micro lab

Presents:

+LANGUAGE PLUS+

Volume Two
Sorting & Searching

written by:

Michael J. Hatlak
and
Curt Rostenbach

A collection of 14 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.

Data convert	Deallocate arrays
ONERR GOTO Fix	Fast Free Memory
Fast Garbage Collection	Input anything
Instring replacement	Invert arrays
Array searching	Quicksort
Substring searching	Variable swapping
Array compaction	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 2"

```
==========================================
!! 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 COMPACT

FORMAT: &CM%,array(n)[,s,e]

WHERE: CM% - Used to invoke routine.

array - Any type array to be compacted.

s - Optional starting position within the array.

e - Optional ending position within the array.

Description

Amper Compact will take all valid data within an array and move it to the front of the array. Valid data is that data which is non-zero in type REAL or INTEGER and non-zero length strings in type STRING.

Example: Hands-on use of Amper Compact

```
]BRUN AMPER COMPACT       | Load module into BASIC.
]DIM ARRAY(5)             | Dimension array.
]ARRAY(1)=0               | Assign some values to array elements.
]ARRAY(2)=1
]ARRAY(3)=0
]ARRAY(4)=2
]ARRAY(5)=3
]&CM%,ARRAY(0)            | Compact array.
```

It is important to note that even though array element zero was specified, it will be unaffected. Proper use of Amper Compact requires a reference to any legal array element, however, the specific element is irrelevant and only used to identify the variable array. Element zero (0) of any array is always a legal reference regardless of the array dimension and is therefore recommended when specifying the array to compact. Element zero is generally ignored or used to contain special information in normal programming practice, therefore, unless specified with the optional array start parameter, it will not be used in the compact operation.

```
]FOR X=0 TO 5:?ARRAY(X):NEXT
0                         | Element 0 not used.
1
2
3
0
0
```

You may compact portions of the array by specifying start and end positions.

```
    ]&CM%,ARRAY(0),0,2      | Compact elements 0 thru 2.
    ]FOR X=0 TO 5:?ARRAY(X):NEXT
     1
     2
     0
     3
     0
     0
```

If the end position is omitted, the end of the array is used, however, the starting position cannot be omitted if supplying an ending position.

Operational Notes

The Amper Compact routine asks APPLESOFT to locate the array element specified and the array definition block. Information from the array variable definition block is used to determine the type and dimension of the array. Optional parameters are then looked for and if found, are used to replace the default values for array start and end. The array is then compacted based on variable type within the limits specified or implied. String arrays are compacted by manipulation of their pointer values so no garbage strings are created.

Programming Notes

The routine works with single dimensional REAL, STRING, or INTEGER type arrays.
The variable array to be compacted must be specified by an array element reference. The particular element is of no consequence as long as it is a legal reference (i.e. within array bounds). Use of element zero (0) of the array variable is recommended since it is ALWAYS a legal reference.
Array element zero is not affected unless specified, this allows it to be used for special purposes, such as non-zero or not-null element count.
The variable CM% is used to invoke the Amper Compact routine, but is not used otherwise. It may be used as a program variable, but the practice is not recommended.

AMPER DATE CONVERT

FORMAT: &DC%,variable$

WHERE: DC% - Used to invoke the routine.

variable$ - string to be converted.

Description

This routine converts date strings to a form suitable for sorting and searching. Date strings are usually stored in the format "MM/DD/YY". This format is not appropriate for anything other than printouts. Amper Date Convert transposes the characters of the above date format to the form "YY/MM/DD". The order of year, month, and day allows simple string comparisons to determine date order. The routine moves the characters around within the string, so no garbage strings are created during transformation.

Operational Notes

Amper Date Convert will not execute unless the supplied string is 8 characters in length! If the routine does not execute, it returns the string untouched. It works on simple strings as well as string array elements.

Programming Notes

String supplied must be eight (8) characters long; otherwise string will not be converted.
Conversion is from "MM/DD/YY" to "YY/MM/DD". Any eight character string may have characters transposed, the string content is NOT tested for "xx/yy/zz" format. If given "ABCDEFGH", routine will return "GHFABCDE".
After converting date string, executing the date convert function two (2) more times on the string will return the original DATE string.
The variable DC% is used to invoke the routine only. It may be used as a program variable, however, the practice is not recommended.

Programming Example

