micro lab

Presents:

+LANGUAGE PLUS+

Volume One
Disk I/O & Keyboard Input

written by:

Michael J. Hatlak
and
Curt Rostenbach

A collection of 16 Ampersand commands for use with APPLESOFT BASIC on your 48K APPLE][/APPLE][Plus or APPLE //e computer with floppy disk drive.

Language Plus+ modules can help you write cleaner and more efficient APPLESOFT BASIC programs.

Language Plus+ adds functions not before available.

Catalog to strings	Deallocate arrays
ONERR GOTO Fix	Fast Free Memory
Fast Garbage Collection	"HELLO" name to string
Instring substitution	Faster Disk I/O
String legalization	String trim
16-bit PEEK	

2699 Skokie Valley Road • Highland Park, IL 60035 • (312) 433-7550

Table of Contents

INTRODUCTION

Welcome to the wonderful world of LANGUAGE PLUS+! The LANGUAGE PLUS+ modules extend the power of APPLESOFT BASIC through the use of the ampersand ('&') command* and are referred to as 'amper' commands or routines. This allows special purpose commands to be created and added to the normal list of APPLESOFT BASIC instructions. These special purpose commands will perform functions normally not available in BASIC, or will replace large, slow subroutines with a single instruction. The amper routines have been written in machine code just the same as the APPLESOFT BASIC language has been written. This means that the routines will execute quickly with minimal space requirements.

This documentation has been written by and for programmers. For those new to the fun, it may be somewhat perplexing. A few pointers are in order to help you.

Presuming you will become as used to these routines as we, the documentation is organized for quick reference. Each command is explained in several sections.

The 'FORMAT' gives a concise layout of required and optional parameters. Each amper routine uses two letter INTEGER variables to specify the function to be performed. Some amper routines may require information to perform their intended function. This extra information will be referred to as 'parameters'.

Each amper routine, as documented in this manual, has a certain order in which it expects its parameters to be. This is its 'format'. Parameters shown within these formats will be generic labels for variables and constants the user is to supply when using the amper routines within a program.

'Constants' are numeric values e.g. 10, 123.45. 'Variables' are BASIC variables as defined within the APPLESOFT BASIC PROGRAMMING REFERENCE MANUAL or the APPLESOFT TUTORIAL. Variable types may be REAL, INTEGER, or STRING.

Some amper routines have optional parameters. These optional parameters will be marked in the format within brackets ('[' and ']'). These brackets are not part of the parameters. They are only used to show the place of optional parameters within the amper routine's format.

```
  FORMAT:  &RE%,file$[,(record,byte)],info$[, info$(n), ... ]
```

To decode the above, '&RE%' is used to invoke the amper routine. 'file$' means that you have to supply a string variable name that will contain the name of the disk file from which to read data. The '$'

* See page 123 of APPLESOFT BASIC REFERENCE MANUAL.

shows that a STRING variable is required. '%' means that an INTEGER variable is required. No specifier means that the variable is a REAL.

'[' and ']' mark off optional parameters and punctuation.
'[, info$(n), ...]' means that added variables are optional.
'(n)' shows that arrays may be used starting at any element.
'...' allows that this can go on indefinitely.

The following is an example of a "real" command: Presume that ML$ contains the name of a mailing list file, and RN specifies a record number within that file. And NA$, AD$, CT$, ST$, and ZP$ represent string variables to store name, address, city, state, and zip respectively. And CM$(0) specifies a list of comments to be read into string array CM$. Using the above format should look like:

```
    &RE%,ML$,(RN),NA$,AD$,CT$,ST$,ZP$,CM$(0)
```

Of course, you would have to read the entire instructions on the individual routines to know what the parameters stand for and how they are used.

The parameters are explained in the section that follows the format declaration.

Capital letters are parameters that must be used EXACTLY. Lowercase letters indicate that the parameters are to be specified by the user.

Sometimes numbers are shown as groups of letters, e.g., 'vvvds' means that in a 5 digit number, the first three digits 'vvv' signify one value, 'd' another value, and 's' another. The exact meanings will be described in the text. In one place 'vvv' stands for volume, 'd'-drive, and 's'-slot. If 25416 is the number, it means volume=254, drive=1, and slot=6.

'Description' explains the purpose of the amper command. Hands on examples will demonstrate the principles of operation for the routine.

'Operational Notes' describe how the routine functions when invoked by the user. This is a step by step explanation of how the routine goes about performing its function. In some routines it is significant in regard to what the routine will perform before an error condition halts its operation.

'Programming Notes' are points that the programmer must be aware of when using the routine within a program. This may be a statement of what kind of error messages will be generated if something goes wrong, or it may give practical and sometimes novel applications of the routine status variable.

Since a thousand words are worth only a small example, we have demonstrated a program with the routine in actual use.

Loading the Amper routines

The Language Plus+ amper routines are loaded into memory by BRUNning the binary B file that contains the amper routines you need for your application.

Only one module may be BRUN at a time, because each binary B file is set to exist at the highest possible location within a 48K disk operating Apple system.

When an amper routine is BRUN, it is loaded high into memory and begins operation by hooking itself into APPLESOFT BASIC via the ampersand command and protects itself by moving HIMEM to a position under the operating phase of the module. Control is then passed to APPLESOFT. The initializing phase of the module will be destroyed in the general operation of APPLESOFT. You cannot make working copies of the modules from memory after they have been BRUN.

The amper routines will remain active in APPLESOFT until an 'FP' command is issued to reset HIMEM. Unpredictable results may occur if attempts are made to use the amper routines after disconnecting them from APPLESOFT.

==
!! IMPORTANT !! IMPORTANT !! IMPORTANT !!
==

The Language Plus+ modules when loaded under program control;

MUST: Be loaded BEFORE the assignment of any string variables!

MUST: Use the CHR$(4) function to signal DOS.

For example. 10 PRINT CHR$(4);"BRUN LANGUAGE PLUS VOLUME 1"

==
!! IMPORTANT !! IMPORTANT !! IMPORTANT !!
==

The amper routine BRUN instructions may be coded in the beginning of the application programs. It is important that no string variables be created before the module is loaded. The loaded module would clobber the string contents as it was being loaded. It is therefore recommended that the user program the DOS BRUN command using the CHR$(4) instead of assigning CHR$(4) to D$ and then issuing the BRUN command.

AMPER CATALOG !

```
FORMAT:  &CA%,array$(n)[,d,s,v]

 WHERE:  CA% - Returned status value.
               if +, # of active files.
                  -, # DOS error message.

   array$( ) - String array used to store file names.
           n - Starting array element.

    array$(n)='xxx VOLUME  SLOT x  DRIVE x  FREE xxx'
  array$(n+1)='pt sss filename--------------------'
      .      .
 array$(n+CA%)=file CA%

             p - Protection.
                 ' ' = unlocked
                 '*' = locked
             t - File Type.
                 'I' = Integer BASIC
                 'A' = APPLESOFT BASIC
                 'T' = Text
                 'B' = Binary
                 'R' = Relocatable
                 'S' = unknown
           sss - File Size in sectors.  NOTE: This number will be
                 actual size; not MOD 256 like normal CATALOG.
      filename - File Name.  NOTE: May contain illegal characters and
                 will be 30 characters in length.

             d - optional Drive selection.
             s - optional Slot selection.
             v - optional Volume selection.
```

D,S,V Notes

If d, s, or v parameters are omitted, current Slot and Drive will be used with Volume set to zero. Constants or Variables may be used in disk drive specification. If constants are used, they are taken to be in Drive, Slot, Volume order. If variables are used, the first letter of the variable may be used to identify the parameter, e.g., if D=2, then &CA%,FILE$(0),D would select file names from drive 2 of the current slot to be read into string array FILE$. If constants and variables are intermixed, variable names take precedence, and constants that follow continue in their order from the last variable, e.g., if V=254 and D=2, then &CA%,FILE$(0),V,D,5 would select drive 2, slot 5, volume 254. If a variable name does not begin with D, S, or V, then the order for constants is assumed.

Description

The Amper Catalog command allows you to read the contents of a diskette's catalog into an array of string variables. It also supplies current volume, slot, drive, and the number of free sectors remaining on the diskette. To use the routine, you must supply the starting element of a string array where the information is to be stored.

