Action! ™

Cartridge System, Run Time, Tool Kit & Bug Sheet

Powerful !

Flexible !

Fast !

OSS ™

Precision Software Tools

A Division of ICD, Incorporated

A Trademark of Fte

Revised by GoodByteXL

Last Edit: 2Ø July 2Ø22

Besides correcting some typos a new customized font was
introduced because the readability on screen as well as the
print quality of the type writer font courier is not that good.

Additionally, a slashed zero was introduced by customizing the
used font. Hope you like it. ☺

Manual created using:

Linux Mint 2Ø.3
OpenOffice 4.1.12
GIMP 2.1Ø.18

a reference manual for

The ACTION! System

comprising

Cartridge Version 3.6
Run Time Package Version 1.4
Toolkit Version 3

A complete programming environment designed
for your ATARI home computer system.

This comprehensive information
provided by Atari enthusiasts
aims at the preservation of

The ACTION! Programming Environment

Revised and enlarged edition V. 3.6 (p) 2Ø22

Preface

As tribute to OSS and ACS this fully edited version is provided
to keep the best and fastest high level programming language on
ATARI 8-bit computers accessible to all being interested in
this great piece of software.

Action! was released by OSS & ACS in 1983.
OSS merged into a division of ICD, Inc. in 1988.
ICD sold their ATARI 8-bit product line to FTe in 1993.
FTe disappeared supposedly in 1996.
Declared as freeware by Clinton Parker in January 2015.

Many of those famous A8 hardware and software products from OSS
and ICD are still available thanks to a worldwide active A8
community. So is this manual.

As my ICD Action! manual fell apart it had to go through the
scanner and some OCR software. And it is completed by the parts
run time package, tool kit and bug sheet #3. All original OSS
material about Action! in one book. Finally!

The new Action! manual in parts I to VI keeps the original
layout for reference to the original paperback and of course
for historical reasons. All entries from the original's table
of contents are found on the same page, except were errors had
to be corrected.

The run time manual in part VII is also kept in the original
layout, except page numbering.

The toolkit in part VIII is adapted to the layout as deemed
necessary.

The appendices needed amendment and changes, so their page
numbering changed. Typos and errors found were corrected and
bug sheet #3 is applied to all parts of the new Action! manual.

ACTION! and other OSS programming languages work best in
conjunction with SpartaDOS X V. 4.47 and up provided by DLT.

Open Office was engaged to create this manual.

Enjoy ACTION! … and may your A8 always be with you!

GoodByteXL, February 2022

P.S.: Special thanks for help to Erhard and Roland. And of
course to everyone else who made me aware of typos and other
errors.

THE ACTION! SYSTEM

Part I: Introduction to ACTION!

Part II: The ACTION! Editor

Part III: The ACTION! Monitor

Part IV: The ACTION! Language

Part V: The ACTION! Compiler

Part VI: The ACTION! Library

Part VII: The Action! Run Time Package

Part VIII: The ACTION! Toolkit

Appendices

Part I: Introduction to ACTION!

Welcome to ACTION! We are here to introduce you to a
complete software development system - one in which you can
perform all of your programming tasks.

If you have programmed your ATARI using ATARI BASIC, you
will discover that ACTION! runs a lot faster, has a better
editor, and is just as easy to learn.

If you have ever done assembly language programming, you
will find that ACTION! is almost as fast as assembly
language, as far as program execution is concerned. You
will also find that programming ACTION! is much quicker and
easier due to the nature of the language, its editor, and
its library of routines.

For those of you with very little or no previous
programming experience, we suggest that you read this
manual very carefully, and be sure you understand one
concept before moving on to the next. We say this because
this is not a tutorial to teach you the ACTION! system, but
rather a reference manual of all the capabilities of the
system.

That is not to say that you will not understand what you
are reading (quite the contrary); it simply means that we
do not discuss every possible programming use of the
concepts involved. We respect your ingenuity and curiosity
and believe that you yourself will find some uses we have
not even dreamed of.

Notes On This Manual

The manual itself is separated into eight parts and a group
of appendices. Each of the parts exclusively discusses one
facet of the ACTION! system, thus enabling you to learn
about the different components of ACTION! without having to
keep flipping pages. Each part is prefaced by a table of
contents, an introduction, and a vocabulary.

The one drawback to separating the ACTION! system into its
component parts is that you will learn everything about one
part before starting the next part. To help alleviate this
problem we suggest that you read the introduction section
of each of the parts before reading one part in depth.
Also, the last chapter of this introduction shows you how
the ACTION! components work together to allow you to run
programs.

Chapter 1: The ACTION! System

The ACTION! system is made up of five different parts:

            The ACTION! Monitor
            The ACTION! Editor
            The ACTION! Language
            The ACTION! Compiler
            The ACTION! Library

The system is completed by two more parts:

            The ACTION! Run Time Package
            The ACTION! Tool Kit

These two parts were originally available separately and for better reference are now merged into this manual.

The Monitor is the boss of the ACTION! System. Through it you can call the Editor, the Compiler, or get access to some system options. This is the monitor's only job, but it is a important one, allowing you to decide which part of the ACTION! system you want to use at any given time.

The Editor is where you create new programs and modify old ones. It does not know anything about the ACTION! language or compiler (that is, it is simply a text editor and does not check language syntax), so you can use it for other word processing or program entry applications. The Editor also allows you to save the text in the editor buffer or read text from a peripheral device (disk drive, cassette, etc.) into the editor buffer.

The ACTION! Language is what you use to communicate with the ATARI machine and tell it to do things. You write a program in the ACTION! Language, and then tell the ACTION! Compiler to translate it into a form the computer can understand (machine language), and then you run the program.

"Why such an involved process? BASIC is not like that." First of all, the process is not that involved once you understand what is going on and why. Secondly, BASIC is not like that because it is an Interpreter, not a Compiler. BASIC translates each line as the program is running, and it takes some time to do that, thereby slowing down the speed of your program. ACTION!, on the other hand, breaks the running and syntax checking of your program into two parts. The Compiler checks your program for proper syntax, and does the translating. When it is through, your program can be run directly, i.e., without any syntax checking.

This makes your program run with incredible speed.

As mentioned in the previous paragraphs, the ACTION!
Compiler translates an ACTION! program into machine code.
The only thing it requires is that the program be in proper
ACTION! form. The compiler will give you an error if you
use syntax which is illegal in the ACTION! Language, just
like an English teacher would give you an error (or red
mark) if you used improper English in class.

The ACTION! system also contains a group of pre-written
routines which you can use in your programs. This group of
routines is called the ACTION! Library, and it enables you
to do all the things you can do in BASIC (i.e., PLOT,
DRAWTO, PRINT, etc.) and much more without writing any
special subroutines of your own.

TECHNICAL NOTE:
Although the ACTION! compiler does translate an ACTION!
source code to 6502 machine language, that compiled code
will not run without the ACTION! cartridge because it (the
code) does some calls to routines in the cartridge. If you
are writing products for resale, a runtime version of
ACTION! which will make your program work without the
cartridge can be licensed from OSS Inc.

NOTE by GBXL:
The latest known ACTION! bug sheet (#3) has been applied to
this manual, which refers to the latest versions of

The Cartridge (3.6).
The Run Time Package (1.4)
The Toolkit (3.0)

Appendix K shows the full text of bug sheet #3.

Chapter 2: How To Write and Run an ACTION! Program

This chapter is designed to let you "get your feet wet" and become more familiar with the ACTION! system. We are going to write a little program in the Editor, Compile that program, and Run it.

When you go to the cartridge from DOS you will be in the ACTION! Editor, so the program can be entered immediately. We are going to assume that you will not make any typos, but if you do, you can use the cursor control keys (<CTRL><up arrow>, etc.) to move around and fix them. When you read the Editor part you will find out about many more editor features and commands, but these are all you need for this program.

Now for the program. Enter it exactly as you see it here (no special commands are required to enter text):

```
PROC hello()

    PrintE("Hello World")

RETURN
```

Before we compile this program, let us discuss what is going on. The 'PROC' and 'RETURN' statements are required by the ACTION! language, and make up the bones of a procedure. The language is structured into a group of subroutines called procedures and functions, with each routine doing a specific task which you define. This might seem strange at first, but it allows you to write programs in components so that you can concentrate on one part of the program at a time. It also makes programs written by others much easier to read.

The above procedure is called "hello", namely because it will print out the line "Hello World" to the screen when the program is run.

The statement starting with 'PrintE' is a library routine call. Here we are making use of one of the prewritten routines in the ACTION! library. This one will print out the specified string, and put out a <RETURN> at the end. This routine call is the only statement in the procedure 'hello' (because it is the only statement between the 'PROC' and 'RETURN').

Now that we have the program in the Editor buffer, how are we going to compile and run it? The Editor certainly cannot do it for us, so we need to get to the Monitor and call the

ACTION! Compiler from there. The Editor command <CTRL><SHIFT>M takes us to the Monitor, so we will use that.

Now that were in the Monitor, we need to call the Compiler to check the syntax of our program and translate it into machine language. This is done by typing the command 'COMPILE <RETURN>' to the Monitor.

