A Reference GUIDE to Using

The ACTION! Run Time Package

with Atari Computers

This manual revised March, 1984

INTRODUCTION

The Action! Run Time Package (which we will call simply "RunTime" from here on) is designed to aid users of the OSS ACTION! cartridge-based language. Specifically, by using RunTime, you can compile an ACTION! program in such a way that the Action! cartridge is no longer needed when running the compiled program.

The primary advantage of using RunTime is that it allows you to give copies of your efforts to your friends, user group members, etc. Remember, though, that you OR your publisher MUST purchase the Commercial License for RunTime if you wish to SELL programs written in Action!

A secondary advantage of using RunTime is that you may produce extrinsic commands (i.e., programs with a ".COM" file name extension) for use with OS/A+ or DOS XL. Again, you could use these new commands at any time, not just when your ACTION! cartridge is installed.

Section 1 of this guide describes how ACTIONI compiles programs, how it builds its symbol tables, and other information you may find useful when compiling programs written in ACTION! We suggest you read Section 1 very carefully.

This documentation then presents you with four possible ways to use RunTime. We suggest that you write and compile a short program using the methods described in Section 2 first. Then you can read the first few
paragraphs of each of Sections 3, 4, and 5 to see if the methods described in each of those sections will be useful to you.

Finally, Section 6 provides a memory map of the most useful and interesting memory locations used by either the compiler or the RunTime system. Many of these locations are discussed in detail in other sections, so section references are provided, if appropriate.

Section 1: How ACTION! Works

1.1 Compiling a Program

When the Action! monitor receives a compile request, it initializes certain of its tables, sets and uses certain memory pointers, and then begins producing 6502 machine code directly into memory, it pays attention to certain system variables which will be described here.

In the discussions which follow, we use square brackets to indicate memory which is pointed to by the named or addressed location. Thus, [$02E7] means "the memory location(s) pointed to by the contents of location $02E7". In general, words which are printed in all capital letters are labels given in the memory map of Section 6.

1.1.1 Memory Allocation

Unless you tell it otherwise, Action! uses memory as follows:

The edit buffer starts at [APPMHI] ([$0E]). This pointer is itself derived as an offset (of about $700 bytes) from [LOMEM] ([$2E7]). The space between [LOMEM] and the initial location of [APPMHI] is used for various semi-fixed buffers, tables, etc.

As you edit your program, Action! changes APPMHI as appropriate.

When you ask to compile your program, APPMHI is copied to CODEBASE ($0491). Also, CODESIZE is cleared to zero.

Symbol table space is allocated from the top of memory downward. The symbol table itself contains symbols for both global and local variables (which part of the table is used for what is controlled by the "hash tables", part of the "semi-fixed" memory mentioned above). The amount of space allocated is determined by STSP ($0495), which may be changed by the user (see Section 3).

As your code is compiled, Action! adjusts [APPMHI] to reflect the top of the compiled code. Also, CODESIZE is incremented to reflect the amount of code generated.

After the code is compiled, the monitor's 'W' command uses CODEBASE and CODESIZE to determine what part of memory to write to the object file.

Note the most important implication of the above: if you do NOT have a program in memory, your code will be generated at the lowest practicable memory address. Supposition: If it can be compiled at the lowest address, or at a higher address determined by the top of the edit buffer, perhaps it can be compiled anywhere. Actually, that supposition is almost true.

The only real limitation is that Action! 's semi-fixed buffers, your compiled program, and your symbol table must, somehow, fit in the memory between the top of DOS ([LOMEM]) and the bottom of the screen memory ([HIMEM]).

Note that if you use DOS XL (the version titled "DOSXL.SUP" on the version 2.3 and above distribution disk), you will automatically be using a LOMEM value, which gains you a significant amount of memory. Unfortunately, the program thus compiled may not then be able to run without the Action! cartridge installed, since it will overlay part of the lowest memory used by any non-DOSXL.SYS version of DOS. However, see Section 4 for information on compiling with an offset and more notes on this subject.

1.1.2 Symbol Table Searches

Whenever the Action! compiler encounters a symbol (e.g., a variable name, a DEFINEd name, a TYPE name, or a PROC name) it always searches for the symbol in three places.

