L.P.C.B.

Peersoft v1.5.6

An Applesoft extension

Benoît GILON

05/01/2016

Peersoft is an Applesoft extension which focuses mainly on performance issues rather than features. Currently, it is to be considered as a complement to the Bananasoft utility already released attempting, among other features, to enrich the Applesoft interpreter by adding new keywords found in other flavours of Microsoft Basic.

An introduction to Peersoft

Peersoft is an Applesoft BASIC extension which aims to provide to the Applesoft programmer features in addition to those offered by plain vanilla Applesoft interpreter alone. Currently, in its present incarnation, Peersoft runs on DOS 3.3 only but a ProDOS version could see the day in months to come if demand level is high enough.

However, three versions of Peersoft exist depending on the CPU detected within the Apple 2 host (either 6502, 65C02 or 65802/16) with relevant code optimizations for the two latter CPUs.

The focus has been put on performance aspects while authoring this software. Here are the main features delivered with this release of Peersoft:

- New instructions for defining a default variable type per name's first character:

Tableau 1: New statements for specifying a variable's default type

New statement	Type of variable	Magnitude	Suffix character
DEFINT	16bits integer	From -32768 to 32767 but see below	%
DEFSNG	Floating point (5 bytes)		!
DEFBYTE	8bits integer (byte)	From -128 to 127 but see below	.
DEFSTR	String	Length from 0 to 255	$

Therefore the program text can be smaller in size by using default typed variables.

Also for the two Applesoft integer subtypes (16 and 8bit integers), a configuration byte has been setup to induce unsigned arithmetic instead of the default signed arithmetic.

If bit b7 of byte $9CE7 (decimal 40167) is set, then 16bit integers are processed in the range of 0..65535 instead of -32768..32767. Same for 8bit integers which are processed in the range 0..255 instead of default range -128..127. This impacts all those code fragments:

1) Integer variable value returning within an expression

2) Integer variable value setting with original or new (see below) syntax schemes

3) Handling of FOR..NEXT loops with integer loop variables.

- A new syntax scheme for setting variable values based upon the result from an arithmetic operation.

  `LET <variableName> += <value>` or

  `LET <variableName> /= <value>` or

  `LET <variableName> -= <value>` or

  `LET <variableName> *= <value>`

  If specified variable is of type integer, then used arithmetic is of type integer too, also the += syntax scheme can be applied to string variables as well. The main benefit, beyond using integer arithmetic when adequate, is to minimize the number of variables references

compared to the original syntax as in the statement `LET <variableName> = <variableName> <operator> <value>` particularly when *variableName* is a multi-dimensional array with same index values.

- Before the release of Peersoft v1.5, an array element from array named `A` could only be referred to by using the expression `A(<expression>, <expression>)` presuming that the array A was defined as having two dimensions (with a `DIM` statement), this is the common way of using array variables from within the program text.
  But now with Peersoft v1.5, the same element could be referred to by using only one dimension specification. The two code extracts below:
  ```
  10 DIM A(4,5):…:S = 0: FOR I = 0 TO 4: FOR J = 0 TO 5:S += A(I,J): NEXT J,I
  10 DIM A(4,5):…:S = 0: FOR I = 0 TO 29:S += A(I): NEXT I
  ```
  Will both compute the sum of every element value into the S variable. This new way of authoring code will simplify some computations upon arrays and thus enhance performance of such computations.
  Also within the original Applesoft interpreter, there was a limitation that an expression value for giving a dimension must be less than 32768. At the time Applesoft was authored this was sensible as the smallest size element type was integer and takes 2 bytes for storage, given the hardware limitation of 64K for 8bits architectures: now this limitation was increased to 65535 thus addressing up to 65536 1 byte elements (such as bytes) within 64Kb (this is useful now that the BYTE sub integer type is added within Peersoft).
- From Peersoft v1.5.6, the loop construct for iterating thru array elements has been further enhanced with the addition of the `FOREACH` new instruction.
  ```
  DIM A(2,2):...:S = 0: FOREACH K,A:S += K: NEXT : PRINT S
  ```
  will compute the sum of array's elements. Note that the handling of the `NEXT` statement includes the update of the current array element as part of its processing. So the Applesoft statement below:
  ```
  CLEAR : DIM A(2,2): FOREACH V,A:V = RND (1): NEXT
  ```
  will initialize the FP array named “A” with random values in the range from 0 to 1.
- A new pseudo variable (“`@`”) , usable in arithmetic expressions and which replicates the value currently stored in the Applesoft floating point accumulator. For instance, the statement:
  ```
  S = 0
  FOR I = 0 TO 9: FOR J = 0 TO 9:S += A(I,J) *  @
  NEXT J,I
  ```
  will compute the sum of squared elements from matrix A. You will notice that the code only refer to every element of matrix A just once and that the * operator between a value and itself is known to be (and actually is) faster than the concurrent expression `A(I,J) ^ 2`.
- A new function statement `IIF()` is provided in order to evaluate one among two expressions based on the value returned by a Boolean expression. Here is a sample:
  ```
  MA = IIF (A > MA,A,MA): REM will compute the MAX(A,MA) and store it into the MA variable.
  ```
  This function is gonna to become a favourite time saver for author and end users alike. Unlike the version in VBA (Visual BASIC for Applications by Microsoft), this function only evaluates two of the three specified sub expressions, the unused one is skipped over by looking for either a level matching close parenthesis or comma, depending of the 1st expression result.

- Integer variables now allowed as loop variables of FOR/NEXT loops, also arithmetic operations/comparisons when running a `NEXT` statements is of integer type whenever the loop variable is of type integer itself. With the current release of Peersoft, this now includes both the original 16bits and the new byte 8bits wide integer subtypes.
- Fixes to some of the Applesoft instructions processing (`ONERR, RETURN and POP`): if you get a look at the Applesoft disassembly listing generated by S-C Documentor and commented by Bob Sander-Cederlof (http://www.txbobsc.com/scsc/scdocumentor/), then you'll find out that Applesoft has many bugs buried in its code.
- Utility routines provided to optimize access to Applesoft variables. For either simple or array variable kinds, time spent to get a reference to a variable is proportional to the number of variables of same kind (i.e. simple or array) that have been defined (i.e. referred to from BASIC program text for simple variables) before the time the current variable (looked up for) has been created. The order of definition of variables within the program flow seldom follows an order with which most used referred to variables or arrays are created first. We tend to define some "constant variables" such as `D$ = CHR$( 4)` at the very beginning albeit the fact that such variable will be much less frequently used than, for instance a loop variable from an inner loop. The goal of the provided utility routines is to **optimize** variable access **after** variables **have been created** and thus their memory storage address defined. Two alternative approaches have been supplied:
    - Physical re-organization of array/simple variables areas where actual variable slots are actually moved within the area;
    - Cache based variable referencing: here the variables storage slots are kept at the same original place within memory. Instead, Peersoft manages a small memory area to keep some variable names and addresses in a safe place. Such variables are looked up for at first place when a reference to some variable need to be returned and thus lookups for such variables are the fastest.
- Availability of co-routines within an Applesoft program: this is a new paradigm for every Applesoft program author. Now he is able to design his programs **as if** they run under a multi tasking environment. The application could be considered as an assembly of phases where the kernel is active (co-routines active and running concurrently) and where kernel is inactive (the kernel is inactive: only one flow is processed by the Applesoft interpreter). Beyond what follows, a subsequent entire chapter has been dedicated to describe its configuration and operation within this document.
    - Because the switch between co-routines will occur at known and arbitrary locations within the interpreter loop, then the context can be kept rather small compared to true multi tasking monitors.
    - Unlike true multi tasking monitors, no hardware generated external signal (IRQ) is used and thus Peersoft co-routines run equally well on the whole Apple 2 range (from the Apple II standard with Autostart ROM to Apple //gs and every emulator environment I currently know of).
- Time oriented optimization by precomputing GOTO/GOSUB target addresses within program text.
- Processing of hardware interrupts from Applesoft subprograms now allowed (with support for mouse and timer VBL interrupts already included).
- Now up to eleven user functions could be defined (compared to just one in plain vanilla Applesoft). Also:

    - Each user defined function can support up to two arguments (compared to just one in plain vanilla Applesoft) (with provided samples as Factorial, LCM, GCD, HSCRN, IDIV and arithmetic binary logic functions);
    - There's support for the definition of user defined function bodies in Applesoft language (which are called “Procedural functions” within this document), thus permitting Applesoft extensions by authors w/o any prior extensive knowledge of 6502 assembly language.
- From Peersoft v1.5.6 version onwards, integer unsigned arithmetic could be used in addition to default signed arithmetic: the arithmetic rules obeyed to is determined by the setting of a configuration bit (msb of byte @ address 40167 decimal)
    - If unset, then the default configuration is active: 16bits integer numbers valid range is -32768 to 32767, 8bits integer numbers valid range is -128 to 127.
    - If set, then unsigned arithmetic is active: 16bits integer numbers valid range is 0 to 65535, 8bits integer numbers valid range is 0 to 255.
    - Whatever the active arithmetic variant, any variable setting statement attempting to exit the valid range as defined above will fail and an ILLEGAL QUANTITY Applesoft error is raised.
    - The FOR/NEXT handling code has been amended in order to support negative step values even when unsigned arithmetic is in effect.
    - Upon program start, as on every occurrence of the `CLEAR` Applesoft statement, default signed arithmetic is reinstated.

Peersoft roadmap

The features described in the previous section are actually implemented in the Peersoft current release. However, not all the features envisioned from the start have already been implemented. Some are yet to be developed from scratch in order to meet the author’s original requirements.

The table below provides some hints about what additional features will be developed and candidate release dates. However, as this project is founded upon the time left only as I am “idle” on both the other (i.e. family and professional/business) aspects of my life, I would suggest to not hold your breath.

Version	Feature(s) implemented	Estimate delivery date
1.7	Generalized user defined functions (`DEF FN`): more args allowed and arg. Variables of any type, not just FP)	30/11/2015
1.6	**New integer subtype BYTE**	**08/09/2015**
1.7	New integer subtypes LNG24 and LNG32	30/11/2015
1.5	New array dimension max value set to 65535 (useful for byte arrays within memory expansion cards: was 32767 originally)	01/01/2015
1.7	Calling Integer BASIC programs from Applesoft programs	30/11/2015
1.7	Inline assembly within Applespft programs	30/11/2015
1.5.5	11 ML user defined functions, each one can accept up to two arguments instead of one originally.	15/08/2015
1.5.5	User defined functions can be coded in Applesoft (“Procedural functions” facility)	15/08/2015

Version	Feature(s) implemented	Estimate delivery date
1.6	**Benefitting from extra memory (thanks to Apple and A.E.) and options for defining arrays on such memory card.**	**08/09/2015**
1.5	Support for remaining utility routines to reorganize the Applesoft variables areas (both simple and array)	01/01/2015
1.7	Merge with Bananasoft utility (using similar technology but focusing on features rather than performance); the name of resulting software could be “fruit salad” but still quite unsure about this ;-)	30/11/2015
1.5	Support for handling of mouse and timer interrupts by Applesoft routines	01/01/2015
1.5	Precomputed GOTO/GOSUB within program text	01/01/2015
1.8	Generalized integer arithmetic for expression evaluation (i.e. the sub expression A% + 1 will be evaluated using integer arithmetic first and only reverting to FP operation in case an overflow occurs.	31/12/2015
1.8	Compilation of Applesoft user defined functions to convert them to machine code callable with `USR<n>(…)` functions (provided by Bananasoft, n from 0 to 9).	31/12/2015

Peersoft physical package description

Peersoft consists of a zip archive (Peersoftv1.5.6.zip) containing:

- One disk image with the .do suffix (DOS 3.3 sector order) containing a complete Merlin 8 DOS 3.3 development environment (v2.48 release) along with auxiliary source files as the 1.5.6 version release is the first one for which sources cannot fit within a single 140K disk image;
- One disk image with the .do suffix providing a complete set of Peersoft source files to build Peersoft (main binary and extra companion utilities) on any CPU variant (6502 plain vanilla, 65C02 or 65816).
- One disk image with the .do suffix which is the Peersoft boot disk with companion Applesoft and ML programs.

Filename on disk	Purpose
PEERSOFTV15.S	Main source file for building Peersoft using Merlin
T.PEERINSTALL	Peersoft PUT file inclusing Peersoft installation stuff.
T.PEERLIST	Peersoft PUT file handling the LIST Applesoft instruction.
T.PEERINTEGARITH	Peersoft PUT file handling the integer arithmetic routines (handling the +=, −=, *= and /= for integer variables) as well as the loop variable increment whenever that loop variable is integer (NEXT statement)
T.PEERAROMBA	Peersoft PUT for all extensions to ROM FRMEVL original subroutine.
T.PEERMTK	Handling of coroutines within Peersoft
T.PEERGOTO	Precomputed GOTO/GOSUB within Peersoft
T.PEERMOUSTIME	Handling of Mouse and Timer events within Peersoft
T.PEERMOTIDATA	Data segment for the handling of the mouse and timer.
T.PEERGLOBALPAGE	Data segment for the Global page Peersoft partition
T.PEERNARRAY	Support for the newarray processing to be actviated for the 1.6 version release.
T.PEERGDATA	Data segment for the precomputed GOTO/GOSUB
T.PEERRGI	Module added to support the new integer subtypes (currently only the BYTE is supported) for the READ, GET and INPUT statements.
CRECON.S	Assembly source file for handling CPU recognition upon Peersoft boot (see HELLO Applesoft program from other disk): object installs as a $0300 subroutine and thus helping launching the proper Peersoft executable file.
TCPRECON.S	Assembly source file for detecting a thunder clock peripheral card or a //gs clock chip: both can be used to measure time spent by Applesoft subroutines within the TF Applesoft program. Object file is loaded as a $0300 subroutine from TF BASIC program.
SMTRECON.S	Assembly source file for detecting a NoSlotClock device within an Apple][, //e or //c and making use of it.
T.PEERNAUXMEM	Contains subroutines that will be located in //e auxiliary memory for the 1.6 version release.
T.PEERFDEF	Module in charge of handling the DEFUSR and DEF<type>new statements.
T.PEERPROCFUN	Module handling the “Procedural functions” stuff within Peersoft.