The compiler takes over and does its job. If it finds an error, it will print out an error message and return you to the monitor. If it finds no errors, the Compiler will return us to the Monitor. From there we can run the compiled program by entering the command 'RUN <RETURN>'. The screen should look like this after you have run the program:

```
    +---------------------------------------+
    |  >                                    |
    +---------------------------------------+
    |   Hello World                         |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    +---------------------------------------+
```

You have written your first ACTION! program!

If you got an error message from the compiler, it means that you did not type the program in properly. You can go back to the Editor from the Monitor by typing 'EDITOR', and can fix your typo. You might note that the cursor is placed at the position where the compiler found the error, so you do not have to look all over for it. Repeat the steps discussed above to re-compile and re-run your program after you have fixed the error.

NOTE:
There is a list of the error codes with their meanings in Appendix C.

NOTE from bug sheet #3:
Running compiled ACTION! programs as '.COM' files under OS/A+ causes those programs to execute twice. It affects all versions of ACTION! under OS/A+. To fix it insert the following as the first global variable you declare:

The ACTION! Programming Environment

```
BYTE RTS=[$6Ø]
; This MUST be the first line in your program,
; aside from comments and SET commands.
```

NOTE by GBXL:
SpartaDOS X as of version 4.47 solves this by providing
special loading modes for these kind of files and others.

Part II: The ACTION! Editor

                    Part II: The ACTION! Editor

                       Chapter 1: Introduction

The Editor is where you create new ACTION! programs and
edit old ones. If you have used a program editor before,
you will notice that the ACTION! Editor is far more
sophisticated than most others: in fact, it could almost be
called a word processor because it does so much.

Although it is capable of many things, you will find that
the ACTION! Editor is easy to work with. If you have never
been exposed to anything but the ATARI screen editor, then
you are in for a pleasant surprise. You can use the ACTION!
Editor for any editing you want to do, not just editing
ACTION! programs. You could do all your editing
(correspondence, programs in other languages, etc.).

1.1 Special Notations and Vocabulary

USAGE OF SINGLE QUOTE MARKS (')

Unless format and context make the use of quotes
unnecessary, commands and special characters will be
enclosed in single quotes.

USAGE OF '<' AND '>'

When talking about a key on the ATARI keyboard, we will
enclose it with the characters '<' and '>', thus:

   <BACK S> (the backspace key)

Some of the keys have more than one label written on them.
In these cases, the label best describing the Editor
command will he used.

There is one exception to the above; the character (A - Z)
and digit (Ø - 9) keys are not enclosed by the angle
brackets.

MULTI-KEYSTROKE COMMANDS

Some of the Editor commands require that you press more
than one key at a time. For these commands, the keys
required are given back to back in the order in which you
should press them. For example,

    <SHIFT><DELETE>

means that you should hold the key marked 'SHIFT' down, and
then press the key marked 'DELETE'.

THE MESSAGE AREA

Throughout this part of the manual the term "message area"
will be used. This area is simply the inverse video line
you will notice at the bottom of your screen when you enter
the Editor. This line normally has

    ACTION! (c)1983 ACS

written on it, but is used by some of the editor commands
to ask you questions, give you information, or report an
error.

When you are using two windows (see section 3.4), the
message area line separates the two windows.

DEFAULT USER RESPONSE

Some of the commands which use the message area to exchange
information with you remember the information you gave the
last time you used the command. This information is called
the "default user response". If the default user response
you see in the message area is what you want, you simply
press <RETURN>. This saves you time because you do not have
to retype the same response many times. If you do not want
to use the default user response, you have the ability to
change either part or all of it.

The ACTION! Programming Environment

1.2 Editor Concepts and Features

TEXT WINDOWS

When you look at your TV or monitor screen, imagine that
you are looking through a window. At any one time you can
only see 23 lines of 38 characters each. This seems very
limiting, and would be if you could not move the window
around. The ACTION! Editor makes it possible for you to
move this window around, both horizontally and vertically,
so that you can look at your whole program.

But the Editor is even more sophisticated than that! If you
are looking through the window, and you want to look at one
line which extends beyond the bounds of the window, you can
move to that line and look at the whole thing. The line
will move to fit into the window, but the window stays
right where it was. When you move off that line, it pops
back into its proper place (with respect to the window).

The Editor evens allows you to split your screen into two
windows, each of which you can control separately. This
enables you to look at two different programs, or different
parts of the same program, at the same time.

TEXT LINES

The ACTION! Editor is designed so that your program can be
read easily. It allows lines up to 24Ø characters long
(even though the window only shows 38 characters at a
time), so you can use indentation to clarify your program
without worrying about making lines too long. The Editor
also allows blank lines, so you can separate the components
of your program with white space.

NOTE: you also have control over the maximum line length,
so you pick a line length you think best (or will fit on
your printer). The Editor even buzzes when you reach the
limit, to let you know it is time to go to the next line.

NOTE: if a text line is longer than the window (if it
extends beyond the left or right bounds of the window), the
character at the edge of the window is shown in inverse
video to make you aware of this.

FIND and SUBSTITUTE

The Editor allows you to search for a given string, and will move the cursor to the first match found in the program.

The Editor also allows an extension of this. You can tell it to search for a given string, and then replace the first match with another string you specify, all with one command.

MOVING TEXT BLOCKS

Have you ever entered a program and wished that a group of lines which you entered at one location could be conveniently moved to another location? In ACTION, this is a snap!

Saving those lines (called a text block) in the copy buffer (that is where a text block is temporarily saved) allows you to move the cursor to where you want the text block placed. You may then "paste" the contents of the copy buffer (the lines you wanted to move) back into the program. You can paste the text block at its original location, and then move somewhere else and paste it there too, thus enabling you to copy text blocks.

CURSOR MOVEMENT

The cursor is controlled not only by the movement keys on the keyboard (e. g., <CTRL><up arrow>), but can also be made to move to specified places in your text through the use of tags and the 'FIND' command.

TAGS

You can mark any location in your text with an invisible marker called a "tag". The Editor allows you to move the cursor to this tag (no matter where the cursor was before) through the use of a simple command.

The number of tags you allowed is limited only to the number of keys on the keyboard, since you must give a one character label to each tag you define.

                    Chapter 2: The Editor Commands

This chapter is devoted to the Editor commands themselves.
Instead of presenting the commands in this form:

        2.2 <CTRL><SHIFT>M
        ----

where you cannot tell what the section discusses (unless
you already know the Editor), the commands are presented by
their function, e. g.:

        2.2 Leaving the Editor

We hope this form makes things clearer and easier to
follow.

Before going into the commands themselves, we should tell
you how to stop execution of a command if you made a
mistake. You can do this by pressing the <ESC> key. Doing
this will get you out of any command safely.

NOTE: Appendix E provides a summary of the Editor commands,
listed by the command itself instead of what it does.

2.1 Getting to the Editor

When you first enter the ACTION! system, you will
automatically be put into the Editor, so there is nothing
involved in "getting to" it. You are already there.

If you leave the ACTION! system and go to DOS (OS/A+, DOS
XL, or ATARI DOS), you will be in the Monitor when you re-
enter ACTION! (there is one exception to this; see NOTE
below). To get to the Editor from the Monitor, you need
only type

        E<RETURN>

This will put you directly into the editor.

NOTE: if you are using OS/A+ or DOS XL, and you execute a
DOS extrinsic command before returning to the ACTION!
system, you will not be put into Monitor as stated above,
but straight into the Editor. This is not the case with
ATARI DOS, since it has no extrinsic commands.

2.2 Leaving the Editor

There is only one way to leave the Editor (aside from turning off the computer):

        <CTRL><SHIFT>M

This command will cause you to go from the Editor to the Monitor, where you may call the other components of the ACTION! system or leave the system altogether and go to DOS.

2.3 Text Entry

There is no special Editor command to allow you to enter text. You simply type it in, as on a typewriter. When you have reached the maximum line length, the Editor will buzz every time you put in another character (see 2.3.2 for more information).

If you want to type a control character, you must press the <ESC> before doing so. This lets the Editor know that the control character should be interpreted as text, and not as an Editor command.

"What happens when I try to type over something I've already written?" The ACTION! Editor allows you two options in this case. Text can be entered in either "Replace" or "Insert" mode.

When in Replace mode, the text you enter will overwrite whatever was there before, replacing the old with the new character by character.

When in Insert mode, the text you enter will be inserted wherever the cursor is, and move all the previous text over without overwriting it.

The Editor command <CRTL><SHIFT>I allows you to change from one mode to another. When you use this command, the mode you have changed to will be printed in the message area (see section 2.5.2 for more information).

NOTE: The Editor is in Replace mode when you first enter it.

If you want to erase all the text in a file, just put the cursor into the window you want to clear, and press <SHIFT><CLEAR>. This will clear not only what you see in the window, but the entire file (see section 2.6.4 for more information).

The ACTION! Programming Environment

2.3.1 Text File I/O

If there were no way to save the program in the Editor
buffer, you would have to retype it every time you wanted
to use it. The Editor allows you both to read and to write
files to any peripheral storage device (Disk Drive,
Cassette, etc.) to save you all this trouble.