Example: Hands-on use of Amper Catalog.

```
     ]BRUN AMPER CATALOG         | Load module into BASIC.
     ]DIM FI$(105)               | Dimension array to hold file names.
     ]&CA%,FI$(0)                | Get filenames.
```

Disk IN USE light will turn on and strings will be loaded.

```
     ?SYNTAX ERROR               | Ignore in immediate mode.
```

Since Amper Catalog uses the keyboard buffer for temporary storage, it's contents are trashed and therefore when BASIC tries to find the next command to execute, it sees garbage and produces the above message. This happens ONLY in immediate keyboard mode and does not occur within a program. However, in immediate mode, if the disk IN USE light did not come on before the SYNTAX ERROR was printed, it is a valid message.

```
     ]?CA%                       | Print file count.
     35                          | Number may be different.
     ]?FI$(0)                    | Print diskette information.
     254 VOLUME  SLOT 6  DRIVE 1  FREE 097
     ]FOR X=1 TO CA%:?FI$;"<":NEXT
     *A 002 LANGUAGE PLUS VOLUME 1          <
      B 032 LOGO                            <
      .  .   .                              .
      R 004 AMPER TRIM                      <
```

The '<' character was printed to show that all the filename strings returned are equal length and padded with spaces at the end.

Open the disk drive door.

```
     ]&CA%,FI$(0)
```

The disk IN USE light should come on, the head should recalibrate and then the APPLESOFT BASIC prompt should appear. Note that '(beep) I/O ERROR' did not occur.

```
     ]?CA%                       | Print status variable.
     -8                          | DOS error 8=I/O ERROR.
```

The error value returned in CA% will always be the negative of the standard Apple DOS error codes*. Your program should always check for error conditions before using the data in the strings.

* See pages 114-115 of THE DOS MANUAL.

Operational Notes

Strings are requested from APPLESOFT as active files are found; therefore, disk operations may stop if garbage collection is automatically invoked by APPLESOFT.
If an error occurs during the operation of the routine, the count of file names stored in the strings will be lost, and replaced with the error number. Whatever file names that have been assigned before the error occurred, will be available in the array.

Programming Notes

Disk information can be extracted from the string using the following:

```
     volume = VAL(array$(n))
       slot = VAL(MID$(array$(n),18))
      drive = VAL(MID$(array$(n),27))
       free = VAL(RIGHT$(array$(n),3)
```

All file name strings are returned 37 characters long.
After Amper Catalog operation, Slot, Drive, and Volume parameters in DOS revert to previous values.
The maximum number of files on a DOS 3.3 diskette is 105. Modified systems may allow more.
The keyboard buffer, $200-$2FF, is used for temporary storage.

Programming Example

This example will read the catalogs from the disks that will be mounted in drive 2 and store them in a file on drive 1.

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 20 D$=CHR$(4):REM CTRL-D
 30 DIM FILE$(105)
 40 PRINT D$;"OPEN CATALOG FILE,D1":D=2
 45 PRINT "MOUNT DISK TO BE CATALOGED IN DRIVE 2"
 50 INPUT "RETURN WHEN READY, 'DONE' TO END:";A$
 55 IF A$="DONE" THEN 300
 60 &CA%,FILE$(0),D:REM GET FILE NAMES
 70 IF CA%=>0 THEN 100
 80 PRINT "-- DISK ERROR #";ABS(CA%);" --"
 90 GOTO 45
100 IF CA%>1 THEN 200
110 PRINT "-- DISK EMPTY --"
120 GOTO 45
200 PRINT D$;"WRITE CATALOG FILE"
210 FOR X=1 TO CA%
220 PRINT FILE$(X):REM SAVE NAMES
230 NEXT X
240 PRINT D$
250 GOTO 45
300 PRINT D$;"CLOSE CATALOG FILE"
310 END
```

AMPER DEALLOCATE ARRAY

FORMAT: &DA%,array(n)[,array$(n),array%(n),...]

WHERE: DA% - Used to invoke Amper Deallocate.

array(n) - array to be erased from memory.

Description

Amper Deallocate will remove from memory any type array and will recover the space used by that array. You may specify as many arrays to be deallocated as you wish by listing them after the command and separating them with commas. The index contained within the parentheses is required, however, its value is not used. Any numeric constant or variable will do. For programming consistency, the use of zero is recommended.

Operational Notes

The routine will remove arrays for the variable storage in the order specified. After each array is removed, array variables higher in memory are moved down over the old array. Therefore, it is faster to deallocate arrays in the reverse order than they were DIMensioned within the APPLESOFT BASIC program.

Programming Notes

If the array specified does not exist, no error will occur.
If multi-dimensional array is DIMensioned, the deallocation request should also specify an equal number of dimensions.
For example, DIM A%(10,20,30,40) should be deallocated with &DA%,A%(0,0,0,0).
The variable DA% is not used except to invoke the deallocation routine when used in conjunction with the '&'. It may be used for data storage, but it is not recommended.

Programming Example

```
 5 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
10 DIM A(100),A$(100),A%(100)
20 REM NOW ARRAYS ARE DIMENSIONED
30 &DA%,A(0),A$(0),A%(0)
40 REM NOW THEY ARE GONE
50 DIM A(100)
60 END
```

NOTE: Line 50 would cause a REDIMENSIONED ARRAY ERROR if A(n) was still in memory.

AMPER FIX

FORMAT: &FX%

Description

This routine formalizes the call to the APPLESOFT ONERR GOTO bug fix. When in the course of creating a bulletproof APPLESOFT BASIC program, the ONERR GOTO statement pair is generally pressed into service. This command pair allows you, the programmer, to keep control of execution when things go west. Problems with the ONERR GOTO materialize after you have directed program execution to continue via a GOTO rather than RESUME. Initially there is no problem, but after several iterations, program execution will leap into the unknown. Apple recommends doing a series of POKEs to install the routine and then execute CALL 768 to cure this malady. This amper routine, once installed, will allow you to clear up the error stack without having to use an undecipherable CALL statement to prime RAM real estate which could contain anything from the ONERR FIX to a sound effect generator. Use of this routine will make your program a teensy bit more readable.

Operational Notes

This is an implementation of the ONERR fix subroutine documented on page 136 of the APPLESOFT BASIC PROGRAMMING REFERENCE MANUAL.

Programming Notes

This routine does not return any values, nor does it require any parameters to function. Use after an ONERR GOTO has transferred control to your error handling routine, and you will be using something other than RESUME to continue program operation.
Until the Amper Fix is loaded, it is not a legal operation.

Programming Example

```
 10 ONERR GOTO 200
 20 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 30 ONERR GOTO 100
 40 INPUT "GIVE ME A NUMBER:";A
 50 INPUT "  ANOTHER NUMBER:";B
 60 C=A/B
 70 PRINT A;"/";B;"=";C
 80 GOTO 40
100 IF PEEK(222)<>133 THEN RESUME
110 &FX%:REM FIX ERROR STACK
120 PRINT "DIVISION BY ZERO!!"
130 GOTO 40
200 PRINT "UNRECOVERABLE ERROR"
210 END
```

AMPER FREE MEMORY

FORMAT: &FR%,variable

WHERE: FR% - Used to invoke Amper Free Memory.

variable - a REAL variable in which to store value.

Description

Amper Free Memory gives you access to the amount of currently free RAM. Note that this is different from the total amount of free space available. To get the total amount, you must use FRE(0). That takes a terrible amount of time while it does an inefficient method of garbage collection to pack all the strings together so it can then compute the difference between the top of variable storage and the bottom of the (just packed) strings. That number does not give you the working amount of storage available. As APPLESOFT BASIC assigns and reassigns strings, the amount of working free memory shrinks until it reaches a point that garbage collection is invoked to clean everything up and set free space back to the maximum. Knowing how much free space you are using and how much is available at any given instance, can help your program decide on a course of action that can optimize operating time. The Amper Free Memory gives you the amount of free storage at the time it is called. This information could be used to decide whether or not to invoke garbage collection manually so an input routine will not suddenly bog down in garbage collection while data entry is taking place. With APPLESOFT BASIC's garbage collection, program operation can be suspended for over 15 minutes!* With Amper Free Memory, you can warn the operator that a pause in operations will occur while you do garbage collection. This will prevent all the situations where the RESET was used, "Because the computer died," and nothing was wrong! In data communications, long pauses can totally disrupt operations. This routine can be used to avoid the use of APPLESOFT's garbage collection during critical operations.

Operational Notes

Amper Free Memory computes the amount of free memory between the top of variable storage and the bottom of string storage. It stores the value in the supplied variable which must be of type REAL. The value will be normalized so the result will not be negative if greater than 32767, and using this function will not invoke the infamous garbage collection routine.

* See Amper Garbage Collect for a (much) faster routine.

Programming Notes

This routine will not invoke garbage collection.
The variable type to store the response must be REAL.
The result will be internally normalized so response will always be positive number, even above 32767.
The response is free space between top of variables and bottom of string space at the time the routine is invoked. It is NOT the TOTAL free space available to the APPLESOFT BASIC program.
Testing responses between two intervals can be used to determine if garbage collection has occurred. If the second response is greater than the first response, garbage collection has occurred.
The variable FR% is not used other than to invoke the amper routine and may be used for storage, however, it is not recommended.

Programming Examples

This example shows a simple usage of the Amper Free Memory routine.