- One disk image with the .do suffix providing a bootable DOS 3.3 with the binary exe files from Peersoft (either machine code or Apple soft sample files).

Filename on disk	Purpose
HELLO	Boot program displaying a menu and prompting the user to select a valid Perrsoft program version to load according to CPU detected
CRECON	Object file loaded by HELLO program and which purpose is to detect host CPU flavour
PEERSOFTV15_6502, PEERSOFTV15_65C02,	One binary executable file per CPU, each file results from assembly from PEEROFTV15.S source files with

PEERSOFTV15_65802	different setting for the KOPT and KOPT16 macros
TF	Applesoft sample program trying to illustrate the features currently included within Peersoft.
TCPRECON	Object file loaded by TF program which deals with peripheral clock/chip detection and time elapsed measurements (Thunder clock and //gs internal battery/clock chip).
SMTRECON	Object file loaded by TF program which deals with peripheral clock/chip detection and time elapsed measurements (NoSlotClock device chip).
TUTORIAL2	Applesoft program file playing with the Peersoft mechanism of co-routines in some unusual ways.
TUTMC	Auxiliary machine code routines (loaded at address $0300). Currently serves to toggle the speaker as a coroutine is swaped-in/swaped-out.
TCUSRFNDEMO	ML program file containing a set of sample USR defined functions to be used by he TF program.
TIMRWAIT	Applesoft program file illustrating the new handling of the Applesoft WAIT statement which can then be interrupted in the middle of a memory scan.

- Peersoft documentation in the form of a LO/PDF file you are currently reading

How to transfer the two disk archives to real 5'1/4 disks on an Apple // hardware

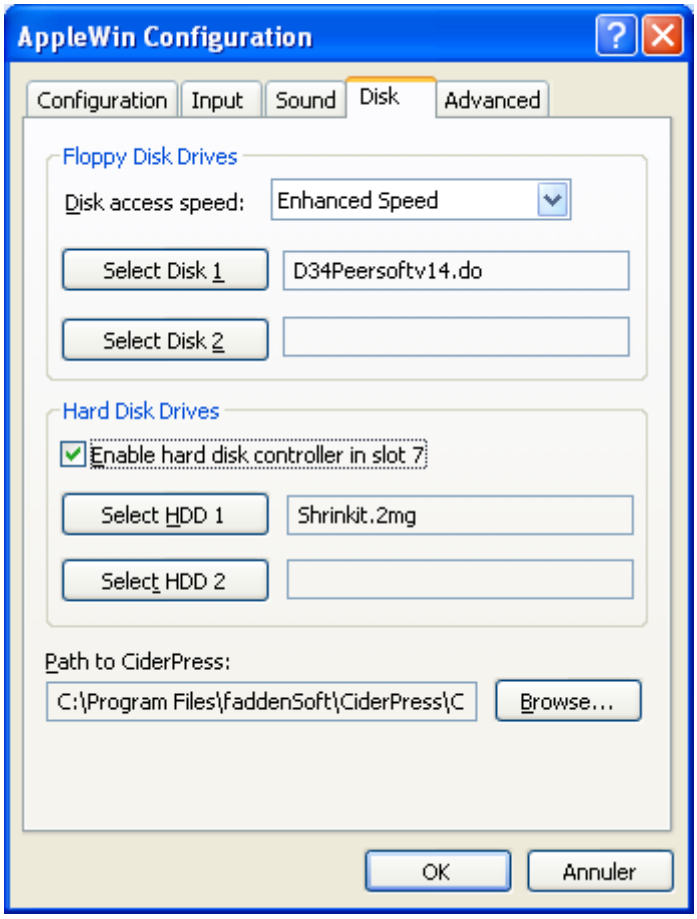
In order to transfer the content of both disks to a native hardware Apple // computer. We will use the components below:

- A pre-formatted disk image (with ProDOS2.0.3 and NuFX Shrink/Unshrink system files); this disk image will grab the DOS 3.3 disk images and put them into a ProDOS 8 archive file. In case you do cannot put your hand on such disk image, you can get the one from my site (URL is http://bgilon.free.fr/apple2/ShrinkIt.2mg).
- An emulator for running the NuFX Shrinkit program on your modern computer. To illustrate this, I am using the AppleWin 1.22 Win32 emulator.
- The CiderPress Win32 program in order to put the NuFX archive files onto a CFFA compact memory.
- The CFFA Compact Flash memory disk drive for Apple //e or //gs (mine is CFFA 2.0, but a newer version has since been released with the ability to directly read .po and .do disk images).

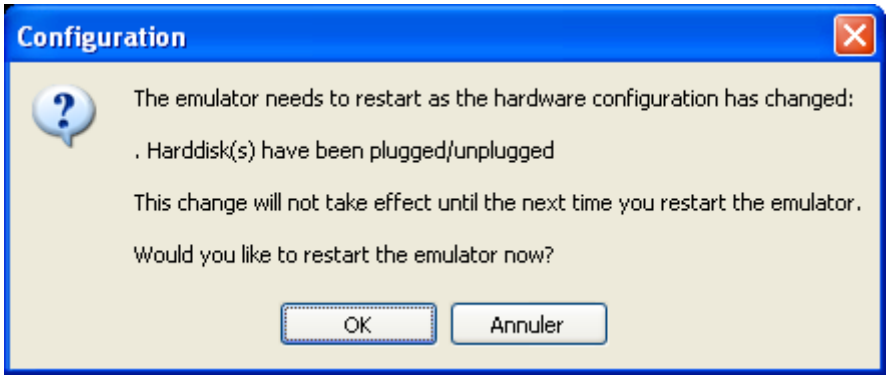
Other paths are available for performing the same tasks (dealing with serial communication interfaces between a "modern" computer and the Apple //).

Configuration of Apple Win 1.22

After having downloaded the disk image containing a bootable ProDOS 8 and the ShrinkIt system file, open the Configuration window by pressing the F8 function key.

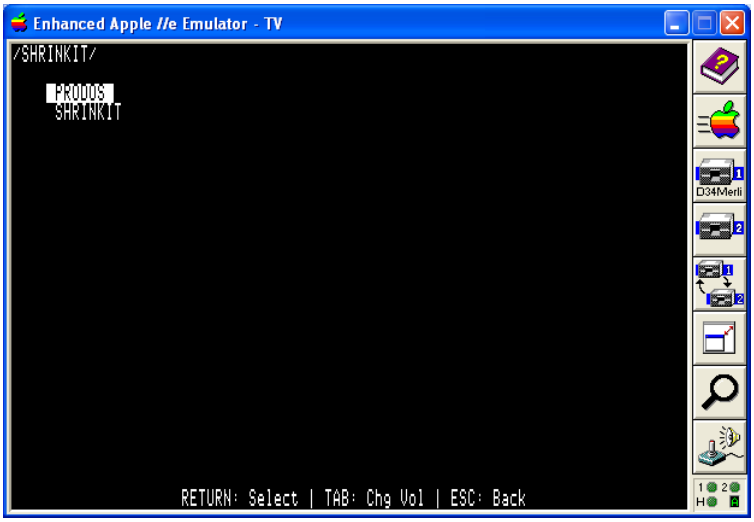


Check the option box labeled “Enable hard disk controller in slot 7” and click on the Select HDD 1 action button. A “choose file” dialog box would open. Navigate to your download directory and select the file. Click OK. A message box could then pop up advising you that AppleWin will reboot due to change in connected bootable devices configuration.



Operation of the Shrinking procedure

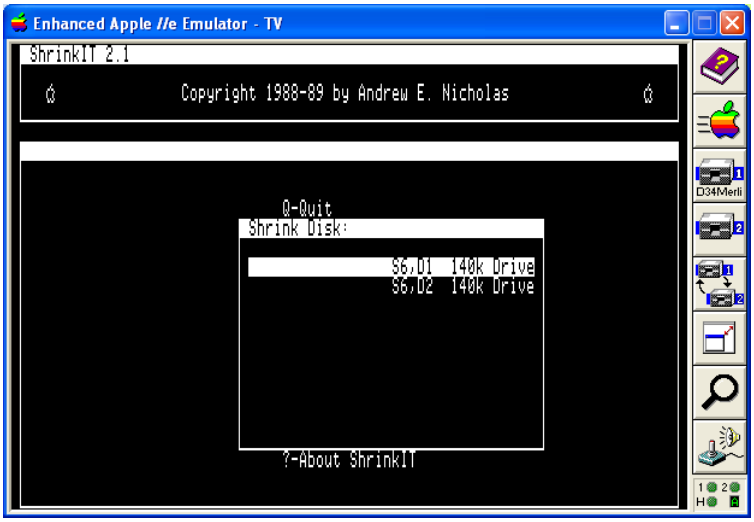
Once the emulator has restarted, then the screen below should pop up.



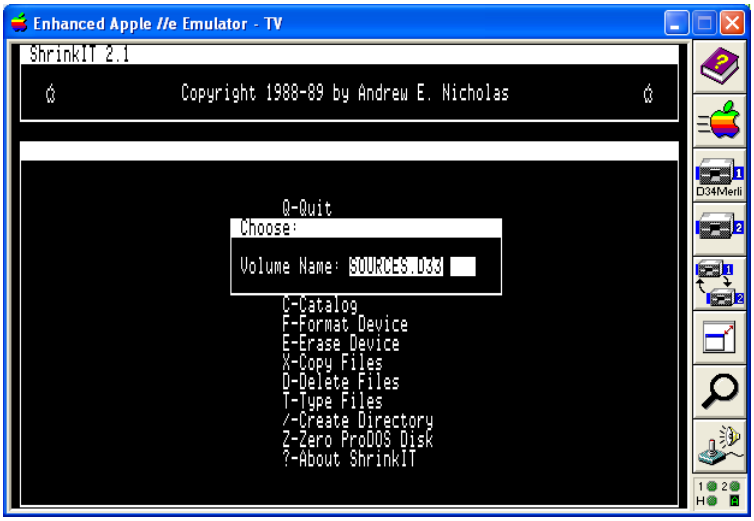
Select the SHRINKIT option, press Return and the screen below should pop up.

Drag and drop the D33Merlin – Peersoftv15.do disk image icon on the drive 1 box with the panel at the right side of the window. And select the Shrink option.

Select the Shrink “Disk” option

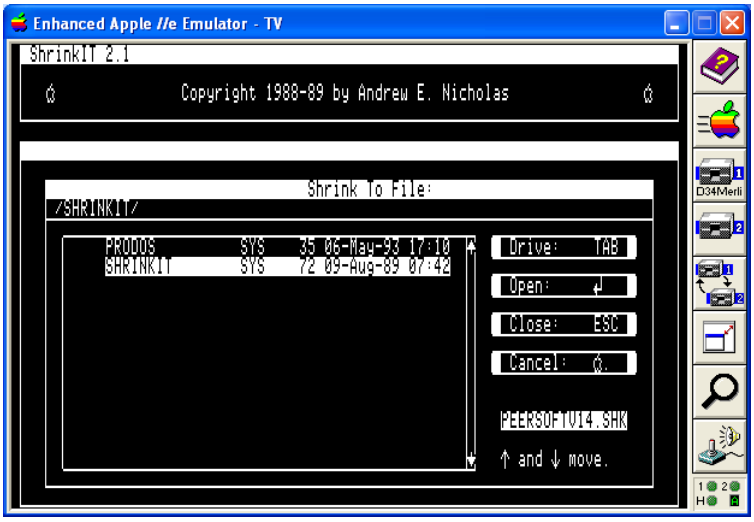


Select the “Shrink Disk on S6, D1 140k Drive” option..

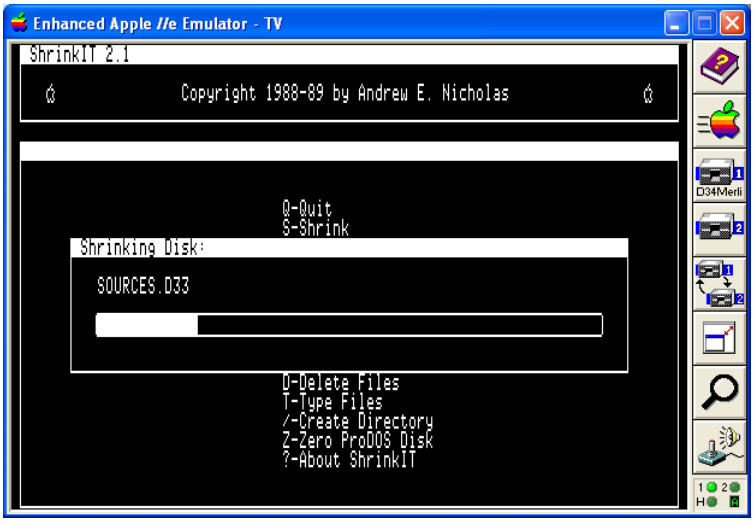


Enter the label for this backup within the archive. Here “SOURCES.D33”.