First, the local symbol table is searched. All symbols defined after the keywords PROC or FUNCTION are encountered (except, of course, the actual name of the PROC or FUNCTION) are considered locals. This would include even the parameters to a PROC or FUNCTION.

Second, the global symbol table is examined. All PROC and FUNCTION names are placed in the global table as well as all names encountered before the first occurrence of a PROC or FUNCTION and all names encountered between a MODULE keyword and the next succeeding PROC or FUNCTION.

[May we suggest that you refer to the Action! reference manual if you are not sure whether a given name is a global or local name.]

Finally, if a name is not found in either the local or global symbol tables, it is assumed to be a system library name. The library built into the Action! cartridge is searched for a matching name. Only if the name is not found here will Action! issue an "undefined symbol" error.

1.1.3 Symbol Table Allocation

When you first boot the Action! cartridge, it allocates certain tables and buffers (which we have called "semi-fixed"). These semi-fixed locations are allocated starting at [LOMEM] and occupy approximately $700 bytes. Of these $700 bytes, $400 bytes are used for two 512-byte "hash" tables--one which will hold up to 255 local symbol pointers and a similar one for global symbol pointers. Action! searches for and stores symbols using a "hashing" algorithm, which significantly speeds up such searches but which necessitates these extra hash pointer tables. ("Hashing" is simply a means of using a mathematical formula on a symbol to produce an index--a hash pointer--into a specially structured table.)

When you ask Action! to begin a compilation, Action! first allocates memory for the symbol tables and their associated pointers. It uses location STSP ($0495) to determine how many pages (of 256 bytes each) to allocate to the main symbol table and allocates roughly from the top of free memory (i.e., just under the display memory) downwards.

Note that, even though there are two hash tables, there is only a single symbol table. This is possible for two reasons. First, since a symbol is never actually searched for directly in the symbol table (because Action! always searches via the hash table pointers), the global and local symbols could actually be mixed with no ill results.

But, second, Action! never adds to both tables at the same time. Action! begins by processing global names, adding all variables, etc., which it finds to the global hash table and thus increasing the size of the symbol table. However, when Action! compiles the name of a FUNCTION or PROC, it automatically switches modes--now all new names are added to the local hash table and, as a consequence, to the END of the symbol table. When a subsequent MODULE, PROC, or FUNCTION keyword is

encountered, Action! wipes out the local hash table and allows the symbol table space it accessed to be reused. Since the local names are always at the end of the global names, this procedure ensures that maximum use is made of the available symbol table space.

A last comment on this subject: this methodology explains why the monitor is able to access the local names of only the last compiled PROC or FUNCTION (as well as all global names, of course).

1.2 Running an Action! Program

Since the Action! compiler produces absolute machine-level code, running a compiled Action! program under any DOS for Atari computers is simplicity itself. One need simply invoke any of the normal loaders (including those built into DOS XL, OS/A+, and Atari DOS), being sure to properly pass the "run address" of the compiled program.

The "run address" may be determined by using the '?' monitor command after compiling the program. For example, assume that the name of the last PROC in your program is MAINPROC. Then using '? MAINPROC' from the Action! monitor will produce a display of the address of MAINPROC (in both hex and decimal, as well as its contents, which are not relevant here).

Note, however, that the 'W' command of the Action! monitor automatically writes not only the compiled code but a properly structured INIT vector (see your DOS manual for definition and clarification) as well. Thus, you normally do not need to concern yourself with knowing the starting address.

Since how a program is loaded and run varies from one DOS to another, we will not try to further describe the process here. We would like to note, however, that giving your compiled program a name with ".COM" as the extension will result in a valid DOS XL or OS/A+ command file, which may then be invoked from the 'Dl:' prompt by using just its name.

1.3 When Your Program is Running

Regardless of whether you compile your program using this RunTime package or not, when your program runs it needs to access a host of library routines. Some of these you know about: they are the various library PROCs and FUNCTIONs listed in your Action! manual.

Others, however, are essentially invisible to you. In an attempt to produce a reasonable compromise between code size and code speed, Action! automatically compiles into your program numerous calls (JSRs) to various support routines. Examples of routines thus provided include multiply, divide, and shift routines.

When your program is compiled with the RunTime package, these routines are supplied from the built-in routines in the Action! library "bank". When you use the RunTime package, you actually compile a set of these routines right along with your own code.

