Table of Contents

Larry's ACTION! TUTORIAL#

by Larry Serflaten

A monthly addition to the SPACE club Newsletter originally appearing from January 1995 through March 1996

1. The Beginning

So, you own an ATARI computer, now what? To more than a few slap-happy owners, the answer is a robust;

"Bring on the GAMES!"

Games are programs. Without any new programmers for the Atari, we won't be seeing many new games. (Period!)

Without twisting any arms, perhaps you know someone who wants to learn to program but doesn't have the means to go out and purchase a new Pentium machine with all the books and software needed to begin. You might steer him/her toward learning to program an ATARI!

If you use a little tact, you could point out that all computers have some things in common,(Memory, Address/Data busses, Video display, Keyboard, and a processor to control it all.) also that skills learned on the Atari can lead to a quicker understanding of the procedures needed to program the bigger machines.

With the permission of the Editor, I will attempt to help the fledgling programmer overcome some of the hurdles in learning a Action! This language is similar to PASCAL and C. I like it because it is very fast, compared to BASIC, and is easier to work with than ASSEMBLY or MAC 65.

The Action! manual was not intended to teach the user how to program. It is a reference manual of all the capabilities of the system. I found it difficult to understand at first, which delayed my getting involved with it for some time. The first hurdle to overcome, (for me anyway!) was to make the decision to try to learn it. You might say it was the biggest hurdle because after making that initial decision, all the other hurdles were very manageable.

I intend to follow the Action! manual as a guide to topic progression. In order to program in Action!, you have to have the cartridge.(See below) Chances are you have the manual as well. I would also like to solicit your questions. We are not all on the same level, so using a little space to answer your questions will allow more advanced users to progress. You can send E-mail to me (Larry S) on the SPACE BBS containing your questions. If you are a registered user, I will reply with E-mail. If I think others would benefit from reading your questions, I will include them at the end of future articles.

We have to begin somewhere, so here is a little synopsis of what will be coming in future issues.

First, I will lightly discuss the Editor, Monitor, Language and Compiler to get the real beginners familiar with loading and running Action! source code. Then I will discuss the same topics in finer detail, adding the Library of supplied routines. After that, it?s up to you. If there is interest enough to warrant continuing further with programming algorithms, tricks and other such advanced concepts, I will be happy to comply.

That being the first installment, I hope I have whet your appetite for our next issue.

The ACTION! cartridge is currently available from CSS for $44.95 + $5 S/H. Order line PH# is (716) 429 5639.

2. Hello World

Occasionally, I must refer you to text in the manual. I must also be brief. You will find explanations here, when the manual is a bit hard to understand. I will use the same keystroke and command notations as the Action! manual, whenever possible. Because of different editions of the Action! manual, I will use the section headings when referring to text in the manual. To find where the text is, I suggest you make book tabs that locate each of the Table of Contents' in the different parts of the manual. On these tabs, print the name of the parts of the Action! system that the tables refer to. You may want to make a big tab for the Error codes appendix, that seems to be the tab I used the most!

Also, specifying a file name is dependent upon the DOS you are using. A file name has 2 parts; 1) An 8 letter file name, and 2) A 3 letter file extension. There are a few DOS's that allow sub-directories. To allow for variances in the DOS types, the file NAME will include the both the name and extension. The device ID, and/or each level of sub-directory will be named the file PATH. The manual uses the term 'filespec' and in some cases 'filestring' to include both the file path and file name. Using a Backus_Naur type method of declaration, the relationship between file spec, path, and name is as follows:

```
OPEN(chan, <filespec>, mode, aux2)
where
filespec = <filepath> : <filename>
```

This means you must use a proper file path, then a colon (:), then a proper filename. To help explain this type of symbolism (as used in the manual), filepath could be further broken down to:

```
<device ID> [|: : <sub-directory>:|]
```

Device ID is the one letter identifier for whichever device you are addressing (D for disk drive). To denote an optional parameter, I will use the '[' and ']'. The |: and :| indicate you can repeat whatever they enclose, one or more times.

The declaration from above now means; you must use a proper device ID, then you may optionally include the colon and a sub-directory name, and you can repeat the colon and sub-directory name as is needed (or desired). The manual breaks it all out, so that the construction of the <filespec> would be:

```
<device ID> [|: :<sub-directory>:|] : <filename>
```

These symbols are explained and used in the Language section, but are included here for those of you who may be reading ahead, and to get them explained at the start.

The Editor is that part of Action! that first appears when you boot up the system. The Editor will read and write text files, as well as allow you to create or edit text. If you desire, you can split the text editing portion into two sections, called windows. This feature allows you to read in two files at one time, useful for including portions of one file into another! The Monitor is the part of Action! that gives you control over the rest of the Action! system. Take time to read about the Editor and Monitor in the first few sections of the manual.

Go ahead and 'get your feet wet' by typing in the indoctrinating "Hello World." program found at the start of the manual. (See "How To Write and Run an Action! Program") The manual leads you step by step to help get your first program typed in, compiled, and run. In "Text File I/O" (Editor section) you can read about saving and loading files from the disk.

No questions yet, next month a little help with error correction, including typos the manual has, and typos you may yet make.

3. The First Mistake

Let's begin with your computer on, and the HELLO WORLD program in the Editor. Add or type in the follow text;

```
PROC stop()
  PrintE("End of program.")
RETURN

PROC hello()
  PrintE("Hello World")
  stop
RETURN
```

Now press <CNTRL><SHIFT> 'M' to get to the Monitor. Enter 'C' to compile the program. You will then be exposed to a compilation error. At this point, the program will not run. Note the Error number and open your manual to the Error Code Appendix. You can try to guess what the error is if given the correct line, however, sometimes you are not given the correct line and you must decide what to look for by the description supplied in the Error Code explanation.

If text was displayed in the Monitors' message area, the cursor would be moved to the start of that same line. If no text was displayed, the cursor is right where it was when you left the Editor. In my case, I had no text line displayed in the Monitors message area, only the error number. Your cartridge may be a different version, and act slightly differently.

The manual explains this error as an Illegal assignment. It is also an Illegal function call, for which there is no other error number. In my manual I have added in the words "Jump should be Jump()"

to remind me to look for this type of error too. Other than this omission, I believe all the error code explanations are right.

To correct the error, add the parenthesis after the word "stop" ("stop()") in the 'hello' procedure, then return to the Monitor to compile it again. It should compile correctly now, if it doesn?t, recheck your text to find any mistakes. If everything is correct, go ahead and run it. Everybody makes mistakes, especially when first learning the language. Here are the typos you should be aware of in the August 1983 Action! manual:

: - = This is the line to look for. : + = This is how it should be. : > = Correct the notes for this line

- MONITOR / RUN - Program Execution
  - >RUN "<filespec>" only works when used on programs saved from the Editor. (When it is still in text form.)
  - -RUN PrintE() <RETURN> does not work.
  - +XECUTE PrintE() <RETURN> is correct.

- LANGUAGE / Arithmetic Expressions
  - >TECH NOTE Using '*', '/', or 'MOD' will not work correctly on large CARD values.
  - LANGUAGE / Record Manipulations
  - -TYPE idinfo=[BYTE level,
  - -TYPE idinfo=[BYTE level (omit comma)

- LANGUAGE / Advanced use of Extended Types
  - >newrecord=idarray+(reccount*recordsize)
  - The pointer will point to the next record not the end of the array.
  - >In the main procedure;
  - -"mode=InputB()" won't work correctly.
  - +"mode=GetD(7)" works fine.

- LIBRARY / The PUT Procedures
  - -PROC PutDE(BYTE channel, CHAR character)
  - +PROC PutDE(BYTE channel) will compile.

- LIBRARY / BYTE color
  - >For Graphics(0) and text windows:
  - +Chr. luminance (color number = 1)
  - +Background (color number = 2)

- LIBRARY / PROC Fill
  - >col and row must be the numbers of the lower LEFT corner of the box.

- LIBRARY / PROC SCopy
  - >The description is wrong, DEST should be dimensioned larger than source, or else identify an array with enough space.

While in the Monitor, enter the line:

```
? $B000<RETURN>
```

The '?' requests the Monitor to display the contents of a memory address. My cartridge displays:

```
45056,$B000 = 6 $0136 54 310
```

This means I have version 3.6 which is the last version I am aware of. If yours is 7 or greater, LET ME KNOW

Until next time!