To save a program in the Editor buffer, you must first put
the cursor into the window which contains the file you want
saved (if you are using only one window, you need not worry
about this). Then you enter the command

        <CTRL><SHIFT>W

In the message area you will see:

        Write

Simply type in the file name you want the program saved to,
and press <RETURN>. The file name must be compatible with
the DOS you are using. If you are not using a DOS, the file
will consist only of a character representing the device (C
for cassette, P for printer, etc.) followed by a colon.

Reading a file into the Editor buffer is just as easy. Move
the cursor to the line preceding the line where you want
the file you are reading in to start, and enter the command

        <CTRL><SHIFT>R

In the message area you will see:

        Read?

Type in the name of the file you want read in, following
the conventions outlined above.

If you are using floppy disks, you can read the directory
on a given disk by replying with the following to the
"Read?" prompt in the message area:

        Read? ?1:*.*

This will read the directory of the disk in drive number 1.
If you want to read the directory of a disk in some other
drive, simply change the '1' in the above example to the
number of the drive. This ability is very useful, because
you need not go to DOS to find out what is on a disk.

2.3.2 Setting the Line Length

As mentioned in the first paragraph of section 2.3, you can set the maximum line length. You can find out how to do this in part III, section 2.5, so we need not show you here.

2.4 Cursor Movement

To move the cursor left one character, press:

        <CTRL><left arrow>

To move the cursor right one character, press:

        <CTRL><right arrow>

To move the cursor up one line, press:

        <CTRL><up arrow>

To move the cursor down one line, press:

        <CTRL><down arrow>

The commands above are simply the normal cursor movement keys the ATARI screen editor understands. The ACTION! Editor, however, allows you some more cursor movement commands designed to increase your program writing speed.

You can make the cursor go to the beginning of the line its on by pressing:

        <CTRL><SHIFT><

and go to the end of the line by pressing:

        <CTRL><SHIFT>>

These two commands will take you to the true beginning or end of the line even if it (the beginning or end) is not visible in the window. The line will simply be shifted over so that it (again, the beginning or end) is visible in the window. When you move the cursor off the shifted line, the line will be moved back to its proper position.

You can go to the beginning of the file by pressing

    <CTRL><SHIFT>H (home), or go with <CTRL><SHIFT>E (end)

to the very end of the file.
2.4.1 Tabs

The ACTION! Programming Environment

You can move the cursor to the next tab stop by pressing
<TAB>.

To set a tab stop, move the cursor where you want the tab,
and then press <SHIFT><SET TAB>.

To clear a tab stop, move to the tab stop you want cleared,
and press <CTRL><CLR TAB>.

2.4.2 Finding Text

The Editor allows you to "find" a specified string of
characters (1 - 32), and can he very useful when skipping
from place to place in your file. To do this enter the
command:

        <CTRL><SHIFT>F

The message area will prompt you with

        Find?

If you have previously used the Find command, you will see
the string you last tried to find following the prompt. If
you want to find the next occurrence of this string, simply
press <RETURN>. If you want to find a different string,
type in the new string and press <RETURN>. You will notice
that the old string disappears as soon as you start typing.

If this is the first time you are using the Find, you will
see nothing following the prompt, and you should type in
the string you want found and press <RETURN>.

This command will start at the current cursor position and
look for the first occurrence of the string you specified.
If the string is found, the Editor will move the cursor to
the first character in the found string and make the window
move to display the surrounding section of text. If the
string is not found, the message area will display the
line:

        not found

2.5 Correcting Text

The following six sections will give you information on how
to correct and delete text from the Editor buffer. The
seventh sections shows you how to undo certain deletions if
you have made a mistake.

2.5.1 Deleting a Character

To delete the character under the cursor (the one the
cursor is flashing on top of), press:

        <CTRL><DELETE>

The characters to the right of the character just deleted
will move left to fill the empty space left by the deleted
character.

To delete the character to the left of the cursor, press:

        <BACK S>

If you are in Replace mode, this will replace the character
to the left of the cursor with a space. If you are in
Insert mode, this will delete the character to the left of
the cursor, and then move all the following characters over
to fill the empty space.

2.5.2 Inserting/Changing a Character

As mentioned in section 2.3, there are two different modes
for text entry: Replace mode and Insert mode. When you
first enter the ACTION! Editor, it is in Replace mode. To
change from one mode to the other, press:

        <CTRL><SHIFT>I

Some of the Editor commands are mode dependent; that is,
they operate differently, depending on the text entry mode.

You can insert a blank character at the cursor position by
entering <CTRL><INSERT>. The text from the cursor to the
right end of the line moves right one space and a blank
space is inserted at the cursor position.

NOTE: If you are in Insert mode, you can simply press the
space bar.

2.5.3 Line Deletions

To delete a whole line, place the cursor on the line you
want deleted, and press:

          <SHIFT><DELETE>

The succeeding lines move up to fill the empty space.

2.5.4 Line Insertions

To insert a blank line above the line the cursor is on,
press:

          <SHIFT><INSERT>

The succeeding lines move down to allow space for the new
blank line.

2.5.5 Breaking & Recombining Lines

To break a single line into two adjacent lines, first
position the cursor on the character you want as the first
character in the second line, and then press:

          <CTRL><SHIFT><RETURN>

NOTE: If you are in Insert mode, simply position the cursor
and press <RETURN>.

Succeeding lines of text are moved down to allow room for
the new line.

To combine two adjacent lines into a single line, first
position the cursor on the first character in the second
line, and then press:

          <CTRL><SHIFT><BACK S>

Succeeding lines are moved up to fill the empty space.

2.5.6 Substituting Text

The ACTION! Editor allows you to substitute a "new" string
for an "old" one. You are prompted for the "new" string,
and then for the "old" one. The Editor searches for the
first occurrence of the "old" string (starting at the
cursor position), and replaces it with the "new" string.

To begin this command, press:

      <CTRL><SHIFT>S

The message area will display the prompt:

      Substitute?

If you have previously used this command, you will see the last "new" string you used following the prompt. If you want to keep this "new" string, simply press <RETURN>. If you want a different "new" string, type it in and press <RETURN>.

If this is the first time you are using Substitute, you will see nothing following the prompt, and you should type in the "new" string you want and press <RETURN>.

After you press <RETURN>, the message area will prompt you with:

      for?

You will see the last "old" string you used following this prompt if you have used the Substitute before. If you want to keep this "old" string, simply press <RETURN>. If you want a different "old" string, type it in and then press <RETURN>.

If this is the first time you are using Substitute, you will see nothing following the prompt, and you should type in the "old" string you want changed and press <RETURN>.

After you press this second <RETURN>, the Editor will try and do the substitution. If it cannot find the "old" string you have given, the message area will show the following:

      not found

If you press <CTRL><SHIFT>S again before you do any other editing, the Editor will execute the same substitution again. This enables you to substitute more than one occurrence of the "old" string with the "new" one without having to keep responding to the "Substitute?" and "for?" prompts.

HINT: you can delete the next occurrence of a string by using this command with the "new" string being nothing. This will substitute the "old" string with nothing, and so (in effect) delete it.

2.5.7 Restoring a Changed Line

The ACTION! Editor allows you to restore a line to its
previous form if you have made an error while editing it.
To do this, you must remain on the changed line and press:

        <CTRL><SHIFT>U

WARNING: if you leave the line and then come back to it,
this command will not work, because the Editor only
remembers what the line was before you started editing it
while you remain on the line.

If you have accidentally deleted a whole line, you can
retrieve it by pressing:

        <CTRL><SHIFT>P

More information about this command may he found in section
2.7.

NOTE: The tags on the changed or deleted line are not
restored.

2.6 Windows

The displayed contents of the central portion of the screen
is called a window. The following five sections describe
the Editor commands used to manipulate, create, and delete
windows. In these sections we use the term "current window"
to mean the window which the cursor is in.

2.6.1 Window Movement

You can make the window scroll up or down one line simply
by moving the cursor. If you try and move the cursor off
the top of the screen, the window moves up one line to keep
the cursor on the screen. The same works with the bottom of
the screen. This type of vertical scrolling could take a
long time if your program were big, so the Editor also
allows you to make the window scroll by the size of the
window itself.

To move up one window, press:

    <CTRL><SHIFT><up arrow>

For the sake of continuity, what was the top line in the
old window is now the bottom line of the new window.
To move down one window, press:

```
    <CTRL><SHIFT><down arrow>
```

For the sake of continuity, what was the bottom line in the old window is now the top line of the new window.

The Editor also allows you to scroll the window horizontally. That is, you can make the window's left margin start at any column (instead of the first column). If a line is longer than the window (if it extends beyond the left or right bounds of the window), the character at the edge of the window is shown in inverse video to make you aware of this.

