**General Information #**

Author: Russ Wetmore
Language: ACTION!
Compiler/Interpreter: ACTION!
Published: Analog #35 (10/ 85)

---

# ON-LINE #

## Getting in on the Action! #

This article, both part one (**ANALOG Computing**, issue 32) and this month's segment, was written for advanced programmers. Don't feel badly if you've dabbled a little in Action! and can't make any sense out of the examples in this article. Some of the concepts are quite advanced and are mainly aimed at the experienced programmer who wants to squeeze more functionality out of the Action! cartridge.

### Modularizing.#

I recently completed a major undertaking in Action! - an integrated three-program package called **HomePak**. All together, these three programs take up about 64K of disk space, not counting the various global subprograms required. like an RS232 handler, character sets, etc.

Two of the programs were too large to compile using standard methods. I faced an interesting decision: recode substantial portions of the program in assembly language (avoiding such being one major reason I did it in a high-level language to begin with) or leave out possible features in order to save space.

I hit upon another, option: compiling the program in pieces. In fact, this saved me time, as I didn't have to compile the whole program every time. Let's face it. Many portions of an Action! program are static variables and arrays that almost never change. Why compile them every time, just to find out their addresses so that the rest of the program can tell where they reside?

There's an "undocumented" feature of the Action! cart you need to know before you can do this. I'll describe it first.

### Compilation offset.#

In page 0, $B5-$B6 is used by the compiler as a compilation offset value. The three **HomePak** programs reside at $3400, which is well above the $2404 address that the cart tells me is my LOMEM value. The manual tells you that you can do the following:

```
SET $E = $3480
SET $491 = $3480
```

to set the base address to $3400, but this throws away a good 4K(!) of memory I need to compile to. A better way of handling it is to compile the program to the LOMEM address, but specify an offset to the compiler. That way, when the program gets written out to disk, it loads at the proper address. You can do this by putting a value in $B5-$B6 (using the set command), which is your base address minus the LOMEM address found at $491. Thus, if your LOMEM value is $2404, and you want your program to load at $4000. you'd put:

```
SET $B5=$1BFC ;(which is $4000-$2404)
```

at the very beginning of your program. The program. when compiled, would reside in memory during compilation at the $2404 LOMEM address, but when written to disk, will appear to load at $4000.

In order for this to work properly, check the value at $491 while the edit buffer is empty. Since any pro-gram in memory pushes up the LOMEM value, you'll have to do your compiling from disk, rather than from memory. It's either that, or check the value every time you want to compile, and alter the program accordingly.

Note: There are a couple of bugs in the current version of the cart that effect the offset value. Negative offsets don't work, so you can't use this trick to compile below the LOMEM address. Also, there is a subtle bug involving type definitions. If you use the $B5-$B6 offset, and your program uses the type construct, you must set the offset to 0 before any type definition - and set it back to its original value afterwards. Example:

```
MODULE  ;Example 1
SET $B5 = $1BFC
;(compile to $4000,
;from LOMEM of $2404)

BYTE
  i. j, k ;some variable definitions

SET $B5 = 0
SET $B6 = 0 ;account for bug
          ;involving TYPE statements
TYPE DISK = [ CARD sector BYTE pos ]
SET $B5 = $1BFC ;return offset to what
       ;it used to be
?
```

Notice that I had to do two set statements, because the Action! compiler will always try to make a set value a byte, if it can. We need to set the card at $B5, so we need to set each byte of the card value.

**Getting down to it.#**

Now we know how to tell Action! where we want our modules to reside. I generally have a file named GLOBALS.H, which is my header file with seldom-changed global variables. I compile this separately, to the desired base address of my whole program.

Once the compilation is finished and I've written the program to a disk file, I use the debugging portion of the monitor to find the end addresses of those variables. (Once a program is compiled - and before any system errors occur - use the program variables in the monitor as you would constants.)