A comment: you have probably heard or read about how the OSS SuperCartridge works and may be aware of the fact that it is constantly switching memory banks as it works. When your program runs, though, it uses only a single bank (where the memory resides), and thus the transition to a RAM-based RunTime package is made easier.

Section 2: Compiling a Program with RunTime

You will recall from Section 1.1.2 that Action! always searches for symbols first in the current "local" library, then in the "global" library, and finally in the built-in system library. This sequence is the secret to being able to produce a RunTime Action! program.

As an illustration, early versions of Action! (3.0 and 3.1) had a bug in the system divide routine. Our (temporary) solution was to provide a listing of an Action! routine (which actually consisted of a set of machine code blocks). By including this subroutine (either directly or via INCLUDE) in your program, you could force the compiler to use the new divide routine instead of the built-in one.

Similarly, the RunTime actually consists of a series of Action! PROCedures and FUNCtions (which in turn consist of mainly machine code blocks) which you include with your program so that the compiler will find their names (in your global symbol table) instead of the built-in names.

2.1 A Simple Compile

The simplest method of compiling of a RunTime version of your program is to use a line of the form

```
INCLUDE "D1:SYS.ACT"
```

as the first line of your program.

The file "SYS.ACT" on your RunTime disk contains the Action! source code (mostly in the form of cone blocks) for ALL the routines in the standard system library. Therefore, by compiling this file at the beginning of your program, you are essentially providing the Action! compiler with a full set of global names which will come before and therefor take precedence over the same names in the built-in system library.

As a trial case, may we suggest that you read in and examine the program called "SAMPLE.ACT" which you will find on your RunTime disk. Notice how it INCLUDEs the file "SYS.ACT". If you wish (and only if you are working on a COPY of your RunTime disk), you may go to the monitor and compile this program. After it compiles, simply use the "Write" command in the monitor to write the object code to disk.

Actually, we have already done this for you. We named our object file "SAMPLE.COM". If you are using OS/A+ or DOS XL, you may now exit to DOS (via the "Dos" monitor command) and (when the "D1:" prompt appears) simply type in "SAMPLE". If you are using Atari DOS, you will have to use the DOS "L" option to load the file "SAMPLE.COM". In either case, the program should run and give the expected results.

Simple, isn't that. May we suggest that you try this technique with one or two of your own programs.

2.2 Selective Use of Libraries

In addition to the complete system library provided as "SYS.ACT", your RunTime disk includes several other library files. They are:

SYSLIB.ACT	SYSIO.ACT	SYSGR.ACT
SYSMISC.ACT	SYSBLK.ACT	SYSSTR.ACT

(There is an additional file, "SYSALL.ACT", which simply INCLUDEs all of the above files. This is equivalent to INCLUDEing "SYS.ACT" as we did in Section 2.1.)

Each of these library files contains a part of the complete RunTime library. To use them, simply INCLUDE the ones you need in the same fashion as we INCLUDEd "SYS.ACT" in Section 2.1. Do not INCLUDE the files which contain only routines you do not use.

Thus if, for example, you knew that your program used no graphics routines, you would not INCLUDE "SYSGR.ACT". Virtually all programs need to INCLUDE "SYSLIB.ACT".

For a complete list and short description of all routines included in each of these libraries, you may read or print the file "SYS.DOC" on your RunTime disk (CTRL-SHIFT-R from Action!'s editor to read the file or

```
TYPE SYS.DOC P:
```

from DOS XL or OS/A+).

Unfortunately, there is no easy way of determining which system library routines your program is using. If you omit a RunTime library, it will get "filled in" from the built-in ROM routines. Thus you will simply have to carefully check your program for library routine calls.

In this vein, there is a program on the RunTime disk which can help you. If you compile AND run the program called "ST.ACT", it will hook itself into the Action! compiler in a unique and useful way: As each PROCedure or FUNCtion is compiled, it automatically then and there prints a list of ALL name references made by the PROC or FUNC. You will still have to check the listing by hand for all references, but at least you don't have to search through lines and lines of source code. (See also the file "ST.DOC" on the RunTime disk.)

Finally, note that the file "SAMPLE2.ACT" on the RunTime disk is another version of "SAMPLE.ACT" which we compiled and ran in the previous section. "SAMPLE2.ACT", though, INCLUDEs only those library routines which it needs. If you compile it and Write it to disk, you will notice there is some (albeit not a terribly large) savings in disk (and, consequentially, memory) space.