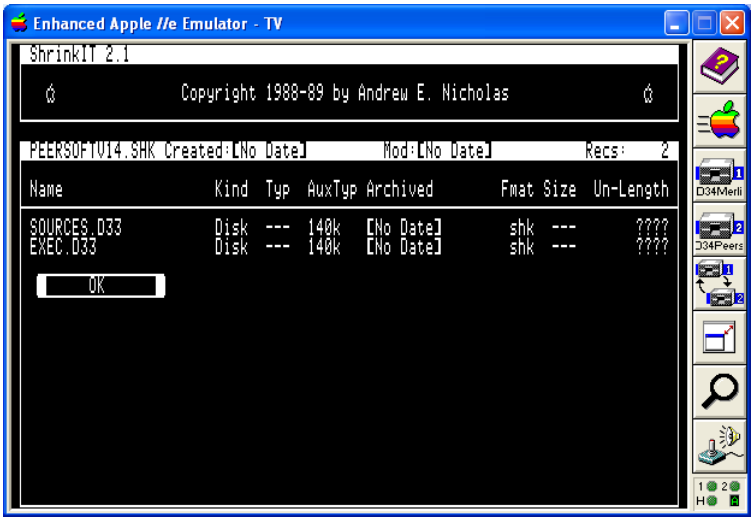
Now enter the name of the ProDOS file which will contain the backup you are about to initiate.



Here I have entered the filename PEERSOFTV156.SHK. Once the RETURN key has been pressed, the progression bar for the shrinking advises you of the… progress so far.



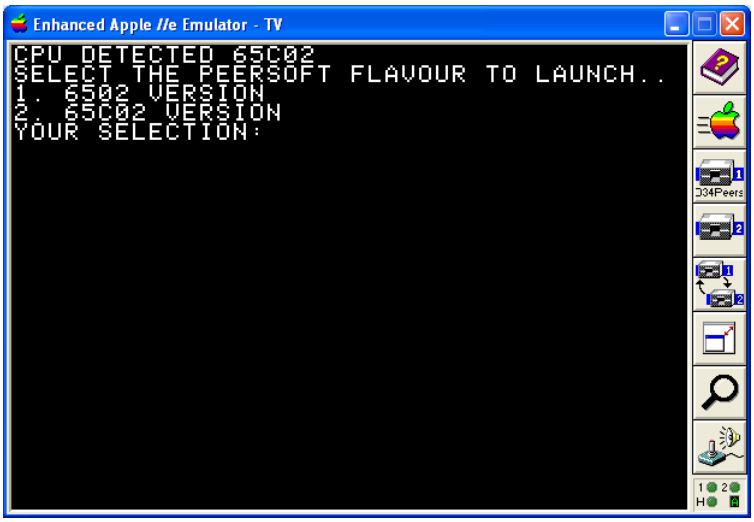
The next step would be to reiterate the procedure above for the D34Peersoftv156.do disk image. Checking that everything is OK at the end of this step can be done by listing the content of the archive which is an option available from the main menu.



Peersoft user manual

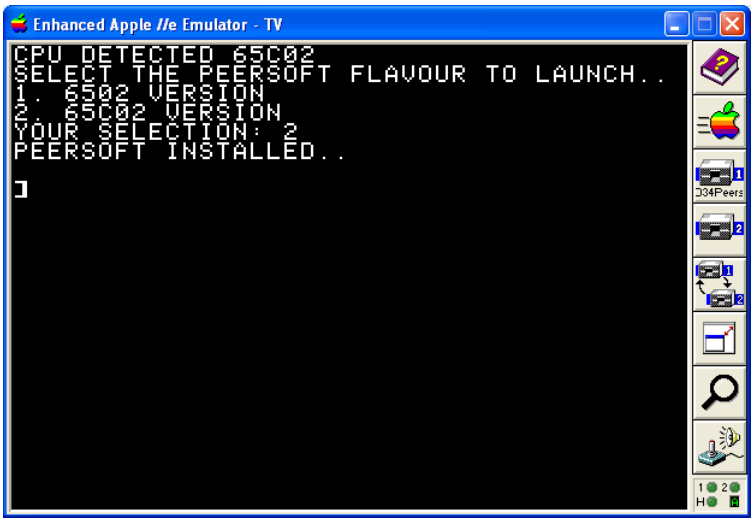
Peersoft executable files come within the D34Peersoft156 (.do archive or real 5’1/4 disk depending on whether you have an emulator or a real hardware on hand.

The relevant disk image is DOS 3.3 bootable, insert it in drive 6 slot 1 and reboot your emulator/computer. The screen appearing should be similar to the one below.



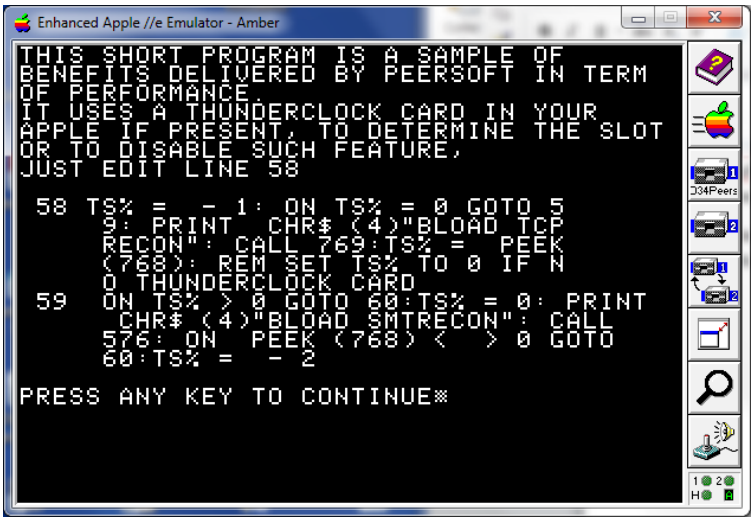
Depending upon the CPU detected on your host environment, more or less choices could be available. At this prompt, you can opt to bypass the Peersoft installation by using the usual <CTRL><C> keystroke. But for the time being, suppose that you selected option 2 to install the Peersoft version which can benefit from the richer instruction set of the 65C02 CPU.

The mention that Peersoft has been installed pops up.

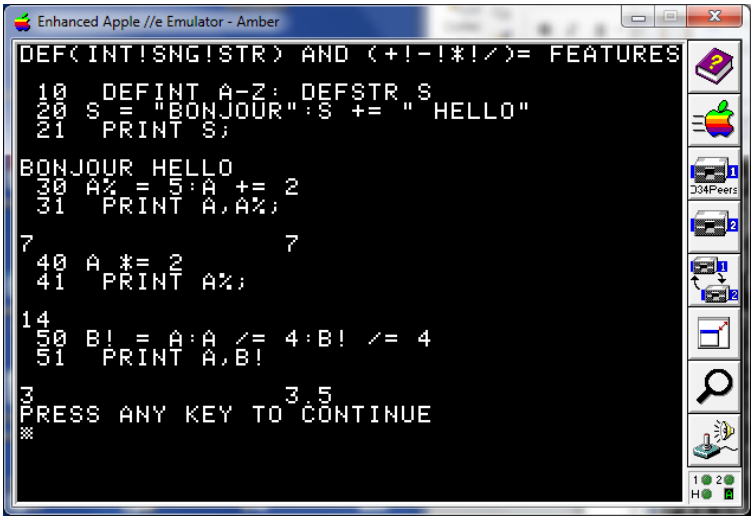


There is an Applesoft named TF which allowed the user to check that the latest build Peersoft showed no regression. In addition, it shows up every implemented feature to interested parties (either programmers themselves or end users).

Just issue the `RUN TF` command from the "`]`" prompt. This leads to screen below. The TF program can make use of the Apple //gs clock chip, a Thunderclock peripheral card (as supported by the Virtual][emulator under Mac OS X), or a SMT NSC chip (as supported by both emulators I used to test Peersoft in emulator environments) to measure time elapsed.



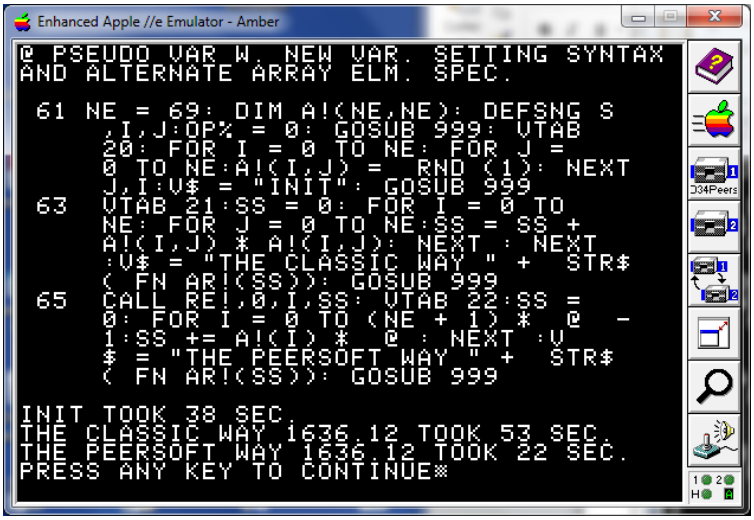
Just press any key on your keyboard to proceed…



This screen shows some new features available by using Peersoft as

- A new way to concatenate strings;
- Default typing for Applesoft variables (using the DEFSTR, DEFINT and DEFSNG statements);
- Some new syntax schemes for altering values of variables.

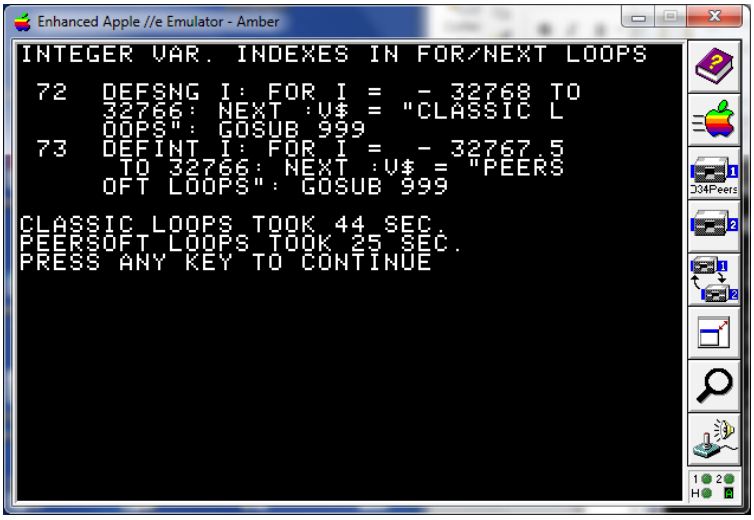
Just press any key on your keyboard to proceed…



The features that are showed within this screen are:

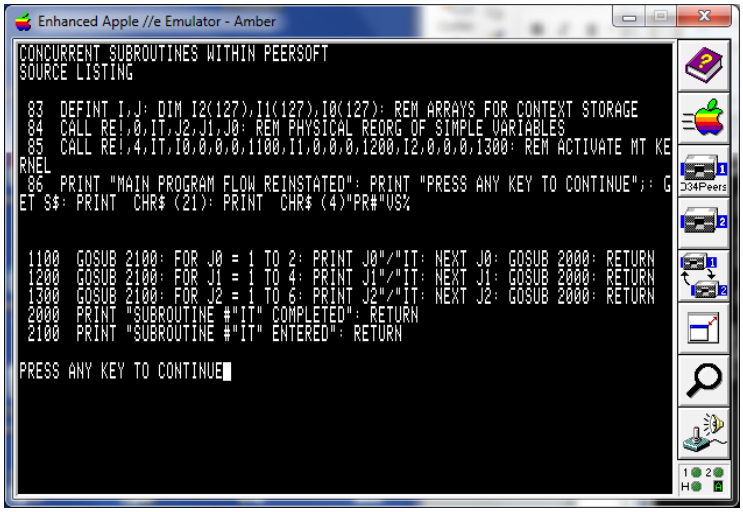
- Use of a utility routine to physically reorganize the simple variables memory area so that the variables "`J`", "`SS`" and "`I`" are looked up first from then on;
- Use of the "`@`" pseudo variable in order to avoid additional references (lookups) to simple variables "`I`", "`J`" and to array "`A!`" and relevant computations.

Just press any key to go to next screen.

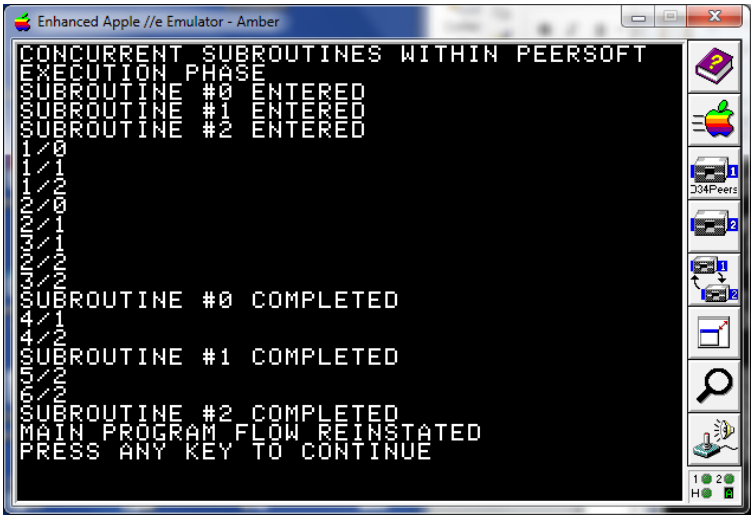


The screen above shows the benefit of adopting pure loop variable of integer type.