To move the window one character to the right, press:

```
    <CTRL><SHIFT>]
```

To move the window one character to the left, press:

```
    <CTRL><SHIFT>[
```


2.6.2 Creating a Second Window

When you first enter the ACTION! Editor there is only one window. You can create a second window by pressing:

```
    <CTRL><SHIFT>2
```

The screen will now look like this:

```
    +---------------------------------------+
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    |                                       |
    +---------------------------------------+
    |   ACTION! (c)1983 ACS                 |
    +---------------------------------------+
    |                                       |
    |                                       |
    +---------------------------------------+
```

The window above the message area is window 1, and the window below it is window 2. You can use each window independently, so you could be working on two entirely

The ACTION! Programming Environment

different files without having to keep clearing window 1
and reading in a file.

NOTE: The size of the window 1 can be set using the Options
Menu available from the monitor. For more information on
how to do this, see part III, section 2.5.

2.6.3 Moving Between Windows

To move from window 1 to window 2, press:

        <CTRL><SHIFT>2

If window 2 does not yet exist, then the Editor creates
window 2, then moves the cursor into it.

To move from window 2 to window 1, press:

        <CTRL><SHIFT>1

2.6.4 Clearing a Window

To clear the text file in a window, move the cursor into
that window (see previous section), and press:

        <SHIFT><CLEAR>

Since this is such a powerful command, the message area
will prompt you with:

        Clear?

Respond with a "Y" or "N". If you have made changes to the
file viewed through that window, and have not saved the
changed version, the message area will prompt you with:

        Not saved, Delete?

to make sure that you know that you have not saved the new
version.

WARNING: This command does not simply delete the portion of
the file visible in the window, but rather deletes the
entire file.

2.6.5 Deleting a Window

To delete a window (i.e., make the window itself go away),
first position the cursor in the desired window, and then
press:

        <CTRL><SHIFT>D

In the message area you will see the prompt:

        Delete Window?

Respond with a "Y" or "N". If you have made changes to the
file viewed through that window, and have not saved the
changed version, the message area will prompt you with:

        Not saved, Delete?

to make sure that you know that you have not saved the new
version.

When you delete a window, the screen space it occupied is
given back to the other window. If you delete window 1,
then window 2 becomes window 1 (since there is only one
window, it must be window 1).

2.7 Moving/Copying Blocks of Text

The ACTION! Editor allows you to move or copy text blocks
through the use of a copy buffer. Whenever you use the
command <SHIFT><DELETE> to delete a text line, that line
is temporarily stored in a region called the copy buffer.
You may then "paste" that deleted line using
<CTRL><SHIFT>P.

The copy buffer is cleared every time you use the
<SHIFT><DELETE>, with one exception. If you use the
<SHIFT><DELETE> consecutively (i.e., without doing any
other commands or text entry between the deletes), the
copy buffer is not cleared. Instead, the second (and
following) deleted lines are also stored in the copy
buffer, thus loading it with a text block.

Now, when you use the <CTRL><SHIFT>P command, the entire
text block in the copy buffer is pasted hack into the text.

Enough of the overview; on to the method itself.

To move a block of text, position the cursor on the first
line of the block, and press <SHIFT><DELETE> until you have

deleted the entire block. Move the cursor to the line above which you want the block to be pasted. Now, simply press <CTRL><SHTFT>P, and the block will be pasted.

To copy a block of text, use the same method as for moving a block, but first paste the block back into its original position before moving to the place where you want it copied. Since pasting does not clear the copy buffer, you can paste the same block (or line) in many different places, thus allowing multiple "copy"s.

2.8 Tags

Tags allow you to mark any location in your text. To set a tag at a given cursor position, press:

    <CTRL><SHIFT>T

The message area will display the prompt:

    tag id:

Enter the one character identification you want for that tag and press <RETURN>. If the id you give already has a tag associated with it, the old tag will be lost, and the id will refer to the new tag.

To move to a specified tag, press:

    <CTRL><SHIFT>G

The message area will display the prompt:

    tag id:

Enter the id character of the tag you want to go to. If the tag exists, the cursor will be moved to it, and the window will be moved to display the surrounding text. If the tag does not exist, the Editor will print

    tag not set

in the message area. This means that no tag with the given id character exists.

WARNING: Any operation which alters the contents of the line (character insertions, deletions, or changes, line breaking or recombining) clears the tags in the line.

HINT: If you use the digits (∅ - 9) as tag ids, you are more likely to remember the id character.

Chapter 3: Comparing ACTION! and ATARI Editors

3.1 Identical Commands

<SHIFT>
     Used in conjunction with letter keys to change the
     case of the letters or used to enter either an
     alternate character or command. Hold the <SHIFT> key
     down while pressing the following key in the sequence
     (e. g., <SHIFT>_, symbolizing the underline character
     "_", means that you should hold the <SHIFT> key down
     while pressing the "_" key).

<CTRL>
     Used in conjunction with one or more other keys to
     communicate a command or special character to the
     editor. Hold the key down while pressing the following
     key in the sequence (e. g., <CTRL><up arrow>,
     symbolizing the command to move the cursor up one
     line, means that you should hold the <CTRL> key down
     while pressing the <up arrow> key).

<ATARI>
     Display succeeding characters in inverse video. Press
     key a second time to return to normal video display of
     entered characters.

<ESC>
     Allows the following control character to be entered
     as text.

<LOWR>
     Shift letter entry to lower case (like unlocking the
     shift lock on a regular typewriter).

<SHIFT><CAPS>
     Shift letter entry to upper case letters only (like
     pressing shift and shift lock keys simultaneously on a
     regular typewriter).

<SHIFT><INSERT>
     Inserts a blank line on the line where the cursor is.
     The line where the cursor was and succeeding text
     lines are moved down to make room for this line.

<CTRL><INSERT>
     Inserts a blank space where the cursor is. Succeeding
     characters on the same line are moved right one
     character to make room for the inserted space.

<CTRL><up arrow>
     Moves cursor up one text line.

<CTRL><down arrow>
     Moves cursor down one text line.

<TAB>
     Moves to the next set TAB location, if any. Do not
     move if no additional TAB exists. Inserts spaces if no
     text (or spaces) exist here already.

<SHIFT><SET TAB>
     Establishes a TAB location at the current position of
     the cursor.

<CTRL><CLR TAB>
     Clears the TAB, if any, at the current cursor
     location.

3.2 Differing Commands

<BREAK>
     This key is not used by the ACTION! editor.

<SHIFT><CLEAR>
     Clears file in the current window. The editor warns
     you when the file has not been saved since the last
     text modification and allows you to cancel the
     command.

<RETURN>
     In Replace mode, this moves the cursor to the
     beginning of the next line. In Insert mode, it inserts
     a <RETURN> into the text.

<SHIFT><DELETE>
     Removes the line the cursor is on (like ATARI screen
     editor). Succeeding lines are moved up to replace the
     deleted line. Can be used repeatedly. Removed line(s)
     is(are) stored in a temporary holding area (called the
     copy buffer) for text copy/move processing. See
     <CTRL><SHIFT>P description and section 2.7.

<BACK S>
     If in Replace mode (see <CTRL><SHIFT>I), then this
     replaces the character to the left of the cursor with
     a space. In Insert mode, this removes the character to
     the left of the cursor and scrolls the rest of the
     line left to fill the empty space.

<CTRL><right arrow>
     Moves cursor right one character, stopping at the end
     of the line. Upon encountering the right margin of the
     window, the editor keeps the cursor within the display
     by scrolling the line contents to the left one
     character.

<CTRL><left arrow>
     Moves cursor left one character at a time, stopping at
     the beginning of the line. When at the left margin of
     the window but not yet at the left end of the line,
     the editor keeps the cursor within the display by
     scrolling the line contents right one character.

3.3 Commands Unique to ACTION! Editor

<CTRL><SHIFT>D
     Deletes the current window from the screen. The window
     contents are cleared from memory and the window itself
     disappears from the screen.

<CTRL><SHIFT>E
     Moves cursor to the end of the file.

<CTRL><SHIFT>F
     Finds a specified group of alphanumeric characters in
     text. If the character string is found, the cursor and
     window are moved to display it.

<CTRL><SHIFT>G
     Finds a user-specified tag anywhere in file (from any
     starting location). If found, the surrounding text is
     displayed and the cursor is positioned at the tag.

<CTRL><SHIFT>H
     Moves cursor to the beginning of the file (home).

<CTRL><SHIFT>I
     Alternates between the character Replace and the
     character Insert modes (the editor starts out in
     replace mode). The mode being switched to is shown in
     the message area. This command affects "<BACK S>" and
     "<RETURN>" handling.

The ACTION! Programming Environment

<CTRL><SHIFT>M
     Goes to the ACTION! Monitor (Part III).

<CTRL><SHIFT>P
     After one or more lines are loaded into the copy
     buffer using <SHIFT><DELETE>, the cursor can be moved
     anyplace in the text. Pressing <CTRL><SHIFT>P causes
     the lines in the copy buffer to be pasted at the
     current cursor location. Succeeding text (with tags)
     is moved down.

<CTRL><SHIFT>R
     Reads a file from a peripheral storage device. The
     file name given must be compatible with the DOS you
     are using. If no device is specified in the file name,
     D1: is assumed.