```
 5 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 2"
10 INPUT"FIRST DATE";A$
20 INPUT"SECOND DATE";B$
30 &DC%,A$
40 &DC%,B$
50 IF A$>B$ THEN PRINT "FIRST DATE IS LARGER.":END
60 PRINT "SECOND DATE IS LARGER."
70 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$(4);"BRUN AMPER ROUTINES VOLUME 2"
10 DIM A(100),A$(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 2"
 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 2"
 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$(4);"BRUN AMPER ROUTINES VOLUME 2"
 10 REM CREATE SIMPLE VARIABLES
 20 A=0:B=0:X=0
 30 REM CREATE LARGE STRING ARRAY
 40 REM FORCING LONG DURATION GARBAGE COLLECTION
 50 DIM A$(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$(4);"BRUN AMPER ROUTINES VOLUME 2"
20 FOR X=1 TO 1000
30 FOR Y=1 TO 1000
40 A$="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 INPUT ANYTHING

FORMAT: &IN%,var$

WHERE: IN% - Used to invoke amper routine.

var$ - String variable into which input will go.

Description

This routine allows the entry of commas (',') and colons (':') into string variables. The APPLESOFT BASIC command, INPUT, treats commas and colons as data separators to mark entry into different variables. There are many instances that a comma and colon are desired as part of the data to be stored in a string, such as addresses and titles, e.g.. "DAVENPORT, IA", "JOHN DOE, PHD" or ratios and times, e.g., "1:1000", "12:45:30 AM". The Amper Input routine allows data of these types to be entered by substituting a PRINT and &IN% for the standard INPUT statement.

Example: Coding differences between INPUT and Amper Input.

```
100 INPUT "ENTER DATA: ";A$
```

The equivalent Amper Input coding would be:

```
100 PRINT "ENTER DATA: ";:&IN%,A$
```

Amper Input may be used in the place of any APPLESOFT BASIC INPUT statements.

Operational Notes

This routine by-passes the normal APPLESOFT input routine so that commas and colons do not cause string termination with the annoying EXTRA IGNORED missive.

Programming Notes

The Amper Input routine may be used in immediate mode, unlike APPLESOFT BASIC, however, since the keyboard buffer is used for text storage, SYNTAX ERROR will probably be returned after use. This does not occur in the deferred mode* within a program.
The variable IN% is only used to invoke the routine, and may be used for data storage, however the practice is not recommended.

* See APPLESOFT BASIC manuals for IMMEDIATE and DEFERRED modes.

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$(4);"BRUN AMPER ROUTINES VOLUME 2"
 10 A$=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 INVERT

FORMAT: &IV%,array(n)[,s,e]

WHERE: IV% - Used to invoke the routine.

array - Any type array to be inverted.

s - Optional starting position within the array.

e - Optional ending position within the array.

Description

Amper Invert will reverse the order of the contents of the specified variable array. This is most useful if you have sorted an array, and you wish to have it in descending order instead of ascending order.

Example: Hands-on use of Amper Invert.

```
]BRUN AMPER INVERT          | Load module into BASIC.
]DIM ARRAY(5)               | Dimension an array.
]FOR X=0 TO 5:ARRAY(X)=X:NEXT | Make array contents ascending.
]FOR X=0 TO 5:?ARRAY(X):NEXT | Print array contents.
0
1
2
3
4
5
]&IV%,ARRAY(0)              | Invert array.
```

It is important to note that even though array element zero was specified, it will be unaffected. Proper use of Amper Invert requires a reference to any legal array element, however, the specific element is irrelevant and only used to identify the variable array. Element zero (0) of any array is always a legal reference regardless of the array dimension and is therefore recommended when specifying the array to invert. Element zero is generally ignored or used to contain special information in normal programming practice, therefore, unless specified with the optional array start parameter, will not be used in the invert operation.

```
]FOR X=0 TO 5:?ARRAY(X):NEXT
0                           | Element 0 not affected.
5
4
3
2
1
```

You may invert portions of the array by specifying start and end positions.

```
     ]&IV%,ARRAY(0),0,2       | Invert elements 0 thru 2.
     ]FOR X=0 TO 5:?ARRAY(X):NEXT
     4
     5
     0
     3
     2
     1
```

If the end position is omitted, the array limit is used, however, the starting position cannot be omitted if an end position is specified.