4. Sing-a-Long!

One more aid in correcting errors is the use of the List option during compilation of your program. To use this option, you can go to the Monitor and enter 'O' for the Option Menu. Continue pressing RETURN (4 times) until the prompt "List on?" is displayed, then enter 'Y' <RETURN>. This option tells the compiler to list the program in the Monitor message area, as each line is being compiled. If it does encounter an error, the listing of the program stops in the approximate location of the error. Whatever caused the error may be listed on the screen, plus all the preceding lines that compiled before the error. This can narrow your search down to about 22 lines of text, instead of how ever many lines there are in your program. Using the List option does slow down the compilation process. You can use it dynamically (you can turn it on and off) during the compile process by including a 'SET List=1' (on) and 'SET List=0' (off) in the text of your program. Try this out to see how it performs. Here is a short program to finish off the section on the Action! Editor and Monitor;

```
MODULE
PROC verse1()
  PrintE("You say good-bye,")
RETURN

PROC verse2()
  PrintE("and I say ""hello!""")
RETURN

PROC song()
PROC main()
  BYTE ch=764

  Graphics(0)
  PrintE("PRESS ANY KEY;")
  song=verse1  song()
  do until ch<255 od
  ch=255
  song=verse2  song()
  do until ch<255 od
  ch=255
RETURN
```

This example gives an indication of why the parentheses are needed in any call to a procedure or function. (Remember Error 10 from the last lesson?) Type it in just as it appears, compile, and run it. In this example, the word 'song' is used both as a procedure name and as a variable. The only distinction between the two uses are the parentheses following the variable name:

```
song=verse1      ;  as a variable
song()           ;  as a procedure
song()=verse1()  ;  will not compile
```

Here is another example program:

```
PROC main()
  BYTE ch=764, atachr=763, tmp
```

```
  PrintE("KEYBOARD TEST PROGRAM")
  PrintE("Press any key;     Esc=exit")
  do
    while ch=255 do od
    tmp=ch
    GetD(7)
    PrintF("ch=%U  atachr=%U (^[%C)%E",tmp,atachr,atachr)
   until tmp=28
  od
RETURN
```

The above example illustrates one method of testing for a keypress. In the main procedure, the BYTE variable 'ch' was used to test for a keypress. The operating system from ATARI uses one address to hold the keyboard value of the most recent key pressed. From this location the computer can tell when you have pressed a key, and will then translate it to ATASCII. The address of this location is 764 or $2FC. Action! allows the assignment of addresses to variables, arrays and procedure names. By declaring the variable to be residing at address 764, the program can watch for any keypresses, just as the computer does. When no key has been entered, and after each translation the address will be set at a value of 255.

Location 763 holds the ATASCII value of your keypress, after it has been entered and translated.

Because the GetD(7) statement will reset ch to 255, tmp was used to temporarily store the value so that it may be used in the PrintF statement. You can use this example to compare the ATASCII values with the actual keyboard values.

Next month, Vo-cab-u-lary!

5. Vocabulary

Last month?s PrintF, in KEYBOARD TEST, does not need ^[between the (and %C. They got printed instead of the Esc character.

In the LANGUAGE section of the ACTION! manual is a listing of the vocabulary of ACTION! system. The words and symbols are listed in alphabetical order, I will list them by function;

- DECLARATIONS- Used in declaring the procedures, functions, or variables;
  - ARRAY -Preceded by BYTE, CARD or INT to declare a list of data
  - BYTE -Declares an 8 bit variable
  - CARD -Declares a 16 bit variable
  - CHAR -Same as BYTE
  - DEFINE -Allows substitution during compile
  - FUNC -Preceded by BYTE, CARD or INT to indicate the start of a function
  - INCLUDE-Adds text from a file, to program
  - INT -Declares a 15 bit variable with a sign bit as the 16th bit
  - MODULE -Indicates the start of global variable declarations
  - POINTER-Preceded by BYTE, CARD or INT to declare a variable reference
  - PROC -Indicates the start of a procedure
  - RETURN -Indicates the end of a procedure RETURN(x) the end of a function
  - SET -Alters memory during compile
  - TYPE -Declares a list of mixed data
  - = -Used in DEFINE and SET to assign equality
  - $ -Indicates Hexadecimal notation
  - ^ -Contents of a Pointer
  - @ -Address of a variable

    - . -TYPE reference
    - [ ] -Start / End of data block
    - " -String contents (delimiter)
    - ' -Character conversion to value
    - ; -Remarks (delimiter)

- PROGRAM CONTROL-used to structure or control program execution;
    - DO -Start of program loop
    - ELSE -FALSE condition of IF, ELSEIF
    - ELSEIF -Dependant IF conditions
    - EXIT -Exit from program loop
    - FI -End of IF, THEN, ELSE conditions
    - FOR -Enables loop parameters
    - IF -Conditional expression
    - TO -upper limit of FOR loop
    - UNTIL -Conditional EXIT from loop
    - WHILE -Conditional entrance to loop
    - ( ) -Precedence setting parenthesis

- OPERATORS- Used in program statements to alter or combine variables;
    - AND -Logical AND
    - LSH -Bit-wise left shift
    - MOD -Remainder after division
    - OR -Logical OR
    - RSH -Bit-wise right shift
    - XOR ! -Bit-wise exclusive OR
    - + - * /-Plus / Minus / Times / Divide by
    - & -Bit-wise AND
    - % -Bit-wise OR
    - = -Assigns equality
    - <> # -Not equal to
    - > -Greater than
    - >= -Greater or equal to
    - < -Less than
    - <= -Less or equal to

```
MODULE BYTE ARRAY
Note= [ 32 16 24 16 24 16 40 ],
Delay=[ 30  8 55  6  8  6 60 ]
DEFINE note_count="7"

PROC Wait(BYTE jif)
BYTE rtclk=20
rtcik=0
WHILE rtclk<jif DO OD
RETURN

PROC Play()
BYTE ARRAY Audio(8)=53760
BYTE c
AUDIO(8)=3
Audio(1)=166 Audio(3)=170
FOR c=0 to note_count-1
  DO
  Audio(0)=Note(c)
```

```
  Audio(2)=Note(c) LSH 1 +1
  Wait(delay(c))
  OD
Audio(1)=0   Audio(3)=0
RETURN
```

When you have this typed in, COMPILE it, then if there are no errors, WRITE the text file to disk. You can alter the Notes and Delay times, but note_count must equal the number of elements in Notes and Delay. If you make something good, write the compiled code, and send it to friends who can simply load it from DOS!

Stay ACTive!

6. Overview

From the top, programs in Action! are made up of procedures and functions which are called upon to do portions of a programmed task. Usually, the parts of a program that must be done repetitively, are coded in separate procedures and called when needed. Also, certain tasks that are common among all programs, are also often written as separate procedures and then included in each program as desired. The reading of the keyboard, may be an example of a task that could be written as a BYTE function, and included in any program that needs user input from the keyboard. The Action! system makes re-using routines very easy to do.

Following along with the Action! manual, Language section, the manual uses special notations which you should read about. You might remember seeing them from an earlier issue. Next comes Fundamental Data Types which are BYTE, CARDinal and INTeger type variables or constants. It may help to remember, the variables invariably change during program execution, while constants consistently remain the same. A 'score' variable might always be changing while a constant number '5' will not.

To let Action! know you want to use a named variable, you must declare it using one of the fundamental types.

: BYTE w,x : CARD y : INT z

Both 'w' and 'x' are 8-bit values. They use one memory location to store values in. Their value may be 0-255. The CARD 'y' is a 16-bit variable, it uses two memory locations in the 6502 LSB/MSB format. That is, if 'y' is assigned to address 1536, (Ex CARD Y=1536) then the least significant byte will be in address 1536 while the most significant byte will be in address 1537. CARDs hold any value from 0 to 65535. INTeger variables are 15-bit values, with the most significant bit used as a sign bit. INTegers hold the values from -32768 to 32767.

As seen before, when declaring variables, you also have an option to assign that variable to a specific address. Another option allows you to assign a value to it at compile time.

```
BYTE ch=764    ; assigns an address
BYTE spd=[3]   ; assigns a value
```

You need to use some sort of compiler or numeric constant in the assignment option. Any numeric or compiler constant works! This can lead to some interesting results:

```
PROC SeeFormat()
BYTE lsb,msb
CARD num=lsb

PrintE("LSB/MSB program    0=exit")
do
```

```
  Print("Enter a number >")
  num=InputC()
  PrintF("%U > LSB=%U   MSB=%U%E",
         num,     lsb,     msb)
  PutE()
until num=0
od
RETURN
```

In this program, the variable 'num' is assigned the same addresses that are used for 'lsb' and 'msb'. Once 'lsb' has been declared, its address becomes "fixed", the 'lsb' identifier is assigned a specific address. Because its address is now a compiler constant, it may be used later, where ever a compiler constant is allowed. It?s the fixed address that is used in the declaration of 'num'.

