Action! and BBS Express! PRO Tutorial#

by Thomas M. Johnson

Available from

Villa Video's Bargain Cellar (414) 265-5149 ExpressNet Node X11

Table of Contents

Lesson 1#

Welcome to the Action!/PRO tutorial. In it, I plan to teach you how to write command modules for BBS Express! PRO. But to write these, you must know the Action! programming language. So this is a 2 for the price of 1 tutorial.

I strongly suggest you print the files of this tutorial out. There are a number of programs that accompany the text part (the part you are reading now) and smaller program included in the text part that may take a long explanation. So instead of trying to view this on the screen and paging around looking for what I am talking about, having it on paper would make things easier.

First thing you should do is turn on your computer with the Action! cartridge plugged in.

You should see a flashing cursor in the upper left hand corner of the screen and the words "ACTION! (c) 1983 ASC" in inverse letters at the bottom. This is the EDITOR mode. This is where you enter and edit your program. The Action! editor is very versatile. I am using it right now to write this tutorial.

I guess we should start with the "standard" first program that you see in ANY book that teaches a programming language. The infamous "Hello, World" example.

To set aside parts of this text that should be entered into the editor, I will indent from here on in.

```
PROC main()

PrintE("Hello, World")

RETURN
```

OK, so now what does it mean? The word PROC is a Action! reserved word (that means you can't name a variable, procedure or function that name).

PROC is short for PROCedure and that simply means it is a section of code.

The word "main" comes down to us from the language C. A short digression is needed. In the language Pascal, the last procedure is the controlling procedure. It can calls all the other procedures and it is not named.

In C, the controlling procedure is the one called "main". But it can be anywhere in the program.

In Action!, we compromise. It is named like C, but it must be last like Pascal. But in Action!, we don't have to call it "main", we can call it anything we want. But most programmers call it main.

Then comes the "()". This mean there are no parameters. The main PROC can NEVER have any parameters. If you are wondering what parameters are, we will cover them in a later lesson.

Next is the "PrintE()" line. This is part of the Action! library. It prints what ever is between the quotes and puts the cursor on the left side of the screen on the next line down.

Why is the PrintE() line indented? In Action!, indention and capitalization do nothing. "PRINTE()", "printe()" and "PrintE()" are exactly the same thing. But to keep programs from 1 programmer to another looking the same, the "PrintE" is used. And it is indented to show that it belongs to the PROCecude "main".

Lastly, is the word "RETURN". This signifies the end of a procedure. When it occurs at the end of the "main" procedure, your program returns control back to the Action! cartridge.

How do you run this program? When it is typed into the editor exactly as shown, you press the CONTROL key, the SHIFT key and the M key at the same time (I will abbreviate this as <C><S>M).

You will hear a buzz and the screen will go blank. There will be a inverse line at the top of the screen. You are now in the MONITOR mode. From here you must Compile your program.

You press C then the RETURN key. If there are errors, you will get a message telling you so. If you have errors, press E and the RETURN key to go back to the Editor. If not, you press R and the RETURN key to Run you program.

This is a good place to end lesson one. If you want to experiment until the next lesson, try adding some more PrintE lines in the program. Like:

```
PROC main()
```

```
PrintE("Hello World!")
PrintE("I am programming in")
PrintE("Action!")

RETURN
```

Lesson 2#

In the last lesson, we went over how to compile and run a Action! program. We also learned that the last PROC in a program is the first one that is run. Finally, we learned our first Action! library PROC - PrintE.

Now we are going to cover varibles. In Action!, you must know what type of values a variable will take on before you can use it. This is so the compiler can set aside the right amount of space for that variable.

In Action!, we have 3 basic variable types: BYTE, INT and CARD.

Note: the keywords BYTE and CHAR are identical and can ALWAYS be used in place of each other.

First the BYTE. A BYTE can have a value from 0 to 255 and that is all. When you add 1 to a BYTE with a value of 255 you get 0. The same when you subtract 1 from 0 you get 255.

BYTE values are used for loops and most other general values.

The next basic type is a INT. INTs can have values from -32768 to 32767. If you need to use negative numbers, a INT is the only way to do it.

Lastly we have a CARD. CARDs can have values from 0 to 65535. Use CARDs when you need larger numbers. But 65535 is the largest number you can have. When you add 1 to 65535 you get 0 for an answer.

The are no real numbers like 5.7 in Action!. Nor are there scientific notation like 5.8E4.

Why does Action! have these restrictions? I know BASIC programmers are say "In BASIC we don't have to declare variables and they can be real and larger than 65535."

But in BASIC, EVERY variable takes up 6 bytes of memory. In Action!, a BYTE takes up 1 byte and INTs and CARDs take up only 2. And this is the way the computer really looks at numbers. The floating point package is the thing that really slows BASIC down.

So we have this chart:

name	size in bytes	low val.	high val
BYTE	1	0	255
INT	2	-32768	32767
CARD	2	0	65535

Ok, so how do you use them? Here is a sample program.

```
  PROC main()
    BYTE i
    Print("Give me a number. ")
    i=InputB()
    PrintBE(i)
```

```
  RETURN
```

The first line we know. It is required by all Action! program to define a PROC.

The next line declares i and j as variable of type BYTE. They will only have a value from 0 to 255.

The Print statement prints what is between the quotes without returning the carriage after.

The i=InputB() line inputs a BYTE from the keyboard and places its value in i.

j=i set the value of j to be the same as the value in i.

PrintBE(j) prints the value of j as a BYTE and returns the carriage.

If a number larger than 255 is entered the Action! system will take what is called the "least significant byte" of that number and place that in i. I won't go into LSB right now. But if you want to experiment...

What if you wanted INTs instead of BYTEs?

```
PROC main()
  INT i
  Print("Give me a number. ")
  i=InputI()
  PrintIE(i)
RETURN
```

OK, the last thing I will do in this lesson is the related Input and Print functions.

```
BYTE b
CARD c
INT  i
```

Just declaring some dummy variables.

Example	Description
i=InputI()	input a INT
c=InputC()	input a CARD
b=InputB()	input a BYTE
Print("hi")	print string without the carriage return
PrintE("hi")	print the string with the carriage return
PrintB(b)	print a BYTE without the CR
PrintBE(b)	print a BYTE with a CR
PrintC(c)	print a CARD without a CR
PrintCE(c)	print a CARD with a CR
PrintI(i)	print a INT without a carriage return
PrintIE(i)	print a INT with a CR

Next we will go into expressions and the IF statement.