Let's take an example. Type this in and save it to disk as EXAMPLE2:ACT:

```
MODULE ; Example 2
; This is my global variable file

SET $B5 = $1BFC
; so program compiles to $4000 from
; $2404 LOMEM. Note: your system
; probably has a different address
; for LOMEM than mine. The value for
; LOMEM will differ depending on what
; DOS you're using, how many drives
; and file buffers you have allocated,
; etc. Do a ?$491 at the monitor
```

```
; with an empty edit buffer to find
; your LOMEN, and subtract it from
; $4000 to get the proper SET value
; for your computer.

BYTE
  two = [2], three = [3] , four = [4]
```

Okay, okay, it's short. But, then, this is just an example, right?

After you've written the file to disk, be sure to clear the source from memory, so your LOMEM value is correct. Compile the module using the command C D:EXAMPLE2.ACT. Once it's compiled, type in W EXAMPLE2.OBJ at the monitor, to write your object file to disk. Now, type this in:

```
 ?two
 ?three
 ?four
```

This tells us the addresses of our byte variables, two, three and four. (You should get the values $4000, $4001 and $4002, respectively.) The last step is to type in ?$E to get the address of the end of the program, which should return the value $2407 (or what-ever your LOMEM value is, plus 3).

Some of you are ahead of me, I can tell - the value returned is the proper value, all right, but relative to the object file as it currently exists in memory. You have to add your set value to it to get the final address, so $2407 + $1BFC (or whatever your values are) = $4003 - which is what we expect it to be.

Now we can start with our second module. Type this in:

```
MODULE  ; Example 3
SET $B5 = $1BFF

; Note that this value is $4003 (the
; address of the byte following the
; first module) minus $2404, my LOMEN
; address. Again, as in Example 2,
; adjust your values accordingly.

; First off, we have to tell this
; module where our globals are:

BYTE
  two = $4000, three = $4001, four = $4002

; Now, for this module's code:

PROC Main()
BYTE
  i, j, k

  i = two + three
  j = three + four
  k = two + three + four

  PrintF("i=%U, k=%U, k=%U%E",i, j, k)
RETURN
```

Save this source file to disk as EXAMPLE3.ACT. Clear the source code from memory, then go to the monitor and type C EXAMPLE3.ACT to compile it. Type W EXAMPLE3.OBJ to write the object code to disk.

We now have two object files on disk. Exit Action! to DOS and type in the following at the DUP.SYS menu:

```
C [RETURN]
EXAMPLE3.OBJ, EXAMPLE2.OBJ/A [RETURN]
```

This appends the second module onto the first. You can now run EXAMPLE2.OBJ, and the result:

```
i=5, j=7, k=9
```

should be printed to your screen.

Using variations of this procedure, you can create programs that are much larger than can be physically compiled. You'll save time, since you won't have to recompile everything. every time.

**ON X GOSUB/GOTO.#**

There's a C language construct whereby you can pass the address of a function to a function. (For those of you who don't know C, you might want to skip over this section; I'm using C here because the examples will serve as a basis for its emulation in Action!) Here's a short example:

```
/* Example 4 */
static void PrtNum(num)
unsigned char num;
{
  printf("We want to print ");
  printf("the number %U here", num);
}

static void PrintANumber(routine, num)
void (*routine)();
unsigned char num;
{
  (*routine)(num);
}

void main()
{
  PrintANumber(PrtNum, 5);
}
```

PrintANumber in the above example takes the address of a function as its argument, and executes it directly. Since the PrtNum routine (actually, the address of Prt Num) is passed, it is executed at the Prin-tANumber call in the main function.