```
  5 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 10 A=0
 20 &FR%,A
 30 PRINT "FREE MEMORY=";A
 40 END
```

The following example demonstrates how string manipulation uses up free space and at what point automatic garbage collection is invoked to clean up the string space.

```
  5 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 10 REM CREATE SIMPLE VARIABLES
 20 A=0:B=0:X=0
 30 REM CREATE LARGE STRING ARRAY
 40 REM FORCING LONG DURATION GARBAGE COLLECTION
 50 DIM A$(1000)
 60 FOR X=0 TO 1000
 70 &FR%,A: REM GET INITIAL SPACE
 80 REM CREATE NEW STRING
 90 A$(X)="ELEMENT "+STR$(X)+" FREE "+STR$(A)
100 &FR%,B: REM GET NEW AMOUNT OF SPACE
110 PRINT X;" FREE: ";B;
120 IF A>B THEN PRINT "  USED: ";A-B
130 IF A<B THEN PRINT "  GARBAGE COLLECTED"
140 NEXT X
150 PRINT CHR$(7);"--DONE--"
```

If the above program is timed while running, it will spend 1 minute and 33 seconds working with 2 minutes and 31 seconds doing garbage collection for a total of 4 minutes and 5 seconds. Add "65 &GA%". Times will then be 1 minute and 34 seconds working and only 14 seconds for garbage collection, giving a total run time of 1 minute and 49 seconds. A reduction of 2 minutes 15 seconds in total run time.

AMPER GARBAGE COLLECTION

FORMAT: &GA%

WHERE: GA% - Returns free space if garbage strings collected otherwise, returns zero.

Description

'Garbage Collection' is the phrase for the operation the computer goes through when doing housekeeping of ASCII character strings. In APPLESOFT, the string variables and pointers are kept in the variable tables which are low in memory. The string contents are stored high in memory and the string variables 'point' to their contents. As new string values are assigned, the new string contents are stored in the free memory pool, the string 'pointers' are set to the new location, and the old string contents are 'forgotten'. This cannot go on forever. Eventually the strings collide with variable storage. When this happens, the 'live' strings are moved to the top of memory over the older strings. APPLESOFT BASIC starts looking through its variable table for the string with the highest location in memory that has not been moved. When it finds one, it is moved high in memory and its pointers are adjusted to the new location. The process continues until all strings have been moved. If you have a large number of strings, this can take astronomical amounts of time.

Operational Notes

The Amper Garbage Collection routine uses a superior algorithm to find 16 strings at a time and move them only if necessary.

Programming Notes

Amper Garbage is set to return immediately unless free space is less than 1K bytes with GA% equal to zero; otherwise, garbage collection is done, and GA% will equal free memory space.
It is recommended to call Amper Garbage frequently to avoid APPLESOFT garbage collection.

Programming Example

```
10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
20 FOR X=1 TO 1000
30 FOR Y=1 TO 1000
40 A$="XVALUE "+STR$(X)+"  YVALUE "+STR$(Y)
50 &GA%:REM TRY TO COLLECT GARBAGE
60 NEXT Y
70 NEXT X
80 END
```

See Amper Free Memory example 2 for execution time information.

AMPER HELLO

```
FORMAT:  &HE%,variable$[,d,s,v]

  WHERE:  HE% - Returned status value.
                if >0 = vvvsd.
                        vvv = volume #.
                          s = slot #.
                          d = drive #.
                     <0 = DOS error #.

    variable$ - String variable that will store 'HELLO' program
                "filename".  NOTE:  May contain illegal characters.

            d - Optional Drive selection.
            s - Optional Slot selection.
            v - Optional Volume selection.
```

D,S,V Notes

If d, s, or v parameters are omitted, current Drive and Slot values default with Volume set to zero. Constants or Variables may be used in parameter specification. If constants are used, the order is Drive, Slot, Volume. If variables are used, the first letter of the variable may be used to identify parameter, e.g., if D=2 then &HE%,FILE$,D would select the 'HELLO' name from drive 2 of the current slot to be read into string FILE$. If constants and variables are intermixed, variable names take precedence and reset constant order. If variable name does not begin with S, D, or V, then constant order is assumed.

Description

Generally one of the hardest tasks when trying to write 'fool-proof' programs, is the handling of disk mounting. There are so many things that could go wrong. Traditionally, the method to determine what disk the user had mounted, was to have a special 'Disk ID' file that contained the name of the diskette. This required trying to open a file on the 'unknown' disk and see if the name matched. Easy?

1. If the disk was mis-mounted or the drive door was left open;
   DISK I/O ERROR occurs.
2. If disk does not belong and is write protected;
   WRITE PROTECT ERROR occurs.
3. If disk does not belong and full;
   DISK FULL ERROR occurs.
4. If disk does not belong and none of the above;
   END OF DATA ERROR, creating 1 sector file.
5. If disk does belong;
   Does name match request?

So for the 'simple' task of mounting a disk, you, the programmer, should have been checking for all the above conditions, along with all the other error conditions that could have occurred elsewhere.

A better solution now exists! Amper Hello allows you to determine what diskette has been mounted in the disk drive, via the 'HELLO' name. Each diskette is initialized with a named file. This name is generally used by DOS to specify which program will be executed after boot up. Unique names can identify program diskettes from data diskettes within applications. Alternately, the volume number can be used for identification. Amper Hello supplies this information as well as slot and drive numbers.

Example: Hands-on use of Amper Hello.

```
    ]BRUN AMPER HELLO           | Load module into BASIC.
    ]&HE%,A$                    | Get 'HELLO' name.
```

Disk IN USE light will turn on and string will be loaded with 'HELLO' name.

```
    ]?HE%                       | Print volume and drive info.
    25461                       | Volume 254, Slot 6, Drive 1.
    ]?">";A$;"<"                | Print 'HELLO' name.
    >LANGUAGE PLUS VOLUME 1<
```

The '>' and '<' were printed to show that the 'HELLO' name returned is variable length and not padded at the end of the string.

Open the drive door.

```
    ]&HE%,A$
```

The disk IN USE light should come on, the head should recalibrate and the the APPLESOFT BASIC prompt should appear. Note that '(beep) I/O ERROR' did not occur.

```
    ]?HE%                       | Print status variable.
    -8                          | DOS error 8=I/O ERROR.
```

The error value returned in HE% will always be the negative of the standard Apple DOS error codes*. Your program should always check for error conditions before using the data in the string.

Operational Notes

The routine examines the boot tracks for the 'HELLO' name and stores it in the string specified. The volume number, slot, and drive are encoded in the status variable.
If an error occurs, the content of the string is not valid.

* See pages 114-115 of THE DOS MANUAL.

Programming Notes

Amper Hello works on DOS 3.3 only; other versions may return garbage.
It reads track 1, sector 9, bytes $75-$92 for 'HELLO' file name.
Disk information can be extracted from the status variable using the following;

```
   volume = INT(HE%/100)
     slot = INT(HE%/10)-INT(HE%/100)*10
    drive = HE%-INT(HE%/10)*10
```

After Amper Hello operation, Slot, Drive, and Volume parameters revert to previous values.
The maximum string size returned can be 30 characters long. The minimum should be 1.

Programming Example

This example program will ask for a specific disk to be mounted, and then test the 'HELLO' name to be sure it is mounted before continuing execution.

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 20 D$=CHR$(4):REM CTRL-D
 30 MOUNT$="DATA DISK":GOSUB 200
 40 GOSUB 100:REM PROCESS DATA DISK
 50 MOUNT$="PROGRAM DISK":GOSUB 200
 60 PRINT D$;"RUN MENU"
 70 END

100 REM SUBROUTINE TO PROCESS DATA DISK
110 RETURN

200 REM DISK MOUNT SUBROUTINE
210 PRINT
220 PRINT "MOUNT ";MOUNT$;" IN DRIVE"
230 INPUT "PRESS <RETURN> WHEN READY";A$
240 &HE%,FILE$
250 IF HE%<0 THEN 300
260 IF FILE$=MOUNT$ THEN RETURN
270 PRINT "DISK MOUNTED IS ";FILE$
280 GOTO 210
290 REM DISK ERROR HANDLER
300 FLASH:PRINT "DISK ERROR":NORMAL
310 GOTO 230
```

AMPER INPUT/OUTPUT

FORMAT

```
   Open:  &OP%,file$[,(length),d,s,v]
 Create:  &CR%,file$[,(length),d,s,v]
   Read:  &RE%,file$[,(record,byte)],variable$[,variable$,...]
  Write:  &WR%,file$[,(record,byte)],variable$[,variable$,...]
  Close:  &CL%,file$
```

Description

These routines were designed to allow the programmer several advantages over normal Apple DOS disk I/O operations. These advantages are:

1) Improved programmer control of disk error conditions.
2) 50% time reduction for I/O operations.
3) Automatic transfer of single dimensional string array contents.
4) Input/Output 'anything', commas, colons, control characters.
5) Can be executed directly from the keyboard.