<CTRL><SHIFT>S
     String Substitution. This commands allows you to
     substitute a new string of up to 32 characters for an
     old one. Can also be used successively to substitute
     multiple occurrences of the old string.

<CTRL><SHIFT>T
     Sets a tag. The position of the cursor in the text is
     marked invisibly by a one-character alpha-numeric tag
     assigned to that location.

<CTRL><SHIFT>U
     Undo text changes. This command restores a changed
     line to its unmodified state. Tags for that line are
     not restored. This command works only on a line not
     deleted by <SHIFT><DELETE> and while the cursor has
     not left that line.

<CTRL><SHIFT>W
     Writes a file to a peripheral storage device. The file
     name must be compatible with the DOS you are using.

<CTRL><SHIFT>]
     Moves the window right 1 column (useful for editing
     files with lots of indentation).

<CTRL><SHIFT>[
     Moves window to the left 1 column.

<CTRL><SHIFT><up arrow>
     Moves the current window up a complete window. For
     continuity between the old and new windows, the top
     line from the last window is the new window's bottom
     line.

<CTRL><SHIFT><down arrow>
>   Moves the current window down 1 window. The new
>   window's top line is pulled from the previous window's
>   bottom line.

<CTRL><SHIFT>1
>   Moves from the second window to the first window. The
>   cursor goes to the previous cursor position, if any.

<CTRL><SHIFT>2
>   Moves from the first window to the second window. If
>   the second window has not been created previously in
>   the current editing session, then the second window is
>   created on the screen. The editor goes to the previous
>   cursor position, if any.

<CTRL><SHIFT>>
>   Moves the cursor to the end of the line and displays
>   that end of the line. If the line is longer than 38
>   characters, then the cursor moves to the end of the
>   line and the line is displayed so that only the
>   rightmost 38 characters show.

<CTRL><SHIFT><
>   Moves cursor to beginning of line and displays the
>   beginning of the line. If the line is longer than 38
>   characters, then the cursor moves to the beginning of
>   the line and the line is displayed so that only the
>   leftmost. 38 characters show.

NOTE: The maximum display line length is 4Ø characters. See
the options menu in Part III, Chapter 2.5.

<CTRL><SHIFT><BACK S>
>   At the beginning of a line, this command deletes the
>   otherwise invisible and inaccessible <RETURN> (EOL).
>   The lower line is appended to the end of the preceding
>   line. The succeeding lines are moved up one line, as
>   needed. At all other times, this command acts the same
>   as <BACK S>.

<CTRL><SHIFT><RETURN>
>   Insert a <RETURN> into the text. The line containing
>   the cursor is broken at the cursor. The portion of the
>   line to the left of the cursor remains on the current
>   line. The remainder of he line is inserted in a new,
>   left-justified line immediately below the left-hand
>   portion or the old line. The window is redrawn, as
>   needed.

Chapter 4: Technical Considerations

4.1 Files from Other Text Editors

The editor cannot handle files which do not contain
<RETURN> characters (EOL's) or have lines longer than 240
characters. Line lengths should be less than or equal to
the line width of your printer for the sake of convenience.

4.2 Key Recognition

During command line entry (in the message area), only the
<ESC>, <BACK S>, and <CLEAR> command keys are recognized.
All text characters are allowed.

During regular text entry, all text characters and commands
are allowed.

4.3 "Out of Memory" Error

This condition can result from an editing session in which
you made quite a few insertions and/or substitutions, or
from typing in a file which is too big (this will occur
very rarely).

When you get this error, immediately write the text file
out to a storage device, and then restart ACTION! (using
the 'BOOT' command in the Monitor). You can then go back to
the Editor and read your text file back in, and continue
working on it.

Part III: The ACTION! Monitor

Part III: The ACTION! Monitor

Chapter 1: Introduction

Part III describes the ACTION! monitor - control center of
the ACTION! system. It connects to all of the functions in
ACTION!.

```
                      +--------------+
                      |              |
+----------+          |    ACTION!   |          +------------+
|          |          |    Monitor   |          |            |
| compiled |----------|    Part III  |- - - -   |    DOS     |
| program  |          |              |          | Appendix D |
|          |          |              |          |            |
+----------+          +--------------+          +------------+
                 /           |          \
                /            |           \
               /             |            \
      +---------+    +----------+    +----------+
      |         |    |          |    |          |
      | ACTION! |    | ACTION!  |    | ACTION!  |
      | Editor  |    | Compiler |    | Options  |
      | Part II |    | Part V   |    |  Menu    |
      |         |    |          |    | Part III |
      +---------+    +----------+    +----------+
```
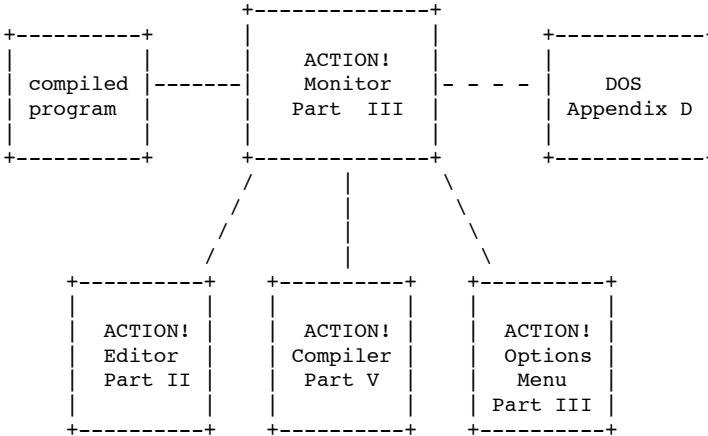
The monitor is characterized by an inverse video line
across the top of the screen, containing the prompt '>' and
the cursor at the left margin.

1.1 Vocabulary
----

term	where defined
<address>	Part IV
<compiler constants>	Part IV
<filespec>	below
<identifier>	Part IV
<statement>	Part IV
<value>	Part IV

When the term '<filespec>' is used in this part, it refers
to a standard ATARI file specifier consisting of a device
(P:, C:, D1:, D2:, etc.) and a file name in the case of
disk drives.

1.2 ACTION! Monitor Concepts and Features

The ACTION! monitor contains two chief features - the
command line and the message area. Both are described
below.

These areas are unique to the ACTION! Monitor. However, the
ACTION! compiler uses a similar screen format (see part IV
on the ACTION! compiler).

You communicate with the ACTION! monitor through the
command line. The command line is the inverted video line
at the top of the screen. It contains both the prompt '>'
and the cursor at the beginning of the line.

Commands are recognized by the first character entered
after the prompt '>'. Thus, 'E', 'Edit', and 'Ejunk' all
tell the ACTION! monitor to call the ACTION! editor. The
various ACTION! monitor commands are summarized in chapter
2.

Below the command line is the message area. The message
area is the large, outlined block in the middle of the
screen. It is a multipurpose area. When a program is
running, it is used to display program results. It can also
be used to trace program execution (see the options menu
choice 'trace' in chapter 2). When either the operating
system or the ACTION! compiler finds an error, the message
area contains the error number and the program text around
the line where the error was found.

In its role as the command center of ACTION, you can move
from the ACTION! monitor to any of several different
ACTION! functions. To get an idea of the relationship
between the various ACTION! parts, see the diagram on the
preceding page. You can execute a compiled ACTION! program
(see the ACTION! Monitor's RUN command in Chapter 2). You
can also call the ACTION! Editor (described in Part II) or
call the ACTION! Compiler (described in Part V). If you are
using disk drives, you can even call the DOS (see the
ACTION! monitor DOS command, mentioned in Chapter 2).

The ACTION! Programming Environment

Chapter 2: ACTION! Monitor Commands

2.1 BOOT - Restarting ACTION!

Sometimes you need to restart ACTION! from the ACTION!
monitor. This might occur after a fatal error or upon
return from DOS. You can restart ACTION! by entering
'BOOT', then pressing <RETURN>.

Examples: BOOT   <RETURN>
             B   <RETURN>

WARNING: text in the ACTION! Editor will be lost. Compiled
programs and their program variables will also be lost.

2.2 COMPILE - Compiling Programs

In ACTION!, a program must be processed by the ACTION!
compiler before it can be run from the monitor. You can
call the ACTION! compiler from the monitor, using:

FORMAT: COMPILE "<filespec>"

The "<filespec>" is an option which allows you to compile
programs which are stored on a peripheral device (disk,
cassette, etc.). If no "<filespec>" is specified, then the
contents of the Editor buffer is compiled. If you are using
two windows, the file in the window which contained the
cursor when you left the Editor is compiled.

If the Compiler finds a syntax error while compiling the
program, the error number and the line on which the error
occurred are display in the Monitor's message area. The
Compiler then returns control back to the Monitor.

NOTE: An ERROR 3 during a compile causes the system to hang
when you return to the editor. Type SET $E=$491^ to the
monitor to reset the ACTION! memory pointer; then go back
to the Editor.

Examples:
     COMPILE   <RETURN> (compile the program in the
     C   <RETURN>   current Editor window.)