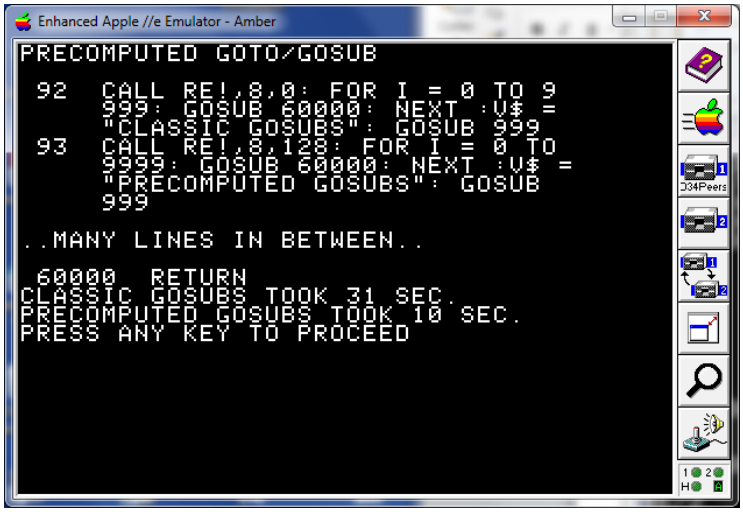
Here is a minimalist segment of code to illustrate the use of co routines within Peersoft (an innovative feature indeed). The first few lines listed set up the environment. And the latter lines form the body of the co routines and subroutines called from within the co routines.



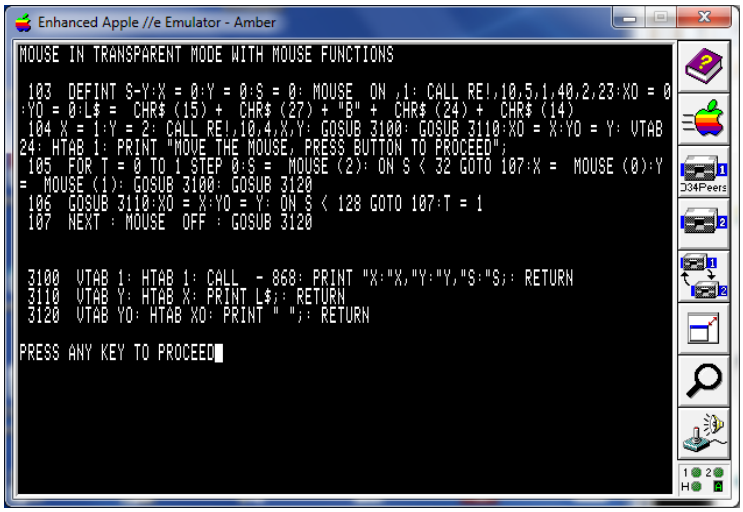
To show what is displayed on the screen resulting from running the activation, just press a key as usual.



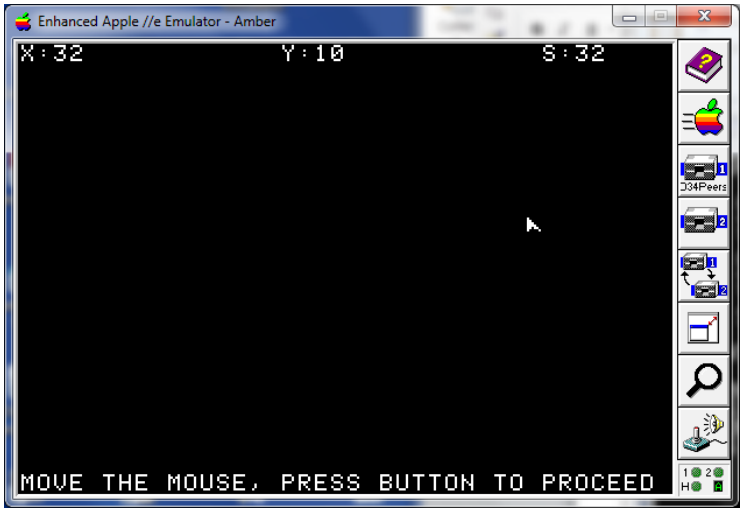
Every co routine is entered and completed and the allocation of CPU to each thread follows a round robin model till all subroutines complete, thus triggering the end of program. The co routine feature is thoroughly described in a subsequent chapter. For the time being, just press a key as prompted by the program; after a while, the screen below is displayed.



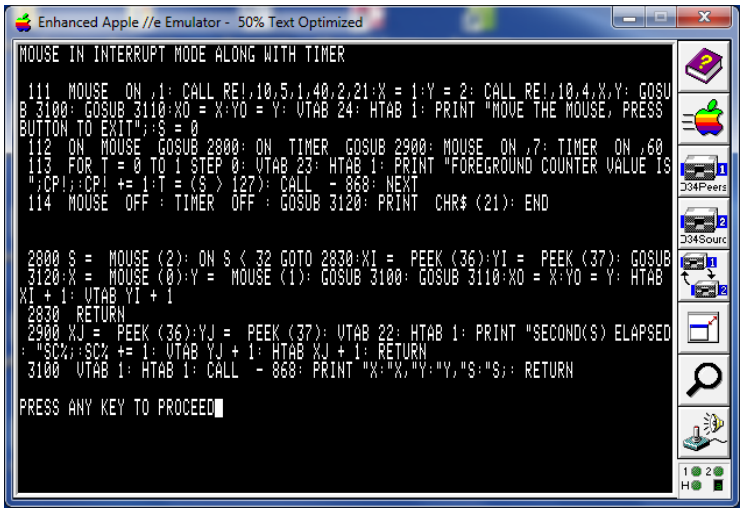
This illustrates a new feature from the 1.5 release, which precomputes the addresses of target lines of `GOTO` and `GOSUB` Applesoft instructions, thus optimizing the handling of such instructions when located in loops (i.e. being run on many occurrences). The reference manual below gives details upon the related utility functions to setup the program behavior (i.e. whether or not performing automatic precomputations). On a medium size program (# of lines < 200), we see that times for handling of `GOTO/GOSUB` instructions are more than halved but could be made even smaller with programs of greater size. The next two screen dumps illustrate another new feature from this Peersoft release (i.e. 1.5).

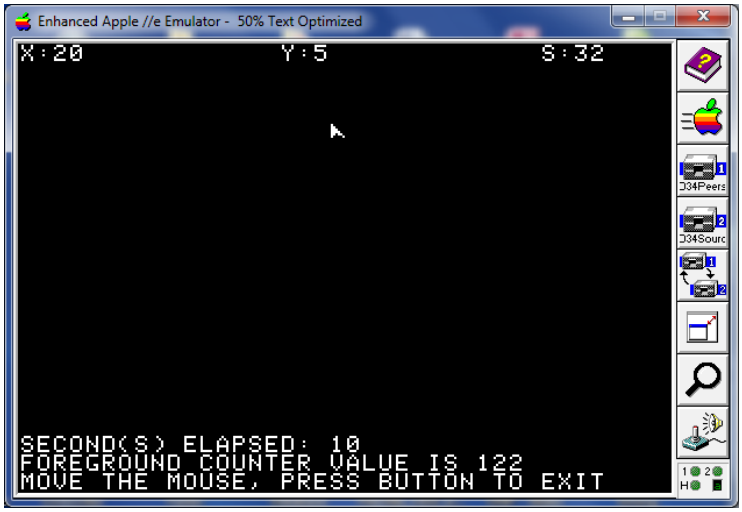


The codlet shown above uses the `MOUSE` instruction to initialize the mouse, some general utility function to setup mouse clamping limits and setting the original mouse position, and the `MOUSE` function to read both the mouse coordinates (X and Y) and the mouse button status. And as the code above is run…



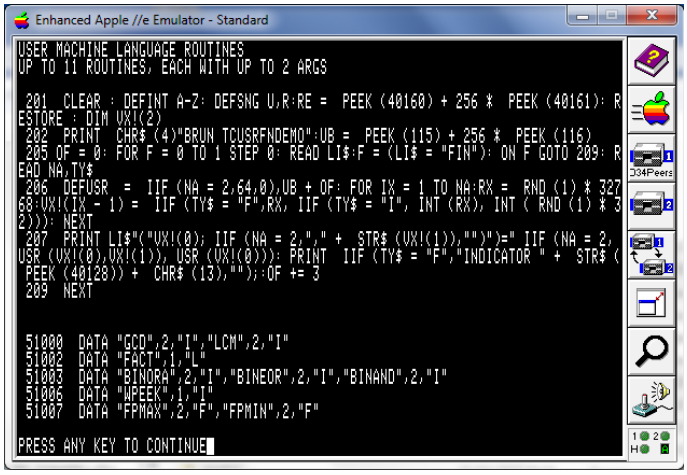
The mouse cursor position is updated as you move the mouse within the clamp “window” (excluding first and last screen lines). Pressing the mouse button will resume normal operation as indicated. A similar screen can be obtained but this time based on a mouse interrupt mode. You must have an enhanced Apple //e (hardware or emulation) or a // c or //GS Apple model in order to see this step (otherwise it is skipped to the program end). Here you’ll see that both MOUSE and TIMER will work asynchronously related to the main program flow (handling by subroutines starting at lines 2800 and 2900 respectively).

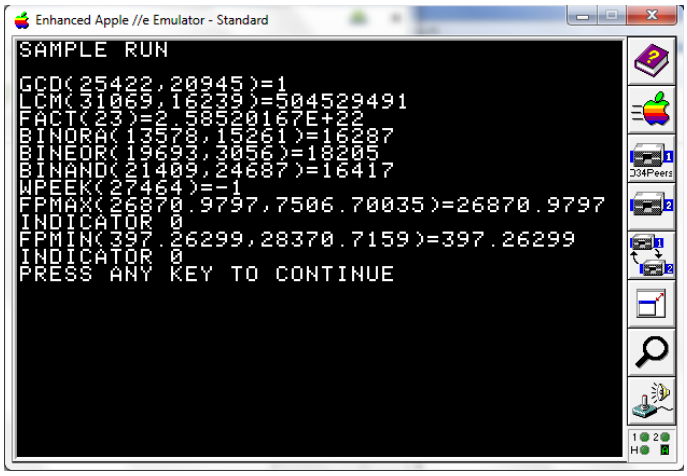




The "*SECONDS ELAPSED*" display screen line is managed by the subroutine registered with the `ON TIMER GOSUB` instruction; Note that as the `TIMER` factor being set to 60 (meaning 60 times 1/60^{e} seconds), the Applesoft routine is only called once every second on a north American Apple 2 hardware.

The *FOREGROUND COUNTER VALUE* display screen line illustrates the foreground processing activity. Well nothing spectacular here… The Applesoft subroutine starting at line 2800 is in charge of updating the cursor character on the display screen within the clamp window limits.





From the 1.5.5 version onwards, Peersoft allows the definitions of up to 11 user defined functions in 6502 Assembly language (only one in original Applesoft), and allows up to two input arguments per user function (just one in original Applesoft), what is shown in the two screen snapshots above.

A sample ML routine library is available and provides the functions below:

- GCD and LCM of two integers (each one can take the form of FP expressions which include 32bits integers);
- Factorial of integers (hélas, submitting input arguments which value is greater than 33 leads to an overflow error);
- Three functions to handle 16bits binary wide operations AND, OR and EOR;
- A function WPEEK for retrieving a 16bits quantity within Apple memory;
- Two functions for finding the MIN and MAX of two FP expressions.

The true innovative feature comes from the fact that such user defined function can now be defined in Applesoft (such UDF being then called "*Procedural Functions*" within this document). Imagine that you have to design a function which, given an input parameter of integer N, will return the sum of integers from 1 to N.

1. A construct, implied by a `DEFFN` Instruction, will not allow counting and control structures within their definitions (even when the appearance of function such as `IIF` might help in some cases), moreover, the type and number of arguments is restricted here;
2. Coding such function in 6502 assembly can be done but this will require some skills from author that are beyond what every Applesoft Application author masters; the new function could then be called as part of a larger expressions just like the ML sample functions referred to from above section;

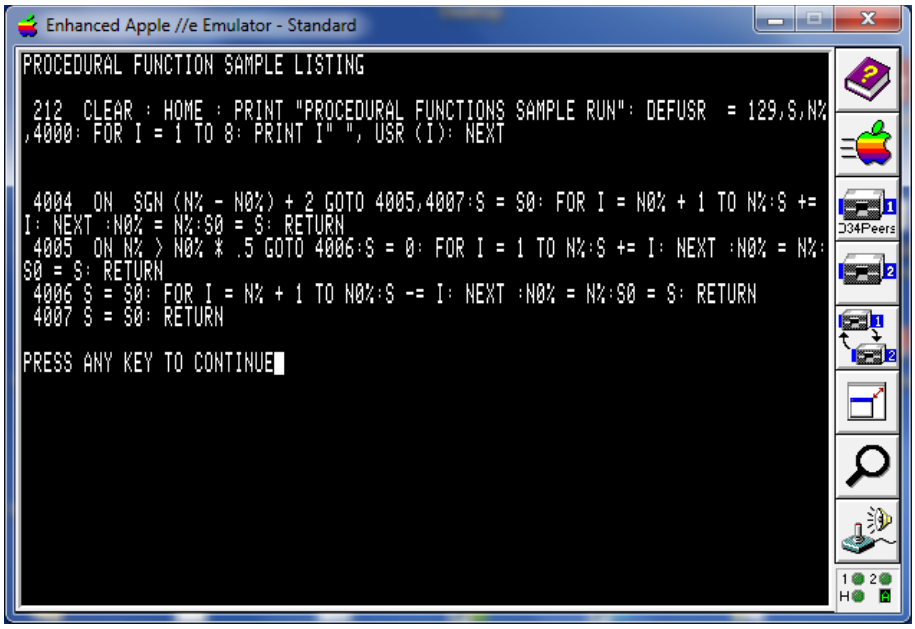
3. One can sacrifice the transparency of calling such functions within more global expressions and use some classic Applesoft constructs as;