Lesson 3

In the last lesson we went over variables. The name of a variable doesn't have to a name like the "i" I used in the examples. In fact, they shouldn't be like that. Variable names should be descriptive to you know what their purpose is. In Action!, the only restriction is that variable names start with a letter. They may contain letters and the underline "_" character later in the name. Which is easier to understand?

```
t=(p*x)+p
```

or

```
total_price = (price * tax) + price
```

This brings us to our next subject, expressions.

Action! supports the following operators.

Operator	Description
-	as in negative numbers. Remember only INT can be negative.
*	multiply
/	divide. This is integer division because Action! doesn't have real numbers. So when you take 5/2 it is equal to 2 NOT 2.5
MOD	This is the remainder when you divide. 5 MOD 2 equals 1 because 5/2=2 with a remainder of 1
+	addition
-	subtraction

Action! also has a number of bit-wise operators but we will cover those later.

I guess a sample program would be good about now. This sample program is a little too long for you to have to type in to your editor. So I guess this is a good time to say that when this occures, the sample program will be called APROG.301

This means it is a program with the ATUTOR series. It goes with lesson 3 and it is the 01 (first) program.

To load the program into your Action! editor, you press <control><shift><R> at the same time. The bottom line of the editor will print "Read?"

Here you type in the filename and it will load. Then you can compile and run the program like normal.

You will notice that the variables are of type INT. I did this so you can see how Action! handles negative numbers. Try -32767 and 4 for sample numbers once.

You will see that Action! does some weird things when you try to go past the boundaries of the variable type. In this case, INT can only be -32768 to 32767.

Try some nicer numbers to show the program works fine. Like 10 and 6. Or any other number you may want to try. You will have to recompile and rerun the program for each different set of values.

You may have noticed a PutE() command in the program. This places a carriage return on the screen. In other words, it just puts a blank line on the screen. I just put it in there to make it easier to read.

The next things we will cover are relationals. Relationals yield a value of TRUE or FALSE. An example might be: 4=7. This is false. Here is a list of the Action! relationals:

a=b tells if a and b are equal

a<>b tests to see if a and b are not equal

a\#b the same as a<>b

a>b is a greater than b

a>=b is a greater than or equal to b

a<b test to see if a is less than b

a<=b is a less than or equal to b

The are 2 more, AND and OR and I will cover them in the next tutorial.

Now we will go into the simplest case of the IF-FI statements. They are used like this:

```
IF <relational is TRUE> THEN

  statement
  statement
  statement

FI
```

If the relational expression between the brackets is true, Action! will execute all the statements between the IF line and the FI line.

If the relational is false, Action! will skip all the statements between the IF and FI and start with the statement after the FI.

The statements between the IF and the FI should be indented a few spaces to show that they are dependent on the IF statement. And it also makes it alot easier to read!

try this program:

```
PROC main()

  IF 4>9 THEN
    PrintE("4 is greater than 9")
    PrintE("Do you believe it?!?")
  FI
  PrintE("done")

RETURN
```

The above program should just print the word "done". If 4 should happen to be equal to 9 today (maybe its Friday the 13th?) then it would print the other 2 lines followed by the word "done".

This is the end of lesson 3, I will leave you with another program that is too big to print here. It is called: APROG.302

Lesson 4

Well, I guess its time we started talking about loops. Loops give us the ability to repeat a section of code over and over again until we want it to stop.

The section of code we want to repeat is marked off with a DO at the start and a OD at the end.

```
PROC main()
```

```
  DO
    Print("Hello!! ")
  OD
RETURN
```

This is print Hello!! over and over and there is no way to stop it other than pressing the BREAK key or the RESET key. So we need a way to control the number of times a loop get executed. Action! provides us with 3 ways.

The first and probably the least powerful is the FOR loop. It is the most used loop. Then why do I say it is the least powerful? Because it executes a loop a fixed number of times.

```
PROC main()

  BYTE i

  FOR i=1 TO 8
  DO
    PrintBE(i)
  OD

RETURN
```

This program just prints the numbers 1 to 8 vertically on the screen. ANY commands can be places between the DO and the OD and they will get executed 8 times.

The next loop controller Action! gives us it the WHILE loop. It looks like this:

```
WHILE (a condition is true)
DO

OD
```

The WHILE loop evaluates its condition at the top of the loop, so if the condition is FALSE to begin with, the loop will never execute. Try program APROG.401 to see.

The last type of loop control Action! gives us is the UNTIL loop.

```
DO
  statements
UNTIL (condition is true)
OD
```

The major difference between the WHILE and the UNTIL is that the UNTIL get evaluated at the bottom of the loop. This means it will be executed at once. After it is executed once, if the condition is FALSE it will get executed again.

```
DO
  PutE()
  PrintE("Press any key or 0")
  PrintE("to exit")
  i=InputB()
UNTIL i=0
OD
```

If the condition i=0 is TRUE the loop exits.

The last loop command I saved for last to mention because in theory, it is never needed. I say in theory because in the REAL world it can be useful.

The command name is EXIT. Lets say your program has to input a large number. And the next numbers are divided into the bug number until the answer gets down below 10. But if the user inputs a 0 to divide into, the is very bad since a division by 0 doesn't exist.

Look at program APROG.402 for this program. You will notice that the divison is not what you would expect. That is because Action! only has whole numbers.

In Action! if you:

```
PrintBE( 1 / 3)
```

That is take 1 divided by 3, you will get 0, NOT .33333 Action! has no decimal notation.

I am getting of the subject but I feel that since it is used in the program it is a good idea to explain it.

Anyway, the EXIT command is good when you use it in emergency situations, when you have to get out of a loop in a hurry.

A EXIT is also useful when you have alot of conditions on when to end a loop.

Like: UNTIL (the user answers yes AND the record number =0 AND you haven't reached the end of the file AND you aren't running out of memory space)

I haven't gone into the actual Action! statements of those situation yet so I put them in words. But you get the picture, there are 4 conditions to end this loop.

The better thing to do is just have 1 or 2 conditions in the UNTIL statement and put the others in IF FI with can EXIT.

MOST other times, if you must have an EXIT in your loop to make it work right, you have chosen the wrong type of loop.

APROG.403 compares doing the same loop with the 3 looping statements. Remember, each time a program must do a loop, it is up to you, the programmer, do decide which type of loop to use. For APROG.403, try and guess which one is BEST. They all work, but one is best for what I am trying to accomplish.

That's it for Action!'s looping commands. In the next file, we won't be going into any new Action! commands. It will cover some editor and monitor commands that will make your writing and debugging of Action! programs faster and easier.