     C "C:"   <RETURN> (compile from cassette)

     C "D1:PRIME.ACT"   <RETURN>   (compile PRIME.ACT
     COMPILE "PRIME.ACT"   <RETURN> from disk drive #1)
Notice that the file name specified in the last example

does not have a device given. If no device is given, the
device D1: is assumed.

2.3 DOS - Transfer to DOS

You can transfer to OS/A+, DOS XL, or ATARI DOS by entering
'DOS', then pressing <RETURN>.

Examples: DOS   <RETURN>
             D   <RETURN>

NOTES: since ATARI DOS and some of its utilities use the
same memory that ACTION! uses, you should always take the
precaution to save all files before going to ATARI DOS.

Exiting to ATARI DOS can cause a system crash if DUP.SYS is
not present on the disk in drive 1.

2.4 EDIT - Transfer to the ACTION! Editor

You can transfer to the ACTION! editor by entering
'EDITOR', then pressing <RETURN>.

Examples: EDITOR   <RETURN>
             E   <RETURN>

NOTE: if you were just compiling a program from the editor
and the compile failed due to a syntax error, you will find
that the ACTION! editor cursor is on the line following the
error.

2.5 OPTIONS - The Options Menu

The options menu allows you to alter certain operational
parameters of the ACTION! Monitor, Compiler, and Editor.
Enter the options menu by entering OPTIONS, then pressing
<RETURN>.

Examples: OPTIONS   <RETURN>
             O   <RETURN>

Each option is displayed in the command line. If you want
to change that option, type in the the value you want, and
press <RETURN>. If you do not want to change that option,
simply press <RETURN>. If you want to exit the options menu
all together, press <ESC>.

NOTE: A summary of the options available may be found in
Appendix G.
Following is a description of each of the options

available. Each description contains the command line prompt for that option, the initial state of that option, and the components of the ACTION! system that option affects (M = Monitor, C = Compiler, E = Editor).

Display   Y   M,C,E

The screen display can be turned off for greater speed during disk I/O and during ACTION! compiler processing. You can turn the screen display off (enter 'N') or you can leave it on (enter 'Y').

Bell   Y   M,C,E

The bell rings whenever errors are encountered in the ACTION! Monitor, Compiler, or Editor. It also rings whenever the ACTION! monitor is called. You can turn that bell off (enter 'N') or you can leave the bell on (enter 'Y').

Case sensitive N   C

When this option is 'Y', distinction is made in variable names between upper and lower case letters (i.e., 'count' differs from 'Count' and 'COUNT') and the language statements (e. g., FOR, WHILE, DO etc.) must be in upper case. However, for the ease of beginning ACTION! programmers, case sensitivity is turned off when you enter the ACTION! system.

Trace   N   C

With this option you can control the compiler's ability to trace program compilation. When this option is enabled ('Y'), the compiler will note in the Monitor's message area every routine call, together with the parameters passed to that routine. See chapter 4 for more information on the usage of this option.

List   N   C

The ACTION! compiler can be commanded to display the current line being compiled in the message area of the screen. Enter 'Y' to enable this listing or enter 'N' to disable this listing.

Window 1 size   18   E

The size of the ACTION! Editor's window 1 is set explicitly. Window 2 size is set implicitly by the relationship with window 1 -- the two windows have a combined size of 23 lines. When there are two windows, each can contain no less than 5 lines and no more than 18 lines.

Enter the number of lines for window 1, then press
<RETURN>. An entry greater than 18 is converted to 18 and
an entry less than 5 is converted to 5.

Line size?   12Ø   E

The line length is the number of characters in the line,
counting from the left margin (see next option). The line
length is used to help you control the size of lines listed
to the printer. The bell sounds whenever the actual number
of characters in the line exceeds the entered line length.
You enter the number of characters of the line length.

WARNING: you can set the line length to a value which is
out of the correct range. The options menu does not check
for this error. Lines longer than 24Ø characters are
arbitrarily shortened by the ACTION! Editor.

Left margin?   2   M,E

The left margin is the starting point for the line count
mentioned earlier. This option is offered so that you can
get the full use of a screen which displays the leftmost 2
characters (not all TV screens can do this!). It is
suggested that you keep the left margin as close to the
left edge of the screen as you feel comfortable with.
Normally set at 2, you can set the left margin as low as Ø
and as high as 39. Enter the appropriate left margin
location and then press <RETURN>.

WARNING: Do not enter a number greater than 39 when using
an ATARI system with the standard display.

EOL character?   (blank)   E

The EOL (End Of Line) character is the character displayed
by the ACTION! editor at the end of a line. Enter the
character which you want to see displayed as the visible
EOL character, then press <RETURN>. Changing from a space
to a visible character would generally only be useful for
removing trailing spaces from lines. However, you may
prefer a visible EOL for reasons of your own. If you desire
such, we suggest any of the ATARI graphics characters
(e. g., <CTRL>T is a solid circle).

2.6 PROCEED - Restarting a Halted Program

Restart a halted program (continue from a stop caused by pressing the <BREAK> key using the 'Break' Library routine) by entering 'PROCEED', then pressing <RETURN>. The program continues as if the interruption had not occurred.

Examples:
     PROCEED   <RETURN>
     P   <RETURN>

2.7 RUN - Program Execution

You can run any program which has just been compiled and is still in the program area. The command has the following formats:

     RUN
     RUN "<filespec>"
     RUN <address>
     RUN <routine>

where <routine> is a valid PROC or FUNC identifier (e. g., for 'PROC Prime()' you would use 'Prime' as the routine identifier).

The first format is used to run a program you have compiled from the Editor buffer.

The second allows you to load a source file stored on a peripheral device, have it compiled by the Compiler, and then it will be run. Remember that the compiled version of your program which has been saved using the 'WRITE' Monitor command must be loaded from DOS. That is, because it is now compiled object code.

The third format allows you to run a program (or routine) which begins at a given address. This is useful when you are trying to debug a program which calls a machine language routine you have written.

The fourth is used to run only one routine from a program which you have compiled.

After program execution, control returns to the ACTION! monitor. When some kind of significant error occurs (e. g., an infinite loop), control does not return to the ACTION! monitor. Such an error requires pressing the <SYSTEM RESET> key in order to return to the ACTION! Monitor.

Additional information on the behavior of running programs

is in the next chapter.

Examples:
        RUN   <RETURN>   (run a program compiled
        R   <RETURN>   from the Editor buffer.)

        RUN "C:" <RETURN> (pull a program from cassette,
                            compile it, then run it)

        RUN "PRIME.ACT" <RETURN>   (pull PRIME.ACT from
        R "D1:PRIME.ACT" <RETURN>   disk #1, compile it,
                                      then run it)

        R $4ØØ <RETURN>   (run a program at address $4ØØ

        RUN 1Ø24 <RETURN> (run a program at address 1Ø24)

        R Prime   <RETURN>   (run the just-compiled
                                procedure 'Prime()')

NOTE: R *<RETURN> locks up the system. <RESET> it.

2.8 SET - Setting a memory value

The SET command in the Monitor works exactly as in the
Language itself, so we will refer you there for a
description of its usage. See part IV, section 7.3.

2.9 WRITE - Saving Compiled Programs

You can write a compiled program (called a binary file) to
disk for later execution directly from DOS by entering
'WRITE', then, in quotes, a valid file specification. The
format is:

        WRITE "<filespec>"   <RETURN>

The binary file in memory is saved to the specified file on
the disk. The file is created, if necessary. If there is
not sufficient room on the disk, or the disk is write-
protected, you are warned with an error message and can try
again. Important: When using SpartaDOS or OS/A+ see note in
part I, chapter 2.

Examples:
        WRITE "PRIME.BIN" <RETURN>   (save a compiled
        W "D1:PRIME.BIN" <RETURN>   version of the
                                          PRIME program to
                                          disk 1)
        W "C:"   (save the compiled program to cassette)
The OS or DOS command to execute a machine language program

The ACTION! Programming Environment

can be used to execute a program saved to by the 'W'
command. See the references mentioned in Appendix D.

NOTE: If a disk error causes a write command to fail, the
IOCB is not properly closed. A disk change before
performing another disk operation will have invalid data
written to the new disk. If such an error occurs, type 'X
Close(1) <RETURN>' to the monitor. You can then erase the
file which caused the error.

2.1Ø XECUTE - Immediate Commands

You can execute any ACTION! language command or any ACTION!
compiler directive (except MODULE and SET) from the ACTION!
monitor. Preface any such command with the command XECUTE,
then the statement(s). Press <RETURN>.

Examples
        XECUTE PrintE("Hello World") <RETURN>
        X trace = 255   <RETURN>

NOTE: using this command is very similar to the BASIC
direct mode.

2.11 ? - Display a Memory Location

You can display the value either of a variable or of a
specified memory location. Enter '?'. Then enter a compiler
constant. Press <RETURN>. The format is:

        ? <compiler constant>   <RETURN>

The ACTION! monitor shows you the actual memory location
(expressed in both decimal and hexadecimal formats),
followed by the printable ATASCII value of that location,
the hexadecimal value of the CARD starting at the specified
location, the decimal value of the BYTE, and the decimal
value of the CARD starting at the specified location. If
the identifier is not in the ACTION! compiler's symbol
table, then a "variable not declared error" occurs.

