ON-LINE

Getting in on the Action!

by Russ Wetmore

Action! is an Atari programmer's dream come true. It is a language not too unlike C or Pascal, but which compiles to very "tight" 6502 machine language. Clint Parker, the author of Action!, has fashioned a remarkable programming environment, where editor, compiler and monitor are all resident at once.

Write your program, compile it, run to test it, then dump right back into the editor with your source code intact, to start making corrections. I've done a couple of major projects using Action! in the past year and can recommend it without hesitation to any serious (or casual) Atari programmer.

There are several *caveats* in creating really big programs (larger than 16K), because of the integrated environment. If you're planning to write such programs, it's necessary to know how Action! creates object code from your source, in order to maximize memory usage. There are also a few bugs that need to be noted.

In this article (and the one next month), I'll show you some tricks I've learned to optimize Action!'s output. These comments all apply to version 3.6—they may work on other versions of the compiler, but have not been tested. They also assume a working knowledge of Action!

Variable allocation.

Allocating free memory.

There isn't a function in Action! that approximates BASIC's FRE(0) command. It isn't as simple as checking the monitor to see where the end of your program is, because Action! tries to help you out by placing some non-initialized arrays beyond the end of your program code, instead of inside your program, where they're declared (specifically, CARD ARRAYs and, generally, BYTE ARRAYs over a page in length).

Luckily, there's an easy method for determining where the end of the program and variable space actually is. The first CARD ARRAY declared in a program is the *last* actually allocated during compilation.

```
MODULE ; Sample 1

CARD
  MEMTOP=$2E5, freemem=[0]
CARD ARRAY
  EndOfProgram(1)

PROC Main()
  freemem=MEMTOP - EndOfProgram
  PrintF("Total free memory=%U%E",
         freemem)
RETURN
```

Static ARRAY variables.

Action! allows you a lot of choices when it comes to variable declaration. For example, ARRAY variable names are actually *pointers* to the ARRAY space. This allows you to do such esoterics as:

```
MODULE ; Sample 2

CHAR ARRAY
  str1="This is a test.", str2

PROC Main()
  str2=str1
  PrintE(str2)
RETURN
```

When you run the program, you'll find that str1 and str2 both "equal" the same string. This is possible because Action! also allocates a *pointer* to the ARRAY, in addition to the ARRAY data itself. When you assign str2 to str1, you're actually just assigning str2's pointer equal to str1's, which is pointing to the ARRAY data.

In many cases, though, this overhead costs memory for arrays that you're never going to reassign, such as string constants. Also, if you were to reference the ARRAY name in a code block, you'd have to go through contortions in order to get to the actual data, because the ARRAY name equals a pointer to the data, which you'd have to access indirectly. Clint very thoughtfully put in a construct that allows you to declare ARRAY variables *without* the associated pointer. Declare the ARRAY with a predefined length of 0. For example:

```
CHAR ARRAY
  str1(0)="This is a test."
```

You won't be able to reassign str1 (you'll get an error if you try), but you will have saved 2 bytes you probably never would have used, anyway. You'll also save 2 bytes every time you reference the ARRAY, because Action! will compile the reference as immediate loads of registers, as opposed to indirect fetches from memory. For example:

```
MODULE ; Sample 3a

CHAR ARRAY
  str1="This is a test."

PROC Main()
  PrintE(str1)
RETURN
```

compiles to:

```
MAIN    LDA str1
        LDX str1+1
        JSR PrintE
        RTS
```

whereas the following:

```
MODULE ; Sample 3b

CHAR ARRAY
  str1(0)="This is a test."

PROC Main()
  PrintE(str1)
RETURN
```

compiles to:

```
MAIN    LDA #<str1
        LDX #>str1
        JSR PrintE
        RTS
```

For similar reasons, you may save memory if you predeclare *all* your variables, ARRAYs or otherwise. For example, when you declare a BYTE variable, you can set its memory address in the declaration. Any variables that follow it in the same statement, though, have extra overhead associated with them. (You can see this effect in the following example.) To test all of these constructs, you can compile a test program then execute the command *?$493* from the monitor, to see the program's length. Try this with the following two examples:

```
MODULE ; Long example

BYTE
  COLOR1=$2C4, i, j, k
CARD
  MEMTOP=$2E5, c, d, e
CHAR ARRAY
  str1="Test1", str2="Test2"

PROC Main()
RETURN


MODULE ; Shorter example

BYTE
  COLOR1=$2C4, i=[0], j=[0], k=[0]
CARD
  MEMTOP=$2E5, c=[0], d=[0], e=[0]
CHAR ARRAY
  str1(0)="Test1", str2(0)="Test2"

PROC Main()
RETURN
```

You'll find that the second example ends up being 19 bytes shorter than the first.

A string shortcut.