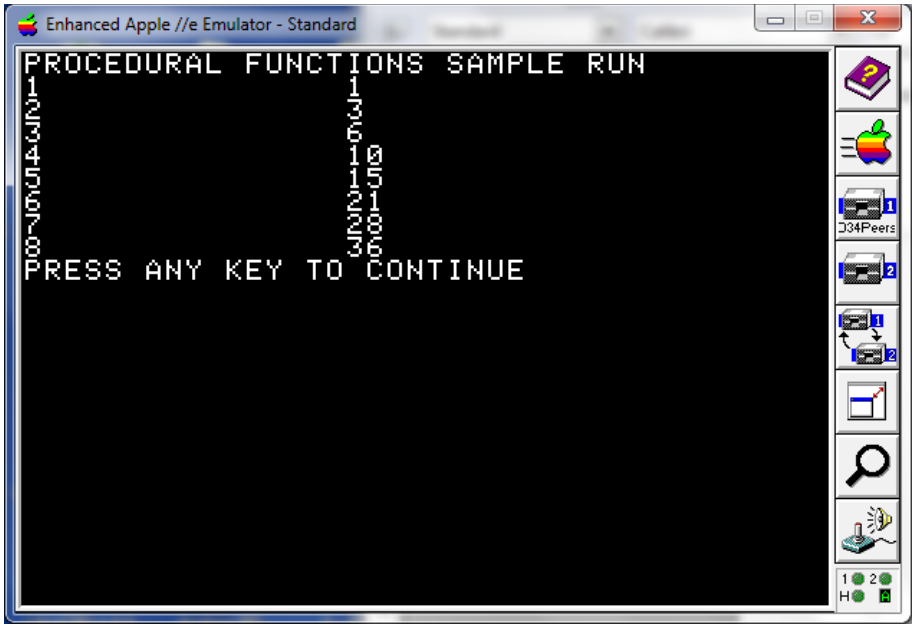
```
P1= <InputExpression1>:P2=<inputExpression2>
GOSUB <LineNumber>
REM Use of result expression from here by using the R variable.
```

But this cannot be used as a sub expression or be part of a `DEFFN` expression.

4. What is allowed with « *Procedural functions* » (PF for short) is the merge of approaches 2 and 3 above. Allowing the definition of functions in Applesoft BASIC including such constructs as loops and tests and the call semantics identical to the option 2.

Briefly stated, one might consider PF as an alternative to multi line user defined functions (as supported by more sophisticated versions of BASIC). The two screen snapshots below illustrate the use of PF in a simplistic way:



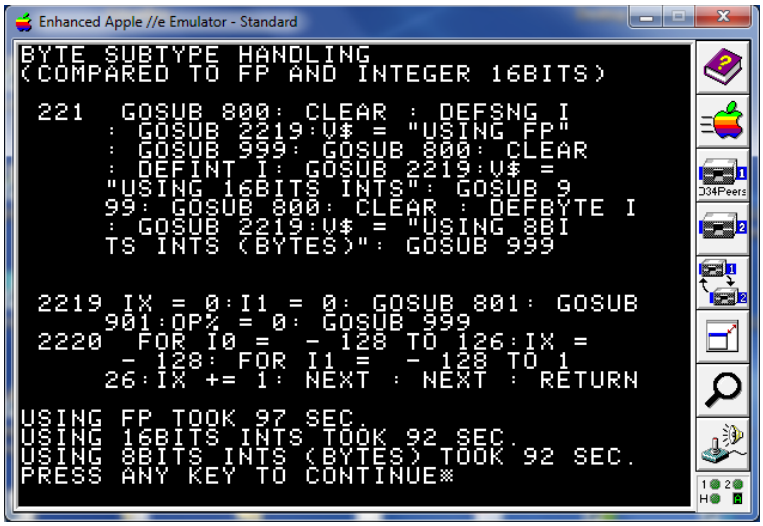


In this sample, we are using:

- a private memory segment to optimize the variable lookups while executing the function body (hence the value 129 instead of 128 for the `DEFUSR` instruction);
- a cache built from variables `S0` and `N0%` contents to avoid restarting the calculation from index 1 within the loop on every occurrence.

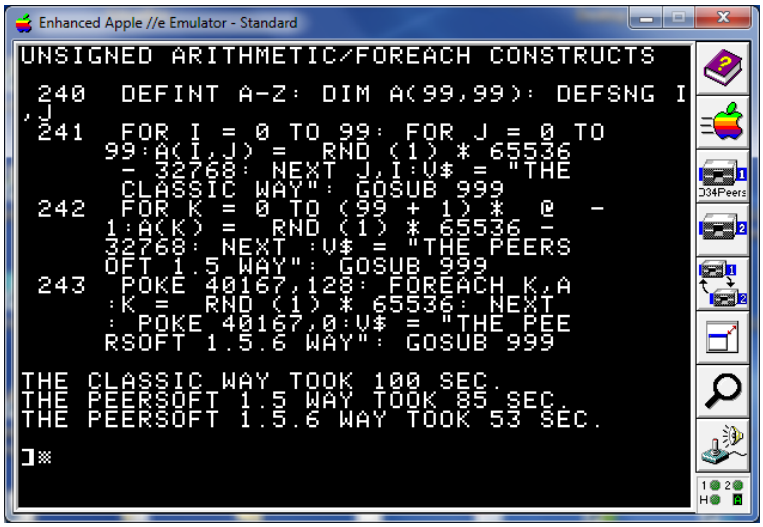
Further details can be found in the relevant reference section.

For the time being, press any key to proceed to next step.



Above is a sample codlet to let the user know what would be the tiny performance gain by using the byte subtype over using more classic FP or 16bits wide integers.

Press any key to proceed to next screen.



The screen above is an illustration of the benefit that can be obtained from using both the unsigned arithmetic variant and the `FOREACH` loop construct to initialize an array.

Peersoft reference manual

Variable default typing

DEFINT A, I-N,Z

To specify the scope of variables involved by every DEF<type> instruction, you just have to list first characters either alone or as part of a range. In the sample above, every variable (simple or array), which first character is “A”,”I”,”J”,”K”,”L”,”M”,”N” or “Z” will have a default type of integer (16bits).

The involved statements are `DEFINT, DEFSNG, DEFSTR` and `DEFBYTE`.

DEF<type> statements can appear anywhere within a program text and be run at anytime within the program flow.

At program start or when a RUN/CLEAR Applesoft statement is run within the program flow, then all variables inherit the Floating point type default.

However, an explicit type specifier (“%”, “$”, the new “!” used for floating point and “.” used for the new integer 8bits subtype) as a variable name’s suffix overrides its default type currently defined.

Thus the statement sequence `CLEAR : DEFINT I:I = 1: PRINT I!` will print 0 on screen.

New syntax scheme for altering variables values

```
A += 3
```

Peersoft will simplify variable value alterations by providing a new syntax scheme. The new `A += 3` being a shortcut for `A = A + 3`.

All four basic operations can be part of the new syntax scheme.

For instance `A -= 3` is a shortcut for `A = A - 3` and `B /= 4` is a shortcut for `B = B / 4`

The new syntax scheme can be included in every context where a variable has its value set within program text. This includes the FOR/NEXT loop construct.

`FOR I += 5 TO 10` is a shortcut for `FOR I = I + 5 TO 10`

Whenever the variable type is integer, then arithmetic operation applied is of the integer kind too. Also the += syntax scheme is also valid for string concatenation whenever the variable is a character string.

`S$ = "BONJOUR": S$ += " HELLO": PRINT S$` will print `BONJOUR HELLO` onto the output display.

@ Pseudo variable

Having written quite a number of applications myself and studied the code from other authors as well, I’ve found out that one pattern that emerge quite often is the use of the same sub expression/variable multiple times within an expression. Sometimes, there is a cost in term of performance to lookup some sub expression/variable (particularly when dealing with multi dimensions array variables). So the idea of implementing the `@` pseudo variable was born.

Every time an expression is evaluated, the Applesoft interpreter will use some constant locations within page zero as main and auxiliary accumulators, large enough to contain an integer, a floating point value, or a string descriptor. The `@` pseudo variable is the simplest in its processing code. All it does is a RTS (actually it's a bit more than that but only by a small amount: cf. source code for further details ;-), this would imply that the returned value will come unchanged from what it was during the last "factor" evaluation.

Beyond the sample code showing up in the previous section : "Peersoft user manual", the `@` could also refer to any content of any type.

`PRINT RIGHT$( A$, LEN( @ ) - 4)` will print the "`A$`" current value with its first 4 characters removed.

Use of the `@` pseudo variable as the first term evaluated within an expression thus possibly referencing the result from a previous statement expression is not recommended, especially from within a co routine.

FOREACH new statement

This new statement will help to implement a new loop construct for easy array lookup and update.

The construct is:

`FOREACH <loopvarName>,<arrayName>:… loop body : NEXT [loopvarName,…]`

The loop variable must be of the same type as the array (either FP, string, 16 or 8bits integers).

Within the loop body, a new pseudo variable '`#`' is defined to return the element index from the start of the array, initialized with zero value as the `FOREACH` statement is processed and incremented by 1 for every `NEXT` statement processed.

IIF() function statement

Likely at many occurrences every Applesoft application author has wished to be able to return one among two values depending on a Boolean criterion. Up to now, he has three methods to code this mechanism:

- Using the Applesoft codlet below:

```
10  ON booleanExpr GOTO 20: LET returnVariable = returnValueIfFalse: GOTO 30
20  LET returnVariable = returnValueIfTrue
30  REM Flow of the program  continues from here
Pro: Can apply to return string or numeric expressions
Pro: BooleanExpr only evaluated once
Pro: Only one of the two expression returnValueIfTrue, returnValueIfFalse is
evaluated
Con: Cannot be summarized within a user defined function definition (DEF FN)
```

- Using the Applesoft sub expression below:

```
booleanExpr * returnValueIfTrue + (NOT booleanExpr) * returnValueIfFalse
Pro: Can be further processed as a subexpression
Con: Boolean expression is evaluated twice
Con: Both returnValueExpression are evaluated even when one will be discarded
(because of being weighted with a zero value).
Con: Work only for numeric return values
```

- Using a prefilled one dimension array (containing two elements)

```
10  DIM VR(1):VR(0) = returnValueIfFalse:VR(1) = returnValueIfTrue
20  REM Then you can use the VR(booleanExpr) as the subexpression.
Pro: can be further processed as a subexpression
Pro: booleanExpr evaluated only once
Pro: VR can be any type (i.e. numeric or string) and so the return value
Con: Only works for constant value returning (the returned expression are not
evaluated each time the VR(booleanExpr) is referred to within an expression.
```

You see that very method described above have its pros and cons. With the new syntax scheme below, I only can perceive Pros to its adoption:

- Just use the `IIF( booleanExpr,returnValueIfTrue,returnValueIdfFalse)`

```
Pro: can be further processed as a subexpression
Pro: booleanExpression evaluated only once
Pro: Just one of the two return value candidate expressions is evaluated (the
other being only scanned for a terminator character)
Pro: the selected expression is actually evaluated at the time of the IIF
function call is processed.
Pro: Can cope with any expression type (either numeric or string).
```

So the IIF can be a time saver both for the Applesoft developer and the end user running his programs.

Integer variables as loop variables within FOR/NEXT loops

I was worried that Integer variables be banned from being used as loop variables within FOR/NEXT loop constructs.

If you try the statement below under plain vanilla Applesoft interpreter, all that is returned is a "`? SYNTAX ERROR`" message.

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

I believed that the use of integer arithmetic for processing the increment and test for final value as part of the `NEXT` statement processing would greatly offer benefits in performance terms.

Hélas (in French in the text), by the time the loop variable appears in the loop body, then all benefits disappear because handling of integer variables is much more costly than of floating point variables just by the fact that the retrieved integer value needs to be converted to floating point whatever the context.

OK now, with Peersoft installed, you can have integer (either 16bits or 8bits wide) variables as loop variables, but I wouldn't tell you more about it… until later this year (or next: cf. section "Peersoft roadmap" for further details on future developments).

Ah yes, be advised that plain vanilla Applesoft is bugged in its integer variable handling too. Have you ever tried to issue a `A% = - 32768` only to get bounced with a `?ILLEGAL QUANTITY ERROR` message?

Users curious about this state of things could study the excellent Web resource already mentioned in this document (http://www.txbobsc.com/scsc/scdocumentor/). Suffice to say that

`A% = - 32767.5` works well and provides the same expected result[1]. From Peersoft version 1.5.6 onwards, a fix is provided.

Another limitation to warn the reader about is that the final value of such loop (using 16bits integer variables) cannot be 32767 (which is the algebraic highest possible value a 16bits integer variable can be bound to). This is because, as the last iteration (the loop variable being equal to the final value) completes, the first operation the NEXT statement does is to increment the loop variable's value (here 32767) with the STEP value (default 1), this add operation causes an overflow within the 6502 and thus the overflow exception is raised to the Applesoft environment, itself raising an "`? OVERFLOW ERROR`" for the Applesoft program. Similar behaviour is to be expected when using 8bits integer variables in loops).

To complete this section on an optimistic side, here is my advice regarding placement of loop variables within FOR/NEXT loops. The point to keep in mind is that:

- It is useless to keep a loop variable at the top of simple variable table, unless either you use it extensively in the loop body or, for whatever reason you have, you still want to use the `NEXT <variableName>` syntax scheme for iterating within a loop. The loop variable placement plays no role in performance aspects when processing a `NEXT` statement with no trailing variable reference as all data upon which the NEXT statement works lies within the stack (including a direct pointer to variable's value within Simple Variable Table); the frame being built as the interpreter enters the loop, just once during processing of the `FOR` statement.