These routines make direct calls into DOS's File Manager package, by-passing the character I/O hooks (no more blinking cursor) and related parsing routines.

Status of operations are passed back to the user in the variable name that was used to invoke the module, allowing local error handling (vs. global ONERR GOTO routine).

Whole single dimensional string arrays may be transferred by specifying element zero (vs. FOR - NEXT loop structure).

Almost any character is valid for input or output. Exceptions are: Carriage return and null ($00) since these are used as record and file terminators.

These routines are compatible with the normal DOS commands. Amper I/O and DOS commands can be used together with no problem.

Operational Notes

See individual routines that follow.

Programming Notes

See individual routines that follow.

Programming Example

See individual routines that follow.

AMPER OPEN

```
FORMAT:  &OP%,file$[,(length),d,s,v]

 WHERE:  OP% - Returned status value.
               if =0 = No error occurred.
                  >0 = DOS error #.

       file$ - String variable containing file name to be opened.

      length - Optional record length description.

           d - Optional Drive selection.
           s - Optional Slot selection.
           v - Optional Volume selection.
```

D,S,V Notes

If d, s, or v parameters are omitted, current Slot and Drive values are used and Volume is set to zero. Constants or Variables may be used to specify disk information. If constants are used, the order is Drive, Slot, Volume. If variables are used, the first letter of the variable may be used to identify parameter. If constants and variables are intermixed, variable names take precedence and reset constant order. If variable name does not begin with D, S, or V then constant order is assumed.

Description

This routine allows the user to open existing files for disk I/O. If the file does not exist, it will not be created. Random access files are specified by enclosing the constant or variable of the length in parenthesis '(...)' following the file name variable.

Example: Hands-on use of Amper Open.

```
]BRUN AMPER I/O          | Load module into BASIC.
]FI$="TEST DATA"         | Assign string value.
]&OP%,FI$                | Open the test file.
```

Disk IN USE light will turn on and file opened for sequential I/O.

```
]?OP%                    | Check operation status.
0                        | Anything else is an error.
]&OP%,FI$,(30)           | Open file as random access.
]?OP%                    | Test status.
0
```

Open the disk drive door.

```
]&OP%,FI$
```

The disk IN USE light should come on, the head should recalibrate and then the APPLESOFT BASIC prompt should appear. Note that '(beep) I/O ERROR' did not occur.

```
    ]?OP%                   | Print status variable.
    8                       | DOS error 8=I/O ERROR.
```

The error value returned in OP% will be equal to the standard Apple DOS error code*. Your program should always check for error conditions before continuing execution.

Close the disk drive door.

```
    ]NF$="PHRED"            | Assign string value.
    ]&OP%,NF$               | Attempt to open file.
```

If previous example followed, drive will recalibrate again and then return the APPLESOFT BASIC prompt.

```
    ]?OP%                   | Print status variable.
    6                       | DOS error 6=FILE NOT FOUND.
```

If operation status was 8 again, re-insert diskette and try again. "PHRED" should not have been an existing file, therefore, the response should have been 6-FILE NOT FOUND. Amper Open will not create files if they are not found. To create files, use the Amper Create routine.

Amper Open was designed for use in applications where you are reading and/or modifying files that have already been created. Use of Amper Open will prevent empty 1 sector files from being created when the wrong disk has been mounted. Amper Open could also be used to test for the existence of a file before using Amper Create to generate a new file.

Operational Notes

DOS buffers are scanned for availability and marked for use if empty. If file name is already open, its buffer is reused and file pointers are rewound to beginning of file.
File name must previously exist on selected drive/slot/volume or OP%=6 - FILE NOT FOUND error code is returned.

Programming Notes

Variable OP% may be used as a conditional flag (e.g., IF OP% THEN error occurred). Location 222 ($DE) will contain the error number as well as the variable. This gives compatibility with ONERR GOTO error handling routines.

Programming Example

See Amper I/O programming examples section that follows AMPER CLOSE.

* See pages 114-115 of THE DOS MANUAL.

AMPER CREATE

```
  FORMAT:  &CR%,file$[,(length),d,s,v]
   WHERE:  CR% - Returned status value.
                 if =0 = No error occurred.
                    >0 = DOS error #.

         file$ - String variable containing file name to be created.

        length - Optional record length description.

             d - Optional Drive selection.
             s - Optional Slot selection.
             v - Optional Volume selection.
```

D,S,V Notes

If d, s, or v parameters are omitted, current Slot and Drive values are used and Volume is set to zero. Constants or Variables may be used to specify disk information. If constants are used, the order is Drive, Slot, Volume. If variables are used, the first letter of the variable may be used to identify parameter. If constants and variables are intermixed, variable names take precedence and reset constant order. If variable name does not begin with D, S, or V, then constant order is assumed.

Description

This routine creates new files for disk I/O. When used on an existing file, it erases the file and re-opens it as an empty file. Random access files are specified by enclosing the constant or variable of the length in parentheses '(...)' following the file name variable.

Example: Hands-on use of Amper Create.

```
    ]BRUN AMPER I/O             | Load module into BASIC.
    ]FI$="NEW FILE"             | Assign string value.
    ]&CR%,FI$                   | Create the new file.
```

Disk IN USE light will turn on and file will be created and opened for sequential I/O.

```
    ]?CR%                       | Check operation status
    0                           | Anything else is an error.
    ]&CR%,FI$,(100)             | Create file as random access.
```

File "NEW FILE" was deleted and recreated as a random access file with record length of 100 characters.

```
    ]?CR%                       | Test status.
    0
```

Open the disk drive door.

```
     ]&CR%,FI$                    | Force an error condition.
```

The disk IN USE light should come on, the head should recalibrate and then the APPLESOFT BASIC prompt should appear. Note that '(beep) I/O ERROR' did not occur.

```
     ]?CR%                        | Print status variable.
     8                            | DOS error 8=I/O ERROR.
```

The error value returned in CR% will be equal to the standard Apple DOS error code*. Your program should always check for error conditions before continuing execution. If an error condition status value is returned, the file has not been created nor opened for subsequent read or write disk I/O. All read and write commands following an OPEN or CREATE that returned an error condition, will return status value 6-FILE NOT FOUND.

Operational Notes

DOS file buffers are scanned for availability and marked for use if empty.
If file name is not found on disk, it will be created and added to the file directory.
If file name exists, it will be deleted before it is opened for subsequent I/O operations.

Programming Notes

If length parameter is not specified, the file type is assumed to be sequential.
Variable CR% may be used as a conditional flag (e.g., IF CR% THEN error occurred). Location 222 ($DE) will contain the error number as well as the variable. This gives compatibility with ONERR GOTO error handling routines.
In the event of an error condition, the file will not be opened for subsequent disk I/O operations.

Programming Example

See examples section that follows AMPER CLOSE.

* See pages 114-115 of THE DOS MANUAL.

AMPER READ

```
FORMAT:  &RE%,file$[,(record,byte)],variable$[,variable$,...]

 WHERE:  RE% - Returned status value.
               if =0 = no error occurred.
                  >0 = DOS error #.

       file$ - String which contains file name being read.

      record - Optional record number.
        byte - Optional byte parameter.
```

Record, byte Notes

If record or byte parameters are specified, they must be inside parentheses '(...)' to be recognized as such. Constants or variables may be used in parameter specification. If constants are used, the order is Record then Byte. If variables are used, the first letter of the variable may be used to identify parameter, e.g., if B=5 then &RE%,FILE$,(B),INFO$ would read from byte 5 of the current record into variable INFO$. If constants and variables are intermixed, variable name takes precedence. If variable name does not begin with 'R' or 'B' then constant order is assumed.

```
   variable$ - String variable where disk text data is to be stored.
               If single dimensional string array variable element
               zero (0) was specified, then the routine will attempt
               to read data into elements 1 thru array dimemsion
               limit.  The number of elements read will be stored in
               element zero.
```

Description

Amper Read allows you to input STRING variables from the specified open disk file. Since the file data is supplied to the routine in string variable form, it does not have to go through lengthy parsing operations within the DOS File Manager. This results in faster operation than equivalent DOS commands. The routine also supports the automatic input of single dimensional string arrays, which yields an even faster execution time over conventional DOS commands. The status variable after each operation gives you, the programmer, more precise control over the handling of error conditions than available through ONERR GOTO constructions.

It is strongly recommended that the user be totally familiar with Apple DOS file structure and how to read both sequential and random access files to completely understand the operation and use of this routine. The user should review Chapter 6: Using Sequential Files* and Chapter 8: Using Random-Access Files# before attempting the

* See pages 47-71 of THE DOS MANUAL.
See pages 81-89 of THE DOS MANUAL.

following examples.