As you can see from above, the compiler usually ignores any spaces you may want to use. An exception to this is while trying initialize a string.

Read the LANGUAGE section of the Action! manual, typing in their program examples. This will get you familiar with using the Editor, Monitor, Compiler and language.

I am still looking to answer your own questions. If you have any questions, send them to me on SPACE BBS, or send a note to the SPACE address including a line with the words 'ATTN: Larry S' on the outside of the envelope.

7. Let's Start Programming

Writing programs is hard work! To get the most out of your efforts, take a little time to plan out your schedule. The chart below will help you organized your program to allow you to create even large programs in a few short hours. Typing and testing may take somewhat longer....

PROGRAM DEVELOPMENT CHART

1. Decide what the inputs and outputs of the program will be.
2. Outline the major tasks that need to be done in the order they are to be done.
3. If there are many decisions to be made by the computer, develop a flowchart.
4. Divide each task into its component parts and look for similarities.
5. Group the similar tasks together to determine which can be combined.
6. Develop a memory map of the computer, showing where in memory the different parts of the program are to reside.
7. Type in/write the MAIN procedure first. Fill out the rest of the program, testing each newly added routine for errors and accuracy.

I will use this chart to write a function that will add two strings of numbers. Assuming I have a game where a player may score in the millions. I want a routine that will increase the score beyond the 65535 limit.

1. For inputs, I will send it the current score, and a string containing the amount of increase: AddTo(score,"500") The output should be placed back into the score string before returning.
2. I must first find the least significant digit of both strings, add these, store the result and calculate a carry if necessary. I must then add, store, and carry the next set of digits, and the next, until there are no digits left to add. Finally, the new value must be placed back into SCORE.
3. There are not that many decisions, the flowchart can wait for another issue.
4. This is a component task: increment. There are repetitive tasks, but they are specifically related to adding two strings. For now, I don't see any advantages to breaking them down, and writing them as separate routines.
5. Nothing to group, its only one routine.

6. SCORE and the other string are provided in the call, I will need a place to hold the results until the operation is done. This string can simply be assigned a location by ACTION!

```
PROC AddTo(BYTE ARRAY s1,s2)
BYTE ARRAY result(15)
BYTE d1,d2,carry,digit,i

d1=s1(0)         ;LSD (digit) of s1
d2=s2(0)         ;LSD of s2

IF d1>d2 THEN    ;Assign LSD of result
 digit=d1+1      ;s1 is longer
ELSE
 digit=d2+1      ;s2 is longer
FI
result(0)=digit  ;Storage string length
result(1)='0     ;MSD just in case
carry='0         ;clear carry
WHILE d1>0 OR d2>0
DO
  result(digit)=carry  ;Handle carry
  IF d1>0 THEN         ;Still more s1
     result(digit)==+s1(d1)-48
     d1==-1            ;Next digit
  FI
  IF d2>0 THEN         ;Still more s2
     result(digit)==+s2(d2)-48
     d2==-1            ;Next digit
  FI
  IF result(digit)>'9 THEN
     result(digit)==-10
     carry='1          ;Calculate carry
  ELSE
     carry='0
  FI
  digit==-1            ;Move to next digit
OD
result(1)=carry        ;Store carry
IF carry='0 THEN       ;Delete leading 0
   FOR i=1 TO result(0)-1
   DO
     s1(i)=result(i+1)
   OD
   s1(0)=i-1           ;New length
ELSE       ;carry=1 so copy as is
   FOR i=0 TO result(0)
   DO
     s1(i)=result(i)
   OD
FI
RETURN
```

It's YOUR turn!

8. Adding To Our Understanding

If you had trouble using the AddTo routine in last month?s issue, it may be due to not supplying enough space to store the score string. Here is a test example that will use AddTo;

```
PROC Test()
BYTE ARRAY score(20)
BYTE i
score(0)=1   ;Initializing score to 0!
score(1)='0
FOR i=0 to 20
DO AddTo(score,"8001")
   PrintE(score)
OD
RETURN
```

This month, we will again use the PROGRAM DEVELOPMENT CHART to make a new routine that will multiply two strings. As you might guess, we will make use of the AddTo routine from last month!

1. Like before, the routine will get two string parameters and store the result in the first parameter.
2. To multiply, the 1st parameter must be added to itself as many times as are indicated in the ones digit of the 2nd parameter. Moving to the tens digit of the 2nd parameter and a value that is ten times greater than what the 1st parameter was, we again add according to the value in the tens place, etc.
3. Flowcharting comes later when there are more conditions to consider....
4. We have a similarity, we need to add one string to another, and luckily it is already available. (Reusable code!)
5. This is only a routine, not too many similarities to try to group them yet.
6. Again we can let ACTION! handle string memory assignments.

```
DEFINE smax="100"
(READ in AddTo in this area)

PROC MultiplyS(BYTE ARRAY s1,s2)
BYTE ARRAY temp(smax)
BYTE digit,value,inc,i

FOR i=0 to s1(0)
DO
  temp(i)=s1(i)   ;Init temp for adding
  s1(i)='0        ;Clear result
OD
s1(0)=1           ;Init result
digit=s2(0)       ;Ones place
inc=temp(0)       ;*10 increment
WHILE digit#0
DO
  value=s2(digit)-48  ;value of digit is
  WHILE value#0       ;the # of additions
  DO                  ;needed
    AddTo(s1,temp)
    value==-1
  OD
  digit==-1       ;Next digit
  inc==+1         ;Increase temp * 10
  temp(0)=inc     ;by adding to length
  temp(inc)='0    ;and storing a new 0
OD
RETURN
```

To help manage the string sizes, I have included a DEFINE statement at the start of this procedure. This statement must manage the string in AddTo also. To allow for this, change the declaration line 'result(15)' to 'result(smax)' in the AddTo procedure.

DEFINE simply substitutes whatever is in quotes with every occurrence of the string listed in the DEFINE statement. The result of this is that the compiler will use 100 every time it finds smax, this makes for easy editing, without having to parse through the entire program looking for places that may need a new value! Changing smax once, effects all other occurrences, pretty neat eh?

Of course we want to see it in ACTION!

```
PROC Test()
BYTE ARRAY sc(smax)
BYTE i
sc(0)=1              ;MUST be initialized!
sc(1)='1
PrintE("Multiplication!")
FOR i=0 TO 15
DO
  MultiplyS(sc,"99")
  PrintE(sc)
OD
RETURN
```

Hey! This could be the start of a huge calculator! Adding subtract and divide, and a temporary register for complex equations, and there it is! Hmmm....

Next month, we get back to BASICs!

9. A Wealth Of Knowledge

If you felt a little challenged these past few issues, don't be alarmed. I want you to have useful examples, which can be used in your own programs, even if you lack the skills to develop sophisticated software. Time spent doing things you know how to do will increase your skills, as well as your library of reusable code! I will supply a few useful routines you can use in your programs, which you can study and learn from. Not to say I am the world?s best programmer, but I do offer help!

We have covered the Editor, Monitor, and Compiler. Open your manual to the Library table of contents. If you have tried to use the BASIC language, you may recognize many of the supplied procedures and functions. In fact, the library is simply a collection of PROC's and FUNC's that may help the BASIC programmer shift to using ACTION! It is the programmers option, to use any of these routines, or none at all. All of them are included in the cartridge.

Turn to the page containing the EOF(8) routine. Lets design a test to see it operate. With your Editor cleared, make a list of 2-3 words and save it with the filename of "D:TEOF". Clear your Editor and type in this test routine:

```
PROC Main()
BYTE c
CLOSE(4)
OPEN(4,"D:TEOF",4,0)
DO
 c=GetD(4) Put(c)
UNTIL EOF(4)
OD
CLOSE(4)
RETURN
```

Compile and run this program. Is the list exactly as you typed it in? Mine has a funny little character at the end. If yours does too, you will benefit from this next routine. Now above the MAIN routine, type in:

```
BYTE FUNC SOF(BYTE a)
BYTE POINTER bp

bp=((a&7) LSH 4) + 835
IF bp^=3 THEN RETURN(1) FI
RETURN(0)

(Main should be here!)
```

Now change the EOF in the Main procedure, to SOF and compile and run it. Isn't that better? Since you now may have a better routine, rename SOF to EOF and change the Main procedure back to EOF. Try it again. Did you notice the funny extra character?

Naming the new routine 'EOF' will in effect replace the EOF array as used in the library. You can do this for any of the routines. The names of the procedures and functions in the library ARE NOT RESERVED names. In fact, this is why a runtime library allows you to make programs that do not need the cartridge. The supplied routines are called instead of the library routines of the same name.

Take the time to look over the library, studying how to call each routine. Write tests for the harder ones to be sure you understand their usage.