Some Applesoft statements processing bugfixes

Every bug fix provided here is an obvious code update to bugs raised, update to be considered as a part of the Web resource already mentioned (http://www.txbobsc.com/scsc/scdocumentor/).

ONERR statement

The current Applesoft implementation for the `ONERR` statement processing erroneously skip the whole physical line after processing instead of just up to next "end of instruction" marker.

This is fixed within this Peersoft release.

FP constant -32768 malformed

The constant, as explained in the Web resource "*SC Documentor*", is missing a byte to be complete, therefore the FP format representation of the integer constant is malformed within ROM and lead to an ILLEGAL QUANTITY ERROR while trying to convert this particular valid value to an integer format for setting an integer type variable. This is fixed within Peersoft v1.5.6.

INPUT statement not collecting the whole entered string

As a benefit from the support for the transparent processing of the *BYTE* subtype, an Applesoft bug illustrated by the codlet below is now fixed (from Peersoft release 1.5.6).

```
10  REM Enter the '10:10' (w/o the quotes) string at prompt
20  INPUT "TIME OF EVENT: ";A$: PRINT A$
```

1 From release 1.5.6 onwards, Peersoft provides a transparent fix to this bug.

This will output the string “10” on display string on plain vanilla Applesoft: actually the entered input string is truncated. This bug was noted/raised by Ivan Drucker at a past Kansasfest session while describing his “Nu Input” package to the audience. From now on, we can say that the Applesoft `INPUT` statement is a bit less “sucky” (to quote Ivan Drucker's own adjective to qualify those Applesoft statements ;-).

RETURN and POP statements

Mouse and Timer handling within Peersoft

The support for mouse and timer (i.e. vertical blanking) within Peersoft is based on:

- The `MOUSE` and `TIMER` new instructions to activate and deactivate the MOUSE and TIMER interfaces, either in transparent or interrupt modes;
- The `ON MOUSE GOSUB` as well as `ON TIMER GOSUB` instructions to setup the Applesoft subroutines called upon interrupts;
- The `MOUSE` function to return status from the mouse Interface (whatever the running mode, i.e. transparent or interrupt based).
- Similarly the `TIMER` function is used to return some useful parameters related to timer event processing.

Please note that only mouse transparent mode is allowed on those hardware configurations: Apple 2, 2+ or //e with unenhanced ROM; No TIMER feature is supported on those old configurations.

This has to do with how the interrupt system is built in those systems and the way some critical zero page locations are used within DOS 3.3. I strongly advise owners of unenhanced ROM Apple //e to switch to a //c compatible ROM.

ON MOUSE GOSUB lineNumber

This syntax scheme of an already existing syntax construct will setup an Applesoft subroutine to handle mouse interrupts (either movement or button presses). The running of a program (`RUN` Applesoft statement or `RUN/LOAD` <filename> DOS commands) will reinitialize such setting. This command should have been met before a "`MOUSE ON`, mode" statement is parsed from the program text.

MOUSE OFF

This statement will shut down the mouse dedicated activity from the mouse interface. The mouse is shut down as a program starts running (either with the Applesoft `RUN` statement or the DOS `RUN` command). As the mouse interface also supports the TIMER activity then the mouse interface is itself shutdown as soon as both the MOUSE OFF and TIMER OFF have been issued either explicitly or not.

MOUSE STOP

This statement is used to temporarily stop calling the Applesoft handling subroutine on every mouse related interrupt occurrence so that a interrupt handling Applesoft subroutine could complete before the next event be taken into account within Peersoft. A `MOUSE STOP` is implicitly performed as the Applesoft mouse event handling routine is entered and the status is reverted back to "`ON`" as the subroutine exists. Note that interrupts are still processed within Peersoft but the calling of Applesoft routine is temporarily deactivated.

MOUSE ON[, mode]

This statement initializes the mouse interface related to mouse activity. Allowed values are given in the table below:

Value	Meaning
1	Transparent mode: no mouse related (i.e. movement or button press) interrupt is further considered within Applesoft evaluation loop

3	Interrupt mode: only mouse movements are notified to foreground Applesoft program
5	Interrupt mode: only button presses are notified to foreground Applesoft program
7	Interrupt mode: both mouse movements and button presses are notified to foreground Applesoft program

MOUSE function

This function will take a single argument and return either the X coordinate of the mouse cursor, the Y coordinate of the mouse cursor or the current mouse button status.

Arg. Value	Meaning
0	X coordinate (16bits signed value)
1	Y coordinate (16bits signed value)
2	Button status (8bit unsigned value)

Utility functions for handling other aspects of mouse management within Peersoft

From the study of the mouse interface API from any Apple model reference manual, you will discover that more API are offered than those described above. In order to support them from Peersoft a new reason code (10) was set up to cope with those needs as shown within this codlet.

```
10 RE = PEEK (40160) + 256 * PEEK (40161)
20  MOUSE   ON ,1: REM Mouse in transparent mode
30  CALL RE,10,5,1,40,2,23: REM Clamp limits set from (1,2) to
(40,23)
40 X = 1:Y = 2: CALL RE,10,4,X,Y: REM Position the mouse cursor to
(1,2)
50  FOR T = 0 TO 1 STEP 0:S =  MOUSE (2): ON S < 32 GOTO 90:X =
MOUSE (0):Y =  MOUSE (1)
60  VTAB 1: HTAB 1: CALL - 868: PRINT "X:"X,"Y:"Y,"S:"S;: ON S < 128
GOTO 90:T = 1
90  NEXT : MOUSE OFF : END
```

The table below gives the complete list of API calls supported by the generic utility function.

Reason subcode	Arguments	Meaning
2	Xm variable, Ym variable, Sm variable	Populates the three variables with the mouse coordinates as read from the mouse interface. Xm, Ym and Sm must be integer variables.
3	None	Clears the mouse to zero position, used for delta mode position determination.
4	X expression, Y expression	Positions the mouse cursor at the location defined by the X and Y

		expressions.
5	Xmin expression, Xmax expression, Ymin expression, Ymax expression	Sets mouse bounds in a window (Xmin, Ymin) to (Xmax, Ymax)
6	None	Reinitializes mouse cursor position to upper left corner of the clamp window.

ON TIMER GOSUB lineNumber

This syntax scheme of an already existing syntax construct will setup an Applesoft subroutine to handle VBL interrupts. The running of a program (`RUN` Applesoft statement or `RUN/LOAD` <filename> DOS commands) will reinitialize such setting. This command should have been met before a "`TIMER ON,` factor" statement is parsed from the program text.

TIMER OFF

This statement will shut down the VBL dedicated activity from the mouse interface. The VBL interrupt is implicitly deactivated as a program begins running (either with the Applesoft `RUN` statement or the DOS `RUN` command).

TIMER STOP

This statement is used to temporarily stop calling the Applesoft handling subroutine on every VBL related interrupt occurrence so that a interrupt handling Applesoft subroutine could complete before the next event be taken into account within Peersoft. A `TIMER STOP` is implicitly performed as the Applesoft VBL event handling routine is entered and the status is reverted back to "`ON`" as the subroutine exists. Note that interrupts are still processed within Peersoft but the calling of Applesoft routine is temporarily deactivated.

TIMER ON[, factor]

This statement alters the mouse interface by enabling user interrupts when a VBL event occurs. The optional factor argument value allows to only fire the Applesoft handling when factor VBL interrupts has occurred thus allowing to fire such routine only every second (in case factor value is set to 60) instead of every 1/60 sec (according to the native VBL interrupt frequency). This is so by implementing an internal two bytes counter which is incremented from zero upon every VBL interrupt till it reaches the factor specified value. If no factor is specified, then 1 is the default value which means that the Applesoft routine will be called every 1/60th sec.

TIMER function

Two possible values for the argument the `TIMER` function can be called at any time during program flow (on eligible host configuration indeed).

- `TIMER(0)` will return the factor value as specified in the TIMER ON statement.
- `TIMER(1)` will return the current value from the 16bits internal counter.

Codlet using the TIMER related instructions

This codlet displays a Chrono on the display screen as your program perform some computations in the foreground.

```
10  CLEAR : ON  TIMER  GOSUB 100: REM SETS UP THE HANDLING ROUTINE
20 S%= 0: TIMER  ON  ,60: REM ONLY CALLS THE ROUTINE EVERY SECOND.
30  FOR …. : REM MAIN PROCESSING (SPENDING SOME *USER* TIME)
40 VTAB 2: HTAB 1:CALL - 868: PRINT "THE MAIN PROCESSING TOOK "S%"
SECONDS TO COMPLETE"
100 S% += 1: VTAB  1: HTAB 1:CALL - 868: PRINT "SECONDS ELAPSED:"S
%;: RETURN
```

New error messages related to Mouse/VBL handling

Here are the new error messages, such message are catchable as the original Applesoft/DOS 3.3 error handler (cf. `ONERR` statement).

Error #	Message	Description
32	MOUSE HARDWARE NOT DETECTED	When no mouse interface was detected upon Peersoft boot, any use of the API described above leads to such error message.
33	UNSUPPORTED HARDWARE CONFIGURATION	When using an Apple 2, 2+ or unenhanced //e and trying to use an interrupt based feature from Peersoft.
34	UNKNOWN APPLESOFT MOUSE EVENT HANDLER	When a MOUSE ON with a mode involving interrupts is issued as the ON MOUSE GOSUB was not already processed.
35	UNKNOWN APPLESOFT TIMER EVENT HANDLER	When a TIMER ON statement is met as the ON TIMER GOSUB was not alaready processed.
36	ILLEGAL MOUSE MODE	An invalid value was submitted
37	ILLEGAL MOUSE OPERATION	The Apple mouse firmware answered with a carry set (this does not include the SERVEMOUSE API called from the Peersoft interrupt handling routine).

Note to Assembly language authors

Whenever Peersoft processes a `MOUSE ON` or a `TIMER ON` instruction which increase the mouse mode value from a non interrupt mouse mode (value < 2) to an interrupt mode (value > 1), then Peersoft claims the IRQV vector in page 3 ($03FE-$03FF) and stores the original content in a safe place ($9A64). Whenever Peersoft handles some interrupt that it cannot handle itself, then it passes on the control to the original IRQV vector claimant.

On the other end, the original IRQV content is restored back to its original content whenever the mouse mode reaches (usually by a program ending, starting or a `TIMER OFF`/`MOUSE OFF` statement processing) the values 0 or 1 from a greater original value.

As a reminder, the Peersoft interrupt system is only in place on suitable host configurations (Applemouse // interface present with enhanced //e or //c, //c+ or //GS).

Note to Applesoft applications authors

Some Applesoft statements might take an indeterminate amount of time before releasing control to the foreground interpreter loop. I foresee three cases in particular:

- The `WAIT` statement which indefinitely waits for some memory cell to conform to a predefined configuration usually set by an external device like the keyboard or a joystick, an interface card memory address cell ($C0nx range) or a memory byte within the 64K when set by a proper interrupt routine;
- The `GET` statement when the current input flow comes from the keyboard;
- The `INPUT` statement when the current input flow comes from the keyboard;

Peersoft in its current incarnation provides a patched version of the `WAIT` statement handling routine so that whenever an interrupt occurs and the related interrupt mode dictates an immediate attention from Peersoft, then:

- The `WAIT` context is saved;
- The registered Applesoft subroutine is immediately run;
- As the Applesoft routine returns then, instead of returning to `NEWSTT`, it returns to a location where the WAIT context is restored and a branch within the loop is run. The loop can then proceed.

Refer to the `TIMRWAIT` Applesoft short sample demo program for an illustration of the applied patch.

The design of similar patches for the other two usage cases would imply to dig into the many layers of input processing (firmware and monitor) (as I did for a previous utility I wrote named Bananasoft for implementing a software only keyboard buffer w/o the help from interrupts).

Co routines within Peersoft

A word of advice: this section describes the working of co routines features within the 1.4 release of Peersoft. Future releases might expose other API to the external entities, in case this interface evolves or others appear, then this section will be updated accordingly within the document you are currently reading.

From now on, an Applesoft program should be considered as a sequence of consecutives phases: some with active co routines (flows of control running in // and on different parts of the program text) and inactive co routines (when a unique flow of control exists and determines the program behavior).

Except when you previously installed another “multi-tasking” environment in your Apple 2, then every Applesoft application has to be considered as a purely sequential unit of flow on time before the adoption of Peersoft in its current incarnation.

Peersoft provides two ways of doing for both transitions (one for from ”purely sequential” to “active co routines” and the other for the other way round).

Activating the co routines (simplistic sample)

For activating the MT (short for "multi threading" kernel) and the co routines which come along, then a unique way of doing this is given below (minimal sample):

```
5 DIM I0%(127),I1%(127), I2%(127)
10 RE! = PEEK( 40160) + 256 * PEEK( 40161)
20  PRINT "ACTIVE CO ROUTINES PHASE ABOUT TO BEGIN ON LINES 1000,
2000 AND 3000"
30  CALL RE!,4,IT%,I0%,0,0,0,1000,I1%,0,0,0,2000,I2%,0,0,0,3000
40  PRINT "ACTIVE CO ROUTINES PHASE ENDED": END
1000  GOSUB 5000: FOR J0% = 1 TO 2: PRINT J0%;"/";IT%: NEXT
1010  GOSUB 5010: RETURN
2000  GOSUB 5000: FOR J1% = 1 TO 4: PRINT J1%;"/";IT%: NEXT
2001  GOSUB 5010: RETURN
3000  GOSUB 5000: FOR J2% = 1 TO 6: PRINT J2%;"/";IT%: NEXT
3001  GOSUB 5010: RETURN
5000  PRINT "CO ROUTINE #";IT%;" ENTERED": RETURN
5010  PRINT "CO ROUTINE #";IT%;" ABOUT TO QUIT": RETURN
```

Address 40160 contains a pointer to the general utility routine within Peersoft.