Lesson 5#

As promised, now we are going to talk about advanced editor and monitor commands. Knowing these will make your writing, editing and debugging of Action! programs many times faster and easier.

In this file, when I use something like this: <C><S>M That means to press the Control, the Shift and the M keys at the same time. And we all know that doing that will take us into the monitor.

There are more commands to help you than I can possibly list. If I did it would take me over 100K. I am just going to cover the ones I use most. I use almost all the commands at sometime or another. Please look at your Action! manual and carefully read over the sections on these commands. They are very important.

I guess we should dive right in with some editor commands. You already know <C><S>R and <C><S>W for reading and writing to and from the editor.

Pressing <C><S>- and <C><S>= (these are the arrow keys) will jump you around the editor 1 screen at a time. You can really get around the editor fast using these.

If you are looking for a certain word or words in you Action! program, the editor has the <C><S>F command. Action! will ask you for the string to search for. And zip you are there or a not found message appear. This only searches from the point you are in, downward. So if you are at the end of your file, no matter what you search for you won't find it because there is nothing below it.

<C><S>S will also find a string of characters but it will also replace it with another string. This will come in very handy when we start programming for BBS Express! PRO.

A word of caution, when you substitute, use a word that does not have the old string contained in it.

Let's say you want to change all the occurrences of the word 'score' in your program into 'u_score'. Changing the first occurrence is great. The cursor will be on the 'u' in 'u_score'.

The next occurrence it will find is the 'score' in 'u_score' so you will get 'u_u_score'.

If you change a existing line but have not yet pressed the RETURN key, <C><S>U will restore it back as good as new.

To delete a line just press <SHIFT><DELETE>. To delete a bunch of line just hold this key down. Oh no! You just deleted the wrong line, well <C><S>P will put the entire block of deleted lines back.

This is also how you move and copy blocks of text. Just delete them, and go to where you want to appear. Press <C><S>P and it is moved.

If you just want a copy somewhere else, make sure you <C><S>P it in the same spot where you deleted it, the move and <C><S>P again.

To erase you file, <Shift><Clear> does the trick.

Remember, look in the Action! manual and read over the commands I did not cover. They are very useful!

Now the monitor. You already know the C command compiles a Action! program and R runs it.

B reboots the Action! system without having to turn your computer off and on. This erases anything you have in the editor and starts you from scratch.

O takes you to the options menu. In here you can change some features of the Action! system and also turn on some debugging flags.

The first option is the Display option. When you turn this on, it speeds up reading and writing to the disk and printer. It also speeds up compiling your program. It does this by turning the screen off while doing these operations. The first thing I always do then I load my Action! cartridge it turn this on.

If that stupid bell is driving you nuts every time you go to the monitor or come across an error while compiling, the Bell flag can turn this off.

The Case Insensitive flag can turn on the.... well I better show you.

In Action!, the variables:

```
score
```

```
Score
SCORE
sCoRe
```

are all the same. Turning this on will make them all different varialbles. It will also make you spell the Action! routines correctly too. PrintE is spelled with a cap P and a cap E. If this is off, the default, you can spell it anyway.

The Trace flag prints the PROC or FUNC you are currently entering when you run your program. We haven't covered PROCs and FUNCs yet but I will say this is VERY, VERY useful when debugging.

Most of the time when Action! finds an error while it is compiling, it will take you right to the spot in the editor where the error occurred. Sometimes Action! doesn't know where the error is and setting List to y will print each line to the screen while that line if being compiled. This in only useful when the Display flag is on. Compile a sample program with this on to see what it does.

Window size, Line size and Left margin we won't go into.

EOF character will change the last character of each line from a blank space to something you can see. The Action! manual say it will "aid visualization of the program."

That's it for the Options menu. Back to the monitor commands. D will take you to DOS. If you are using DOS 2.0 or 2.5 the editor will be erased so be sure to save it before you use this. If you have DOS XL or SpartaDOS, the editor remains. To get back to Action! from DOS use the B (to cartridge) command in DOS 2.0-2.5 and CAR in DOS XL and SpartaDOS.

You can compile a Action! source program without having to load it into the editor. Just put a filename after the C for compile. Make sure you use the whole filename and device and make sure there are "quotes" on both ends of the filename.

C "D:APROG.402"

will compile APROG.402 off disk drive #1. This is useful when your program won't compile because it is out of memory.

W "D:MYPROG.CMD" will write the compiled code to disk. That way you don't have to recompile it each time. Using R "D:MYPROG.CMD" will load and run your Action! program if it was saved using W.

If you have the Action! runtime package, this is how you write programs that don't need the cart. to run.

This is also how you save BBS Express! PRO command modules.

Well that's about it for now. I have barely scratched the surface with the number of commands you have available to you. But, These bare minimums will help you get around easier.

Lesson 6

Well, now it is time to cover the most powerful of all the Action! Print routines. It is called PrintF for print formatted.

This baby does it all!

The first thing PrintF must have it a string in "quotes." If PrintF is used like this, it is just like Print.

command	output
PrintF("Hello")	Hello
Print("Hello")	Hello

But if some special characters appear inside the "quotes", PrintF can do anything you want.

The first of the special characters is the %E. That is a percent sign followed by an E. This can appear anywhere in the "quotes" and Action! will do a RETURN where the %E appears.

Here is a comparison of the old way versus PrintF

old way

```
PutE()
PrintE("What is your name?")
PrintE("Only 10 characters please.")
PutE()
```

new way

```
PrintF("%EWhat is your name?%EOnly 10 characters please.%E")
```

It may look a little harder to read but it actually takes up less memory and time. Why? Because with each call to an output procedure, Action! must open a channel to the screen, print what you want and then close the screen. So the old way there are 4 opens and closes. With PrintF there is only 1.

The next special character is the %U. What does the U stand for? It means Unsigned. INTs are signed with a +/- so it must be used by BYTEs and CARDs. But how does it know what value to print where the %U is?

Assume a and b are BYTES.

```
a=5 b=10
```

old way

```
PutE()
Print("The sum of ")
PrintB(a)
Print(" and ")
PrintB(10)
Print(" is ")
PrintBE(a+b)
```

new way

```
PrintF("%EThe sum of %U and %U is %U%E",a,b,a+b)
```

Both print the line:

```
The sum of 5 and 10 is 15
```

After the second quote, there is a comma and then the BYTEs that get substituted. The first BYTE in that list goes with the first %U. And so on with the rest.