Again, we have written the compiled file to disk using the file name "SAMPLE2.COM". Follow the instructions above for running the program.

Section 3: Compiling With Large Symbol Tables

You will recall from Section 1 we mentioned that, by default, Action! supports only up to 255 Global symbols (as well as up to 255 Local symbols). The limit on the length of any given symbol (name) is greater than the limit on the length of a line, so virtually any name is valid. However, the total space occupied by names and Action!'s associated type bytes, values, etc., cannot exceed the space reserved via STSP ($495).

This section will discuss how to bypass two of the three limitations noted above. Note that there is currently no way to have Action! recognize more than 255 different local symbols. We do not feel that this is a limitation: if you have a PROCedure or FUNCtion which uses this many symbols, it should probably be broken into two or more subroutines anyway.

3.1 Increasing Your Symbol Table Space

By default, Action! reserves 2K Bytes (2048 bytes, Eight "pages" of 256 bytes each) of RAM for its symbol table. To change the space reserved, you need simply change the contents of STSP (location $495). You must change STSP before you do a compile, since Action! initializes its symbol table pointers, etc., when you give the Compile command from the monitor.

For example, to allow up to 3K Bytes of symbol table space, simply give the command

```
SET $495=12
```

to the Action! monitor and then Compile.

Remember, the contents of STSP is the number of 256-byte pages to be reserved.

HINT: If you have a program which you know will need a particular amount of symbol table space, simply place a SET similar to the one above at the beginning of the program. The program will NOT compile the first time, because it will run out of symbol table space. However, the SET will have taken place, and if you simply compile it again the proper amount of space will then be reserved for you.

3.2 Increasing the Number of Global Symbols

Your RunTime disk contains a file named "BIGST.ACT". Simply compile and run this program and you may then use up to 510 global symbols.

Action! has a flag (BIGST, $4C4) which tells it that you wish to allow an expanded global symbol table. The mechanism Action! uses to accomplish this is very simple: When BIGST is set, Action! splits the global symbol table into two parts, using two separate hash tables, based solely on the first character of each symbol. Action! uses the contents of location FRSTCHAR(location $4AD) to determine which character defines the splitting point.

After determining which character you wish to split your symbol table on (usually either 'a' if you keep upper and lower case distinct or 'M' if you don't), simply Read the file "BIGST.ACT" into the editor and change it to reflect your choice. Then compile and run the program. So long as you do not reBoot Action!, the big symbol table option will be in effect.

By the way, note that Action! uses [STG2] ([$CE]) as the hash table for the other 255 globals. You can set STG2, BIGST, and FRSTCHAR yourself, but letting "BIGST.ACT" do it for you is generally easier and safer.

Section 4: Compiling at a Particular Address

In Section 1, we noted that Action! places your compiled code directly in memory. Normally, it places the object code directly above the edit buffer, which in turn is above Action!'s "semi-fixed“ RAM and thus above DOS. In this section we discuss methods for telling Action! where you wish to place your code.

4.1 Directing the Code Storage Address

So long as you have no program in the edit buffer, you may think of the memory from the top of Action!'s semi-fixed RAM to the bottom of the symbol table space as your "free" RAM. You may ask Action! to place your object code anywhere in this space.

You may determine exactly what the bounds of this space are from the monitor. Simply use a '? $E' command to determine the bottom of this space. Remember that [$0E] ([APPMHI]) define the current "code pointer" for Action! If you haven't compiled anything yet, then APPMHI points to where code WILL be stored.

The top of this space may be determined via a '? $B0' command. Actually, location $B0 (STBASE) contains a single BYTE value (so be sure and look at the least significant byte of the contents of $B0 after using '?'). This byte value is the page number of the start of the symbol table (less 1, actually).

Now, if you compile your program and then again look at the contents of APPMHI (or at CODESIZE), you know how big your compiled program is. If it does not occupy all of the "free" memory, you may, if you wish, move it upward within the free memory.

Basically, Action! needs both APPMHI (otherwise labeled CODE) and CODEBASE (location $491) SET to the initial code address. You do this by simply including two SETs at the very beginning of your program. For example, if I would like my object code located at location $5000, I would put these two lines as the first two lines of my program:

```
SET $E=$5000
SET $491=$5000
```

ONCE MORE: The important thing to remember, here, is that your compiled object code MUST fit between [$0E] and the bottom of the symbol table.