The arguments are described in the table below:

Table 2: Arguments for activating the MT kernel

Argument		Description
4		Reason code meaning: I would like to activate the MT kernel with co routines defined by following parm values.
IT%		It is the name of the Applesoft variable (must be simple integer type variable) which will hold the current thread index value from 0 to *NumCoRoutines – 1*. Peersoft updates this value upon every context switch.
1st co routine	I0%	Name of the integer type array which will contain the context storage for the 1st co routine, structure of this array is given in a section below.
	0	This parameter defines whether the co routine has a private error handling routine of its own. This parameter should be considered as a bit string here where, for our purpose only the two lsb interest use. Three values are possible here: 0: implies that no error handling at all while the co routine is the one run by CPU. That means that no segment exists in the context dealing specifically with the error handling, making its size smaller and its store and retrieval faster. Whenever the context is restored, a zero is stored in the ERRFLG flag page zero location. 1: Private error handling which instructs Peersoft to cater for dedicated error handling segment within stored context for this co routine. The co routine should however, execute an ONERR GOTO nnn instruction in its own flow of control. 2: The co routine relies on the status of the "global environment" (ie error handling status as the CALL RE!, 4,... is run), a context segment for dealing with error handling is created iif the ERRFLG

Argument		Description
		(page zero location $D8 meaning an ONERR handler is active) is set upon the CALL RE!,4,... is processed by Peersoft. The role of other bits (b2b7 from the value are described in a subsequent section).
	0	This parameter is the address of a machine language subroutine (ending with a RTS instruction) called whenever the co routine is about to be active (gain the 6502 CPU). The sub routine must not change any register value (cf. Push and Pull 6502 instructions)
	0	This parameter is the address of a machine language subroutine (ending with a RTS instruction) called whenever the co routine is about to release control and the corresponding context be stored in the context storage area (see array I0% description above). The sub routine must not change any register value (cf. Push and Pull 6502 instructions)
	1000	This is the co routine starting Applesoft BASIC line number. Consider that, internally, the CALL RE!, 4... does a GOSUB to this line number upon co routine activation.

Arguments descriptions for 2nd and 3rd co routines are similar in their description as the ones for the 1st co routine's arguments. Up to 8 co routines can be active at the same time.

Activating the co routines (Not so simplistic approach)

Now suppose that a particular co routine needs to have a dedicated environment for text cursor positioning.

The context segment representative of text cursor positioning could be summarized within the table below:

WNDLFT, WNDWDTH, WNDTOP, WNDHGHT for text window setting on the display screen and

CH, CV, BASL, BASL+1 for cursor location within the window.

Deactivating the co routines

Beyond the natural and normal way of returning to a single flow for the Applesoft application (let every co routine return to the statement following the `CALL RE!,4,...` instruction (by using a combination of `RETURN/POP` statements themselves).

The fastest way is based upon a new reason code for the Peersoft general utility: `CALL RE!,5` is the instruction to insert within the code of a co routine (including its error handling procedure, dedicated or shared). Such statement **must** be run as the MTK is active.

Peersoft data structures and hints for performing usual tasks from Applesoft programs relating to co routines

For reference by assembly language programmers, here is the structure of the Peersoft global page and of every integer type array variable used by Peersoft for context storage purposes.

Table 3 : Peersoft global page

Address (decimal)	Address (hexadecimal)	Description
40159	$9CDF	A call to this address will branch to the Peersoft "general utility" routine already described in a previous section. An alternate way is to get the vector stored at (40159+1, 40159+2) and calling it directly (cf. sample Applesoft above)
40158	$9CDE	Peersoft version byte: currently a $15 value is stored at this location (meaning 1.5)
40157	$9CDD	Number of instructions between two context switches (default to 10, setup whenever Peersoft is loaded from disk).
40156	$9CDC	Bit 7 set iif the MT kernel is active. A call to CALL RE!,4,... will set it up. This flag is reset whenever the MT kernel is terminated, usually as the last co routine returns to the global environment.
40155	$9CDB	Number of ticks that the currently running co routine will last before next context switch. At every context switch, Peersoft copies the $9CDD slot into this slot, upon running an Applesoft instruction, the context switch occurs only if the value from this slot, decremented by one, reaches zero.
40154	$9CDA	Bit 7 set if context switch temporarily inhibited while a critical section of code is run by the current co routine.
40167	$9CE7	Bit 7 set and allowing a UDF returning a 16bits integer as a result, could be returned as a signed value or not.

What happens when the co routines are established?

A `GOSUB` stack frame is created in the stack segment of every co routine's context. This GOSUB frame indicates that the return points to just after the `CALL RE!,4,...` Applesoft statement. Obviously, the stack pointer for every established co routine is decremented by the frame size (5 bytes including the GOSUB token).

Peersoft marks a co routine as being completed when, as this co routine is being run by the CPU, the current stack pointer reaches the original stack pointer value taken as the CALL RE!, 4.. Applesoft statement was parsed.

Hints and tips

How to release control to other threads from the current co routine?

Just use the `POKE` statement `POKE 40155,1` just before the location where you want to release control. While parsing the next statement, Peersoft will decrement this value to 0 and thus a context switch will be triggered (saving the current context, and restoring the next active co routine declared within the kernel. Be advised that this could be the same co routine as the current one in case no other is still active.

How to temporarily disable the context switch?

While a critical section of Applesoft code is being run within the current co routine, no context switch should occur in order to let this section of code appear as being atomic. A simple way to fit this requirement is to insert a `POKE 40154,128` statement at the beginning of your critical section

code. In order to reinstate switches for giving a chance for other co routine to flow normally, then use the `POKE 40154,0` statement at the conclusion of this section.

Also, as the decrement operation is not processed while the switching is inhibited, it is a good idea to insert a `POKE 40155,low_value` just in the vicinity of the former `POKE` statement (as I did in my tutorial example Applesoft programs).

Having private variable sets (no collision between co routines)

The current solution I propose is to get arrays of variables with at least one dimension indexed by the context index value. In the tutorials from the disk, I used two arrays (`XH()` and `XV()`) to store cursor data (line and column where cursor lies in two dedicated integer arrays) and all PRINT statements or cursor position setting statements being run in critical sections of code.

Committing suicide or assassination (of other threads)

Beyond the usual way to mark a thread as completed (i.e. using RETURN or POP instructions in order for the stack pointer to reach its initial value), an alternate and more intrusive way would be to force a specific byte from Peersoft memory to $FF value, thus Peersoft will consider the relevant co routine as completed. Here is the code segment which performs just that action.

```
AD =  PEEK (40152) + 256 * PEEK (40153): POKE AD + 8 + IT%,255
```

Where IT% being the current context index implies suicide and IT% being unequal to current context index (but still between 0 and 7) meaning assassination.

Structure of the context storage

Every context is stored within a dedicated integer type array variable (one dimension) which layout is described in the table below

Table 4 : Context Storage layout

Offset	Page zero	Description
Header for housekeeping by Applesoft		
0 and 1	N/A	Name of the array (two bytes)
2 and 3		Offset from the beginning of this array to next array variable or to end of memory area
4		Number of dimensions (must be 1 for Peersoft usage).
5 and 6		Value of first (and last) dimension
Constant segment (general use)		
7	N/A	Offset to stack segment (always populated)
8	N/A	Operation mode for context. B0b1 provides an indication whether the local error handling is in use or not. In case local error handling is in use, whether the global environment is used for such context or not; B2b7 provides options for additional context switch operations. The one being shown within the tutorial is the display cursor backup/restore operations.
Constant segment for monitoring context switches		
9 and 10	N/A	Address of machine language routine to be called whenever the co routine is paged in. This routine must not alter register values from the calling environment (unless pushed on stack) and must return with a RTS (after possible Pull from stack instructions). High byte is $FF if no routine registered.
11 and 12	N/A	Address of machine language routine to be called whenever the co routine is paged out. This routine must not alter register values from the calling environment (unless pushed on stack) and must return with a RTS (after possible Pull from stack instructions). High byte is $FF if no routine registered.
Core segment (always populated)		
13	REMSTK ($D8)	Current stack pointer for this pointer (only byte at offset 8 is meaningful)
14 and 15	CURLIN, CURLIN+1	Current Applesoft line # for the co routine
16 and 17	TXTPTR ($B8), TXTPTR+1	Current text pointer within program text for the co routine
18 and 19	OLDTEXT, OLDTEXT+1	Text pointer of last instruction parsed by interpreter exec loop
Local Error handling segment (optional: see value at offset 8)		
20 and 21	TXTPSV ($F4), TXTPSV+1	Points to the first character of line # as ONERR GOTO statement is parsed.
22 and 23	CURLSV ($F6), CURLSV+1	Line # where the ONERR GOTO is located
24	ERRNUM ($DE)	Error # when an error occurs
25	ERRSTK	Stack pointer as the error occurs (so that RESUME could branch back to the faulty statement)
26 and 27	ERRLIN ($DA), ERRLIN+1	Applesoft line # where the error occurred (so that RESUME could branch back to the faulty statement)
28 and 29	ERRPOS ($DC), ERRPOS+1	TXTPTR pointer of the statement raising the error.
30	ERRFLG ($D8)	Only bit 7 is meaningful here.
Stack segment (variable size)		
<ValueAt Offset 7> and above	N/A in page zero: within hardware page 1	From private stack pointer to global environment stack pointer value, bytes extracted from hardware stack (page 1) from offset given by REMSTK value at offset 8 from this structure) to the REMSTK known as the `CALL RE!, 4, …` was parsed.

