we better NEVER get to here !

500 rem the fancy data !         8421842184218421
510 DATA 03C0                   !      ****      !
520 DATA 0C30                   !    **    **    !
530 DATA 1008                   !   *        *   !
540 DATA 2004                   !  *          *  !
550 DATA 4002                   ! *            * !
560 DATA 4E72                   ! *  ***  ***  * !
570 DATA 8A51                   !*   * *  * *   *!
580 DATA 8E71                   !*   ***  ***   *!
590 DATA 8001                   !*              *!
600 DATA 9009                   !*  *        *  *!
610 DATA 4812                   ! *  *      *  * !
620 DATA 47E2                   ! *   ******   * !
630 DATA 2004                   !  *          *  !
640 DATA 1008                   !   *        *   !
650 DATA 0C30                   !    **    **    !
660 DATA 03C0                   !      ****      !
```

Notice how much easier it is to use the hex data. With FIND, the hex to decimal conversion is easy, too.

The factor slowing this program the most is the SIN and COS being calculated in the movement loop. If these values were pre-calculated and placed in an array this program would move!

ERROR NUMBER DECRIPTION

1 - BREAK KEY ABORT

While SET 0,1 was specified, the operator hit the BREAK key. This trappable error gives the BASIC A+ programmer total system control.

2 - MEM FULL

All avaiable memory has been used. No more statements can be entered and no more variables (arithmetic, string or array) can be defined.

3 - VALUE

An expression or variable evaluates to an incorrect value.

Example: An expression that can be converted to a two byte integer in the range 0 to 65235 (hex FFFF) is called for and the given expression is either too large or negative.

```
A = PEEK(-1)
DIM B(70000)
```

Both these statments will produce a value error

Example: An expression that can be converted to a one byte integer in the range 0 to 255 hex(FF) is called for and the given expression is too large.

```
POKE 5000,750
```

This statement produces a value error.

Example: `A=SQR(-4)` Produces a value error.

4 - TOO MANY VARS

No more variables can be defined. The maximum number of variables is 128.

5 - STRING LEN

A character beyond the DIMensioned or current length of a string has been accessed.

Example:

```
1000 DIM A$(3)
2000 A$(5) = "A"
```

This will produce a string length error at line 2000 when the program is RUN.

6 - READ, NO DATA

A READ statement is executed but we are already at the end of the last DATA statement.

7 - LINE #/VAL > 32767

A line number larger than 32767 was entered.

8 - INPUT/READ

The INPUT or READ statement did not recieve the type of data it expected.

Example: INPUT A

If the data entered is 12AB then this error will result.

Example:

```
1000 READ A
2000 PRINT A
3000 END
4000 DATA 12AB
```

Running this program will produce this error.

9 - DIM

Example: A string or an array was used before it was DIMensioned.

Example: A previously DIMensioned string or array is DIMensioned again.

```
1000 DIM A(10)
2000 DIM A(10)
```

This program produces a DIM error.

10 - EXPR TOO COMPLEX

An expression is too complex for Basic to handle. The solution is to break the calculation into two or more Basic statements.

11 - OVERFLOW

The floating point routines have produced a number that is either too large or too small.

12 - NO SUCH LINE #

The line number required for a GOTO or GOSUB does not exist.
The GOTO may be implied as in:

```
1000 IF A=B THEN 500
```

The GOTO/GOSUB may be part of an ON statement.

13 - NEXT, NO FOR

A NEXT was encountered but there is no information about a FOR with the same variable.

Example:

```
1000 DIM A(10)
2000 REM FILL THE ARRAY
3000 FOR I = 0 TO 10
4000 A(I) = I
5000 NEXT I
6000 REM PRINT THE ARRAY
7000 FOR K = 0 TO 10
8000 PRINT A(K)
9000 NEXT I
10000 END
```

Running this program will cause the following output:

```
0

ERROR-  13 AT LINE 9000
```

NOTE: Improper use of POP could cause this error.

14 - LINE TOO LONG

The line just entered is longer than Basic can handle. The solution is to break the line into multiple lines by putting fewer statements on a line, or by evaluating the expression in multiple statements.

15 - LINE DELETED

The line containing a GOSUB or FOR was deleted after it was executed but before the RETURN or NEXT was executed.
This can happen if, while running a program, a STOP is executed after the GOSUB or FOR, then the line containing the GOSUB or FOR is deleted, then the user types CONT and the program tries to execute the RETURN or NEXT.

Example:

```
1000 GOSUB 2000
1100 PRINT "RETURNED FROM SUB"
1200 END
2000 PRINT "GOT TO SUB"
2100 STOP
2200 RETURN
```

If this program is run the print out is:

```
GOT TO SUB

STOPPED    AT LINE 2100
```

Now if the user deletes line 1000 and then types CONT we get

ERROR- 15 AT LINE 2200

16 - RETURN, NO GOSUB

A RETURN was encountered but we have no information about a GOSUB.

Example:

```
1000 PRINT "THIS IS A TEST"
2000 RETURN
```

If this program is run the print out is:

```
THIS IS A TEST

ERROR- 16 AT LINE 2000
```

NOTE: improper use of POP could also cause this error.

17 - BAD LINE

If when entering a program line a syntax error occurs, the line is saved with an indication that it is in error. If the program is run without this line being corrected, execution of the line will cause this error.

NOTE: The saving of a line that contains a syntax error can be useful when LISTing and ENTERing programs.

18 - NOT NUMERIC

If when executing the VAL function, the string argument does not start with a number, this message number is generated.

Example: `A = VAL("ABC")` produces this error.

19 - LOAD, TOO BIG

The program that the user is trying to LOAD is larger than available memory.

This could happen if the user had used LOMEM to change the address at which Basic tables start, or if he is LOADing on a machine with less memory than the one on which the program was SAVEd.

20 - FILE #

If the device/file number given in an I/O statement is greater than 7 or less than 0, then this error is issued.

Example: `GET #8,A`

will produce this error.

21 - NOT SAVE FILE

This error results if the user tries to LOAD a file that was not created by SAVE.

22 - 'USING' FORMAT

This error occurs if the length of the entire format string in a PRINT USING statement is greater than 255. It also occurs if the length of the sub-format for one specific variable is greater than or equal to 60.

23 - 'USING' TOO BIG

The value of a variable in a PRINT USING statement is greater than or equal to 1E+50.

24 - 'USING' TYPE

In a PRINT USING statement, the format indicates that a variable is a numeric when in fact the variable is a string. Or the format indicates the variable is a string when it is actually a numeric.

Example:
```
PRINT USING "###",A$
PRINT USING "%%%",A
```
Will produce this error.

25 - DIM MISMATCH

The string being retreived by RGET from a device (ie. the one written by RPUT) has a different DIMension length than the string variable to which it is to be assigned.

26 - TYPE MISMATCH

The record being retreived by RGET (ie. the one written by RPUT) is a numeric, but the variable to which it is to be assigned is a string. Or the record is a string, but the variable is a numeric.

27 - INPUT ABORT

An INPUT statement was executed and the user entered cntl-C (return).

28 - NESTING

The end of a control structure such as ENDIF or ENDWHILE was encountered but the run-time stack did not have the corresponding beginning structure on the Top of Stack.

Example:

```
10 While 1 : Rem loop forever
20 gosub 100
100 ENDWHILE
```

Endwhile finds the GOSUB on Top of Stack and issues the error.

29 - PLAYER/MISSILE NUMBER

Players must be numbered from 0-3 and missiles from 4-7.

30 - PM GRAPHICS NOT ACTIVE

The user attempted to use a PMG statement other than PMGRAPHICS before executing PMGRAPHICS 1 or PMGRAPHICS 2.

31 - FATAL SYSTEM ERROR

Record circumstances leading to this error and report it to us immediately.

32 - END OF 'ENTER'

This is the error resulting from a program segment such as:

```
SET 9,1 : TRAP line# : ENTER filename
```

when the ENTER terminates normally.

APPENDIX J

COMPATIBILITIES

The following incompatibilities are between Atari Basic and BASIC A+ are known to exist:

1. BASIC A+ and Atari Basic SAVEd program files are NOT COMPATIBLE !!! However, the LISTed form of all Atari Basic programs IS compatible with BASIC A+.
   Solution: use Atari cartridge to LOAD all SAVEd programs, then LIST these programs to a diskette, then go to BASIC A+ and ENTER them and (optional) then SAVE them in BASIC A+ form.

2. Various documented RAM locations do not agree. The only three locations known to be of any significance are now deemed to be too volatile to document. Instead, alternative methods of accessing their purposes are provided:
   STOPLN -- contained line # where a program stopped or found an error -- NOW accessible via ERR(1).
   ERRSAV -- contained the last run-time error number -- NOW accessible via ERR(0).
   PTABW -- the 'tab' size used by PRINT when 'tabbing' for a comma -- NOW accessible via SET 1,<ptabw>.

3. By default, BASIC A+ allows the user to enter program text in lower case, inverse video, or upper case characters. Atari Basic allowed only upper case (non-inverse video) characters. Normally, this is not a problem; however, REMarks and DATA statements ENTERed which contain inverse video and/or lower case characters will find that these characters have been changed to normal video, upper case.
   Reason: BASIC A+ changes all inverse or lower case character strings NOT ENCLOSED IN QUOTES.
   Solutions:
   a. Put quotes into REMarks and DATA statements as needed.
   b. SET 5,0 -- this will disable entering of lower case and inverse characters; but if you are ENTERing an Atari Basic program, there will be none of these anyway.

4.

L$)
y
O,

This paragraph does not apply to version 3.04

------ now compatible with Atari BASIC ----

st
ly,
a

use

these bytes at all, so unless you have custom drivers
the difference is unnoticable.

5. Similarly exotic: When OPENing a file, th- (usually)
dummy parameter normally set to ze-
OPEN #file,mode,O,FL$). A- er, AS
WELL AS THE MODE par- Atari
Basic. With es.
In Ata- econd
p in
A in,
th to
re
NO UX2
th ne
exo ble
via rollow this
exam
al,FILE$
e+256*special,O,FILE$
Agair y situation to have occur. The
BASIC s chosen because of its compatibility with
some capabilities.

This paragraph does not apply to version 3.04 ----
now compatible with Atari BASIC ----
------

6. ATARI vs. APPLE II: If you are a software author, there are
obvious advantages in having one BASIC A+ which will run
programs unchanged on two machines. Excepting for GRaphics
capabilities, Player/Missile Graphics, SOUND, and some game
controls, BASIC A+ is completely compatible on the two
machines. Even graphics are compatible to some degree, but
see the Apple II BASIC A+ manual for more details.

7. Cartridge convenience: If you did not purchase OS/A+ (why not?)
BASIC A+ may seem a little awkward to use, what with having to
LOAD it via the DOS menu, etc. Partial solution: after
duplicating the OSS master disk, RENAME the file BASIC.COM to
AUTORUN.SYS on any Atari DOS version 2S or 2.8 master disk.
Then, when you turn on the power, DOS will boot and immediately
run BASIC A+. Of course, you must still use RUN AT ADDRESS
to return to BASIC A+ after going to DOS, but you should need
to do that less frequently now that BASIC A+ gives you so
many extended DOS-like commands. Good luck. And try OS/A+
soon -- remember it INCLUDES (at NO extra charge) an Editor/
Assembler/Debug package upward compatible with Atari's
cartridge (sound familiar ?) .

SYNTAX SUMMARY AND KEYWORD INDEX

All keywords, grouped by statements and then functions, are listed below in alphabetical order. A page number reference is given to enable the user to quickly find more information about each keyword.

STATEMENTS

page	syntax
32-H	*BGET #fn, addr, len
32-H	*BPUT #fn, addr, len
9	BYE
24	CLOAD
26	CLOSE #fn
43	CLR
48	COLOR aexp
9	CONT
25	*CP
24	CSAVE
28	DATA <ascii data>
35	DEG
12-B	*DEL line [,line]
41	DIM svar(aexp)
41	DIM mvar(aexp[,aexp])
32-A	*DIR filename
25	DOS
36-A	*DPOKE addr,aexp
48	DRAWTO aexp,aexp
22-A	*ELSE {see IF}
9	END
22-A	*ENDIF {see IF}
22-B	*ENDWHILE
25	ENTER filename
32-B	*ERASE filename
15	FOR avar=aexp TO aexp [STEP aexp]
28	GET #fn, avar
16	GOSUB line
17	GOTO line
45	GRAPHICS aexp
18	IF aexp THEN <stmts>
18	IF aexp THEN line
22-A	*IF aexp : <stmts> ELSE : <stmts> ENDIF
32-A	*INPUT "...",var [,var...]
25	INPUT [#fn,] var [,var...]
10-A	*[LET] svar=sexp [,sexp..]
10-A	[LET] avar=aexp
10-A	[LET] mvar=aexp

10	LIST [filename]
10	LIST [filename,] line [,line]
26	LOAD filename
48	LOCATE aexp,aexp,avar
12-A	*LOMEM addr
26	LPRINT [exp [;exp...] [,exp...]]
12-A	*LVAR filename
78	*MISSILE pm,aexp,aexp
71	*MOVE fromaddr,toaddr,lenaexp
10	NEW
15	NEXT avar
26	NOTE #fn, avar,avar
20	ON aexp GOTO line [,line...]
20	ON aexp GOSUB line [,line...]
26	OPEN #fn, mode,avar,filename
49	PLOT aexp,aexp
75	*PMCLR pm
76	*PMCOLOR pm,aexp,aexp
75	*PMGRAPHICS aexp
77	*PMMOVE pm[,aexp] [;aexp]
76	*PMWIDTH pm,aexp
28	POINT #fn, avar,avar
35	POKE addr,aexp
20	POP
49	POSITION aexp,aexp
28	PRINT [#fn]
28	PRINT exp [[;exp...] [,exp...]] [;]
28	PRINT #fn [[;exp...] [,exp...]] [;]
32-C	*PRINT [#fn,] USING sexp , [exp[,exp...]]
32-B	*PROTECT filename
28	PUT #fn, aexp
35	RAD
28	READ var [,var...]
10	REM <any remark>
32-B	*RENAME filenames
21	RESTORE [line]
16	RETURN
32-I	*RGET #fn, asvar [,asvar...]
32-H	*RPUT #fn,exp[,exp...]
11	RUN [filename]
29	SAVE filename
69	*SET aexp,aexp
50	SETCOLOR aexp,aexp,aexp
57	SOUND aexp,aexp,aexp,aexp
29	STATUS #fn, avar
15	STEP {see FOR}
11	STOP
32-G	*TAB [#fn], avar
18	THEN {see IF}
15	TO {see FOR}
12-A	*TRACE
12-A	*TRACEOFF
22	TRAP line
32-B	*UNPROTECT filename
22-B	*WHILE aexp
30	XIO aexp,#fn,aexp,aexp,filename
28,32-C	? {same as PRINT}

FUNCTIONS

page	syntax
33	ABS(aexp)
35	ADR(svar)
37	ASC(sexp)
34	ATN(aexp)
79	*BUMP(pmnum, aexp)
37	CHR$(aexp)
33	CLOG(aexp)
34	COS(aexp)
36-A	*DPEEK(addr)
36-A	*ERR(aexp)
33	EXP(aexp)
40-B	*FIND(sexp, sexp, aexp)
35	FRE(0)
60-A	*HSTICK(aexp)
33	INT(aexp)
38	LEN(sexp)
34	LOG(aexp)
59	PADDLE(aexp)
60-A	*PEN(aexp)
79	*PMADR(pm)
59	PTRIG(aexp)
35	PEEK(addr)
34	RND(0)
34	SGN(aexp)
35	SIN(aexp)
34	SQR(aexp)
59	STICK(aexp)
60	STRIG(aexp)
38	STR$(aexp)
69	*SYS(aexp)
36-B	*TAB(aexp)
36	USR(addr [, aexp...])
38	VAL(sexp)
60-A	*VSTICK(aexp)

EXPLANATION OF TERMS

exp	- EXPression
aexp	- Arithmetic exp
sexp	- string exp
var	- VARiable
avar	- Arithmetic var
svar	- String var
mvar	- Matrix var (or element)
fn	- File Number
line	- line number (can be aexp)
pm	- Player/Missile number (aexp)
[xxx]	xxx is optional
[xxx...]	xxx is optional, and may be repeated
addr	- ADDRess aexp, must be 0 - 65535
<stmts>	one or more statements

NOTE: keywords denoted by an asterisk (*) not in Atari Basic.

APPENDIX L

BASIC A+ MEMORY USAGE

This section describes memory usage INTERNAL to the BASIC A+ interpreter, in what was ROM in the Atari Basic cartridge. See the memory map (appendix D) and memory locations (appendix I) for RAM locations.

Throughout this section, hex addresses are used exclusively. Whenever three addresses are given together separated by slashes (e.g., 4000/6000/8000) they represent the three values associated with systems which have 32K, 40K, and 48K bytes of free RAM available.

CHARACTER GRAPHICS RESERVED AREA 4000/6000/8000

1K bytes of memory are reserved for character graphics. By reserving this memory at fixed locations (at least for any given machine size), the task of writing character set manipulators is greatly reduced.
P.S.: You can find the address of this area via the following subterfuge:

```
  Charactergraphicsaddress = (PMADR(0)-9000)&(14*4096)
```

NOTE: if you do not intend to use character graphics, you can use this area for assembly language routines, etc.

COLDSTART 4400/6400/8400

Where BASIC A+ comes upon loading from disk. Entering at this address performs the equivalent of a NEW.

WARMSTART 4403/6403/8403

Equivalent to where Atari Basic goes when the RESET key is used. Does not destroy any program, but does close files, etc.

JUMP TO TEST FOR BREAK 4406/6406/8406

BASIC A+ checks for the user's use of the BREAK key at the end of executing each line. Exotic driver's might make use of this fact to cause pseudo-interrupts to BASIC A+ at this point. Write for more details, but otherwise don't touch this.

THE SET/SYS() DEFAULT VALUES 4409/6409/8409

Upon execution of NEW, the set of 10 default byte values (SET 0 through SET 9) are moved from this location to 'RAM'. If you would like to change a default, POKE these default values and then save BASIC A+ via OS/A+.
4409 (etc.) is SET 0, 440A is SET 1, etc.

CURRENT TOP OF BASIC A+ approx. 7800/9800/B800

But we expect to add features, so if you wish to customize BASIC A+ in this area we suggest you work from the next address(es) down:

DEFINED	TOP OF BASIC A+	7B00/9B00/BB00

This is where Players from Player/Missile Graphics start in PMG.1 mode. Also, the area from 7C00/9C00/BC00 up is used by Atari's OS ROM upon RESET and power up to initialize the graphics screen.