We can carry this concept a little further - by using arrays of addresses to functions. This gives us the tools we need to do our emulation of BASIC's ONxGOSUB function:

```
/* Example 5 */

/* Global declarations */

/*
  FUNCPTR is typed as a pointer to a
  function returning void (no value)
*/
typedef void (*FUNCPTR)();

/*
```

```
  Here, we have to tell the compiler
  ahead of time what we're up to:
  we're using these names as functions
  returning void
*/
void Print1(), Print2(), Print3();

/*
  routines is an array of pointers
  to functions returning void
  (n'est-ce pas?)
*/
FUNCPTR routines[] =
  { Printl. Print2, Print3 };

static void Print1()
{
  puts ("Subroutine number 1\n");
}

static void Print2()
{
  puts("Subroutine number 2\n");
}

static void Print3()
{
  puts ("Subroutine number 3\n");
}

void main()
{
  unsigned char i;
  for (i = 0; i <= 2; ++I)
    (*routines[i])();
}
```

This little program does a lot. First, it executes a "for" loop for the values between 0 and 2. The "pointer" to the desired function is fetched (routines[i]). which is then executed directly. Routines is an "array of pointers" to functions, with three elements (numbered 0 to 2).

This example has the same function as BASIC's ONxGOSUB. The equivalent BASIC would be:

```
0 REM BASIC version of C code
10 FOR X=1 TO 3 20 ON X GO5UB 100,200,300
30 NEXT X
40 END
100 PRINT "Subroutine #1":RETURN
200 PRINT "Subroutine #2":RETURN
300 PRINT "Subroutine #3":RETURN
```

**Translating to Action!#**

We can carry these same basic concepts over to Action! There's an eccentricity of the compiler that we need to know first. We can't declare an array of procs or funcs, because such a declaration requires a constant at compile time.

We can, however, declare a code block that includes proc and func addresses, and point an array name to it. For example, to emulate the C example above in Action!. we'd do the following:

```
MODULE ; Action! version of Example 5
; First, let's define the PROC's
; to be called:
PROC Print1=*()
 PrintE("Number 1")
RETURN
PROC Print2=*()
 PrintE("Number 2")
RETURN
PROC Print3=*()
 PrintE("Number 3")
RETURN
; Next, we define a dummy PROC which
; holds the addresses of the PROC's
; we want to execute:
; (We can't define these in a
; CARD ARRAY because they're NOT
; constants and Action! would choke
; on them.)

PROC dummy=*() [
  Print1 Print2 Print3 ]

; Now, a MODULE statement because
; we have to declare a variable:

MODULE ; for CARD ARRAY declaration

; This declares a CARD ARRAY that
; points (suprize!) to "dummy"
CARD ARRAY ptrary = dummy
; This routine does a JMP indirect
; to the address passed to it:

PROC Indirect=*(CARD address) [
  ; ("address" is passed in the A and
  ; X registers)

  $85 $AE ;STA $AE save low byte
  $86 $AF ;STX $AF save high byte

; NOTE! To change this to emulate ON x GOTO rather than
; ON x GO5UB, add this line here:
;
; $68 $68 ;PLA/PLA pull off
          ; return address

  $6C $AE $00 ] ;JMP ($AE)
; jump indirect to routine, which
; RTS's itself to the calling PROC
; Now, our version of the
; C "main" function:

PROC main()
BYTE i
  FOR i = 0 TO 2 DO
```

```
    ; Fetch address of routine to
    ; call ( ptrary(i) ) and execute
    ; it (via "Indirect" PROC)
    Indirect(ptrary(i))
  OD
RETURN
```

Notice the indirect procedure. We have to do this, because there we have to jump "indirectly" to the routine address. Another way of handling this would be to jump indirect directly into the card array, but this would require self-modifying code (which is a no-no).

To convert the above to emulate BASIC's ONxGOTO, we just insert two PLAs in the indirect procedure, to pull the return address off the stack.

**Arrays of arrays.#**

The last foray we're going to make right now into extending Action!?s functionality is the concept of "arrays of arrays." Action! arrays want to be only one-dimensional, which is prohibitive in a lot of real world programming needs.