4.2 Compiling With an Offset

Since the Action! cartridge, DOS, and Action!'s buffers and tables occupy fixed or semi-fixed RAM locations, you often cannot place your Action! code in the actual memory locations that you want to use. For example, if you wanted to write a program which replaced all or part of DOS, you could not do it by simply SETting location APPMHI.

But have no fear. Action! has provided for you. Action! allows you to compile code into one set of memory locations even though it is designed to run at a different set of locations!

The mechanism Action! uses is simple: there is a location called CODEOFF ($B5) which contains a 16-bit address offset. By default, CODEOFF contains a zero, so code is generated which is designed to run at the same addresses at which it is stored. When you change CODEOFF, though, strange and wonderful things can happen.

Everytime Action! generates an address for a PROC, FUNC, variable, etc., it uses the actual location defined by [APPMHI]. However, every time Action! compiles a REFERENCE to such an address, it adds CODEOFF to the address. For example, suppose that as Action! compiles it sees the following source code fragment:

```
SET $B5=$1000 ; set CODEOFF to 4K Bytes
; assume that APPMHI contains $4000 at this point
PROC P()
        …
PROC Q()
        P()
        …
```

The compiler "knows" that the PROCedure named "P" is located at address $4000. Yet, when it compiles PROCedure "Q" and encounters the reference to "P", it generates the equivalent of

```
        JSR P+[CODEOFF]
          or
        JSR P+$1000
          or
        JSR $5000
```

Since Action! ignores any overflow/carry which results in adding CODEOFF to an address, we could 'SET $B5=$F000' and effectively subtract $1000 from each address instead (remember, Action! does not allow negative compiler constants except via this mechanism).

As an example, then, let us suppose that we do indeed wish to replace DOS. Thus we want a program which will run at location $700. Let us further suppose that we are using Action! with a DOS which causes a LOMEM of $2100. Thus the initial contents of APPMHI will be approximately $2800 (plus a little). We might, then, start our Action! program with the following lines:

```
SET $E=$2F00 ; just to make the setting ...
SET $491=$2F00 ; ...of CODEOFF easier
SET $B5=$D800 ; equivalent of $B5= -$2800
```

And, lo and behold, if we dumped the compiled code we would find that we had indeed generated code designed to run at location $700.

Now, if we use the Write command from the Action! monitor, Action! automatically adjusts the starting and ending addresses for our object code file so that it will be LOAded in (via LOAD in DOS XL, the L option of Atari DOS, etc.) at the offset address! In other words, Action! has done all the hard work for us.

Special Note: Sometimes, though, you do not want the code you have generated loaded into its intended running address. In our example, we certainly wouldn't want DOS to try and Load our program at $700: we would wipe out part of DOS and surely do nasty things to our system. Presumably, we would want our code to Load in where it was generated. Then we would have a small routine which would move the code to its intended address and run it.

You may accomplish this purpose by simply noting the values of CODEBASE and APPMHI at the end of your compile. Then go to DOS (via the 'D' command of the monitor) and SAVe that part of memory. It will now LOAd where it was compiled, so you will have to somehow have a routine which will move it and run it (may we suggest simply appending such a routine--written in assembly language and placed, say, in page 6--to your main program).

FINAL NOTE: This offset technique may also be useful if you have an Action! program which almost, but not quite, fits in its allotted "free" RAM. Since arrays (other than small BYTE arrays) are allocated semi-dynamically after the end of your program (and may thus occupy the symbol table's space, for example), they do not affect the size of your compiled code. Thus you may "recover" the $700 bytes "lost" to Action!'s semi-fixed RAM by coding an offset of $F900. The space thus gained is not huge (1700 bytes or so), but it may make all the difference to you.

On this same note, remember that using DOS XL can save you up to 5K bytes of RAM during a compile. Then, if you remove the Action! cartridge to run your program, DOS will have to move LOMEM higher (since it will now all reside at $700 up), but HIMEM will have moved up by 8K bytes. Some work with offsets, etc., here could be very beneficial when you are working with very large programs.