Operational Notes

The Amper Invert routine asks APPLESOFT to locate the array element specified and the array definition block. Information from the array variable definition block is used to determine the type and dimension of the array. Optional parameters are then looked for and if found, are used to replace the default values for array start and end. The array is then inverted based on variable type within the limits specified or implied. String arrays are inverted by manipulation of their pointer values so no garbage strings are created.

Programming Notes

This routine works with single dimensional REAL, STRING, or INTEGER type arrays.
The variable array to be inverted must be specified by an array element reference. The particular element is of no consequence as long as it is a legal reference, i.e., within array bounds. Use of element zero (0) of the array variable is recommended since it is ALWAYS a legal reference.
Array element zero is not affected unless specified, this allows it to be used for special purposes, such as a counter.
The variable IV% is used to invoke the Amper Invert routine, but is not used otherwise. It may be used as a program variable, but the practice is not recommended.

AMPER MATCH

```
FORMAT:  &MA%,variable,array(n)

 WHERE:  MA% - Returned array index.
               -1 = value not found.
               >0 = matching element.

   variable - Variable with value to search for in array.

      array( ) - Single dimensional array to search.

            n  - Starting element for search.
```

Description

This routine searches variable arrays at high speed. An APPLESOFT BASIC loop to search an array can scan approximately 300 elements per second. Amper Match can search, on the average, 14000 elements per second!

Operational Notes

Arrays will be searched from the starting element to the end of the array for the target value. After constant overhead of .08 second, arrays elements are searched at the rate of 13000 per second for real variables, 15000 per second for integer variables, and depending on contents, 13000 to 19000 per second for strings.

Programming Notes

Arrays are one dimensional only.
Both variables must be of the same type.
Constants, other than array start index, are not allowed.

Programming Example

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 2"
 20 F=0:Y=0:DIM A(4000): REM CREATE VARIABLES
 30 PRINT "BUILDING LIST"
 40 FOR X=0 TO 4000
 50 A(X)=INT(RND(1)*1000): REM BUILD LIST OF RND NUMBERS
 60 NEXT
 70 PRINT "SEARCHING"
 80 FOR X=0 TO 999: REM FIND ALL OCCURRENCES OF X IN A(N)
 90 PRINT X;" -";:Y=0:F=0: REM INITIALIZE SEARCH
100 &MA%,X,A(Y): REM SEARCH ARRAY FOR X
110 IF MA%>-1 THEN PRINT " ";MA%;:F=1:Y=MA%+1:IF Y<4000 THEN 100
120 IF F=0 THEN PRINT " NOT FOUND";
130 PRINT:NEXT:END
```

AMPER QUICKSORT!

FORMAT: &QS%,array(n),array%(n)

```
   WHERE: QS% - Used to invoke routine.

     array(n) - Array to be sorted.
                May be REAL or STRING.

    array%(n) - Array to be co-sorted.
                Must be INTEGER.
```

NOTE: The ENTIRE array is sorted and REQUIRES array boundary values to be set up before sorting. See Programming Notes section.

Description

Amper Quicksort uses a sorting algorithm that is faster than Bubble and Shell-Metzner sorts. The routine will sort a specified real or string array. An integer array will be rearranged to follow the movement of the original array to the sorted array, this allows matching data to the array being sorted to be reordered to coincide again. This routine sorts the entire array specified. Array boundaries must be set up for proper operation.

Example: Hands-on use of Amper Quicksort.

```
     ]BRUN AMPER QUICKSORT    ¦ Load module into BASIC.
     ]DIM A(3001),A%(3001)    ¦ DIMension data and follower array.
     ]X=RND(-1)               ¦ Set random seed.
     ]FOR X=1 TO 3000:A(X)=RND(1):A%(X)=X:NEXT
                              ¦ Fill with data and order.
     ]A(0)=-1E36              ¦ Mark lower limit.
     ]A(3001)=1E36            ¦ Mark upper limit.
     ]?A(1),A%(1)             ¦ Print sample values.
      .738207502     1
     ]&QS%,A(0),A%(0)         ¦ Sort array.
```

The REAL array will be sorted into numerical order between elements 0 and 3001. The INTEGER array which initially contained the count of 1 through 3000 in elements 1 through 3000 will be rearranged to follow the sorting of the REAL array.

```
     ]?A(1),A%(1)             ¦ Print new values.
     3.97660905E-04   2671
```

The current value in A(1) used to be in element 2671 before the sort.