Peersoft co routines tutorial

Within the Applesoft program listing below,

- The arrays `I0`, `I1` and `I2` serve as context storage areas useful for switching between co routines;
- The arrays `XH` and `XV` serve as memory place where to store screen cursor locations (horizontal for `XH` and vertical for `XV`) for every co routine implemented here (the number of them being 3).
- Variable `XC` serve as an indicator that the user issued a Ctrl-C keystroke while the program was running. Therefore it is set to a non zero value at line `2901` (part of the shared general error handling routine beginning at line `4000` (see `ONERR` statement at line `2` and part also of the dedicated (i.e. private) error handling for co routine #1 beginning at line `2900`);
- RE holds the address where to call the Peersoft general utility routine with appropriate parameters.

```
1 CLEAR : DEFINT I-N,X: DIM I0(127),I1(127),I2(127),XH(2),XV(2)
2 PRINT CHR$ (4)"PR#0": TEXT : HOME :XC = 0: ONERR GOTO 4000
3 PRINT CHR$ (4)"BLOAD TUTMC": VTAB 1: HTAB 15: PRINT "TUTORIAL 2"
4 XH(0) = 1:XV(0) = 2:XH(1) = 1:XV(1) = 21:XH(2) = 1:XV(2) = 6: DEF FN DR(A) = PEEK
(A) + 256 * PEEK (A + 1): DEF FN AR(CX) = INT (CX * 100) * .01
5 RE = FN DR(40160): POKE 40157,4: REM # OF APPLESOFT INSTRUCTIONS RUN BETWEEN TWO
SWITCHES
6 CALL RE,4,IT,I0,2,0,0,1000,I1,1,768,774,2000,I2,2,774,768,3000
7 VTAB 1: HTAB 1: PRINT "PROGRAM ENDED, PRESS ANY KEY";: GET A$: HOME : END
999 REM FIRST COROUTINE: MONITOR EVERY CONTEXT INCLUDING ITSELF
1000 AD = FN DR(40152):OF = 0:NT = 0:SO = PEEK (AD + 17):SL = 0:LX = - 1: GOSUB
5010: PRINT " RUNNING TASKS STATUS (";SO"/";: GOSUB 5000:XH = XH(IT):XV = XV(IT)
1002 FOR JT = 0 TO 7: ON PEEK (AD + 8 + JT) < 255 GOTO 1003:NT = JT - 1:JT = 7
1003 NEXT JT: FOR J0 = 0 TO 1 STEP 0: GOSUB 1100:JF = 1
1004 FOR JT = 0 TO NT: GOSUB 1200: NEXT JT
1005 J0 = JF: NEXT J0: RETURN
1099 REM
1100 ON PEEK (40157) = LX GOTO 1102: POKE 40154,128: HTAB XH: VTAB XV:LX = PEEK
(40157)
1101 PRINT LX;")";: CALL - 868: POKE 40155,1: POKE 40154,0
1102 RETURN
1199 REM PRINT A CONTEXT CONTENT (JT)
1200 IF PEEK (AD + JT + 8) < 255 AND JT < > IT THEN JF = 0
1201 OF = PEEK (AD + JT + 8) * 256 + PEEK (AD + JT):XV(IT) = 3 + JT:XH(IT) = 1:
GOSUB 5010: CALL 777,OF,JT: GOSUB 5000
1202 RETURN
1999 REM SECOND CONTEXT: PROCESS SOME KEYBOARD INPUT FROM USER
2000 BS$ = CHR$ (8):CU$ = CHR$ (127) + BS$: POKE 49168,0: ONERR GOTO 2900
2001 GOSUB 5010: PRINT SPC( 6);"DIVISION EXEMPLE": GOSUB 5000:LY = XV(IT): FOR J1 =
0 TO 1 STEP 0
2002 XH(IT) = 1:XV(IT) = LY: GOSUB 5010: CALL - 958: PRINT "ENTER NUMERATOR: "CU$;:
GOSUB 2801: ON M$ = "" GOTO 2004:VN = VAL (M$)
2003 GOSUB 5010: PRINT "ENTER DIVISOR: "CU$;: GOSUB 2801: ON M$ < > "" GOTO 2005
2004 J1 = 1
2005 ON J1 = 1 GOTO 2007:VD = VAL (M$):VR = FN AR(VN / VD): GOSUB 5010: PRINT
"RESULT: ";VR;" <RET> TO PROCEED"CU$;: GOSUB 2851: ON XC = 1 OR ES% = 1 GOTO 2004:
GOTO 2007
2006 POKE 40154,128: VTAB 24: HTAB 1: PRINT MO$;
```

```
2007 NEXT : RETURN
2800 REM INPUT SUBROUTINE
2801 GOSUB 5000:M$ = "":LM = 0:ES% = 0: FOR JS = 0 TO 1 STEP 0
2802 GOSUB 2861: ON ES% = 0 AND XC = 0 GOTO 2803:M$ = "":LM = 0: GOTO 2809
2803 ON JS = 1 GOTO 2809: ON A < > 8 OR LM = 0 GOTO 2804:LM -= 1:M$ = LEFT$( M$,LM
+ (LM = 0)): PRINT " "A$A$;CU$;: ON LM > 0 GOTO 2804:M$ = ""
2804 ON A < 31 GOTO 2809:LM += 1:M$ += A$: PRINT A$;CU$;
2809 GOSUB 5000: NEXT
2810 GOSUB 5010: CALL - 868: PRINT : GOSUB 5000: RETURN
2850 REM GET RETURN SUBROUTINE
2851 GOSUB 5000:ES% = 0: FOR JS = 0 TO 1 STEP 0
2852 GOSUB 2861:JS = (ES% = 1) OR (XC = 1) OR (A = 13): GOSUB 5000: NEXT : GOSUB
5010: CALL - 868: GOSUB 5000: RETURN
2860 REM GET KEYBOARD ENTRY
2861 ON PEEK (49152) > 127 OR XC = 1 GOTO 2862: POKE 40155,1: GOTO 2861
2862 GOSUB 5010: IF XC = 0 THEN GET A$:A = ASC (A$)
2863 ON XC = 0 GOTO 2864: PRINT "#ABORTED#!";:JS = 1
2864 ON A < > 27 GOTO 2865: PRINT "<ESCAPED>";:JS = 1:ES% = 1
2865 ON A < > 13 GOTO 2866:JS = 1
2866 RETURN
2900 ON PEEK (222) < > 255 GOTO 2902
2901 XC = 1:A$ = CHR$ (3):A = 3: PRINT CHR$ (7);: RESUME
2902 ON PEEK (222) < > 133 GOTO 2903:EL = FN DR(218): ON EL < > 2005 GOTO 2903:MO$
= "DIVIDE BY ZERO ERROR":J1 = 1: CALL - 3288: GOTO 2006
2903 PRINT CHR$ (7);: GOTO 4003
2998 REM 3RD CONTEXT MAIN ROUTINE, JUST PRINT SOME STAR CHARACTERS
2999 REM AS A BACKGROUND ACTIVITY
3000 FOR J2 = 0 TO 1 STEP 0:J2 = J1: GOSUB 3008
3001 PRINT "*";: GOSUB 5000: NEXT
3002 FOR J2 = 1 TO 4: GOSUB 3008: PRINT MID$ ("OVER",J2,1);: GOSUB 5000: NEXT :
RETURN
3008 XV(IT) = INT ( RND (1) * 15) + 6:XH(IT) = INT ( RND (1) * 40) + 1: GOSUB 5010:
RETURN
4000 IF PEEK (40156) < 128 THEN VTAB 23: HTAB 1: CALL 771: END
4001 ON PEEK (222) = 255 GOTO 2901
4003 XH(IT) = 1:XV(IT) = 23: GOSUB 5010: CALL 771: GOSUB 5000: CALL RE,5
4998 REM STORE CURSOR POSITION INTO CONTEXT AND RELEASE CONTROL TO MT
4999 REM EXPECTS TO BE CALLED WHILE CONTEXT SWITCHES INHIBITED
5000 XH(IT) = PEEK (36) + 1:XV(IT) = PEEK (37) + 1: POKE 40155,1: POKE 40154,0:
RETURN
5009 REM INHIBIT CONTEXT SWITCH AND RESTORE CURSOR POSITION FROM STORED CONTEXT
5010 POKE 40154,128: VTAB XV(IT): HTAB XH(IT): RETURN
```

New way to declare up to 11 user defined functions

Eleven `DEFUSR[n]` instructions and ten `USR<n>` functions, `n` from 0 to 9 have been added to implement this feature.

Simple syntax scheme for usual classic ML routines

You can define such entries for those routines by using the simple syntax below :

```
DEFUSR[n] = <parm1>
```

`n` being optional here, if `n` is omitted, then the preexisting user function vector is considered. The parm1 value is known to be the entry point address of such routine within 6502 memory. That value must be > 255.

As a `USR[n]` is processed within an expression,

- Only argument's value is left in `FAC` (zero page slot) ready for the ML routine to retrieve (unmodified behaviour from classic Applesoft) ;
- Only argument's type is left in `VALTYP` (zero page slot) ready for the ML routine to make use of (unmodified behaviour from classic Applesoft).

More elaborate syntax scheme for ML routines dealing with two arguments

As soon as the ML routine has to deal with two input arguments (instead of just one), then an alternative syntax form must be used for its declaration.

```
DEFUSR[n] = <parm1>,<parm2>
```

Where *parm1* is a running mode for such function whose value here should be 64 (further details in table below) and *parm2* is the entry point ML routine address.

As the `USR[n](arg1,arg2)` is processed,

- *arg1* value is available within the `ARG` (second/auxiliary Applesoft FP accumulator) ;
- *arg1* type (value of `VALTYP`) is kept in the page zero cell at address $BE ;
- *arg2* value is left in `FAC` ;
- *arg2* type is left in `VALTYP` ;

I would suggest that you take a look at the TCUSRFNDEMO.S source code within the archive for samples of how such user defined functions can be built and used from Applesoft.

Elaborate syntax scheme for declaring Procedural functions

According to the number of input arguments such functions can deal with (either 1 or 2), declaring such functions can take either one of the syntax schemes below:

- `DEFUSR[n] = <parm1>,<OutputVariableName>,<InputVariableName>,<beginLineNumber>`
  iif the function only deals with one input argument, *parm1* value being either 128 or 129 (see table below).

- `DEFUSR[n] = <parm1>,<OutputVariableName>,<InputVariable1Name>,<InputVariable2Name>,<beginLineNumber>`
  iif the function deals with two input arguments, *parm1* value being either 192 or 193 (see table below).

Within this release of Peersoft, the variables being used as bridges between the calling and called environments must be numeric (either integer or FP). This might evolve in future releases.

To discriminate between PF declarations and machine language routines entry point declarations, one should play on the *parm1* value, first argument to the `DEFUSR` instruction.

If *parm1* value is > 255, then the `DEFUSR` declares a classic ML UDF entry point, otherwise the table below applies:

Bit #	Meaning when set to 1	Meaning when set to 0
7	Implies a PF declaration	Implies a ML routine declaration
6	Two arguments are expected	Only one argument is expected
1	When declaring a PF, implies that a separate data segment be used when evaluating (function body running). This is called a “dynamic PF” in the document.	When declaring a PF, implies that the data segment will be the same between the caller and callee environments. Such PF are called static.

Some rules should be obeyed in the PF management and more precisely when the PF could or could not be called.

- You cannot call a PF from within a PF;
- When co routines are actives, it is safer to call PF from a uniq co routine;
- You should avoid calling dynamic PFs from interrupt processing Applesoft subroutines.

Some of the above constraints might be obsolete in future releases of Peersoft.

New error messages related to PF management

Error #	Message	Description
38	EMBEDDED PF NOT SPPORTED IN THIS RELEASE	Returned as soon as a PF is called from within the body of a PF.
39	ILLEGAL OP WHILE PF IS ACTIVE	

Some reference for sample functions provided

The *TCUSRFNDEMO* binary file holds the executable code for some ML coded functions I might foresee to use some day. The table below describes each of them (the source file is also provided in the source disk).

Entry point offset within binary file	Description	Remark
0	Computes the GCD of two integers	Input arguments are expected to be FP so that the whole 32bits integer value range (as part of an FP) is processed.
3	Computes the LCM of two integers	Same as for the GCD function
6	Computes the Factorial of an integer expression	Only integers from 0 to 23 are computable. Greater values lead to an Overflow Error exception.
9	Computes the binary OR of two integer expressions	If binary OR returned value bit15 is 1 and a particular configuration parameter (bit 7 of cell at address $9CE7 within Peersoft global page) is set, then 65536 is added to original result. If both b15(value) and b7($9CE7) are set then returned value range is [0, 65535] instead of [-32768, 32767].
12	Computes the binary EOR of two integer expressions	Same remark as above
15	Computes the binary AND of two integer expressions	Same remark as above
18	Return the 16bits value at some 6502 address.	Same remark as above
21	Returns the MAX of two FP expression	Also sets an external indicator flag according to which of the two expressions is the MAX. (cell at address $9CC0 within Peersoft global page). Returned value is 1 iif the second parameter is the max expression otherwise 0.
24	Returns the MIN of two FP expressions	Same remark as above: returned value is 1 iif the second parameter is the min expression otherwise 0.

Storing arrays on expansion memory

From Peersoft v1.6 revision, the application author can now specify the storage location of some of the arrays used by their applications, this by using an alternate syntax within the `DIM` instruction.

On an Apple //c, //gs or Apple //e hosting a 80 col. extended memory card, the line below:

```
10  DIM A%(10,10)#1
```

will reserve a 16bits integer array of 121 user bytes (+ 7 header bytes) within extended 80 col. memory expansion card, the amount of space reserved in main memory is exactly 17 bytes.

Some rules to keep in mind:

- Apart of the richer syntax for declaring arrays due to be located in auxiliary memory, the operation of such arrays is completely transparent to the Applesoft application author. Such arrays could be numeric or strings.
- When an array that is located in auxiliary memory bank is of string type, that would imply that the strings themselves are stored in auxiliary memory too (same bank). That's the reason why the page zero 16bits locations VARTAB, ARYTAB, STREND, FRETOP and HIMEM are handled within the Peersoft routines in auxiliary memory.
- Peersoft is designed to keep the performance overhead to access array elements in auxiliary memory minimal, managing a 1 element cache in main memory array layout.
- Definitive 1.6 release will cope with more memory segments where to store arrays.

FREE function update to cope with multiple memory segments

In order to return the actual free amount per memory segment then the argument has been given a meaningful role here (he didn't have any under plain vanilla Applesoft).

- If the argument's value is numeric 1, then the returned value is the amount of free memory within auxiliary memory bank (after a possible garbage collection);
- Otherwise (other value or value type string), the free function returns (after a possible garbage collection) the amount of free memory within main bank as it always returned.

New layout for an array entry within the array variable memory area

Here is the layout of an arrray structure on plain vanilla Applesoft

Offset	Item	Description
+0,+1		Offset from current cell to next array with array variable area in memory

Offset	Item	Description
+2,+3	Array name	Encoded array name (array type can be determined from bit 7 of both bytes)
+4	# of Dims (n)	Value in range 1..255
+5,+6	# of elements in 1st dimension	Integer order (hi byte then lo byte)
...	...	...
+(3+n+n),+(3+n+n+1)	# of elements in last dimension	Integer order (hi byte then lo byte)
+(5+n+n),+(5+n+n+1)	1st array element value	Slot is either 2, 3 or 5 bytes wide.
...	...	Other elements

Here are the changes applied by Peersoft to the layout of the array structure

Byte at offset +4 has changed its structure, now its layout looks like:

- b0 to b3: # of Dims – 1 (0 means 1 dimension and 7 means 8 dimensions);
- b4 to b5: memory bank where the array elements are: 00 value means 48K main memory and 01 means auxiliary memory bank (within 80 col. Extended memory).
- b6 to b7: subtyping for integer type arrays: 00 for bytes and 01 for 16bits ints (other values are reserved for future use).

The new array variable layout implies that a new garbage collection be put in place to handle such string arrays in main memort, that's why the Fast Garbage Collector from Randy Wiggington is provided, sitting in the language card and transparently called whenever a new variable creation is taking place.

In case “b4 to b5” value is non zero meaning the array content is stored in an extended memory, then the content of structure starting at offset +(5+n+n) from above table is as follows

Offset	Item	Description
+(5+n+n),+(5+n+n+1)	Starting address of sister structure in extended memory	Natural order (i.e. lo then hi bytes)

Offset	Item	Description
+(5+n+n+2),+(5+n+n+3)	# of elements in whole array	Natural order
+(5+n+n+4),+(5+n+n+5)	Offset from sister structure base address to current element cell within extended memory	Used for 1 element cache processing
+(5+n+n+6), ...	Uniq element value cached in main memory	Referred to by offset above within sister structure. Slot is either 1, 2, 3 or 5 bytes

In the same vein, in the case “b4 to b5' value from cell at offset +4 from main memory structure above in non zero, then a sister structure exists in auxiliary memory which holds every element value. The layout is a replicate of main memory structure except that:

- # of dimensions is 1 whatever the actual # of dimensions of array;
- # of elements in this dimension is set to the total # of elements (at offset +(5+n+n+2), +(5+n+n+3) from main memory structure);

The reason for such header fields to be there (instead of just the elements raw values) is for allowing an unmodified garbage collection routine (from ROM) to correctly handle such array (ie. Skipping numeric arrays and processing string arrays the usual way).

Contents