Example: Hands-on use of Amper Read sequential.

```
    ]BRUN AMPER I/O           | Load Module into BASIC.
    ]FI$="TEST FILE"          | Assign string value.
    ]&OP%,FI$                 | Open file.
    ]?OP%                     | Check success of open.
    0                         | If not 0, check error.
    ]&RE%,FI$,A$              | Read A$
    ]?RE%                     | Check operation.
    0                         | Should be zero.
    ]?A$                      | Look at input.
    LINE 1 FROM TEST FILE     | Should look like this.
    ]&RE%,FI$,A$:?RE%:?A$     | Get another and print
    0
    LINE 2
    ]&RE%,FI$,A$,B$,C$        | Get three more.
    ]?RE%:?A$:?B$:?C$
    0
    LINE 3
    LINE 4
    LINE 5
    ]&RE%,FI$,A$              | Try one more.
    ]?RE%:?A$
    5                         | Error 5=END OF DATA.
    LINE 3                    | Variable unchanged.
```

As you can see, there are a few differences between normal DOS commands and Amper I/O. First, we are able to do this without getting 'NOT DIRECT COMMAND' errors. Second, we can only input STRING variables. Third, we did not get the '(beep) OUT OF DATA' error when we tried to read past the end of our test file*.

```
    ]&OP%,FI$                 | Rewind file.
    ]?OP%                     | Always check result.
    0
    ]DIM A$(5)                | Create an array.
    ]&RE%,FI$,A$(0)           | Read in an array.
    ]?RE%:?A$(0)              | Print status values.
    0                         | Read was OK.
    5                         | Number of strings read.
```

Whenever element zero (0) of an array is specified, and only when element zero is specified, does Amper Read attempt to input as many strings as the variable specified is dimensioned to.

```
    ]FOR X=1 TO VAL(A$(0)):?A$(X):NEXT
    LINE 1 FROM TEST FILE
    LINE 2
    LINE 3
    LINE 4
    LINE 5
```

* Nor will we get ANY standard DOS errors using these routines.

Example: Hands-on use of Amper Read random access.

```
     ]&OP%,FI$,(100):?OP%     | Open file with record length 100.
     0
     ]&RE%,FI$,(1),A$         | Read record 1
     ]?RE%:?A$                | Print result.
     0
     RECORD 1 LINE 1 OF 3
     ]&RE%,FI$,B$,C$          | Read two more strings from record 1.
     ]?RE%:?B$:?C$            | Print results.
     0
     RECORD 1 LINE 2 OF 3
     RECORD 1 LINE 3 OF 3
     ]&RE%,FI$,(0,12),A$      | Read record 0 byte 12.
     ]?RE%:?A$
     0
     TEST FILE
```

When specifying record and byte offset values, record is implied as being first, and byte offset second. To specify byte offset alone, we must assign it to a variable that begins with the letter 'B'. For more information on the byte parameter, see page 69 of THE DOS MANUAL.

```
     ]BY=7                    | Go after byte 7 of file.
     ]&RE%,FI$,(BY),B$        | Record # will equal zero.
     ]?RE%:?B$
     0
     FROM TEST RECORD
```

When using arrays and the random access files, it is important to remember that the array reads are sequential within the specified record.

```
     ]&RE%,FI$,(0),A$(0)      | Read 5 lines of record 0.
     ]?RE%:?A$(0)
     0
     5
     ]FOR X=1 TO 5:?A$(X):NEXT
     LINE 1 FROM TEST FILE
     LINE 2
     LINE 3
     LINE 4
     LINE 5
     ]&RE%,FI$,(1),A$(0)      | Try to read 5 lines from record 1.
     ]?RE%:?A$(0)
     5                        | Error 5-OUT OF DATA!
     3                        | However, 3 strings were read.
     ]FOR X=1 TO 5:?A$(X):NEXT
     RECORD 1 LINE 1 OF 3
     RECORD 1 LINE 2 OF 3
     RECORD 1 LINE 3 OF 3
     LINE 4                   | Unchanged from previous read.
     LINE 5                   | Unchanged from previous read.
```

The user may intermix simple strings, single array elements, and single dimensional array reads on one statement, and may extend the parameter list as long as necessary. It is important to note that when the end of the record or file is encountered, RE% will be set to 5-OUT OF DATA, and the unused portion of the parameter list will be ignored.

Operational Notes

The Amper Read routine searches the DOS I/O buffers looking for the file name specified in the string given; if not found, RE% will be set to 6-FILE NOT FOUND and control returned to BASIC. If the file buffer is found, the routine will then look for the appearance of an open parenthesis '(' to signify Record and/or Byte parameter specification. If not found, file read will proceed from the current position within the file. If Record and Byte parameters were specified, the routine will look for variable names beginning with 'R' or 'B' to identify parameters. If constants or variables not beginning with 'R' or 'B' are found, then parameters are taken to be in record, byte order. String variables are then processed one at a time from the parameter list. If end of record or end of file marker ($00) is encountered in reading the disk file, RE% will be set to 5-OUT OF DATA and the rest of the parameter list will be ignored and control returned to BASIC. If a disk I/O error occurs, RE% will be set to 8-DISK I/O ERROR and control returned to BASIC.

Programming Notes

Amper Read allows inputting of commas and colons into the string variable. Carriage returns <CR> or nulls <$00> terminate string input.
The variable list to be read in may be extended indefinitely as needed. If 'End of Data' or 'End of File' is encountered before list is exhausted, RE% will be set equal to 5 and the remainder of the list will be ignored and their contents unchanged.
If file name has not been previously OPened or CReated, then RE% will be set to 6 - File Not Found.
String expressions are not allowed for specification of file name.
Variable RE% may be used as a conditional flag (e.g., IF RE% THEN error occurred). Location 222 ($DE) will contain the error number as well as the variable. This gives compatibility with ONERR GOTO error handling routines.

Programming Example

See Amper I/O programming examples section that follows AMPER CLOSE.

AMPER WRITE

```
FORMAT:  &WR%,file$[,(record,byte)],variable$[,variable$,...]

WHERE:  WR% - Returned status value.
              if =0 = no error occurred.
                 >0 = DOS error #.

       file$ - String containing file name to which you are writing.

      record - Optional record number.
        byte - Optional byte parameter.
```

Record, byte Notes

If record or byte parameters are specified, they must be inside parentheses '(...)' to be recognized as such. Constants or variables may be used in parameter specification. If constants are used, the order is Record then Byte. If variables are used, the first letter of the variable may be used to identify parameter, e.g., if B=5, then &WR%,FILE$,(B),INFO$ would write to byte 5 of the current record into variable INFO$. If constants and variables are intermixed, variable name takes precedence. If variable name does not begin with an R or a B then constant order is assumed.

variable$ - String variable containing text data to be stored on disk. If single dimensional string array variable element zero (0) was specified, then the routine will attempt to write data from elements 1 thru array dimension limit if variable$(0)="" or elements 1 thru VAL[variable$(0)], e.g., if DIM INFO$(20), &WR%,FILE$,INFO$(0) and INFO$(0)="10", then INFO$(1) thru INFO$(10) will be written to the file.

Description

Amper Write allows you to output STRING variables to the specified open disk file. The routine supports the automatic output of single dimensional string arrays. The status variable after each operation allows more precise control over the handling of error conditions than available through ONERR GOTO constructions.

It is strongly recommended that the user be totally familiar with Apple DOS file structure and how to write both sequential and random access files to completely understand the operation and use of this routine. The user should review Chapter 6: Using Sequential Files* and Chapter 8: Using Random-Access Files# before attempting the following examples.

* See pages 47-71 of THE DOS MANUAL.
# See pages 81-89 of THE DOS MANUAL.