For those of you wanting to dig a little deeper into programming, type this in:

```
PROC See()
BYTE i=$E0,j=$E1

i=0
j=i+1
RETURN
```

After compiling this short procedure, use "* See" in the Monitor to see the compiled machine language code.

You can use the '*' (Dump) option in the Monitor to check out how the ACTION! system changes your text into machine language. By typing in short programs, you can manually disassemble the code to help you learn machine language. It will come in useful when you have larger programs that you want to optimize. You can expect to cut out about half of the machine code that ACTION! uses, if you want to do some optimization.

That?s the ACTION! system, next month we take a brief look at the ATARI computer system. After a few key points about the ATARI computer, we can then, finally, get to creating our own library.

10. The Effect Of Change

To program your Atari, you really must understand how the memory is organized, how it is used, and which memory locations cause what effects.

A real good book on this subject is; MAPPING THE ATARI from Compute! Books.

Basically, your computer has RANDOM ACCESS (RAM) and READ ONLY (ROM) memory. RAM is used by the applications to hold program, and user data. ROM is used by the manufacturer to hold the OPERATING SYSTEM (OS), and special purpose chips. The chips may allow user access, but generally you cannot write to memory locations above 40960 ($A000-$FFFF).

The special chips perform several of the functions necessary to talk with the outside world. Pokey handles serial port Input and Output (I/O) and interrupt requests, PIA handles the controller jacks, and ANTIC handles the video screen. The OS contains the code and data needed to operate these chips. The PIA and ANTIC are actually microprocessors, the ANTIC chip is even run by a little program!

The OS uses some RAM to hold the variables it needs to use; margins, colors, cursor position and many other attributes change as the computer operates. Knowing where the OS keeps its variables allows the programmer to alter them to produce subtle and major changes in the way the computer operates.

Using these variables is very easy in Action! Simply by assigning an Action! variable to the address of a hardware register, or OS variable, your program can manipulate them as needed. The OS sometimes uses shadow registers which are memory locations used to hold values used in another register. Locations 704-712 are such registers. You can change these color registers at any time, but the change will not be seen until after a vertical blank interrupt. If you instead change the hardware register, the change will take place immediately:

```
PROC registers()
BYTE ch=764, random=53770,
     wsync=54282, toggle
CARD ARRAY regs=[710 53272]
BYTE POINTER bp
PrintE("Hardware/Shadow    Esc=Exit")
PrintE("Press any key to alternate")
toggle=0
bp=regs(toggle)
PrintE("Shadow")
do
  if ch<255 then
     toggle==!1
     bp=regs(toggle)    ;Change registers
     if toggle=0 then
        PrintE("Shadow")
     else               ;Toggle=1 (!)
        PrintE("Hardware")
     fi
     if ch=28 then EXIT ;Esc pressed
     else ch=255        ;Reset ch
     fi
  fi
  wsync=1               ;Wait for sync
  bp^=random & $E6      ;Dark colors only
  wsync=1
od
bp=regs(0)              ;Restore original
bp^=148                 ;color
RETURN
```

Location 710 is a shadow register of the hardware register at location 53272. The variable wsync is used to synchronize the processor with the display screen. Any time wsync is written to, the CPU is halted until the scanning beam is at the start of a new scan line. This program demonstrates the difference between the shadow and hardware register used to give color to the background. Changing the shadow register will change the background color, only after the vertical blank period (the time when the scanning beam is turned off, going back to the top of the screen). Changing the hardware register will cause a color change in the middle of displaying the screen. The new value will take effect as soon as it is stored (which happens every couple of lines due to wsync). Remove the wsync statements to see a new effect.

As this was written, SPACE BBS was in the process of being moved to a new location. Perhaps the best way to receive your questions is by mail, otherwise, don't hesitate to ask me about Action!

Next month, practical examples!

11. Off And Running

One of my earliest questions was; Why won?t this program work without the cartridge???

I will save you the pain of pulling your hair out and tell you. If you want to write programs that will run without the cartridge installed, you must follow these 5 rules;

1. No errors. Any error will halt your program. You can tell if your program has an error by running it with the cartridge installed. If it works OK, the Monitor message area will be clear when you exit from your program.
2. No library calls. You have to supply all the necessary code to run. Any call to the library will generate an error when run without the cartridge.
3. No more than 2 bytes of parameters in any function or procedure call. More than 2 requires a little parameter saving routine found in the cartridge. That?s 2 BYTEs or 1 CARD and no more.
4. No MOD, multiplication, or division. You can add and subtract to your heart?s content, but you can't try to multiply. These also call a cartridge routine.
5. No shift operations using a variable, or using a constant larger than 4.

If you follow these rules, your program should run without the cartridge. The two rules I have had trouble with are 2, no library calls, and 4, no multiplication. How can you read the keyboard if you can?t use OPEN(4,"K:",4,0)? As shown earlier, you can monitor address 764. How about opening a file? Offsets depend on using multiplication for tabular data....

Here are a few solutions to get around the need for the cartridge. Of course you have to write programs that won?t cause an error, number 1 is rather evident. Number 2, as you saw earlier, you can write your own routines and use them in place of the library calls. For the few that you will need, this is a must. To eliminate rule 3, you can store and retrieve the needed parameters in an array, passing only the address of the array, and picking out the parameters in the called routine. Number 4 is hard to eliminate, except by writing the code needed to multiply or divide two numbers. The same with rule 5, if you must shift according to some variable, you will have to write code or a routine to do the entire shift using no more than 4 shifts at a time.

A LSH or RSH must be followed by a constant number, but that shift can be in a FOR/NEXT loop. If the needed constant is larger than 4, two lines that total the number needed may be used:

TO CODE THE STATEMENT: USE: X= X LSH Y FOR Z=1 TO Y DO X==LSH 1 OD X= X LSH 7 X==LSH 4 X==LSH 3

To help you get started on your very own library, I have adapted the multiplication program found on page 171 in ATARI ROOTS to Action! The major changes include the storing of the Accumulator and X register to $E0 and $E1 respectively. Action! expects the value of a function to be returned in location $A0 and $A1, these locations were used instead of $C0, $C1. This function will multiply two BYTE values and return a CARD value. Often that is enough to squeak by in a pinch!

```
CARD FUNC MultiplyB=*(BYTE a,x)
[$85 $E0 $86 $E1 $A9 $00 $85 $A0 $A2
 $08 $46 $E0 $90 $03 $18 $65 $E1 $6A
 $66 $A0 $CA $D0 $F3 $85 $A1
 $60]  ;$60 is a RETURN command

PROC Test()
```

```
BYTE i,j
CARD z
FOR i=10 TO 200 STEP 15
DO  FOR j=0 TO 200 STEP 50
    DO  z=MultiplyB(i,j)
        PrintF("%U*%U=%U%E",i,j,z)
OD  OD
RETURN
```

The '=*' after the word MultiplyB simply means to omit the allocation of memory for the parameters. Action! passes the parameters using the accumulator and X register. No reference was made in the function to either 'a' or 'x' so there is no need to reserve memory for them. The function obtained the values directly from the accumulator and X register.

Although the MultiplyB routine will run without the cartridge, the Test routine calls a library procedure. Guess what?s coming next issue!

12. Kilroy Strikes Again!

These months sure take a long time, don't they? This month we make use of the old MultiplyB routine and add a little more to your library!

```
MODULE
CARD savmsc=$58,x=$55
BYTE y=$54

CARD FUNC MultiplyB=*(BYTE a,x)
[$85 $E0 $86 $E1 $A9 $00 $85 $A0 $A2
 $08 $46 $E0 $90 $03 $18 $65 $E1 $6A
 $66 $A0 $CA $D0 $F3 $85 $A1 $60]

BYTE FUNC AscToInt(BYTE chr)
BYTE i
i=chr&128             ;Inverse bit...
chr==&127             ;Gone!
IF chr<32             ;Conversion
 THEN RETURN(chr+64+i)
ELSEIF chr<96
 THEN RETURN(chr-32+i)
FI
RETURN(chr+i)         ;Implied ELSE

PROC CLS()
BYTE POINTER bp
FOR bp=savmsc to savmsc+959
DO bp^=0 OD
RETURN

PROC EchoS(BYTE ARRAY sa)
BYTE ARRAY dst
BYTE i
i=0
dst=MultiplyB(y,40)+savmsc+x-1
DO
 i==+1
 dst(i)=AscToInt(sa(i))
UNTIL i=sa(0)
OD
RETURN
```

You might want to save this portion for later use, then type in the Test program to see it in use. These and the test program will run, in object form, from DOS! Make sure, then save them.