```
    ]?A(2211),A%(2211)          | Print old element 1*.
     .738207502           1
```

The information in the INTEGER array could be used to restore the REAL array back to its original order, but it is intended for situations where you have matching information in different arrays. For example, suppose array NA$(n) contained names, AD$(n) contained the matching addresses, CT$(n) for city, ST$(n) for state, and ZP(n) for zip codes. If any of these arrays were then sorted, the other array contents would no longer match up. The follower INTEGER array allows you to reorder the other arrays to match, or do translation of index to match. If ZP(n) was sorted with A%(n) as the co-sorted array; ZP(1) would match with NA$(A%(1)), AD$(A%(1)), CT$(A%(1)), and ST$(A%(1)).

Example: A simple sort.

```
    ]DIM ARRAY$(4),ARRAY%(4) | Create small array.
```

Now fill the arrays with data.

```
    ]ARRAY$(0)=""               | Setup lower limit.
    ]ARRAY$(1)="APPLE":ARRAY%(1)=1
    ]ARRAY$(2)="CIDER":ARRAY%(2)=2
    ]ARRAY$(3)="BAKER":ARRAY%(3)=3
    ]ARRAY$(4)=CHR$(255)        | Mark upper limit.
```

Sort ARRAY$ with ARRAY% to follow movement of elements.

```
    ]&QS%,ARRAY$(0),ARRAY%(0)
```

Print the arrays to show order and movement of elements.

```
    ]FOR X=1 TO 3:?ARRAY$(X),ARRAY%(X):NEXT
    APPLE     1
    BAKER     3
    CIDER     2
```

Operational Notes

The arrays are located in memory and the dimension of the array to be sorted is used to define the starting limits of the sort. During the sort, it is the array boundary values that determine the limits of the sort. It is absolutely IMPERATIVE that the user preset the array boundaries. As the elements are sorted, the corresponding elements of the co-sorted array are swapped element for element.

Programming Notes

REAL and STRING arrays may be sorted, the co-sorted array must be INTEGER and present.
The dimension of the INTEGER array must be EQUAL TO or GREATER THAN the dimension of the array to be sorted. If not, APPLESOFT BASIC may crash with unpredictable results.

* Since the RND seed was set, this example is rigged.

```
========================================
!! IMPORTANT !! IMPORTANT !! IMPORTANT !!
========================================
```

The array boundaries MUST be set up BEFORE the sort.

```
              REAL array
              ----------
            array(0)=-1E36
        array(last element)=1E36

             STRING array
             ------------
             array$(0)=""
     array$(last element)=CHR$(255)
```

```
========================================
!! IMPORTANT !! IMPORTANT !! IMPORTANT !!
========================================
```

The variable QS% is used to invoke the amper routine. It may be used as a program variable, however, the practice is not recommended.

Programming Example

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 2"
 20 D$=CHR$(4):REM CTRL-D
 30 PRINT D$;"OPEN TEXT FILE"
 40 PRINT D$;"READ TEXT FILE"
 50 INPUT COUNT
 60 DIM A$(COUNT+1),A%(COUNT+1)
 70 FOR X=1 TO COUNT
 80 INPUT A$(X)
 90 A%(X)=X
100 NEXT
110 PRINT D$;"CLOSE"
120 A$(0)="":A$(COUNT+1)=CHR$(255)
130 PRINT "SORTING"
140 &QS%,A$(0),A%(0)
150 PRINT "SORT COMPLETED"
160 PRINT D$;"OPEN TEXT FILE INDEX"
170 PRINT D$;"WRITE TEXT FILE INDEX"
180 PRINT COUNT
190 FOR X=1 TO COUNT
200 PRINT A%(X)
210 NEXT
220 PRINT D$;"CLOSE"
230 END
```

AMPER SUBSTRING

```
FORMAT:  &SS%,var1$,var2$[,s]

  WHERE:  SS% - Location of var1$ within var2$.  0 if not found.

        var1$ - String for which you are searching.

        var2$ - String to be searched.

            s - Optional start position within var2$ to begin
                looking for var1$.
```

Description

This routine allows you to search a character string for a specified substring. The routine will return the starting position of the substring within the searched string.