If you work with strings at all, you probably know that the length of a declared string is always the first ("zeroth") byte of the ARRAY. As such, you probably use a construct similar to:

```
MODULE ; Sample 4a

CHAR ARRAY
  str1="Test"

PROC Main()
  PrintF("Length of %S is %U%E",
          str1, str1(0))
RETURN
```

You can save considerable memory (11 bytes each occurrence!) by declaring a separate BYTE variable:

```
MODULE ; Sample 4b

CHAR ARRAY
  str1="Test"

BYTE
  str1len=str1

PROC Main()
  PrintF("Length of %S is %U%E",
          str1, str1len)
RETURN
```

By making the declaration *str1len = str1*, we're setting str1len's memory location equal to the "zeroth" byte of str1, hence str1len will always be equal to the length of str1 (if you don't point str1 elsewhere). The reason for the memory savings is simple. In the first example, the compiler is given the address of the start of the ARRAY and an offset to the actual byte desired. This compiles to something similar to:

```
LDA str1       ;Fetch address of array
STA $AE        ;Save for indirect ref
LDA str1+1     ;Fetch high byte
STA $AF        ;Save...
LDY #0         ;We want 0'th element
LDA ($AE),Y    ;Fetch string length
```

If we declare a BYTE variable outright, though, it will already be pointing to the proper memory location, and no calculation is needed to find it. Thus, the compiler produces something like:

```
LDA str1len ;Fetch string length
```

which, I think you'll agree, is much cleaner. You can apply this principle to *any* portion of a declared ARRAY that isn't going to move, that you need to access.

PROC and FUNC addressing.

In the Action! manual, reference is made to "addressing routines." Besides the example given, there's little said about how useful this construct can be.

Forward references.

Action! is a one-pass compiler. Most compilers use a two-pass method, where the entire source program is scanned first to build a symbol table of variable addresses. Thus, on the second pass, if a variable is used before it is declared, the compiler can look it up in the symbol table to find its address.

Action!, however, only makes one pass through a program for speed reasons. This means that *every* procedure or function is supposed to be previously declared before you reference it. Sometimes this isn't feasible, but how do you get around it?

One other feature of Action! is the ability to reassign PROCs and FUNCs to different memory locations from where they are first compiled. If you run the following example:

```
MODULE ; Sample 5

PROC Num1() PrintE("ONE") RETURN

PROC Num2() PrintE("TWO") RETURN

PROC Main()
  Num2=Num1 Num2()
RETURN
```

you'll get the result *one* printed to the screen, because we've "pointed" Num2 to Num1's address. Using this same concept, we can forward reference a PROC or FUNC before it is declared!

```
MODULE ; Sample 6

PROC DUMMY()

PROC Num1() DUMMY() RETURN

PROC Num2() PrintE("TWO") RETURN

PROC Main()
  DUMMY=Num2 Num1()
RETURN
```

In Num1, we've actually forward referenced Num2 indirectly, by setting DUMMY to be equal to Num2.

An indirect detriment.

Unfortunately, as in the case of non-initialized ARRAYs, the overhead for such indirection is the default case. I have very rarely used the addressing feature and, even then, only in cases where I was too lazy to redo the necessary routines properly.

Action! compiles normal PROC references in a manner similar to this example:

```
MODULE ; Sample 7

BYTE
  test

PROC DUMMY()
RETURN

PROC Main()
  PrintBE(test)
RETURN


test      .DS 1

DUMMYvec JMP DUMMY
DUMMY    RTS

Mainvec  JMP Main
Main     LDA test
         JSR PrintBE
         RTS
```

If you were to do the assignment DUMMY=Main, what the compiler would actually produce is:

```
LDA #<Main
STA DUMMYvec+1
LDA #>Main
STA DUMMYvec+2
```

so that the resulting code at DUMMYvec would actually become JMP Main. If you don't ever use this feature, though, every time you declare a PROC or FUNC, you're actually throwing in a JMP to the next instruction.

The way to avoid this automatic inclusion of the JMP command is to use the construct:

```
PROC procname=*()
```

You save three bytes and a little overhead in speed when you declare routines this way. One important note—this construct will *not* work if you're passing variables to a routine, unless the first thing encountered in the routine is a code block. This is because of the way that Action! handles saving its zero-page working variables.

Modularizing programs.

You can also use this construct to "modularize" your programs. This is important if you're trying to compile large programs. Frequently, you'll run out of symbol table space or, worse yet, run out of memory to compile to because the cartridge eats 8K of space itself, in addition to other overhead.

You can compile all of your constant strings and low level routines, for example, separately from your main program and reference them in your program through equates and routine addressing. You can then use the SET command in the second module to compile the second module above the first, then append your files together to get the final object file. I'll go more into detail on how to do this next issue.

Also, next time I'll cover ARRAYs of ARRAYs (string ARRAYs, for example), an Action! version of BASIC's "ON x GOSUB" and "ON x GOTO" commands, plus other surprises. □

Russ Wetmore has been involved in the home computer industry for over six years. He's probably most widely known for his best-selling, award-winning Atari game program ***Preppie!*** *He has also shown his talent as a composer/arranger whose work has been heard on national TV. Russ is President of Star Systems Software, Inc., a research and development firm specializing in entertainment and home productivity programs for a host of computers.*