MODULE simply lets the compiler know that you are declaring variables that you want available to ALL FOLLOWING routines. Here we declare a variable to use the systems pointer to screen memory (savmsc), the cursor X position and, cursor Y position.

MultiplyB, Hello! Reusable code is FUN!

AscToInt is a routine that changes ASCII to Internal code. This code is used by the system to determine what characters to display on the screen. Why the people at Atari didn't simply use ASCII is not a big concern NOW! You can use the next routine to play around with different values. It basically strips off the inverse bit, then calculates the new value needed, and adds back the inverse bit before returning.

CLS is my ClearScreen routine. Changing the 0 to some other value will fill the screen with a different character.

EchoS will duplicate whatever is in quotes including non-printable characters. To make it as fast as possible, I did not include any error checking or alter X or Y. With 40 bytes in a line, 'dst' has to be Y*40 bytes from the top (savmsc). It also has to be X number of spaces from the left edge, then offset by 1 to align with the incoming string. If not offset, then 'dst(0)' would be at the X and Y position, but 'sa(0)' is not supposed to be printed, it holds the length of the string. The first byte of the string is actually in sa(1). Offsetting allows 'dst(1)' to be at the correct X and Y position, thereby setting up a means of direct (1 to 1, 2 to 2, etc.) assignments.

Using these routines as examples, perhaps you can write your own Echo, EchoB, EchoI, and EchoC. I used echo because, I saw the name used in another program, to do about the same thing, and because it keeps it separate from Print and its options which provide error and boundary checks. Of course you'll want to test them:

```
PROC Test()
BYTE ch=764
CLS()
X=11 Y=10
EchoS("KILROY WAS HERE!")
x=4 y=12
EchoS("And he wants you to press a key!")
DO UNTIL ch<255 OD
x=2 y=16
ch=255
RETURN
```

Remember to set X and Y each time!

13. Details, Details, Details!#

Much of computer programming involves the use of expressions. Action! supports arithmetic, bit-wise and, relational expressions. Arithmetic operators, as their name implies, perform arithmetic functions. Bit-wise operators are used to manipulate values on a bit-by-bit level. Relational operators allow for combining variables or simple expressions to form much more complex expressions.

Arithmetic operators (*, /, +, -, MOD) are fairly self evident. Their use is the same as normal arithmetic! To save space, and your valuable attention, I won?t try to explain how to use them!

- The 'and' operator '& is often used to mask bits in a register, or limit the number of bits in an operation.

- The 'or' operator '% is often used to add bits into a register.
- The 'xor' operator '! has several uses, to toggle bits, subtract, and may be used to add and remove bits in a register.

Masking is produced by 'and'ing a value with another value which contains each bit set that is desired in the result: : turn==+1 : turn==&3 > insures turn will only count from 0 to 3 and then wrap around back to 0 again. Using a mask value of 10 will not limit the count to 10. It will limit turn to be either 0,2,8 or, 10, if that was desired....

The following is an example using a lot of bit-wise manipulation, it converts only alpha-numeric characters to their lower case normal video equivalents:

```
BYTE FUNC ToLower(BYTE a)
BYTE t
t=a&$1F
IF (a&$70)=$30 THEN
   IF (a&$0F)<10 THEN
      RETURN (a&$3F)     ;Its a number
   FI
ELSEIF (a&$7F)<$41 THEN  ;Too small
ELSEIF t>0 AND t<$1B THEN
   RETURN(t%$60)         ;It's a letter
   FI
   RETURN(32)            ;Too large
PROC TEST()
BYTE A,B
A=0
DO  B=TOLOWER(A)
    PRINTF("  %C%C = %C%E",27,A,B)
    A==+1
UNTIL A=255 OD
```

These are the characters that will be accepted;

```
               NORMAL       INVERSE
   Numbers   $30 - $39    $BO - $B9
   Capitals  $41 - $5A    $C1 - $DA
   Low case  $61 - $7A    $E1 - $FA
```

You might know that the only difference between normal and inverse is that bit 7 is set for the inverse video characters. Once this bit is stripped away, we can limit our efforts to the values under the NORMAL column. The high nibble (a nibble is 4 bits) of every Number is a 3. We can test for that, then a test to be sure the low nibble is 9 or less will insure we have a number to work with. Numbers need no translation, once we have one, we can simply pass it back. Returning a&$3F will strip off the inverse bit of the incoming variable.

There are several characters between 0 and $41, To insure no attempt is made to translate them, a simple 'less than' test will eliminate them. Since the next statement after the IF/FI is a RETURN(32), valuable space can be saved by letting control 'fall through' out of the IF/IF condition as opposed to including a RETURN(32) after the THEN.

We know the alphabet has only 26 letters. In both CAPS and low case, the letter A has a lower nibble value of 1. It stands to reason that the value for Z must be 25 steps up from the value of A. 26=$1A, so each bit of the low nibble is needed along with one bit of the upper nibble. The command t=a& $1F strips away all attributes from 'a and gives us a value in the range of 0 to 31. A previous test has taken care of values too low to use, so we know this is going to be a letter. If this value is equal to or between 1 and 26, it must be a letter. That test is performed, and if true, the value $60 is added to it to bring it up to the range of lower case letters.

Next month; RELATIONAL operators.

14. Deal Me In!

```
DEFINE DECKS="4"

MODULE
BYTE ARRAY shoe(52)
;INCLUDE: (From my tutorial #12)
;Global variables; savmsc
;                  x, y
;Routines;         MultiplyB
;                  AscToInT
;                  CLS
;                  EchoS
PROC Echo(BYTE a)
BYTE POINTER dst
dst=MultiplyB(y,40)+savmsc+x
dst^=AscToInt(a)
RETURN

BYTE FUNC PickCard()
BYTE crd,r=53770,vc=54283
DO
  crd=(r&$38)%(vc&$07)
UNTIL crd<52 AND shoe(crd)<DECKS
OD
shoe(crd)==+1
RETURN(crd)

BYTE FUNC ShowCard(BYTE a)
BYTE ARRAY pip="A23456789TJQK",
           suit=",.p;"  ;(Use Control key)
BYTE v,s

s=(a&3) +1             ;Mask out suit
v=(a RSH 2) +1         ;Remove suit
Echo(pip(v)%128)       ;Display card
y==+1
Echo(suit(s)%128)
y==-1
IF v<10 THEN           ;Calculate value
   RETURN(v)
FI
RETURN(10)

PROC Main()
CARD tot,cnt
BYTE i,ch=764

FOR i=0 to 51          ;"Shuffle"
DO shoe(i)=0 OD
tot=MultiplyB(DECKS,52);Total cards
cnt=0                  ;Init counter
DO
  CLS()                ;Clear screen
  y=O                  ;Init Y
  DO
    x=3                ;Init X
```

```
    DO
      i=PickCard()
      ShowCard(i)
      cnt==+1          ;Increment counter
      x==+2            ;Next column
    UNTIL cnt=tot OR x=37
    OD
    y==+3              ;Next row
  UNTIL cnt=tot OR y=21
  OD
  x=11 y=23
  EchoS("Press any key...")
  ch=255
  WHILE ch=255 DO OD   ;Wait for keypress
UNTIL cnt=tot          ;All cards out
OD
CLS()
ch=255
RETURN
```

This month you get a full blown program to shuffle cards and display them on the screen. This entire program can be run from DOS. If you have not written an Echo procedure, I have supplied one. You must supply the routines you typed in from issue #12. When typing in the ShowCard function, use the card characters in the suit string. I used printable characters so that our editor could easily print out the program: You must hold down Control when typing the characters between quotes.

Some casinos use 4 decks in what they call a shoe, for their Blackjack tables. Other games use less. DECKS is given to allow for an easy method of determining how many decks are to be shuffled together.

The first relational operator is in the PickCard function. In 4 decks of cards, there are only 52 different cards, and only 4 of any one card. Both conditions must be met before the card is accepted. The next relational operator is in the Main procedure. I know I can?t draw more cards than are in the shoe, nor do I want to attempt to print a card somewhere off the screen. At the start of the loops, X or Y is initialized. I may or may not run out of cards while in either of the loops. As you can see, if cnt=tot, then every loop will be exited from. The statement; IF cnt=tot THEN EXIT FI, would work. I like keeping things small, adding another IF condition would create the need for additional code.

Further discussion coming next month.

15. A Bit Of Digression

Now that you have fully scrutinized the nifty routines I included in last month?s issue, you may have noticed that ShowCard was a function, yet used as a procedure. This is completely acceptable. When used this way, the value returned is ignored. You can use the ShowCard function to find out the value of the card. Aces thru Tens count as their pip value, while face cards count as ten. If need be, the card itself can be used to determine suit, in the same manner as ShowCard.