4.3 Using Large Assembly Language Modules

Since you can direct Action!'s code generation, you can obviously "tell" it to reserve any given area of memory. This implies that you may assemble code for some specific address range, make a list of the subroutine entry points and/or variables to be accessed from Action!, and compile an Action! program which avoids the assembly language area. If the Action! program equates PROCedures, FUNCtions, and variable names to locations within this area, the assembly language routines, etc., may be used interchangeably with Action! routines.

Here is a small example of what we are discussing:

```
Assembly language:
         *=$3000
 LSH3                    ; FUNCTION: left shift argument by 3
         ASL A
         ASL A
         ASL A           ; left shift 3 times
         STA $A0         ; put where Action! puts function
         LDA #0          ; ...return values
         STA $A1
         RTS
 MASK    .BYTE  1,2,4,8,16,32,64,128 ; set of bit masks

Action!:
 BYTE FUNC LSH3=$3000 (BYTE N)
 BYTE ARRAY MASK(0) = $300A
```

For this particular example, you would probably be better off putting the small routine and array directly in your Action! program, via code blocks. But for larger, more complex operations, etc., this technique is very workable.

Section 5: Compiling ROMmable Code

If you have just finished reading Section 4, you should have a pretty good idea of how to ask the Action! compiler to produce code which will run in the normal cartridge space (i.e., $A000 to $BFFF, where Action! itself resides). Presumably, you know how to compile your code somewhere safe in RAM with CODEOFF set such that the code will run in ROM space (e.g., compiling to $6000 in RAM with CODEOFF set to $4000).

However, there is still a rather sticky problem: what do we do about variables? Normally, Action! compiles in such a way that global variables, PROCedures, FUNCtions, and local variables all share the same address space (i.e., they are all mixed up together, according to Action!'s own schemes). What we need is some way to tell Action! to keep programs and variables separate.

5.1 RAM and ROM Variables

Actually, there is one very simple way: simply assign addresses to ALL your variables. When you make a declaration such as

```
BYTE Temp = $D4
```

Action! assumes you know what you are doing. All references to "Temp" actually become references to location $D4.

There is a second class of variables which need no special care: those which aren't really "variable". If you initialize the contents of a variable or array (or string) and then never change its contents, then you actually want that variable in ROM. A declaration of the form

```
BYTE ARRAY Bits(0)=[1 2 4 8 16 32 64 128]
```

will generate and initialize an 8-element byte array. Presuming that you never store into Bits(n), the array actually should be in ROM.

But the vast majority of variables in most programs fall into neither of the above two categories. They are variables which we intend to change and which we want the compiler to assign space for.

Truthfully, Action! was not designed to produce code with variables and program separated. But the workings of the SET compiler instruction let us access a sophisticated method which we feature here.

Before reading further, it might be a good idea to read or print the listing of the file "KALROM.ACT", supplied on your RunTime disk. This is a somewhat smaller version of the famous Action! Kaleidoscope demo, but this version is designed to be compiled into ROM!

We call your attention to the two DEFINEs at the head of the program:

```
        DEFINE RAM =   "SET $682 = $E^
                        SET $B5  = $C800
                        SET $E   = $680^"
        DEFINE ROM =   "SET $680 = $E^
                        SET $B5  = $5800
                        SET $E   = $682^"
```

Note also the various SETs a little further in the program:

```
        SET $E=$6000          SET $491=$6000
        SET $B5=$5800         SET $680=$5800
```

And then let us note, before explaining how all this ties together, that this program will compile at address $6000, where APPMHI and CODEBASE ($E and $491), are initially set. The code will be compiled to run at address $A800, the sum of APPMHI and CODEOFF ($E and $B5).

The RAM used by this program will be compiled at $5800 (the initial value of location $680, see below) and be placed, when the ROMmed code is run, at location $2000 ($5800 + $C800, the alternative value for CODEOFF, ignoring the overflow from the addition).

HOW IT WORKS: The initialSETs (not the ones in the DEFINEs) are given values which will start Action! producing code designed to reside at $A800, as we noted. When the compiler reaches the label "RAM", though, it executes the SETs defined thereby. Specifically, it saves the current value of the code pointer (APPMHI) in a "spare" location ($682) via "SET $682=$EA". Did you remember that you can use constant pointers in a SET? "$E^" simply means "the contents of location $E".