Example: Hands-on use of Amper Substring.

```
]BRUN AMPER SUBSTRING   | Load module into BASIC.
]A$="TEST"              | Assign strings.
]B$="THIS IS A TEST WITH A TEST STRING"
]&SS%,A$,B$             | Search string.
]?SS%                   | Print result variable.
11                      | Found at column 11.
```

The starting position within the searched string may be specified to find multiple occurrences.

```
]&SS%,A$,B$,SS%+1       | Search from last position found plus 1.
]?SS%
23
```

String expressions and constants may be used to specify either string parameter.

```
]&SS%,"PHRED",B$        | Look for "PHRED".
]?SS%
0                       | Not found.
```

A useful technique is to test an input string against a list of commands in a constant string. See the second example program for this technique in use.

Programming Notes

Amper Substring returns its results in the calling variable (SS%). You may perform a logical test on this variable to determine if the search was successful.

```
    IF SS% THEN <search succeeded>
```

If you wish to find all occurrences of one string within another, you could continue searching the string until SS% went to zero. Each time you are successful store SS%+1 into s so that the next search begins one character beyond where the previous search found a match. String expressions and constants may be used for parameters.

Programming Example 1

This program requests a string to find and a string to search. The substring, if found, will be highlighted in the searched string.

```
  5 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 2"
 10 INPUT"STRING TO LOOK FOR";A$
 20 INPUT"STRING TO SEARCH IN";B$
 30 &SS%,A$,B$
 40 IF NOT SS% THEN 90
 50 PRINT B$
 60 HTAB SS%
 70 PRINT A$
 80 END
 90 PRINT "STRING NOT FOUND!"
100 END
```

Programming Example 2

This program uses the Amper Substring to search a table of commands to get a branch number to take in an ON..GOTO.

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 2"
 20 GR: REM SET LORES GRAPHICS
 30 COLOR=7: REM PLOT IN WHITE
 40 X=20:Y=20: REM CENTER IN SCREEN
 50 HOME
 60 PLOT X,Y
 70 PRINT "U-UP D-DOWN R-RIGHT L-LEFT E-END: ";:GET A$:PRINT A$
 80 &SS%,A$,"UDLRE": REM FIND COMMAND
 90 ON SS% GOTO 120,140,160,180,200: REM SELECT ROUTINE
100 PRINT "-- UNKNOWN COMMAND --": REM ZERO FALLS THROUGH ON..GOTO
110 GOTO 60
120 Y=Y-1:IF Y<0 THEN Y=40
130 GOTO 60
140 Y=Y+1:IF Y>40 THEN Y=1
150 GOTO 60
160 X=X-1:IF X<0 THEN X=40
170 GOTO 60
180 X=X+1:IF X>40 THEN X=1
190 GOTO 60
200 TEXT:HOME
210 END
```

AMPER SWAP

FORMAT: &SW%,variable1,variable2

WHERE: SW% - Used to invoke routine.

variable1 - Variable to be swapped with variable2.

variable2 - Exchanged with variable1.

Description

This routine swaps the contents of the variables specified quickly without requiring additional space being used and garbage strings being created.

Operational Notes

Variables are swapped without intermediate storage and/or new string assignments. Numerics have their contents swapped. Strings have their pointers swapped; string contents are unaffected. Since only the three bytes describing the string position is swapped and the string contents are not moved, large strings are swapped as fast as small strings.

Programming notes

Variables to be swapped must be of the same type; mixed mode swapping is illegal.
The variable SW% is only used to invoke the routine. It may be used as a program variable, but the practice is not recommended.

Programming example

```
 10 PRINT CHR$(4);"BRUN AMPER ROUTINES VOLUME 2"
 20 DIM A(100)
 30 FOR X=0 TO 100
 40 A(X)=RND(1):REM CREATE RANDOM LIST
 50 NEXT X
 60 F=0
 70 FOR X=1 TO 100:REM BUBBLE SORT
 80 IF A(X-1)>A(X) THEN &SW%,A(X-1),A(X):F=1
 90 NEXT X
 95 IF F=1 THEN 60:REM SORT UNTIL ZERO
100 END
```

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	CM	DC	DA	FX	GC	IN	IS	IV	MA	QS	SS	SW	WD
Language Plus Volume 2	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
Garbage Group				xx	xx								
Internals Group			xx	xx	xx								xx
String Group				xx	xx		xx				xx		
Sort Group	xx	xx	xx	xx	xx	xx		xx		xx		xx	
Search Group				xx		xx			xx		xx		

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

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

The String Group is for routines that search and modify strings.

The Sort Group is for data input and sorting.

The Search Group is for string and array searching.

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