Example using a 4ØØ/8ØØ machine:

        +--+
        | >? $FFFE |
        +--+
        | 65534,$FFFE = s $E6F3 243 59123 |
        | |
        +--+
Same example using a XL/XE machine:

```
+---------------------------------------+
|  >? $FFFE                             |
+---------------------------------------+
|  65534,$FFFE = , $CØ2C 44 49196       |
|                                       |
+---------------------------------------+
```

NOTE: the results might not be what you expect because
memory has been altered since the compile - see SYMBOL
TABLE in Part V.

* - Memory Dump

Starting from a specified memory address, you can display
the memory contents of sequential locations in a format
identical to that described just above. Simply enter '*'
and the <address>. The format is:

        * <address> <RETURN>

The monitor returns a list of the memory contents in the
variety of formats (mentioned above) at the rate of one
line per memory location. You can stop the listing by
pressing <space bar>. You can temporarily halt the listing
by entering <CTRL> 1. Press <CTRL> 1 a second time to
continue the listing.

Example:

```
+---------------------------------------+
|  >* $6ØØ                              |
+---------------------------------------+
|  1536,$Ø6ØØ  = * $ØØØØ Ø Ø            |
|  1537,$Ø6Ø1  = * $ØØØØ Ø Ø            |
|  1538,$Ø6Ø2  = * $ØØØØ Ø Ø            |
|  1539,$Ø6Ø3  = * $ØØØØ Ø Ø            |
|  154Ø,$Ø6Ø4  = * $ØØØØ Ø Ø            |
|  1541,$Ø6Ø5  = * $ØØØØ Ø Ø            |
|  1542,$Ø6Ø6  = * $ØØØØ Ø Ø            |
|  1543,$Ø6Ø7  = * $ØØØØ Ø Ø            |
|  1544,$Ø6Ø8  = * $ØØØØ Ø Ø            |
|  1545,$Ø6Ø9  = * $ØØØØ Ø Ø            |
|  1546,$Ø6ØA  = * $ØØØØ Ø Ø            |
|  1547,$Ø6Ø8  = * $ØØØØ Ø Ø            |
+---------------------------------------+
```

Chapter 3: Program Debugging Facilities

You have probably written programs which do not work the
way that you expected, not because of syntax errors, but
simply because something you are doing (or think you are
doing) is not executing properly. With the ACTION! Monitor
and its options menu you can debug your program step by
step to determine where the error is occurring.

The TRACE Option

     One of the options available in the options menu is
     'Trace'. If this option is enabled ('Y'), you can
     follow your program's execution. When the trace is
     on, every time a routine is called its name and
     parameters are displayed on the screen. You might be
     able to discover what is going wrong simply by
     looking at the order of the routine calls and/or the
     parameters being passed. If this is so, fantastic! If
     not, you probably need to do some major debugging.

The first thing you need to do before doing any major
debugging is to stop your program sometime during its
execution. There are two ways to do this in ACTION!: the
BREAK> key and the Library routine 'Break'.

The <BREAK KEY>

     Although the <BREAK> key is disabled during use of
     the ACTION! Editor, it is usable during program
     execution with certain restrictions. The <BREAK> key
     will stop program execution only if you are:

         1) doing some sort of I/O
         2) calling a routine with more than 3 parameters

     These might seem strange circumstances, but there is
     a good reason for them. The ACTION! system itself
     does not check to see if the <BREAK> key has been
     pressed during program execution, but the system does
     make calls to CIO in the above two circumstances, and
     CIO checks to see if the <BREAK> key has been
     pressed.

Library PROC Break()

If you want program execution to stop at any given place,
simply make a call to this Library routine at that point.
This routine acts exactly like the <BREAK> key, except that
it works under all circumstances. Using this method to stop
a program is more reliable than pressing <BREAK> because

you know exactly where you are in the program when the program stop occurs.

NOTE: you may use this routine more than once in one program if you want to break execution at more than one place.

Now that you have stopped the program, you can use the Monitor commands '*' and '?' to look at the value of the variables you are using. If this method of debugging is used with the 'Trace' option on, you can even find out where you are in your program (if you are using the Library 'Break', you already know where you are) and so look at the variables local to the procedure you are in as well as the global ones.

If this method does not work, we can only suggest that you insert diagnostic 'Print' statements into your program (e. g., PrintE("In loop FOR x=1 to 1ØØ") PrintBE(x) might be used to debug a FOR loop which has run amuck).

The ACTION! Programming Environment

Part IV: The ACTION! Language

```
                Part IV:  The ACTION! Language
                -----------------------------


                    Chapter 1: Introduction
                    -----------------------
```

The ACTION! language is the heart of the ACTION! system. It
incorporates the good points of both C and PASCAL and, at
the same time, is the fastest high level language available
for ATARI 8-bit home computers. If you have a background in
BASIC or some other unstructured language, you will find
ACTION! a welcome change because its structure is similar
to the way we structure ideas in our own minds. You can
actually look at an ACTION! program someone else has
written and understand what is going on, without having to
wade through a thousand GOTOs and undeclared variables.

Program structure is simple in ACTION!, because programs
are built component by component. The components are groups
of related statements which accomplish some task. When you
have written components for all the tasks required in your
program, it is a simple matter to execute them. It is very
similar to a list of chores, such as

```
        1.)  Make your bed
        2.)  Clean your room
        3.)  Dust the living room furniture
        4.)  Wash the Dog
```

except that the computer will do the tasks in the order in
which you present them, not in whatever order it likes
best.

Having separate components also makes it very easy for you
to do a single task over and over, or do the same in ten
different situations and places.

The only requirement this structured approach imposes is
that a program must consist of proper components (in
ACTION! they are called procedures and functions) for it to
be valid. A program usually contains many components, but
at least one is required. This is not a restrictive
requirement at all, as you will soon find out. In fact, it
makes your program more comprehensible to yourself and
others.

NOTE: when compiling and running a program with many
routines, the last routine is considered to be the main
one, so you should use it to control your program.

                    Chapter 2: ACTION! Vocabulary

In our discussion of ACTION! we will use some terminology
that we should explain. We will use as little jargon as
possible, but some is required to differentiate between
parallel but different concepts later on. What terms we do
not present here will he explained when they are first
used. Before going into the special notations used in this
part, we will give you a list of the keywords in ACTION!. A
"keyword" is any word or symbol the ACTION! compiler
recognizes as something special, whether it be an operator,
a data type name, a statement, or a compiler directive:

        AND   FI   OR   UNTIL   =   (
        ARRAY   FOR   POINTER   WHILE   <>   )
        BYTE   FUNC   PROC   XOR   #   .
        CARD   IF   RETURN   +   >   [
        CHAR   INCLUDE RSH   -   >=   ]
        DEFINE   INT   SET   *   <   "
        DO   LSH   STEP   /   <=   '
        ELSE   MOD   THEN   & $;
        ELSEIF   MODULE   TO   %   ^
        EXIT   OD   TYPE   !   @

WARNING: You may not use the above keywords in any context
other than the one defined in the ACTION! language;
specifically, you may not use these words as identifiers.

2.1 Special Notations

When discussing the language, we use some terms which
might be unfamiliar to you, so their meanings are
presented here. The list is in alphabetical order, with the
symbols at the end.

Address   An address is a location in memory. When you
            tell the computer to put something into memory,
            you must give it an address, just like you give
            the post office the address of the destination
            of a letter on the letter's front. In the
            computer there are only house numbers, no
            streets, no cities, states, and no zip codes.
            So an address to the computer is simply a
            number.

Alphabetic Any letter of the alphabet, in either upper
            (ABC) or lower (abc) case. "Alphanumeric"
            includes the digits "Ø" through "9" as well.

Identifier Throughout the manual we will refer to the

48

names you give to variables, procedures, etc. as identifiers. We do this because names in ACTION! must follow some guidelines:

1. They must start with an alphabetic character
2. The rest of the characters must be alphanumeric, or the underline (_) character.
3. They may not be keywords.

These rules must be obeyed when you wish to create an identifier, otherwise you will get a syntax error.

MSB, LSB   MSB stands for "Most Significant Byte", and LSB stands for "Least Significant Byte". In the decimal system we have significant digits, not bytes. For example, the most significant digit of '54' is '5', and the least significant is '4'. If you are unfamiliar with the byte storage system, do not worry. You can program very well in ACTION! without knowing anything about the internal workings of the computer.

Note that two-byte numbers stored and used by ACTION! are generally in LSB, MSB order, as is conventional on 6502-based machines.

$ The dollar sign, when used in front of a number, tells the computer that the number is hexadecimal (the base 16 number system; useful when working directly with the computer), not the customary decimal.

Examples:
        $24FC $0D
        $88 $F000

The ACTION! Programming Environment

;   The semicolon is the comment symbol, and
             everything on a line after it is ignored by the
             compiler.