Then what does a %C stand for? %C prints the value associated with it as a character. The ATASCII value for the letter 'A' is 65. So,

```
PrintF("The letter %C.%E",65)
```

will print:

The letter A.

This may not seem very useful but when you want to print special graphics characters it is real nice.

Here is a list of special characters and how they output the data in the list.

format char	description
%I	INT
%U	CARD (the U stands for Unsigned) and BYTE
%C	print as a character
%H	a Hexdecimal number
%E	the RETURN character
%%	output the percent sign
%S	output as a string (we'll cover this in a later lesson)

So what is wrong with PrintF? Well, it can only print to the screen. Not to a external device like a disk or your printer. Don't worry about that now, we'll cover that in a later lesson also.

Try APROG.601 for an example of using PrintF for handling some complex output formats.

Well that's is for this instalment. Short, sweet and to the point. Next, we will go into the heart of Action!, the ability to call PROCs.

Lesson 7#

Now its time to learn about PROCs. What is a PROC? We already know about PROC main() but what else is there?

A PROC is an independent block of code that can be called. Ideally, a PROC should use only local variables and not global. We'll get into that in a minute.

You define a PROC the same way you define main().

```
PROC my_proc()

  local variables

  statements

RETURN
```

The "my_proc" can be any name that isn't already used by the Action! system.

For you BASIC programmers, a PROC is like a GOSUB. But much more powerful.

PrintE and the most of the other routines we've looked at so far are PROCs that are built in to the Action! cartridge.

Let's take a look at a sample.

```
PROC my_proc()

  BYTE a

  a=10
  PutE()
  PrintBE(a)

RETURN

PROC main()

  BYTE a

  a=5
  PutE()
  PrintBE(a)
  my_proc()
  PutE()
  PrintBE(a)

RETURN
```

Can you guess what will be printed? The output will look like this:

```
5
10
5
```

First 'a' is set to 5 in main(). Remember that the last PROC is always the first one run. The value of 'a' is then printed. Then when the computer gets to the line "my_proc()" it goes up to the spot where "PROC my_proc()" is and starts running there.

Ok, now we have another 'BYTE a' and we already declared a BYTE a. These 2 "a's" are as different as apples and oranges. They are local to the PROC they were declared in.

So 'a' is set to 10 and is printed. When the RETURN statement is executed, the computer goes to the next line after the call to the PROC. In this case it will print 'a' again.

And it prints 5. But in my_proc we set a=10. Once again, these are 2 different variables because each can only be used by the PROC it is defined in.

A even more powerful feature of PROCs it the ability to pass it parameters. Parameters and local variables allow PROCs to not know or care anything about the outside program. A PROC will have a specific task and it won't care what called it or why. Take a look at APROG.701 for an example.

The call to the PROC in main() looks like "square(num)". Action! takes whatever num is equal to and places in the variable "number" in PROC square(BYTE number).

You can have a number of parameters and they can be any type. You can even mix types. To pass 2 BYTEs and 1 CARD to a PROC use this:

```
PROC m(BYTE a,b CARD c)
```

and the call would look like:

```
m(10,30,5000)
```

There are commas between like variable types and a space between different types.

You can have PROCs call other PROCs. It's important to understand the use of local variables and PROCs. Even more so of the PROCs. Next time we will cover FUNCs, which are similar to PROCs, and the opposite of local variables, globals.

Lesson 8#

Do you remember what a function is from high school algebra? It take an number, performs some operation on it and return only 1 number back. Like:

```
y = f(x)
```

where f(x) = x^2

So if x=3 then y=9.

In Action!, a function is like a PROC only it has a type associated with it. We can write the x^2 function in Action! like this.

```
CARD FUNC squared(BYTE x)

  CARD y

  y = x * x

RETURN(y)

PROC main()

  CARD answer

  answer = squared(3)
  PrintCE(answer)

RETURN
```

And it will output a 9.

Notice that there is a variable type before the word FUNC. This tells what is going to be returned. A BYTE, CARD or INT can be used. As you can see, the rest is alot like a PROC. The next different thing is the RETURN statement.

Whatever is on the left side of the equals sign in the call, in this case the variable 'answer' will get the same value as what is inside the () after the word RETURN in the FUNC.

You can see that the input routines we've been using so far are really FUNCs. When you do a:

```
value = InputB()
```

the first line of InputB in reality looks like this.

```
BYTE FUNC InputB()
```

meaning it returns a BYTE to value.

When we first talked about IF FI, I said that the conditional after the IF returned a TRUE or a FALSE. In reality, the FALSE is the number 0. And TRUE is actually any other number. The most common values for TRUE are 1 or 255 but ANY positive value will due.

So if you wrote a FUNC that returns a 0 or a 1, you could use it in place of a conditional. This is a powerful use of FUNCs.

```
BYTE FUNC values_match(BYTE x,y)

  BYTE rval

  rval=0
  IF x=10 AND y=20 THEN
    rval=1
  FI
RETURN(rval)


PROC main()

  BYTE v1,
       v2

  PrintE("Please give me 2 numbers.")
  Print("v1 = ")
  v1=InputB()
  Print("v2 = ")
  v2=InputB()

  IF values_match(v1,v2) THEN
    PrintE("Your values match!!")
  FI

RETURN
```

This is almost a trivial sample of the power this type of construct. If you have seen a program I wrote running on BBS Express! PRO called "The Wheel", 50% of that program is BYTE FUNCs that return a 1 or a 0 to IF FI statements.

Now, what if you need a variable that will be used by alot of PROCs and FUNCs? Do you keep sending it as a parameter? Well, you could, but using globals would be better.

We know that using a local variable doesn't affect any of the other variable used outside that PROC or FUNC. What if we want to change something in the outside program? Again, global variables are the answer.

A global variable is a variable that is not declared between a PROC and its RETURN or a FUNC and its RETURN. They are declared at the start of the program and between PROCs and FUNCs.

The compiler knows a variable is a global by use of the work MODULE. The word MODULE is assumed as the first line your program. But it never hurts to put it in. If you declare globals anywhere else in your program (like between PROCs etc.) then you MUST have the MODULE there.

```
MODULE


  BYTE score


  PROC you_win()
```

```
    PrintE("You won this game!!!")
    score = score +100
  RETURN

  PROC main()

    you_win()
    PutE()
    Print("Your score is: ")
    PrintCE(score)

  RETURN
```

You can see that the value of score in the main() was increased by the "score = score +100" in you_win(). You can't do this with locals because when you change a local (this includes parameters too) it is only changed within that PROC or FUNC.