You might also wonder why the routines that were just a few lines long, were not simply written into the Main procedure. One reason is that Action! was designed to make use of RE-USEABLE code. Any task that may be needed repeatedly should be in its own routine. There is a limit to how many routines you can have, namely 1024 bytes for global variable and routine names. The smaller names you use, the more of them you can use. I have written several large programs and have yet to run out of room for the names. I usually end up pressed for memory to hold all the needed code. A second reason is that it makes for more readable code. A different programmer may want to make use of your code. By keeping routines small, focused, and well documented, other programmers may be

better able to reuse your code. It is much easier to read when there are few global variables, and each routine uses only the variables it needs to do its task.

You may think Echo was a small routine, believe me, it isn't. Translated to ML (Machine Language) it grows dramatically. You can benefit from substituting your own optimized ML code for the tasks that are repeated often. To show you how much difference there can be, I have supplied an optimized ML version of the AscToInt function. This function is used several times in one EchoS call. Each byte of code we save counts as one cycle for each letter in a string of characters. This savings will result in increased response time.

Load the old AscToInt routine into your Editor, compile it, and while still in the Monitor type; ? AscToInt <RETURN>. Record the left most number to paper. Now type; `* AscToInt <RETURN>` and get ready to stop the listing using the <CNTL><1> key combination. Stop the list when you see a diamond after the equals sign, this is the RETURN byte. Record the leftmost number here and subtract the first number from it. I find it to be 48 bytes long, whereas, this new function does the same thing in only 27 bytes. That works out to about 21 EXTRA cycles needed in the code written by Action!

Assuming a 19 byte string, on only a half of a line of text, 400 cycles can be saved using the new routine. Print just three lines of text and you can expect to gain over 2000 cycles!

```
BYTE FUNC AscTolnt=*(BYTE a)
[$85 $E0     ;STA $E0       Save Accu.
 $29 $80     ;AND #$80      Mask out
 $85 $E1     ;STA $E1       inverse bit
 $45 $E0     ;EOR $E0       Gone!
 $C9 $20     ;CMP $20       a<32
 $B0 4       ;BCS +4  >+    Jump if false
 $09 $40     ;ORA $40  |    add 64
 $90 6       ;BCC +6 >-|-+  Jump to RTS
 $C9 $60     ;CMP $60 <+ |  a<96
 $B0 2       ;BCS +2  >--|-+Jump if false
 $E9 $1F     ;SBC $1F    | |subtract 32
 $05 $E1     ;ORA $E1<---+-+add Inv bit
 $85 $A0     ;STA $A0       Still in Accu.
 $60]        ;RTS
```

I listed it in this form for those of you trying to optimize your own code. You can compare the two routines, which operate in much the same fashion. One bit of razzle-dazzle, you may notice I didn't clear or set the carry flag, even though I performed branches that depend upon it. The CMP instruction affects the carry, if operand >= accumulator, carry is set. Also 'SBC $1F' supposes it needed a borrow (Carry is clear) so it takes one away also. The total subtracted from the accumulator is 32 ($20).

Zero page memory, used because of its high access speed, is not hard to find. Making efficient use of it is another matter. Action! uses $A0-$A1 to store returned values from functions, and, $A3-$AF to Store parameters. I use $E0-$EF.

Next month, flowcharts!

16. Caught In The Flow

Many people know how to use a map. Fewer people know how to make one. The same could be said for a newspaper, toaster, baseball glove, microwave, or a flowchart. They are all tools, used when we want to do a specific task. When it comes time to order and list all the logical steps and branches of a new program, a flowchart is your tool to gain perspective.

To demonstrate my point, I will play the role of plant manager, you are my shipping process engineer. We are about to tool up for 12 different products of merchandise. Our automatic shelves will accept any of the items from the end of a conveyor. You are to design the conveyor system that will move each item to its proper shelf. The products will arrive from the plant on one conveyor in random order. Each of the 12 products are differentiated by either color, size or, weight. You get to use scanners, scales, gates, and conveyors to build the warehouse storage system.

Scanners measure color and size, scales measure weight, gates divert the box to another conveyor if the (scan/weight) test fails. This information, combined with a complete description of the 12 products, will provide you with the main reason why flowcharts are used. If you attempt to submit a design, you will have to have a record of it for others to inspect. You need a piece of paper to express such a large complicated system. A flowchart is simply a representation of the complete design, drawn out on paper.

Of the 12 items: : 3 are black, 3 are red : 3 are blue, 3 are black and red

(Scanners designed to detect black will pass the black boxes, and the black and red boxes also. The same for scanners that detect red, will pass both the red boxes and the boxes that are black and red.)

- Black boxes are small, medium, or large. Black items weigh 10 Lbs.
- Red has small, medium, and large boxes. Red items weigh 10 Lbs.
- Blue has 1 large and 2 medium boxes. One of the medium blue weighs 20 Lbs, the other blue boxes weigh 50.
- Black and red has 2 small and 1 med. One of the small black and red items weighs 20 Lbs, the others weigh 50.

This brief description should give you a different size, weight or color for each of the twelve products.

You must now devise a path for the items to follow using as few scanners, scales, and gates as possible, making sure all the items get transported to a shelf and that no shelf has more than one type of item stored on it. Once you have given this some thought, try to determine the least amount of gates, scanners, and scales you would need to do the job. Then see if you can reduce your plan to using only that amount.

The perspective you gain when you build your flowchart (floorplan) is included in the ability to conceptualize each action needed to accomplish the desired outcome. The first action needed in the floorplan is a scanner and gate to divide up the incoming boxes of product. You can label that action as a test and branch, but what test would be best suited to be the first test? There are several variables, and perhaps a few different solutions, what is the first thing to consider? If this sort of problem interests you, perhaps you will make a good programmer.

Most flowcharts consist of input/output blocks (slanted rectangles), decision blocks (diamonds), processing blocks (rectangles), termination or beginning blocks (ovals), and the lines and arrows that connect them all. You can use this system to solve a problem like the one above, or you can devise your own system. When you have completed the flowchart, you have created an algorithm, a special method for solving a problem. Programs typically contain many algorithms, because they often do many things. The test and branch from above is a decision type operation. Input from the plant, of course, is input, with lines as conveyors. The (automatic) shelf could be thought of as a save or store process. Whatever works for you, may be useful!

When a user has to wait for the computer to finish processing, the computer is most often being slowed down by many iterations in a loop structured algorithm. The programmer must use a good algorithm, coupled with the speed of machine language commands, to reduce the wait time. Moving from flowcharts to actual machine code is part of the programming process and is the subject of

the next issue. Advanced users of ACTION! might want to "look in" and give me feedback and/or assistance!

17. Victory

From the last issue:

The 12 different boxes of product need 11 gates to provide 12 conveyors to the 12 automatic shelves. My set of eleven tests began with a scale weighing out any package less than 15 Lbs. If your solution has something different, that is perfectly acceptable, provided it meets the needs of the job. Basically the first test should divide the group of products roughly in half, but it is not mandatory. The important part is that it performs as required, and you can check that by using your flowchart to see where each box would get diverted to. The criteria was that the boxes should end up on separate shelves. You might imagine the supervisor may want to invest more cash into the system and ask you to add in the needed equipment to reject certain boxes which fail in a unique way, and return all rejects on a new conveyor to the plant.

Such is the way it goes in programming, just when you think your done, somebody shows up and wants to change the program. A good flowchart helps in modifying an already complete program. With a flow chart, the programmer can make necessary changes to the algorithm, and can quickly add in the needed code in the proper spot. This is just a reminder to always document your programs, either by the use of a flow chart, or in the source code itself. You never know when you may decide to add more stuff to a finished program.

Moving from words in a flowchart, to code on the computer can be made easy to do. The more details included in a flowchart, the easier it is to translate into code. Flowcharts also help in optimization of algorithms. Careful inspection of a flow chart can reveal redundant loading of a variable, or testing of a value, that may be consolidated or reduced. Looking at the whole process, might give insight to a different algorithm, or method, that will do the same thing even quicker. Sometimes a better algorithm will be enough, and should be attempted first, but there are occasions when assembly is the best choice. Remember you need only to rework the code that causes a delay, or that will make a perceptible difference. There is no big advantage in optimizing the part of a program that is already performing well.

As if to fly in the face of good advice, look back at issue #12's procedures. Here we had a few good routines that were made even better by translating part of them to machine code (AscToInt). EchoS has a loop in it, one of the most basic programming structures, and the most time consuming. Translating this routine to machine code will provide an example of the loop structure as it appears in machine language.