Example: Hands-on use of Amper Write sequential.

```
     ]BRUN AMPER I/O             | Load module into BASIC.
     ]FI$="NEW FILE"             | Assign string value with file name.
     ]&CR%,FI$                   | Create output file.
     ]?CR%                       | Check success of create.
    0                            | If not 0, check error.
```

String expressions are not allowed for specifying output to file, so we have to assign the data to string variables first.

```
     ]A$="STRING 1"
     ]B$="STRING 2"
     ]C$="STRING 3"
```

Now we may write our data to the file.

```
     ]&WR%,FI$,A$,B$,C$          | Write data.
     ]?WR%                       | Check status of write.
    0                            | Should be 0.
```

To output an array, we can specify the entire array or just a portion of the array. When element zero (0) of a single dimensional string array is specified in the output parameter list, Amper Write will examine the content of that element to determine how much of the array to send to the disk file. If the string is null, elements 1 through the array dimension are written to the disk.

```
     ]DIM A$(3)                  | Dimension string array.
     ]A$(1)="STRING 1"           | Assign string values.
     ]A$(2)="STRING 2"
     ]A$(3)="STRING 3"
     ]&WR%,FI$,A$(0)             | A$(0)="" by default.
     ]?WR%                       | A$(1) thru A$(3)
    0                            |   have been written to disk.
```

If string array element zero has a numeric value (e.g. "10"), the routine will then write elements 1 through the numeric value to the disk file.

```
     ]A$(0)="2"
     ]&WR%,FI$,A$(0):?WR%        | A$(1) and A$(2) will be
     0                           |   written to disk.
     ]A$(0)="10"                 | Try something funny.
     ]&WR%,FI$,A$(0):?WR%        | Remember DIM A$(3).
     107                         | Error 107=BAD SUBSCRIPT.
```

If the numeric value of element zero exceeds the array dimension, the routine will output elements 1 thru the dimension limit, and then return 107-BAD SUBSCRIPT error when attempting to output the next (nonexistant) array element.

```
     ]A$(0)="JUNK"               | Assign non-numeric.
     ]&WR%,FI$,A$(0):?WR%        | Try array write.
     53                          | Error 53-ILLEGAL QUANTITY.
```

If element zero is non-numeric, error 53-ILLEGAL QUANTITY will be returned without writing to the disk file.

Example: Hands-on use of Amper Write random access.

```
    ]&CR%,FI$,(100):?OP%        | Recreate file with record length 100*.
    0
    ]&WR%,FI$,(1),A$,B$,C$      | Write to record 1.
    ]?WR%
    0
    ]A$(0)=""                   | Make A$(0) legal.
    ]&WR%,FI$,(2),A$,A$(0),B$:?WR%
    0                           | Write to record 2.
    ]&CL%,FI$                   | Files must be closed to assure
    ]?CL%                       |    all data is saved.
    0
```

The user may intermix simple strings, single array elements, and single dimensional array reads on one statement, and may extend the parameter list as long as necessary.

Operational Notes

The Amper Write routine searches the DOS I/O buffers looking for the file name specified in the string given; if not found, WR% will be set to 6-FILE NOT FOUND and control returned to BASIC. If the file buffer is found, the routine will then look for the appearance of an open parenthesis '(' to signify Record and/or Byte parameter specification. If not found, file write will proceed from the current position within the file. If Record and Byte parameters were specified, the output will be positioned within the file as directed. String variables are then processed one at a time from the parameter list. At the end of each string, a carriage return is added to the file to separate strings. If any error conditions occur, the status variable WR% will be set appropriately, the rest of the parameter list (if any) will be ignored and control will return to BASIC.

Programming Notes

Amper Write allows outputting of commas and colons from the string variables and a carriage return <CR> follows the output of each string variable.
The variable list to be written may be extended indefinitely as needed.
If file name has not been previously OPened or CReated, then WR% will be set to 6 - File Not Found.
String expressions are not allowed as file name or output parameters.
Variable WR% may be used as a conditional flag (e.g., IF WR% THEN error occurred). Location 222 ($DE) will contain error number.

Programming Example

See Amper I/O programming examples section that follows AMPER CLOSE.

* We could have opened the file with &OP%, but &CR% clears file.

AMPER CLOSE

```
FORMAT:  &CL%,file$

 WHERE:  CL% - Returned status variable.
               if =0 = no errors occurred.
                  >0 = DOS error #.

        file$ - String variable which contains file name to close.
```

Description

This routine terminates disk file I/O activity. Since the file data is supplied to the routine in string variable form, it does not have to go through lengthy parsing operations within the DOS File Manager. This results in faster operation than equivalent DOS commands. The status variable after operation gives you, the programmer, more precise control over the handling of error conditions than available through ONERR GOTO constructions.

Operational Notes

If file name is open for REad operations, file buffer is released to DOS. If file name was open for WRite operations, buffers are written to disk file, catalog information is updated, and file buffer is released to DOS.

Programming Notes

File name must be specified for proper syntax; mass closures are not allowed.
If file is not open, CL% will be set to 6-FILE NOT FOUND.
Variable CL% may be used as a conditional flag (e.g., IF CL% THEN error occurred). Location 222 ($DE) will contain the error value as well as the variable CL%. This is for compatibility with ONERR GOTO error handling routines.

Programming Example

See Amper I/O programming examples section that follows.

AMPER I/O EXAMPLES

EXAMPLE: Serial Read and Write

FUNCTION: Copy a sequential file.

Description

This program will read one sequential disk file, and copy its contents into another sequential file. Enter 0 to specify current drive and slot. Note that drive and slot specifiers are reversed from the default order and that the first letter of the variable name is used to identify the parameters.

```
 10 TEXT:NORMAL:HOME
 15 PRINT CHR$(4);"BRUN AMPER I/O"
 20 INPUT " INPUT FILE NAME:";AFILE$
 25 INPUT "              SLOT:";SA
 30 INPUT "             DRIVE:";DA
 35 PRINT
 40 INPUT "OUTPUT FILE NAME:";BFILE$
 45 INPUT "              SLOT:";SB
 50 INPUT "             DRIVE:";DB
 55 PRINT
 60 &OP%,AFILE$,SA,DA:IF OP% THEN 200
 65 &CR%,BFILE$,SB,DB:IF CR% THEN 200
 70 &RE%,AFILE$,INFO$
 75 IF RE%=5 THEN 100
 80 IF RE% THEN 200
 85 &WR,BFILE$,INFO$:IF WR% THEN 200
 90 GOTO 70

100 &CL%,AFILE$:IF CL% THEN 200
105 &CL%,BFILE$:IF CL% THEN 200
110 PRINT "-- DONE --"
115 END

200 PRINT
205 PRINT "--DOS ERROR--";OP%;"-";CR%;"-";RE%;"-";WR%;"-";CL%;"--"
210 END
```

AMPER I/O EXAMPLES

EXAMPLE: Amper Serial Read and Random Write.

FUNCTION: Read sequential file and convert to random access.

Description

This program scans a specified sequential disk file to find the largest string and then creates a random access file using the largest string size as the record length. Note that record size is MAX+1 to allow for the carriage return added to the end of each string when output.

```
 10 TEXT:NORMAL:HOME
 20 PRINT CHR$(4);"BRUN AMPER I/O"
 30 INPUT " INPUT:";AFILE$
 40 INPUT "OUTPUT:";BFILE$
 50 &OP%,AFILE$:IF OP% THEN 300
 60 MAX=0
 65 REM SCAN FILE FINDING LARGEST LINE
 70 &RE%,AFILE$,INFO$
 80 IF RE%=5 THEN 120
 90 IF RE% THEN 300
100 IF LEN(INFO$)>MAX THEN MAX=LEN(INFO$)
110 GOTO 70
115 REM CREATE NEW FILE USING RECORD LENGTH
117 REM OF LONGEST STRING IN INPUT FILE
120 &CR%,BFILE$,(MAX+1):IF CR% THEN 300
130 L$=STR$(MAX+1)
135 REM SAVE RECORD LENGTH IN RECORD 0
140 &WR%,BFILE$,(0),L$:IF WR% THEN 300
150 R=0:REM CLEAR RECORD COUNT
155 REM REWIND INPUT FILE
160 &OP%,AFILE$:IF OP% THEN 300
170 &RE%,AFILE$,INFO$
180 IF RE%=5 THEN 230
190 IF RE% THEN 300
200 R=R+1
210 &WR%,BFILE$,(R),INFO$
220 GOTO 170
230 &CL%,AFILE$:IF CL% THEN 300
240 &CL%,BFILE$:IF CL% THEN 300
250 PRINT
260 PRINT "-- DONE --"
270 END

300 PRINT
310 PRINT "-- DOS ERROR --";OP%;"-";CR%;"-";RE%;"-";WR%;"-";CL%;"--"
320 END
```

AMPER INSTRING

```
FORMAT: &IS%,var1$,var2$,s

 WHERE: IS% - Used to invoke the function only.

      var1$ - The string to be placed into target string.
              Note: May be any legal string expression.

      var2$ - The target string.

          s - Required start position at which placement begins.
```

Description

Amper Instring will place one string into another at a specified position and will not generate any garbage strings in the process. This is useful for formatting output for a printer.

Operational Notes

The routine locates the target string and places the source text directly into the target string. It does not create new strings.

Programming Notes

WARNING: APPLESOFT creates string pointers that point into the program text! This means that if the target string specified was created: A$="DOG", the program text in memory will be changed. The way to avoid this would be: A$=MID$("DOG",1), forcing the string to be stored in high memory.
When the replacement string is larger than the target string, only the characters that can fit into the target string will be substituted.