Let's take a simple two-dimension byte array. An array of arrays can basically be considered to be an array of pointers to arrays. Since pointers are actually cards in disguise, it follows that, to create an ar-ray of arrays, we need to do the following: (1) declare the individual byte arrays; and (2) declare a card array of the addresses of the individual arrays.

We have the same problem we had before - we can't declare an array using values which aren't constants. But we know how to get around that now, right? Here's an example:

```
MODULE ; Example 6
; Declare our individual arrays:
BYTE ARRAY
  one()   = [ 1, 2, 3 ],
  two()   = [ 4, 5, 6 ],
  three() = [ 7, 8, 9 ]
; Declare a dummy PROC with the
; addresses of the BYTE ARRAYS:

PROC dummy=*() [ one two three ]

; MODULE statement because we're
; declaring a variable:

CARD ARRAY
  ary_of_arys = dummy

; Now, our main procedure, which
; illustrates how to access our doubly
; subscripted arrays:

PROC main()
BYTE i, j
BYTE ARRAY bary

  ; loop for first subscript:
  FOR i = 0 TO 2 DO

  ; fetch address of array:
  bary = ary_of_arys(i)
```

```
  ; loop for second subscript:
  FOR j = 0 TO 2 DO
    PriniF("Array(%U)(%U) = %U%E",
            i, j, bary(j))
    OD
    PutE()
  OD
RETURN
```

You should get the following output when you run this example:

```
Array (0) (0) = 1
Array (0) (1) = 2
Array (0) (2) = 3
Array (1) (0) = 4
Array (1) (1) = 5
Array (1) (2) = 6
Array (2) (0) = 7
Array (2) (1) = 8
Array (2) (2) = 9
```

You can, of course, carry this out ad infinitum - as many layers as you like - by declaring card arrays for each layer of arrays.

Another typical use of arrays of arrays in programming is "string arrays", where strings are considered to be arrays of characters (as in C and Action!). I'll give a more useful example here:

```
MODULE ; Example 7
; This subroutine prints out an
; English explanation for the user
; when a system error occurs (Only
; errors 128 through 144 are given
; for space reasons)

DEFINE LASTERR = "144"

CHAR ARRAY
  s128() = "BREAK key abort",
  s129() = "IOCB already open",
  s130() = "Nonexistent device",
  s131() = "IOCB Write only",
  s132() = "Illegal handler command",
  s133() = "IOCB not Open",
  s134() = "Illegal IOCB number",
  s135() = "IOCB Read only",
  s136() = "End of file",
  s137() = "Truncated record",
  s138() = "Device timeout",
  s139() = "Device NAK",
  s140() = "Serial frame error",
  s141() = "Cursor out of range",
  s142() = "Serial bus overrun",
  s143() = "Checksum error",
  s144() = "Device done error",
  generic() = "Error %U!%E"

PROC dummy=*() [
  s128 s129 s130 s131 s132 s133 s134
  s135 s136 s137 s138 s139 s140 s141
```

```
  s142 s143 s144 ]

MODULE ; for variable declaration

CARD ARRAY
  errstrs = dummy

PROC PrintError(BYTE errnum)
  IF errnum > 128 THEN
    IF errnum > LASTERR THEN
      PrintF(generic, errnum)
    ELSE
      PrintE(errstrs(errnum - 128))
    FI
  FI
RETURN
```

I'll leave it as an exercise to you, to figure out how this last example works. It's much like the preceding example, if that's any help.

That's it for this journey into esoterica. The Action! language has many capabilities that most people will never see or use. I hope I've at least sparked some of you to do more investigative work.

Next month is letters/feedback time. I've gotten a lot of response to my articles on piracy - some pro and a surprisingly high number on the con side. I'll share some of the more representative ones with you next month.

---

*Russ Wetmore has been involved in the computer industry for over six years. He's probably best known for his game **Preppie!** and is president of Star Systems Software, Inc., a research and development firm specializing in entertainment and home productivity programs.*

---