First the flowchart:

1. Save STRING address
2. Calculate DESTINATION address
3. Initialize counter
4. Load byte from STRING
5. Save byte to DESTINATION
6. Decrement counter
7. Test for end condition
8. Loop to "4" until done
9. RETURN

Now the code:

```
PROC EchoS=*(BYTE ARRAY S)
   ;1. Save STRING address
```

```
[$85 $A2       ;STA $A2  Save S LO
 $86 $A3       ;STX $A3  Save S HI
   ;2. DEST=(Y*40)+SAVMSC+X
 $A5 $54       ;LDA $54  Y pos
 $A2 40        ;LDX #40
 $20 MultiplyB ;JSR MultiplyB
 $18           ;CLC
 $A5 $58       ;LDA $58  Screen LO
 $65 $55       ;ADC $55  X pos
 $65 $A0       ;ADC $A0  (Y*40) LO
 $85 $A4       ;STA $A3  DEST   LO
 $A5 $59       ;LDA $59  Screen HI
 $65 $A1       ;ADC $A1  (Y*40) HI
 $85 $A5       ;STA $A4  DEST   HI
  ;3. Init counter
 $A0 $00       ;LDY 0
 $B1 $A2       ;LDA ($A2),Y
 $A8           ;TAY
 $88           ;DEY
  ;START=START+1  S(0) is string Length!
 $18           ;CLC
 $E6 $A2       ;INC $A2
 $D0 $02       ;BNE 2
 $E6 $A3       ;INC $A3
   ;4.  LOOP start, load byte
 $B1 $A2       ;LDA ($A2),Y
 $20 AscToInt  ;JSR AscToInt
   ;5. Store byte
 $91 $A4       ;STA ($A4),Y
   ;6 Decrement counter
 $88           ;DEY
   ;7 & 8. Test and Loop until done
 $10 $F6       ;BPL (LOOP)
   ;9. RETURN from whence we came
 $60]          ;RETURN
```

As you can see, most of this procedure is spent setting up the parameters for the loop. The loop itself is quite small, only 10 bytes (Two bytes are used to indicate where AscToInt is located). This loop can still be optimized further, if desired, by moving all the code from the AscToInt routine to its proper place in the loop. This results in eliminating the need for jumping (using time to store values on the processor stack and in the program counter) which is time consuming when done repeatedly. The code from the AscToInt routine is needed for this loop to function, so moving it 'in line' will make the loop larger, but will actually reduce the execution time.

Were you wondering how to call a procedure using machine language? You can also call ACTION! routines that you have written. ACTION! passes variables using the A and X registers, and returns function values in $A0 and $A1.

18. Breaking The Record!

BYTES, CARDS, POINTERS, and ARRAYS can be used for most of your programming. There comes a time when you must work with a large amount of data, and a mixed set of BYTES, CARDS, and ARRAYS. At times it makes sense to organize your data so that it may be easily processed, other times it makes just as much sense to organize your data for optimum use with files on a disk. Records are used to help in both of these areas.

Records allow a programmer to group the fundamental data types together (BYTES, CARDS, and INTEGERS). The grouping must be declared before it is used. The records become real useful when

you set up a block of memory to hold an array of records. This is no more than a BYTE array, which is divided into individual records, one record right after the other. Each record can be accessed using a record POINTER, which steps through the array, in steps the size of one record.

```
TYPE DAY=[BYTE hi,lo   ;Record FORMAT
          INT change]

BYTE ARRAY memory(400) ;Records storage

DAY POINTER temp       ;Record ID

DEFINE RECORD_SIZE="4" ;2 BYTES + 1 INT
DEFINE MAX_DAYS="9"    ;99 max!

PROC Make_Record()
BYTE i,r=53770,oldtemp
INT chg

  temp=memory             ;# 1a

  oldtemp=75
  FOR i=0 TO MAX_DAYS
  DO
    chg=r&7
    chg==-4
    temp.change=chg
    temp.hi=oldtemp+chg
    temp.lo=temp.hi-5-(r&3)
    oldtemp=temp.hi
    temp==+RECORD_SIZE    ;# 1b
  OD
RETURN

PROC SHOW()
BYTE i

Make_Record()

Graphics(0)
PrintE(" ")

printE("DAY   High   Low   Chg")

  FOR i=0 TO MAX_DAYS
  DO
    temp=memory+(RECORD_SIZE*i) ;# 2
    PrintF(" %U     %U     %U    %I",
       i,temp.hi,temp.lo,temp.change)
    PrintE(" ")
  OD
RETURN
```

You can read your manual to get further information on declaring records. This program is nothing special, it creates a list of each day?s high and low temperature and indicates the change of the high temp from one day to the next. Take note, there are two methods used to access the records. Method #1 sets 'temp' to the start of the storage area (1a) and steps through each record by adding RECORD_SIZE to 'temp' (1b) each pass through the loop. Method #2 calculates the proper location of each record (#2). This random access method of locating each record is simple to use and works well for records that reside completely in memory.

Because each record can be accessed in a uniform manner, the need for special data handling code is reduced. To give you a little practice in using records, try to write a routine that would allow you to enter the data for this program. You can have the user input the high and low temperatures, and the computer calculating the amount of change in high temperature from the previous day. When you have that routine working, write a program to keep track of the time of the sun rising and setting each day. Also keep track of how many hours of daylight there are in each day and the change in daylight hours from one day to the next. The more practice you get, the better you become.

Records are a very useful tool. As will be shown next month, they can be used to provide an easy means of working with data objects (the manual calls them 'virtual records'), we're going to call them worms!

19. An OBJECT Lesson

```
INT xx=$E0, yy=$E2   ;DIR parameters
TYPE worm = [BYTE X1,X2,X3,X4,
                  Y1,Y2,Y3,Y4
             INT  DX, DY]
DEFINE SIZE="12"
DEFINE COUNT="80"   ;1 - 99 MAX WORMS
DEFINE SPEED="250"  ;0 - 255
worm POINTER wm
BYTE ARRAY memory(1200),
           mask=[$7F $BF $DF $EF
                 $F7 $FB $FD $FE]
CARD ARRAY screen(96) ;PLOP/FIND table
;----------------------------------------
;         ( Clip and Save!)
PROC PLOP(BYTE PX,PY) ;Faster PLOT
BYTE ARRAY ROW        ;command
BYTE XB,XV,PM
ROW=screen(PY)   XB=PX RSH 3
XV=PX &7         PM=COLOR LSH (7-XV)
ROW(XB)=(ROW(XB)& mask(XV)) % PM
RETURN

BYTE FUNC FIND(BYTE PX,PY)
BYTE ARRAY ROW        ;Faster LOCATE
BYTE XB,XV,PM         ;command
ROW=screen(PY)   XB=PX RSH 3
XV=PX & 7        PM=mask(XV)!255
RETURN(ROW(XB)&PM)
;----------------------------------------
PROC PICK_DIR()
BYTE R=53770
  xx=0 yy=0
  IF R<20 THEN      yy=1
  ELSEIF R<60 THEN  xx=1
  ELSEIF R>175 THEN xx=-1
  ELSEIF R>225 THEN yy=-l
  FI
RETURN

PROC START_UP()
BYTE R=53770,I
CARD SCRMEM=88
;---------Clip this too-----------
 GRAPHICS(6+16)     ;160 x 96, 1 color
```

```
 screen(0)=SCRMEM   ;Setup array for
 FOR I=1 TO 95      ;PLOP and FIND
 DO screen(I)=screen(I-1)+20 OD
 COLOR=1            ;Border
;--------------------------------
 PLOT(0,0)      DRAWTO(159,0)
 DRAWTO(159,95) DRAWTO(0,95)
 DRAWTO(0,0)
 wm=memory
 FOR I=0 TO COUNT     ;Random placement
 DO                   ;of worms
  wm.X1=(R&127)+25
  wm.Y1=(R&63)+15
  wm X2=wm.X1  wm.X3=wm.X1  wm.X4=wm.X1
  wm.Y2=wm.Y1  wm.Y3=wm.Y1  wm.Y4=wm.Y1
  PICK_DIR()   wm.DX=xx     wm.DY=yy
  PLOT(wm.X1,wm.Y1)
  wm==+SIZE
 OD
RETURN

PROC MOVE()
BYTE I,NX,NY,Z,J=$AA,R=53770
 wm=memory
 FOR I=0 TO COUNT
  DO
  COLOR=0      PLOP(wm.X4,wm.Y4)
  wm.X4=wm.X3  wm.X3=wm.X2  wm.X2=wm.X1
  wm.Y4=wm.Y3  wm.Y3=wm.Y2  wm.Y2=wm.Y1
  COLOR=1
  Z=FIND(wm.X1+wm.DX,wm.Y1+wm.DY)
  IF Z=0 THEN          ;OK to move
     IF R<240 THEN     ;Do move
        wm.X1==+wm.DX wm.Y1==+wm.DY
     ELSE              ;Change direction
        FOR J=0 TO 3
        DO
          PICK_DIR()
          IF FIND(wm.X1+xx,wm.Y1+yy)=0
          THEN EXIT FI
        OD
        wm.DX=xx  wm.DY=yy
     FI
  ELSE                 ;Not OK to move
     FOR J=0 TO 7      ;Find new direction
     DO
       PICK_DIR()
       IF FIND(wm.X1+xx,wm.Y1+yy)=0
       THEN EXIT FI
     OD
     wm.DX=xx
     wm.DY=yy
  FI
  PLOP(wm.X1,wm.Y1)
  wm==+SIZE
 OD
RETURN

PROC WAIT()         ;Slow down display
```

```
BYTE I,J
 I=255
 WHILE I>SPEED
 DO
   J=255
   WHILE J>SPEED
   DO J==-1 OD
   I==-1
 OD
RETURN

PROC MAIN()
BYTE K=764
 START_UP()
 DO
  MOVE()  WAIT()
 UNTIL K<255
 OD
RETURN
```