The expansion of the RAM definition also causes CODEOFF ($B5) to be changed and APPMHI ($E, also called CODE) to be loaded from the contents of location $680, another "spare" chunk of memory. (Did you remember that $680 was initialized to $5800, just for this purpose?)

When Action! encounters and expands a "ROM" definition, the effective opposites happen: APPMHI is saved in $680, CODEOFF is changed to the value needed for ROM generation, and APPMHI is reloaded from $682, where it had been saved by the "RAM" definition.

Whew! It all seems complicated, but once you have set up the DEFINEs for "ROM" and "RAM" the rest is easy.

The only other thing to watch out for is just WHEN do you use these ROM and RAM definitions? Generally, you simply code "RAM" just before you define some variables you want to reside in RAM. In the case of local symbols, then, you code "RAM" just before defining them and "ROM" just after doing so.

There is just one place which is a little tricky: after compiling some ROM-based definitions of global variables, you need to code "RAM" to cause the parameters and local variables of the next PROCedure or FUNCtion to be compiled in RAM. However, due to the method by which Action! generates code and address references, you must code "RAM" after the keyword PROC or FUNC.

Again, we refer you to the listing of "KALROM.ACT" for further examples and techniques. You will note the BYTE ARRAYs in the beginning being generated in ROM. These are invariant masks, as we discussed above. Also, note that it does not matter whether "ROM" or "RAM" was last coded when you define variables which are assigned to specific addresses.

PROCedures and FUNCtions which receive no parameters and have no local variables may be considered completely ROM-resident. Code block PROCs and FUNCs which use only the parameters passed in the registers (remember, the first three bytes of parameters are passed in A, X, and Y) may include the notation "=*", as shown in several PROCs in the example, and will generate no actual variable storage.

5.2 Other Considerations

Once you have tackled the general problem of separating RAM and ROM space, there are a few other things to watch out for when producing ROMmable Action! code.

5.2.1 FOR loops

In general, you cannot use FOR loops in Action! code which is to be placed in ROM. When Action! encounters a statement of the form

```
FOR LoopVar=Begin TO Finish STEP Increment
```

it realizes that it needs space to store the "Finish" and "Increment" values. If these values are not constants, they are evaluated at run time and stored in-line among the compiled code!

This is not a major matter: you can easily modify the above FOR loop to be a WHILE loop instead. For example:

```
LoopVar=Begin
WHILE LoopVar <= Finish
DO
        …
LoopVar ==+ Increment
OD
```

A little lengthier than the equivalent FOR loop, but actually no less efficient in most cases.

5.2.2 PROCedure variables

Action! allows PROC names to be used in expressions, including assignments to a PROC name. For example, you are allowed and encouraged to handle your own errors via the following (paraphrased from the Action! reference manual):

```
PROC HandleError()
        …
RETURN

…

SaveError = Error
Error = HandleError

…
```

Action! handles PROC names in this fashion thanks to a usually invisible mechanism: Each PROC or FUNC is compiled to start with a JMP instruction. Normally, the target of the JMP is the byte immediately following the JMP, the actual code for the PROC or FUNC.

When you assign a value to a PROC (as in 'Error = HandleError', above), the code generated actually modifies the last two bytes of the JMP instruction, the target address.

Unfortunately, when a PROC or FUNC is in ROM, you obviously can NOT modify the target of the JMP instruction. If you desparately need this capability, may we suggest the following scheme:

```
PROC HandleError = $600 ( )
        ; or any other "safe" address
BYTE Hjmp = $600 ; same address
…
PROC RealHandler( )
        …
RETURN

…
MAIN()
        Hjmp = $4C ; a JMP instruction
        HandleError = RealHandler
        …
        ; and now you can assign to 'HandleError'
        ; as and when you wish
```

The important part of this "trick" is that you MUST set the JMP instruction in place "by hand", as we did in the first line following MAIN().

5.2.3 Action!'s System DEVICE

Many of the I/O routines in the Action! library (both the cartridge library and the RunTime version) perform their operations to a channel (file) defined by the contents of a location called DEVICE. For example, PRINT() and INPUTS() both use DEVICE.

Normally, Action! initializes the contents of DEVICE to zero. You can thus easily change the default output channel by simply OPENing a file on another channel and placing a new channel number in DEVICE.