Programming Example

```
  5 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 10 A$=MID$("THIS IS A ---- STRING",1)
 20 B$="TEST"
 30 &IS%,B$,A$,11
 40 PRINT A$
 50 END

]RUN

THIS IS A TEST STRING
```

AMPER KEYboard INput ?

```
FORMAT:  &KE%,type,variable$,length[;]

 WHERE:  KE% - Returned status value.
               1 = Operation complete.
               2 = Null string implied.
               3 = Field backspace requested.
               4 = CTRL-C keyed.

        type - String type selection.

               ABS(type) - 1 = alphanumeric string.
                           2 = alpha string.
                           3 = zero fill numeric.

               SGN(type) + CR required.
                         - length terminates.

   variable$ - String variable where input is stored.

      length - Length of string.

           ; - Optional carriage return suppression.
```

Description

Amper Keyin is designed to control what the user types in response to requested input. Amper Keyin replaces the INPUT statement in certain circumstances. Those circumstances are when the programmer wishes to restrict the user from entering too many characters, non-alphabetic characters, or non-numeric characters. Amper Keyin does not allow the user to move the cursor around the screen (via ESC key sequences) nor terminate program execution at input time (via CTRL-C). Amper Keyin allows the programmer to easily construct input screens where the user can move back and forth on the screen keying in responses. The programmer can also specify inputs that are terminated by number of characters entered rather than by a carriage return.

It is recommended to read the 'Operational Notes' before attempting the following examples.

Example: Hands-on use of Amper Keyin.

```
    ]BRUN AMPER KEYIN       ¦ Load module into BASIC.
    ]&KE%,1,A$,5            ¦ Input 5 characters to A$.
    _____                   ¦ 5 underscores appear as prompt.
```

You can now enter 5 characters. The ESC keys for cursor movement do not work nor are control characters* allowed. Note that when you try

* Except CTRL-A, CTRL-B, CTRL-C, and CTRL-X, which are special. (CTRL-H and CTRL-U are the arrow keys '<--' and '-->'.)

to enter a sixth character, the cursor does not move and you get a bell. If you press RETURN, control will return to BASIC.

```
     ]?KE%                        | Print status variable.
     1                            | If not 1, a special key was pressed.
     ]&KE%,-1,A$,5                | Try again with length terminating.
```

This time, after 5 characters were entered, control returned to BASIC. The '1' or '-1' in the above examples specified that alphanumeric (any printable character) input was requested. A positive '1' specified that a carriage return was necessary to terminate input, and a negative '1' specified that the length alone would terminate input.

```
     ]&KE%,3,A$,5                 | Try inputting zero fill numerics.
     00000                        | 5 zeroes and the cursor is on right.
```

This time only numbers are allowed and they scroll from right to left as on an adding machine. Decimal points are not allowed, as on an adding machine. After typing a few digits, type CTRL-X. The number is cleared to zeroes again and control has not been returned to BASIC. Type '123' and then RETURN.

```
     ]?KE%                        | Print status variable.
     1                            | 1=operation complete.
     ]?A$                         | Print input variable.
     00123                        | Note leading zeroes fill string.
     ]&KE%,2,A$,5                 | Input alpha only string.
```

Now, only alphabetic (A-Z) characters are allowed. Type CTRL-X and two letters. Now type CTRL-A. You get a bell, telling you it is not allowed. Type CTRL-X, and now try CTRL-A. This time it was allowed.

```
     ]?KE%:?A$                    | Print status and input variables.
     1                            | 1=operation complete.
     00123                        | No change from previous value.
```

In that example, we "accepted" the previous value of the input variable. Since we were at the beginning of the field and the string length and requested length matched, it was a legal operation. Note however, that the 'type' of input was not validated. This method of operation allows us to create input screens where the user may move ahead without having to re-input duplicate data. The user can signal to the program that a backward movement is requested by typing CTRL-B to an input request.

```
     ]&KE%,1,A$,10                 | Prompt for 10 characters.
```

This time try CTRL-A. It is not allowed since the length of the requested string is not the same as the current length of the string. Type CTRL-B.

```
     ]?KE%:?A$                    | Print status and input variables.
     3                            | 3-field backspace requested.
     00123                        | Unchanged from previous value.
```

As you can see, the previous value of the string is untouched, even though a longer string was requested*. The status variable however, shows that the user wants to back up an input field. It is up to you, the programmer, to decide in program logic whether that is a valid request or not. The same action occurs when the user types CTRL-C, except KE%=4. Amper Keyin does nothing other than to inform the program of the user's wishes. It is the program's (and programmer's) responsibility to honor or ignore the request. See the example program for this in action.

```
      ]&KE%,1,A$,10                ¦ Request 10 characters.
```

Type 'KEYIN' and press RETURN.

```
      ]?KE%:?">";A$;"<":?LEN(A$)
      1                            ¦ 1-operation complete.
      >KEYIN    <                  ¦ The string is padded with spaces.
      10                           ¦ The requested length.
```

All strings are returned the requested length, padded at the right with spaces. If zero fill numeric requested, the left is padded with zeroes. This is for the situations where we are dealing with fixed length information. The excess spaces can be removed by using Amper Trim, which is faster and easier than trying to append spaces to ends of random sized inputs.

```
      ]&KE%,1,A$,10                ¦ Request 10 characters.
```

Press RETURN in response to the prompt.

```
      ]?KE%:?">";A$;"<":?LEN(A$)
      2                            ¦ 2-null string IMPLIED.
      >         <                  ¦ The string is all spaces.
      10                           ¦ The requested length.
```

Once again, the response is padded to the requested length, but the status variable lets us know that RETURN was pressed without the entry of any characters. This allows us to deal with situations where null field entries may or may not be allowed.

The optional semi-colon at the end of the command is to suppress the carriage return that is printed after each input field. When putting prompts at the end of the screen, the carriage return will cause the text to scroll. The semi-colon option is to avoid such action. The carriage returns are printed at the end of inputs so that DOS statements following will be recognized, so be aware that using the semi-colon option with this command has the same effect upon DOS commands as a PRINT statement ending in semi-colon.

Operational Notes

When Amper keyin for alphanumeric or alpha only is requested, the prompt will be: Underscore characters for the length of the string. For zero fill numerics, zeros will be printed and numeric input will roll in from right to left.

* Nothing really significant about that, but that's how it works.

The user has several options while inputting:

1) Type the requested input.
2) Type RETURN immediately.
3) Use the previous value of entry.
4) Request to back space a field.
5) Restart entry.
6) Abort entry.

1) When the user keys a response, the routine checks the keystrokes for proper range before accepting into string. Invalid input rings bell. RETURN terminates input. If length termination option is used, input ends when character count matches length parameter, otherwise RETURN is required to continue. Strings are returned padded to requested length, and KE%=1.

2) If character count is zero when RETURN is keyed, strings are returned padded (with spaces) to requested length, and KE%=2 to indicate null string was implied.

3) If user keys CTRL-A and length of previous string value matches requested string, old string is 'accepted' as returned value and KE%=1. If lengths do not match or CTRL-A is not the first character typed, BELL will sound and keystroke will be ignored.

4) CTRL-B displays the old string value which is unchanged since the user is requesting to backspace to a previous input field. KE%=3 and it is the programmer's responsibility to honor or ignore this request to back up.

5) CTRL-X restores initial prompt and input is reset. This response will not affect old string value or terminate input sequence.

6) CTRL-C displays the old string value and is unchanged as in CTRL-B response and returns KE%=4. The programmer can handle or ignore this response. ONERR GOTO will not respond to this input and user can do local error handling.

Programming Notes

This routine is designed for controlled user input of single or multiple fields. The recommended program structure is to input into a string array. String parameters with matching VTAB and HTAB values should be kept in arrays. Input is then initiated by executing VTAB and HTAB to starting screen position. This is followed by Amper keyin function call with parameter values and string elements specified which is then followed by an ON KE% GOTO x,y,z,t statement.

For more complex entry structures, a preliminary ON KE% GOTO statement can trap CTRL-B and CTRL-C responses, followed by an ON...GOTO statement for individual array entry validation routines. These routines could use another ON KE% GOTO to check for null entries and then validate string input for proper response to continue or re-issue input request.

Programming Example

This example program is a rudimentary mailing list data entry program.