Try to resist the urge to use all globals in your programs. Globals make bugs nearly impossible to find. And there are times when you don't NEED a variable to change in the outside program.

When I first introduced PROCs I said that they were independent blocks of code. Using globals makes a PROC dependent on the rest of the program.

Lesson 9#

An ARRAY is a group of numbers that are the same type. Whether that means they are a BYTE, INT or CARD.

A string is something like this: "Hello, how are you today?"

So why are these 2 concepts begin presented in the same lesson? Because to a computer, they are the same thing. A string is just an ARRAY of BYTE values. But since a BYTE and a CHAR are EXACTLY the same thing, we will use CHAR just so we know that we are using that ARRAY as a string instead of just holding numbers. There is one small difference between strings and BYTE ARRAYs and we cover that in a little bit.

An ARRAY is a indexed group of numbers. Lets say you have 4 people playing a game and you want to keep track of all their scores. You could use 4 different variables like this:

```
PROC print_score()

  BYTE score1,
       score2,
       score3,
       score4

  PrintF("Score 1: %U",score1)
  PrintF("Score 2: %U",score2)
  PrintF("Score 3: %U",score3)
  PrintF("Score 4: %U",score4)
RETURN
```

Everytime you needed to do something with a persons score, you have to figure out who is playing and then use their particular score variable.

A better way is to use an ARRAY.

```
PROC print_score()

  BYTE ARRAY score(5)
  BYTE player

  FOR player=1 TO 4
  DO
    PrintF("Score %U: %U",player,score(player))
  OD
RETURN
```

Now any time you need a persons score, you just place that player's number in score( ) and it is given to you.

The declaration of an ARRAY is

```
type ARRAY name(dimension)
```

type can be any of the fundamental types; BYTE (CHAR), INT or CARD. The dimension is how big the ARRAY is going to be. In Action!, ARRAYs go from 0 to dimension-1. That is why we declared score(5), now the player numbers go from 0 to 4. We could have used score(4) but then the numbers would go from 0 to 3 and humans usually don't think that way.

The type has nothing to do with how big an ARRAY can be. The type is what the ARRAY will hold. For example, you can have this:

```
BYTE ARRAY total_pay(5000)
```

Now you can have 5000 employees but each can only have a pay of 0 to 255. (Sounds familiar huh?)

```
total_pay(1) = 210
total_pay(2) = 170
total_pay(3) = 250
total_pay(4) = 100
        .
        .
        .
total_pay(4999) = 30
```

If you want to input a number into an array you could use:

```
PROC main()

  CARD ARRAY lottery(10)
  BYTE i

  FOR i=1 TO 9
  DO
    lottery(i)=InputB()
  OD
RETURN
```

You use ARRAY just like any normal variable, but you must subscript "(i) or something" each time you use it.

So, what is the difference between a ARRAY of BYTEs and a string? The zeroth (0th) BYTE of a string holds its length. For this reason, you can't do a direct assignment, the compiler won't let you. Here's what I mean:

```
  CHAR ARRAY prompt(30)

  prompt="What is your guess? "
```

That is illegal. Instead, Action! provides you with a ton of PROCs and FUNCs to make working with strings easy.

To assign a string variable some text, you use:

```
SCopy(prompt,"What is your guess? ")
```

This just copies what ever is second into what ever is first. So you can also use:

```
SCopy(again,prompt)
```

This will make the variable "again" equal to the same text as the variable "prompt". This is assuming that "again" was dimensioned to the same size or bigger than "prompt".

Printing strings is easy, we just use the normal Print() and PrintE() we have always been using. Just now, you use the string name inplace of the text.

```
PrintE(prompt)
```

Inputing strings is a little different that inputing normal variables. Here we use:

```
InputS(prompt)
```

Notice that it is a PROC and not a FUNC.

Action! also has PROCs that take part of one string and copy them into another (sub strings) and taking a string and copying into part of another (appending or inserting). Look in you Action! manual on how to use these PROCs.

Another thing you can't do in Action! is compare string like this:

```
IF prompt=again THEN ...
```

You must use a built in FUNC that does this for you.

SCompare(prompt, again)

This FUNC returns an INT value. It return a number <0 if prompt is less than again. It return 0 if they are equal. And it returns a number >0 if prompt is greater than again.

This is great for alphabetization. In fact, APROG.902 does a little alphabetizing.

Before I let you go I have to tell you a little something I left off of ATUTOR.003.

In an IF statement, if the conditional is not true you can have an ELSE statement.

```
IF a>b THEN
  PrintE("A is less than B")
ELSE
  PrintE("A is greater than or")
  PrintE("equal to B")
```

```
FI
```

Action! will execute the second part only if the first is not true. You can also have multiple IFs.

```
IF a>b THEN
  PrintE("A is less than B")
ELSE
  IF a=b THEN
    PrintE("A is equal to B")
  ELSE
    PrintE("A is greater than or")
  FI
FI
```

In fact, this happens so often that we have a special statement for it: the ELSEIF. The next example is the exact same thing as the above one.

```
IF a>b THEN
  PrintE("A is less than B")
ELSEIF a=b THEN
  PrintE("A is equal to B")
ELSE
  PrintE("A is greater than or")
FI
```

The reason I am mentioning this now is APROG.901 uses both the ELSE and the ELSEIF.

Lesson 10#

Well, I guess its time to talk about files. Now, when I say files, I mean both files and devices. The only Atari device that uses files is the disk drive. But when it comes down to using files and devices, things are done the same.

Your Atari has a few devices built in to it when you turn it on.

E: - input and output - This is the standard Atari editor that is used. When you do a PrintE() or other simple output, this is the device it goes to. Also, when you do a InputB() this is the device it comes from. It is the screen and the keyboard combined.

C: - input and output - This is the cassette player.

P: - output only - This was originally meant to stand for parallel but Atarians now know it as printer. It makes sense that it is for output only, right?

S: - output only - This is the screen. This is the device that is opened by the Atari when you do a Graphics command. (We'll cover Graphics later in the tutorial.)

K: - input only - The keyboard. When you input information from the keyboard, nothing is echoed on the screen.

There are 2 more devices that we call standard, even though they aren't built in to you computer. They have to be loaded in before you can use them.

D: - input and output - Your disk drive. DOS must be loaded before you can use it. Also, a filename must be supplied when you first open this device.

R: - input and output - The RS232 port on the 850 or P:R: Connection. This is used by modems that need the interface.

There is also 1 custom device that is "famous" enough to merit me mentioning it here.

T: - input and output - This is your 1030 or XM301 modem.