When using the RunTime package, you must take responsibility for initializing DEVICE. You may do this by coding

```
DEVICE = 0
```

in your code. Or, if you intend to never change the contents of DEVICE, you might code a declaration of

```
BYTE DEVICE = [0]
```

as an early global variable. See the file "SYS.DOC" on the RunTime disk for other comments.

5.2.4 File Names

The library in the Action! cartridge automatically adds a "D:" filename prefix to a filename if the filename does not begin with a device name (e.g., “D2:", "P:",etc.). The RunTime library does NOT do so.

Be sure that your Action! programs include sufficient filename validation.

Section 6: Action! Memory Map

The important locations used by the Action! compiler and RunTime are given here in memory location order. The address, label used by internal routines, and a short description of each location are all given. If changing a location might be useful to you, the description will say so and possibly point you to another Section for more information.

The labels given here are shown in mixed upper and lowercase, as used by Action!'s author. In other Sections of this document, these labels are shown in all upper case, simply to make them easy to distinguish from surrounding text.

In the listing which follows, a period between the address and the label indicates a system location which is two bytes long, in normal 6502 low/high format. It may, of course, thereby be an address pointer.

Addr	Label	Description
000E.	code or APPMHI	The "location counter" used by Action! (Also Atari OS's "application program high memory".) Points to where next byte of code will be stored during an Action! compile. Generally should only be changed before a program is compiled (Section 4.1), but can be changed with caution to produce ROMmable object code (Section 5.1).
009B.	buf	Address of Action!'s edit buffer. More importantly, though, this buffer is also used by the library OPEN procedure to validate the filename it is passed. It must be initialized to a valid address when a compiled program is run with the cartridge. CAUTION!! OPEN in the RunTime library does NOT validate filenames. See Section 5.2.4 and comments about the use of locations $500-$5FF, below.
00A0 to 00AF	args	This portion of zero page is used to store function and procedure parameters and as temporaries for evaluation of many expressions. Parameters start at $A0 and work up. Temporaries start at $AF and work down.
00B0	stBase	High byte of address of start of symbol table.

00B1.stGlobal Location of 512-byte hash table for global symbols.

00B3.stLocal Location of 512-byte hash table for local symbols.

00B5.codeOff Offset between compiled-at address (as determined by "code", location $0E) and compiled-for address (e.g., when compiling code to be placed in ROM or in place of the DOS code). See Section 4.2.

00B7 device Current default device number. If changed to a valid OPENed file number, all library output normally sent to the screen will go to that file instead. Described in the Action! reference manual, but see also the source code comments on the RunTime disk and warning in Section 5.2.3 re ROMmable code.

00CE.stG2 Used only if the "big symbol table flag" (bigST, $4C4) is set. This is the address of the 512-byte hash table used for the second half of the global names. See the file BIGST.ACT on the RunTime disk and Section 3.2.

0491.codeBase The first address used to store code in the current compile. It is preserved for later use with the "Write" monitor command. See Sections 1.1 and 4.1.

0493.codeSize The number of bytes of code generated by the current compile. See Section 1.1.

0495 stSp Symbol Table SPace. Simply the number of 256-byte pages available for the symbol tables (both locals and globals). May be easily altered as needed by the user. See Section 3.1.

049A list Action!'s "List the program as it is compiled" flag. Changed by the "LIST?" query in the Options. Can also be changed at any point in a compile via a SET.

04AD frstChar When a big symbol table is in use (see $04C4), this character determines the division point between the lower and upper halves of the global symbol table. See also Section 3.2 and the file "BIGST.ACT".

04C0 bckgrnd Background color. Use at your own risk.

04C4 bigST A flag. If set, the global symbol table is divided into two parts (see also $04AD). Thus you may use a total of 510 global symbols. See also Section 3.2 and the file "BIGST.ACT" on the RunTime disk.

04CB Error A JMP to the current error handling routine. As far as the compiler is concerned, this is the address of the Error procedure. See the Action! reference manual for a method of substituting your own error handler. See section 5.2.2 for comments re ROMmable code.

0500-05FF A buffer used by the RunTime library OPEN routine. When OPEN is passed a filename, it moves the name here and appends a RETURN ($9B) character. The cartridge routines do this differently, using a buffer pointed to by BUF (location $9B, see above) instead. You can easily change the location of this buffer by changing the DEFINEs at the beginning of the RunTime library.