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 1"
 15 D$=CHR$(4):REM CTRL-D
 20 DIM V(8),H(8),T(8),A$(8),L(8)
 30 FOR X=1 TO 8
 40 READ V(X),H(X),T(X),L(X):REM LOAD TABLES
 50 NEXT X
 60 PRINT D$;"OPEN NAME AND ADDRESS FILE,L73"
100 HOME:REM DISPLAY TITLES
110 PRINT "NAME AND ADDRESS ENTRY"
120 PRINT
130 PRINT "   NAME:"
140 PRINT "ADDRESS:"
150 PRINT "   CITY:"
160 PRINT "  STATE:      ZIP:"
170 PRINT "  PHONE: (   )    -"
180 PRINT
200 X=1
210 VTAB V(X):HTAB H(X):REM MOVE TO SCREEN POSITION
220 &KE%,T(X),A$(X),L(X):REM INPUT STRING
230 ON KE% GOTO 300,210,240,500
240 X=X-1:IF X=0 THEN 200:REM BACKUP
250 GOTO 210
300 X=X+1:IF X<=8 THEN 210:REM CONTINUE TO NEXT
310 VTAB 9:HTAB 1:REM CHECK ENTRY
320 PRINT "ABOVE CORRECT?";
330 &KE%,-2,A$,1
340 ON KE% GOTO 350,330,240,330
350 IF A$="Y" THEN 370
360 GOTO 200
370 PRINT D$;"WRITE NAME AND ADDRESS FILE,R";R
380 FOR X=1 TO 8
390 PRINT A$(X);:REM CONCATENATE FIELDS
400 NEXT X
410 PRINT
420 PRINT D$
430 R=R+1:REM INCREMENT RECORD COUNT
440 GOTO 200
500 PRINT D$;"WRITE NAME AND ADDRESS FILE,R";R
502 PRINT:REM TERMINATE FILE WITH NULL
505 PRINT D$;"CLOSE NAME AND ADDRESS FILE"
510 END
600 DATA 3,9,1,20 :REM NAME
610 DATA 4,9,1,20 :REM ADDRESS
620 DATA 5,9,1,15 :REM CITY
630 DATA 6,9,-2,2 :REM STATE
640 DATA 6,17,-3,5 :REM ZIP
650 DATA 7,11,-3,3 :REM AREA CODE
660 DATA 7,16,-3,3 :REM PREFIX
670 DATA 7,20,-3,4 :REM SUFFIX
```

AMPER STRING LEGAL

FORMAT: &SL%,var$

WHERE: SL% - Used to invoke Amper String Legal.

var$ - string to be legalized.

Description

This routine converts any unprintable (and therefore invisible) characters to legal (visible) characters. This is useful in checking input responses for proper results. This routine can also make control characters visible from catalog file names when used with Amper Catalog.

Example: Hands-on use of Amper String Legal.

```
]BRUN AMPER STRING LEGAL | Install module in BASIC.
]A$="ABCD"               | Assign string variable.
]&SL%,A$                 | Legalize string.
]?A$
ABCD                     | No change, character string legal.

]A$=A$+CHR$(9)+A$        | Add invisible character.
]?A$                     | Print string.
ABCDABCD                 | 8 visible characters.
]?LEN(A$)                | Print length of string.
9                        | String really has 9 characters.
]&SL%,A$                 | Legalize string.
]?A$                     | Print legalized string.
ABCDIABCD                | CHR$(9) (CTRL-I) changed to 'I'.
```

Operational Notes

If during the scan of the supplied string the routine encounters any illegal ASCII codes it will convert them to their legal counterpart. A control-G will be converted to a normal G.

Programming Notes

Amper String Legal does not translate lowercase characters to uppercase characters.
The value SL% is used only to invoke the amper routine. It may be used as a user variable, but it is not recommended.

AMPER TRIM ?

FORMAT: &TR%,var$

WHERE: TR% - Used to invoke Amper Trim.

var$ - string to be trimmed.

Description

Amper Trim is used to remove trailing spaces from the end of string variables. This is useful for trimming output to disk files and printers. This routine works very well in conjunction with Amper Catalog and Amper Keyin to reduce input strings to working minimum.

Example: Hands-on use of Amper Trim.

```
]BRUN AMPER TRIM            | Install module into BASIC.
]A$="TEST STRING        "   | Assign string value.
]&TR%,A$                    | Trim string.
]?A$;"<"                    | Print string.
TEST STRING<                | '<' printed to show end of string.

]A$="          "            | Assign string all spaces.
]&TR%,A$                    | Trim string.
]?">";A$;"<"                | Print string.
><                          | Trimmed to null string.
```

Operational Notes

Amper Trim will shorten a string to its smallest possible size by trimming off any trailing blanks that may exist. The string is shortened by modifying the length parameter of the variable, this method does not generate garbage strings*. The string supplied may be either simple or array in type.

Programming Notes

The variable TR% is used to invoke the Amper Trim routine, but is not used otherwise. It may be used as a program variable, but the practice is not recommended.

* Strings that are inactive and need to be 'garbage collected'.

AMPER WORD

FORMAT: &WD%,var1,var2

WHERE: WD% - Used to invoke Amper Word routine.

var1 - location to peek.

var2 - variable into which value will be placed.

Description

This routine is the 16-bit equivalent of BASIC's 8-bit PEEK command. Many times it is advantageous to know the value in a 16-bit pointer made up of two 8-bit bytes. Two 8-bit bytes used together as a unit are generally referred to as a 'word'. In the 6502 microprocessor architecture, the lower 8 bits are stored first in memory, followed by the high order 8 bits.

This routine is a machine language replacement for the basic code:

```
A = PEEK (B) + PEEK (B+1) * 256
```

The equivalent Amper Word command is:

```
&WD%,B,A
```

This routine is excellent for peeking out pointers for Applesoft or DOS.

Operational Notes

The contents of the specified location and the next location are combined into a 16-bit word and then converted to a floating point number.

Programming Notes

The address range specified must be 0-65535 ($0000-$FFFF).
Extreme caution must be used in the 49152-53247 ($C000-$CFFF) range since this is the control, I/O, and peripherial card region!
Please note that in some instances APPLESOFT stores its values in high, low order rather than low, high order.
The WD% variable is only used to invoke the Amper Word routine and may be used as a program variable, however, the practice is not recommended.

Use with the RELOCATING LINKING LOADER (tm)

The Language Plus+ amper routines are provided in relocatable R file format so that they may be used in conjunction with Micro Lab's RELOCATING LINKING LOADER.

The RELOCATING LINKING LOADER allows the modules to be custom combined into binary BRUNable B files. You can combine routines from other Language Plus+ Volumes as they become available.

The RELOCATING LINKING LOADER allows routines to be organized and positioned in memory as per the user's needs.

Experienced Assembly Language and APPLESOFT programmers may write their own custom amper routines and link them into our routines.

You do not need any experience with Assembly Language to use the RELOCATING LINKING LOADER with the Language Plus+ modules.

Routine Groupings

The following table describes the combination of Language Plus+ modules in each binary B file on the diskette.

File name	CA	DA	FX	FR	GC	HE	IS	IO	KE	SL	TR	WD
Language Plus Volume 1	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
Directory Group	xx		xx			xx	xx	xx		xx	xx	
Garbage Group				xx	xx							
Entry Group			xx	xx	xx	xx	xx	xx	xx		xx	
Internals Group		xx	xx	xx								xx

The Directory Group is designed for use in program applications where disk directories are read and stored in files.

The Garbage Group is for applications that will involve massive amounts of string manipulations.

The Entry Group is for data entry programs.

The Internals Group is for programs that require manipulation of arrays and program pointers.

CONCLUSION

We hope that the users of these amper routines will find as much utility from the routines as we at Micro Lab have. These routines were initially written to provide high-speed and high-level approaches to several programming problems we encountered in writing commercial grade APPLESOFT BASIC programs. These problems were in the areas of user data entry, array content searching, sorting, and disk I/O error control. The use of the Amper vector within APPLESOFT with calls in APPLESOFT itself, afforded many capabilities in the design of these extra commands.

As we wrote these routines, we developed a style that eased the development of other routines. To commercially release our library of routines, we reworked the structure of older amper routines up to the standard that we had evolved.

This 'Standard' is the use of two letter INTEGER variable names for the invocation of the separate amper routines. We did this to allow for more than 26 routines with unique names, and to allow the invocation variable to be used as a returned status variable.

If we had used 'named' routines like '&READ,FILE$,INFO$', rather than '&RE%,FILE$,INFO$', it would have required an extra variable to return the outcome of the operation, and it would have required two separate parsing routines to identify the function. One parsing routine would have been required to identify the token or character names and another to identify variables to be used. The second one already exists in APPLESOFT and is faster than searching through multi-letter operation names.

Using variable names rather than words, allows the routines to be smaller and faster. The user then has only a mild inconvenience of dealing with arbitrary two letter routine invocation names when using LANGUAGE PLUS+. Also, APPLESOFT will never have problems when parsing the line containing the call to the amper routine because of reserved words within 'name' of the routine. We found this method to be the least troublesome while trying to co-exist with APPLESOFT.

Since these routines have evolved over a period of time, some are more sophisticated than others. In the future we plan to add to our library of routines, and we will issue revisions to some of these routines as they develop.

Mike Hatlak Curt Rostenbach