OK, those are the devices. How do you use them? Good thing you asked. The first thing you must do before you can use a device, is open it.

```
Open(1,"K:",4,0)
```

The number "1" is the device number. After you open a device, from the then on you refer to it by its device number.

The "K:" is what device you want to open. It is a string and must either be in "quotes" or a CHAR ARRAY variable.

The next number is the command. Use this chart to find out which command to use.

Access	Value
Input Only	4
Output Only	8
Input and output	12
Append to end of file	9
Disk drive directory	6

Since a K: device can only do input, the number "4" is an easy choice.

Just because a device can do input and output DOES NOT mean you should choose the number 12. If you want to output to a disk file use 8. The reason the command number 12 is there is in a situation like this:

If you want to change the 5th BYTE in a file, you read the first 4 and then over write the 5th.

The last number "0" is required and there must always be a "0" there.

If you know BASIC, you will see there is a little difference between the order Action! uses and the order BASIC uses.

In BASIC it is like this:

```
OPEN #1,4,0,"K:"
```

The arguments mean the same things, just a different order.

To output to a file, you use the standard Atari Print PROCs except you put a D (for device) at the end in the proper place. For example, if you want to print a string with a carriage return on the end to a device, you'd use:

```
PrintDE(2,"Hi there")
```

The number "2" is the device number you assigned it in the Open(). This can be done to any device that can do output and the statement will be exactly the same.

Here are the output PROCs.

Proc	output
PrintD	string
PrintDE	string with CR
PrintBD	BYTE
PrintBDE	BYTE with CR
PrintCD	CARD
PrintCDE	CARD with CR
PrintID	INT
PrintIDE	INT with CR

The input statements are close to the normal input FUNCs as well.

```
b=InputBD(1)
```

Input a byte from device number 1.

Here are the input FUNCs.

```
InputBD - BYTE
InputCD - CARD
InputID - INT
```

And for strings:

```
InputSD(3,name)
```

This get a string from device number 3 and places it in name.

There are 3 more statements you can use with devices.

c=GetD(1) - get a single character from device number 1

PutD(1,c) - put a single character on device 1. PutDE(1,c) - same but CR after

InputMD(3,name,20) - Get a string from device number 1 and only 20 characters long.

When you are done with a device, you must always close it.

Close(4) - close device number 4.

There are a few things that are different when you use D:

The first is you must supply a filename. It can be 8 characters long with a 3 character extension.

```
Open(3,"D:MYFILE.DAT",8,0)
```

If you use a DOS that has subdirectories, they can be included in the device string as well.

If you use Open a disk file with a command number of 8, it will create a new file if it does not already exist or erase the old file if it does.

When you Open a disk file with command number 4, it always starts reading at the beginning.

It is a good idea to get in the habit of close a device number before you open it just to make sure it is ok to use. If you try to open a device number when it is already open, you program will bomb.

The device numbers that you are allowed to use are 1 though 7.

Lastly, every time you compile an Action! program, a BYTE ARRAY is automatically declared for you. The ARRAY is called "EOF". EOF holds either a 0 or a 1 so it can be used in a IF, WHILE or UNTIL statement.

When using disk or cassette files, you cannot try to read past the end of the file. If you do try, the program will crash.

So you have 2 choices, either you have to save the number of entries in that file and use a FOR loop to read in exactly that number of entries. Or use EOF.

EOF holds a 1 if you are at the end of file and 0 otherwise. For example:

```
EOF(2)
```

If device number 2 is at the end of the file, then this will be a 1. The best way to use EOF is in a WHILE loop because a WHILE loop checks before the loop is run the first time.

```
PROC read()

  BYTE character

  Close(1)
  Open(1,"D1:ATUTOR.001",4,0)
  WHILE EOF(1)=0
  DO
    character=GetD(1)
    Put(character)
  OD
  Close(1)
RETURN
```

This PROC prints ATUTOR.001 on the screen character by character. The file ATUTOR.001 must be on the disk in disk drive #1 for this program to work. Also, you may have noticed that I didn't call it main() this time. Remember, you don't have to call it main().

This type of PROC might be good if you wrote a game and this will print the instruction file to the screen for the user.

You may also have noticed that reading character by character is not the most efficient way to do this. A better way would have been to read the whole line in with a InputSD() and echo it with a PrintE(). But to so this the last line must end with a CR before the end of file. If you KNOW that it does, you can use the next PROC instead. But, if you are unsure, you don't want to make assumptions. (By the way, ATUTOR.001 does have a CR before the end of file so it is OK to use this PROC)

```
PROC better_read()

  BYTE ARRAY line(50)

  Close(1)
  Open(1,"D1:ATUTOR.001",4,0)
  WHILE EOF(1)=0
  DO
    InputSD(1,line)
    PrintE(line)
  OD
  Close(1)
RETURN
```

Lesson 11#

We have just finished pretty much of the basics of Action! Now it's time to get into the more advanced topic that Action! offers.

The first of these is the POINTER. A POINTER does not hold a value like a "normal" variable. Instead it tells us where that value physically is inside your Atari. To declare a POINTER we first have to declare what type it will point to.

```
BYTE POINTER a

CARD POINTER b
```

There are 2 new operators that can be used with POINTERS. If we have a variable i of type INT we can use this:

```
INT POINTER g

  g=@i
```

This means g points to i. Well, it really mean that we have assigned g to point to the same memory location as i. g holds the memory location of i.

Another new operator we now have is:

```
  g^=5
```

This says to put the value 5 into the memory location that g points to.

Try APROG11.001 and try and follow what is going on. You can see that POINTERs are a nice way of simulating the BASIC POKE command. But Action! has even better ways to accomplish that.

They say "A picture is worth a thousand words" to I'll try it now. I'll try and show you this PROC graphicly.

```
PROC m()

  CARD c

  CARD POINTER cp

  c=10
  cp=@c

  cp^=6

RETURN
```

Ok, we have 2 objects to begin with: I'll use ? to mean we don't know for sure that these value hold since we didn't assign them to anything yet.

```
  ?               ?
  cp              c
```

When the assignment c=10 come up the variable look like this:

```
  ?               10
  cp              c
```

When cp=@c is executed, cp is assigned to point to the memory location that c occupies. NOT to the value in c.

```
  -------|        10
  cp     -------> c
```

Then, we change the value in the location cp points to to 6.

```
  -------|        6
  cp     -------> c
```

I hope that helped a little.