As stated at the start of this monthly column, I intended to give you routines you can use, and a little insight into programming with ACTION!. This issue gives you new PLOT and LOCATE routines that are faster than those supplied with ACTION!. In addition to the clip and save section, part of the START_UP routine must be included to initialize the screen array. You may remember it is faster for the computer to look up data in a table than it is to calculate the needed values every time. This is what the array does in the PLOP and FIND procedures.

20. Number Twenty!

Last month?s program should have revealed the cookie-cutter approach by using an array of records. Each worm had 4 pixel segments and its own direction of travel stored in a record. The MOVE procedure handled all the worms in the same manner, which eased the programming task. This same method can be used for other programs where you have many identical, or nearly identical, objects. Some examples might include soldiers engaged in a war, rooms of a haunted castle, employee files, days in a calendar, and many, many more.

Although you can?t directly declare a BYTE array in a record, you can coax ACTION! to store arrays in records. This month you will see how you use and access a record that has a BYTE array in it. The program asks you to enter three student names, with three test scores each. It then displays the student name and average of the test scores, along with their grade. At the bottom of the display, the records themselves are shown, with spaces inserted between the name, test scores, average, and grade. This section shows you how the record would look (minus the spaces) if it were stored in a disk file.

```
TYPE CLASS=[BYTE namlen
            CARD n1,n2,n3,n4,n5
            BYTE test1,test2,test3,avg,
                 grade]
DEFINE CLASS_SIZE="16"

CLASS POINTER student
BYTE ARRAY memory(l00)

PROC Enter_Data()
BYTE ARRAY prompt= "Enter test score "
CARD av,gr,tot

  PrintE("Enter NAME of student;")
```

```
  InputS(student)
  IF student.namlen>10 THEN
     student.namlen=10
  FI
  PrintF("%S 1; ",prompt)
  student.test1=InputB()
  tot=student.test1
  PrintF("%S 2; ",prompt)
  student.test2=InputB()
  tot==+student.test2
  PrintF("%S 3; ",prompt)
  student.test3=InputB()
  tot==+student.test3
  av=tot/3
  student.avg=av
  IF     av>=92 THEN student.grade='A
  ELSEIF av>=84 THEN student.grade='B
  ELSEIF av>=76 THEN student.grade='C
  ELSEIF av>=68 THEN student.grade='D
  ELSE               student.grade='F
  FI
RETURN

PROC MAIN()
BYTE i,j,x=85,y=84
BYTE POINTER ptr

student=memory      ;Locate storage space
FOR i=l to 3
DO
   Graphics(0)
   PRINTF("Student number %U%E",i)
   Enter_Data()
   student==+CLASS_SIZE
OD
Graphics(0)
student=memory
PutE() x=2 y=2
PrintE("NAME         AVG  GRADE")
y=4
FOR i=1 TO 3
DO
  x=2  Print(student)
  x=16 PrintB(student.avg)
  x=22 PrintF("%C%E",student.grade)
  student==+CLASS_SIZE
OD

ptr=memory                 ;Show records
FOR i=1 TO 3
DO
  FOR j=0 to 15
  DO
    y=18+i x=2+j            ;Insert spaces
    IF x>12 THEN x==+2     ;Tests
      IF x>17 THEN x==+l   ;Avg
         IF x>19 THEN x==+l;Grade
         FI
      FI
```

```
    FI
    put(ptr^)
    ptr==+1
  OD
OD
RETURN
```

This program runs funny, in that whenever the name is less than 10 characters long, the records still hold old data behind the student name. See if you can fix this problem. To see this effect, run the program and enter AAAAAAAAAA for the first name, then run it again and enter Z for the first name. You will see that part of the name AAAAAAAAAA is still in memory.

21. Final Episode

Up to now I have discussed the different aspects of a program for the computer. Now it?s time to discuss how to turn those programs in the computer, to files on the disk for everyone to enjoy.

After you have typed in a program, you are required to enter the Monitor to compile and run the program. The Monitor will also save the compiled code to disk. The file produced from the Monitors write command (W "D1:MYPROG.EXE") is executable from the disk. After you have written a program file to the disk from the Monitor, you can then call up DOS and run the file from the command line prompt. If you have supplied all of the necessary code for execution, you will be able to run the file even without the ACTION! cartridge installed. Those files are the type of files you can send to your friends, who may not have the ACTION! cartridge. For a discussion on how to get programs to run without the cartridge, refer back to my column #11.

The DOS file system has a method of determining what type of file is stored on the disk. By using the first two bytes in a file, it can determine if the file was meant for direct execution, or for use with the BASIC cartridge, and so on. If the file begins with two 0's, it is probably an executable file, BASIC uses 255, 255 for its tokenized form. The next four bytes in the file tell DOS where to put the stored data in memory, and how much to put there. By using the two CARD values as the start and stop memory addresses of the code segment, it can load the executable code anywhere in memory. The Monitor supplies all these bytes for you when you use the write command. DOS also allows for the appending of files, so that some code may be loaded in at one address, and other code may be loaded in elsewhere. When the computer has loaded data into the last (stop) address, DOS checks to see if there is more data in the file. If it finds more, it checks to see if the next two bytes are 0,0 (which does happen when files are appended). If not and there is still more data in the file, DOS will assume the next four bytes are the start and stop addresses of another segment of code (or data, depending upon who put what, where!) This process will repeat until there is no more data in the file.

The ATARI operating system also has two special addresses for use in handling the executable files. These addresses are RUNAD (Run Address = 736, 737) and INITAD (Init Address = 738, 739). When data is loaded into INITAD, the ATARI Operating System (OS) interrupts the load process and begins executing code at the address pointed to by INITAD. When that code has finished (via a RETURN statement), the loading process resumes. When data is loaded into RUNAD, the loading process is allowed to complete, then execution of code begins at the address pointed to by RUNAD. You might now see how professional programmers put a little title screen up while their program is loading. A short program to put text on the screen could be loaded in, the INITAD could be used to run that code, and then loading of the main program could proceed. Using ACTION! it is real easy to do things like this. When the Monitor writes a program to the disk, it also includes a short segment to load INITAD with the starting address of the main procedure. Appending files written by the Monitor will always act in the manner stated above, some part loads and runs, then another loads and runs, and another, and another, as many times as there were files appended together. Try it yourself, and put it to good use!

With all the routines and tips I have let you in on, you are now ready to undertake your own projects with confidence. With just a little advanced planning, almost anything a computer can do, can be done using the ATARI. It is up to you, the user of the computer, to put the computer to use. It doesn?t do any work sitting on a shelf. You don?t have to rely on some distant programmer to write the program you want to use, you can write it yourself. It seems the younger generations pick up the new technology quicker than old folks, which is all the more reason to introduce young kids to the ATARI computer. It is easy to use, and simple to program, providing a stepping stone to the larger machines for those kids who may want to pursue a career involving some aspect of programming a computer. The bulk of computer programming is done by those who have a need. The limits of what a computer can do is bound up only by the imagination of those people willing to attempt the programming of their computer. My advice; reach for the stars!

22. Conclusion

It has come time to close this column and move on. I want to thank Mike Schmidt our newsletter editor for the great job he has done organizing, printing, and mailing our club newsletter to all our members. All the people in the club who volunteer their time to help SPACE provide support for the ATARI community, deserve a hearty thanks. It?s the members, who support SPACE through membership and the purchase of DOMs, that are to be congratulated on keeping this group together for so long. Nice going. This is my last submission, I had a fun time, and I hope you gained a little more confidence in your own efforts.