Have you ever wanted to change the value of parameter you passed to a PROC or FUNC? Well, before now you have to create a global and not use a parameter at all. POINTERs allow you to keep PROCs and FUNCs independent by not using globals and allowing you to change the value of a parameter outside of the PROC.

```
PROC add(BYTE POINTER a)

  a^==+10
RETURN

PROC main()
  BYTE a

  a=5
  PrintBE(a)
  add(@a)
  PrintBE(a)
RETURN
```

Well, I bet you are saying, "That's nice, I MIGHT use that once in a while, but big deal otherwise." I am going to introduce 2 more types of new Action! ideas then you will see the importance of POINTERs. We are going to leave POINTER for a moment to talk about 2 things. The first is VERY short. It is the Action! directive DEFINE. You use DEFINE to clarify your programs and make them easier to read. And they take up no extra memory so you can use tons of them without any overhead.

DEFINEs simply substitute one thing for another. We know from a previous APROG that Put(125) clears the screen. But you really can't tell that just by looking at it can you? But CLS looks like it might mean clear screen. So we can use:

```
DEFINE CLS = "Put(125)"

PROC m()

  CLS
  Print("hi")
```

Or you can use it like this:

```
DEFINE MAX = "25"

PROC any()

  FOR i=1 TO MAX
  DO
```

```
    ;read in values
  OD
  FOR i=1 TO MAX
  DO
    ;print out values
  OD
```

This way, if you want to increase the maximum number of entries, you only have to change the value in MAX, and all the times it is used are changed.

The next thing is records. You may have noticed that when you declare an ARRAY, all the items have to be the same type. All INTs, all CARDs etc. Records allow us to mix types into one object.

```
TYPE new = [ BYTE player_number,
                  points_per_game

             CARD season_total,
                  lifetime_total ]

new bball_player
```

We have just made a new type of variable. When we use the

```
new bball_player
```

it is just like

```
INT g
```

"new" is the type and bball_player is the variable name. To use a record we have "dot notation". That is just a fancy way of saying we do this:

```
bball_player.player_number=12
bball_player.points_per_game=28
bball_player.season_total=300
bball_player.lifetime_total=InputC()
```

And to retrieve this information we can just:

```
PrintBE(bball_player.player_number)
```

I think this is a good time to stop. We have gone over alot in a short time and this stuff is pretty hard to get the hang of right away. And it is probably hard to think of what you can use this for. And your also saying, "Hey, you said there would be more with POINTERs!" We'll go over how to get more out of records using POINTERs next time.

Lesson 12#

Well, as you probably noticed in the last file that POINTERs and records aren't all that useful by themselves. In fact, I decided not to include a sample program about records because I couldn't think of a good example.

But, when you mix POINTERs and records you get alot of power. You may have tried to make a ARRAY of records. This is illegal in Action! But, if you use POINTERs to accomplish this it will work.

Also, you cannot have a ARRAY as a field in a record. Since strings are just CHAR ARRAYs, you can't, for example, associate a name with other information about a person. Again, POINTERs make this possible too.

Even more, you can't have ARRAYs of strings. You guessed it, POINTERs to the rescue again. I will be covering all of these in this lesson. You don't have to fully understand records and POINTERs to use the concepts I am introducing. You can just copy the routines, etc. and modify them for whatever use you have in mind. But, a good understanding of POINTERs will allow you to even more powerful things in Action!

First, how do you make an ARRAY of records in Action!? First you have to decide what your record will look like.

```
TYPE employee=[CARD ssnumber1
               BYTE ssnumber2
               CARD ssnumber3
               BYTE department,
                    salary ]
```

Now we have to count up the total number of bytes in the record. CARDs and INTs take up 2 bytes and BYTEs take up 1. So, here we have 2 CARDs and 3 BYTEs for a total of 7.

```
DEFINE size = "7"
```

We have to decide how many records we want to hold. Our company is kind of small, we only have 6 employees. So, 6 employees each taking up 7 bytes. We have to reserve 42 bytes of memory to hold our information because 6*7=42.

```
BYTE ARRAY company(42)
```

Now its time to make that POINTER I have been talking about.

```
employee POINTER info
```

'employee' is the type of record that the POINTER will point to. 'info' is the name of the POINTER.

Lastly, you need an equation to figure out where in memory this really is. This equation is the same one we will use for all advanced record and POINTER manipulations.

```
info = company + (counter * size)
```

That's it!

label | use info | The name we gave our POINTER company | The ARRAY we used to reserve memory. counter | This is the record number we wish to look at. Since we have 6 employees this will be a number from 0 to 5. This can be a constant like 3 or a variable like in a FOR loop. size | The number of bytes in a record in our DEFINE line.

Ok, so how do you use it? Easy... If we want to enter employee number 3's information.

```
info = company + ( 3 * size)
info.ssnumber1 = 392
info.ssnumber2 = 80
info.ssnumber3 = 4593
info.department = 3
info.salary = 7
```

And if employee #2 got a raise to paycode #10

```
info = company + ( 4 * size)
info.salary = 10
```

If you want to print employee #0's social security number:

```
  info = company + ( 0 * size)
  PrintC( info.ssnumber1)
  Print("-")
  PrintB( info.ssnumber2)
  Print("-")
  PrintCE( info.ssnumber3)
```

The numbers in the record can be used just like any other variable.

```
  info.department = 4 * 5
```

Or whatever. Here is a little picture that I hope will help you to see what is going on. In case you didn't know, when you declare a ARRAY in Action!, the ARRAY name is actually a POINTER to where the ARRAY is in memory. So, let's say our ARRAY starts at memory location 16000. Also, I'll use - to represent bytes. So a CARD would have 2 bytes, --.

```
  TYPE employee=[CARD ssnumber1
                 BYTE ssnumber2
                 CARD ssnumber3
                 BYTE department,
                      salary ]


record 0     record 1    record 2
-- - -- - -|-- - -- - -|-- - -- - -|
^
|
16000
```

To get record 1 we use this formula:

```
  info = company + ( 1 * size)
```

Sticking in the numbers:

```
  info = 16000 + ( 1 * 7)
  info = 16007
```

So, we move our pointer over 7 bytes to location 16007.

```
record 0     record 1    record 2
-- - -- - -|-- - -- - -|-- - -- - -|
            ^
            |
            16007
```

To get record 2:

```
  info = 16000 + ( 2 * 7)
  info = 16014
```

So, we move our pointer over 14 bytes from location 16000 to 16014

```
record 0     record 1    record 2
-- - -- - -|-- - -- - -|-- - -- - -|
                        ^
                        |
                        16014
```

But records can't contain strings. So POINTERs must be used again to trick Action!

What if you wrote a game where there are multiple levels. When the first player dies, the second player takes over, right where he left off. Well, you have to keep track of the players score, level and name. The score and level are easy with records, but the name? We'd use this:

```
TYPE record = [BYTE level
               CARD score
               BYTE name]
```

Why only 1 BYTE for the name? This is just the first BYTE of the players name. We'll save more space for it in a second. Count up the bytes without the name BYTE. 1 BYTE and 1 CARD = 3 bytes

```
DEFINE offset = "3"
```

We'll save 20 bytes for the name. So add the length of the name and the offset value for the size.

```
DEFINE size = "23"
```

How many players maximum can play our game? We'll just say 8. So, 8 * 23 = 184

```
BYTE ARRAY players(184)
```

Now we need 2 POINTERs. One like normal, and 1 to point to the name.

```
record POINTER active_player

CHAR POINTER his_name
```

The first POINTER we use just like we have been before.

```
active_player = players + (count * size)
```

But to get the name we use:

```
his_name = active_player + offset
```

That's it! We'll so some assignments to record number 4.

```
active_player = players + (4 * size)
his_name = active_player + offset

active_player.level=1
active_player.score=0
InputS(his_name)
```

Now lets output player 6.

```
active_player = players + (6 * size)
his_name = active_player + offset

Print("Level: ")
PrintBE(active_player.level)
Print("Score: ")
PrintCE(active_player.score)
Print("Name:  ")
PrintE(his_name)
```

Ok, the last situation like these is if you need an ARRAY of different size strings. Let's say a you need a program to keep track of your customers first name, lastname and the last date the ordered from you. We'll say the date is of the form mm/dd/yy.

How big to we want each field?

field | size in bytes firstname | 11 lastname | 14 date | 9

Try to declare your sizes 1 bigger because strings go from 0 to 1 less than their size. This is needed because, remember, the 0th byte is the length of the string.

There are no records involved this construct. Here we just use POINTERs to get around. The first DEFINE always starts at 0.

```
DEFINE firstname = "0"
```

If you want to the firstname to be 11 bytes, the lastname must start at the 12th byte. 0 + 12 = 12

```
DEFINE lastname = "12"
```

If the lastname is 14 bytes long the date must start 15 bytes later than the lastname. This means the 27 byte overall. 12 + 15 = 27

```
DEFINE date = "27"
```

The total size is 27 + 9 = 36

```
DEFINE size = "36"
```

How many customers do you have? Let's just say 100. 100 * 27 = 2700

```
BYTE ARRAY data(2700)
```

Since we don't have a record only characters, our POINTER is just:

```
CHAR POINTER ptr
```

The POINTER is just like normal

```
ptr = data + (counter * size)
```

Once again, that's it! For customer #53:

```
ptr = data + (53 * size)

Print("First name: ")
InputS(ptr + firstname)

Print("Last name: ")
InputS(ptr + lastname)

Print("Last order date: ")
InputS(ptr + date)
```

APROG12.002 is a full featured phone book based on this last subject.

Lesson 13#

Well, here we are at the last lesson of Part 1. And I have to apologize for this one. There are a lot of little things I want to mention. Most of them have nothing to do with any of the others.

The first has to do with assigning variables and routines to specific memory locations. You saw how you can use POINTERs to change memory locations (like the background color of the screen). But there is a better way. When you declare a variable, you can assign it to a location.

```
BYTE bkgnd=710
```

Now any change you make to bkgnd will go into location 710.

```
bkgnd=0
```

This will change the background color to black. You can also assign PROCs to specific memory locations too. That way, if there is a routine loaded into memory, you can call it easily.

```
PROC machine_routine=$3500 (BYTE ARRAY s)
```

In Part 2 of this tutorial we will use this quite a bit.

There is another thing you can do with variables where you declare them. You can assign them to a value using [square brackets\].

```
INT i=[-32]
BYTE b=[0]
```

So, if you do a:

```
PrintIE(i)
PrintBE(b)
```

you will get

```
-32
0
```

Square brackets are also used for inserting machine language subroutines in Action! programs. You can use hex values or decimal. You can even access Action! variable. Here's an example:

```
[$A9 my_val $2A $8D $CD $04]
```

Action! is very fast but sometimes you can use that little extra burst from machine language.

If you want your Action! code to compile to a specific memory location you just insert this at the start of your program.

```
SET 14 = $4000
SET $491 = $4000
```

Now you program will compile to location $4000.

- ◦ SET IS NOT LIKE A POKE! SET CHANGES

MEMORY LOCATIONS AT COMPILE TIME, NOT WHILE IT IS RUNNING!!!**

If you create a bunch of global routines that you in all your programs, you don't have to type them in each time. They can be loaded in while the program is compiling. That way they are compiled in with your code. You can INCLUDE a file anywhere as long as if is above whatever PROCs are going to call them. You usually INCLUDE files after the global variable and before your first PROC.

I have included a Public Domain runtime package. If you INCLUDE this when you compile your programs, they will work without the Action! cartridge. It would look like:

```
INT global1,
    global2

INCLUDE "D:RUNTIME.ACT"

PROC My_first_proc()
     .
     .
     .

RETURN
```

I mentioned that Action! predeclares some variables for you. There is an important one that HAS to be mentioned.

Normally, if you Action! program finds an error, it aborts. Your can trap errors by making PROCs that will execute when an error occures.

```
PROC my_error(BYTE errno)

  Print("I found an Error number ")
  PrintBE(errno)
RETURN
```

A BYTE parameter is always needed. To make Action! go to this PROC instead of aborting, you just put in:

```
Error = my_error
```

Lastly, Action! has the same graphics command built in as BASIC. I am not going to go into great detail on graphics. There have been thousands of magazine articles written on the subject. I will just list the BASIC graphics command and their Action! equivalents.

BASIC | Action! GRAPHICS 0 | Graphics(0) SETCOLOR 1,0,0 | SetColor(1,0,0) COLOR 2 | color=2 POSITION 3,4 | Position(3,4) PLOT 10,10 | Plot(10,10) DRAWTO 20,20 | DrawTo(20,20)

I have not listed all the routines in the Action! library. There are sound and game controller routines as well as a ton more.

Take a look at your Action! manual again. Hopefully you will understand it better than the first time you opened it and said "What does that mean?!?"

Well, I hope you all have enjoyed this tutorial. Action! is a great programming language and Action! programmers are very friendly.

Stay tuned for Part 2, writing programs for BBS Express! Pro.

Tom