                 Examples:
                     ;This is a comment
                     This is not and will cause a compiler
                     error
                     ; This comment has a ; semi-
                     ; colon in it
                     var=3   ;comments can come
                                ;after executable
                                ;statements
                     ;this is a 3 line comment
                     ;
                     ;with a blank line in it

< and >   Whatever is between these two symbols is used
             to define some part of a format. It is never a
             keyword, and usually is a term describing what
             goes in its place in the construction (e. g.,
             <identifier> means a valid identifier should be
             used).

{ and }   Whatever is between these is optional in the
             format construction ({<identifier>} means that
             a valid identifier may be used here, but is not
             required).

|: and :| As in music, these symbols denote repetition.
             Anything between them is repeatable from zero
             times on up (e. g., |:<identifier>:| means that
             you could have a list of zero or more
             identifiers here).

| This symbol shows an 'or' situation (e. g.,
             <identifier> | <address> means you could use
             either an identifier or an address, but not
             both.

Chapter 3: Fundamental Data Types

Before discussing the Fundamental Data Types, something
must be said about variables and constants, since they are
the basic data objects the computer manipulates.

3.1 Variables

Legal variable names must be valid identifiers. Other than
this there is no restriction on variable names. Because a
working knowledge of functions and procedures is required
before discussing the scope of a variable, the topic is
presented later in section 6.3.

3.2 Constants

There are three types of constants in ACTION!: numeric
constants, string constants, and compiler constants.

Numeric constants may be entered in three different
formats:

        1) Hexadecimal
        2) Decimal
        3) Character

Hexadecimal constants are represented by a dollar sign ($)
in front of the number.

Examples:
        $4AØØ
        $ØD
        $3ØØ

Decimal constants require no special character to define
them as decimal.

Examples:
        655ØØ
        2
        324
        46

NOTE: Both hexadecimal and decimal numeric constants may
have a negative sign in front of them, thus:

                        -$8C
                        -436Ø

The ACTION! Programming Environment

Character constants are represented by a single quote (')
preceding the character. Characters are numeric constants
because they are internally represented as one byte
numbers, as per the ATASCII character code set.

Examples:
        'A
        '@
        '"
        'v

String constants consist of a string of zero or more
characters enclosed by double quotes ("). When stored in
memory, they are preceded by their length. The double
quotes are not considered as part of the string; if you
want a " in your string, place two double quotes together
(see examples).

Examples:
        "This is a string constant"
        "a "" double quote in a string"
        "58395"
        "q"   (a single character string constant)

Compiler constants are different from the above types of
constants, in that they are used at compile time to set
certain attributes of variables, procedures, functions, and
code blocks, and are not evaluated at run-time. The
following formats are valid:

        1) A Numeric Constant
        2) A Predefined Identifier
        3) A Pointer Reference (see section 8.1.2)
        4) The Sum of Any Two of the Above

We have already talked about the first format, but the
other three require some explanation. When you use a
predefined identifier (i. e., a variable, procedure or
function name) in a compiler constant, the value used is
the address of that identifier. The third format allows
pointer references as compiler constants. The last one
permits you do simple addition of a combination of any two
of the other three types. Here are some examples which show
the valid formats in use:

        cat   ;uses the address of the variable 'cat'
        $8D00 ;a hex constant
        dog^   ;a pointer reference as a constant
        5+ptr^ ;5 plus the contents of the pointer 'ptr'
        $80+p ;evaluates to $80 plus the address of 'p'

3.3 Fundamental Data Types

Data types allow humans to make sense out of the stream of
bits the computer understands and manipulates. They allow
us to use concepts we understand, so we need not know how
the computer does what it does. ACTION! supports three
fundamental types and some advanced extensions of these
(see chapter 8 for the extended types). The basic ones are
BYTE, CARD, and INT, and each is detailed below. All of the
fundamental types are numeric, and so allow you to use
numeric format when entering data.

3.3.1 BYTE

The type BYTE is used for positive integers less than 256.
It is internally represented as a one-byte, unsigned number
-- its values range between Ø and 255. At first glance this
might seem a useless type, but it has two worthwhile
applications. When used as a counter in loops (WHILE,
UNTIL, FOR) program speed will increase because it is
easier for the computer to manipulate one byte than many.

Also, since characters are represented inside the computer
as one-byte numbers, BYTE is also useful as a character
type. In fact, the ACTION! compiler allows you to use the
keywords BYTE and CHAR interchangeably, so those of you
with PASCAL or C experience can use CHAR when dealing with
characters and feel more at home.

3.3.2 CARDinal

The CARD type is very similar to the BYTE type, except that
it handles much larger numbers. This is because it is
internally represented as a two-byte unsigned number. Hence
its values range from Ø to 65,535.

TECHNICAL NOTE: A CARD is stored in the LSB, MSB form which
is standard on 65Ø2-based machines.

3.3.3 INTeger

This type is like BYTE and CARD in that it is integer only,
and can be entered in numeric format, but that is where the
similarity ends. INT allows both positive and negative
numbers ranging from -32768 to 32767. It is internally
represented as a two byte signed number.

TECHNICAL NOTE: INTs are stored LSB, MSB like CARDs.

3.4 Declarations

Declarations are used to let the computer know that you
wish to define something. For example, if you want the
variable 'cost' to be of the type CARD, somehow you have to
tell this to the computer. Otherwise the computer will not
know what to do when it sees 'cost'.

Every identifier you use must be declared before it is
used, whether its a variable, procedure, or function name.
Variable declarations will be explained here, followed by a
note about numeric constant declarations; procedure and
function declarations are explained in chapter 6.

3.4.1 Variable Declaration

The procedure for declaring a variable is the same no
matter what fundamental type you want it to be. The basic
format is:

```
 <type> <ident>{=<init info>} |:,<ident>{=<init info}:|
```

where

```
     <type>        is the fundamental type of the
                   variable(s) being declared
     <ident>       is an identifier naming the variable
     <init info>   allows you to initialize the value
                   of the variable, or define the
                   memory location of that variable
```

'<init info>' has the form:

        <addr> | [<value>]

where
        <addr>   is the address of the variable, and
                        must be a compiler constant

        <value>   is the initial value of that varia-
                        ble, and must be a numeric constant

NOTE: an explanation of <, >, {, }, |:, :|, and | can be
found in the vocabulary (chapter 2).

Notice that you can optionally have more than one variable
declared by one <type>. You can also optionally tell the
compiler where you want each variable to reside in memory
or initialize the variable to a value. The following
examples should help clarify this format:

        BYTE top,hat   ;declare 'top' and 'hat' as BYTE
                          ;variables

        INT num=[Ø]   ;declare 'num' as an INT varia-
                          ;ble and initialize it to Ø

        BYTE x=$8ØØØ,   ;declare 'x' as BYTE, placing it
                          ;at memory location $8ØØØ
            y= [Ø]   ;declare and initialize 'y'

        CARD ctr=[$83D4],   ;declares and initializes
             bignum=[Ø],   ;three variables as CARD
             cat=[3ØØØØ]   ;type

In the last two examples you may note that the variables
need not be on the same line. The ACTION! compiler will
keep reading in variables of the type given as long as
there are commas separating them, so remember not to put a
comma after the last variable in a list (strange things
will happen if you do).

Variable declarations must come immediately after a MODULE
statement (see section 7.4) or at the beginning of a
procedure or function (see sections 6.1.1 and 6.2.1). If
you use them anywhere else, you will get an error.

3.4.2 Numeric Constants

Numeric constants are not explicitly declared. Their usage
declares their type. A numeric constant is considered to be
of type BYTE if it is less than 256, otherwise it is
considered to be of type CARD. For all practical purposes,
negative constants (e. g. -7) are treated as type INT:

        Constant   Type
        --------   ----
        543   CARD
        $ØD   BYTE
        $F42   CARD
        'W   BYTE

Chapter 4: Expressions

Expressions are constructions which obtain values from
variables, constants, and conditions using a specific set
of Operators. For example, '4+3' is an expression that
equals '7' as long as we take the '+' operator to mean
addition. If the operator were '*' instead, multiplication
would result, and the expression would equal '12' (4*3=12).
ACTION! has two types of expressions, arithmetic and
relational. The example given above is an arithmetic
expression. Relational expressions are those which involve
a 'true' or 'false' answer. '5 >= 7' is false if we take
'>=' to mean "is greater than or equal to". This type of
expression is used to evaluate conditional statements (see
section 5.2.1). A conditional statement in every day life
might be, "If it is five o'clock or later, then it is time
to go home." An ACTION! relational expression for this
might be:

                    hour >= 5

You yourself make this check (and many others)
automatically when you look at a clock, but the computer
needs to be told exactly what to check for.

Before going into the expressions themselves, we need to
define the operators that apply to each type of expression.
After that we will discuss each expression, and then go
into some special extensions of relational expressions.

4.1 Operators

ACTION! supports three kinds of operators:

            1) Arithmetic operators
            2) Bit-wise operators
            3) Relational operators

As suggested by the names of the first and last, they
specifically pertain to an expression type. The second
class of operators performs arithmetic and addressing
operations at bit level.

The ACTION! Programming Environment

4.1.1 Arithmetic Operators

The arithmetic operators are those we commonly use in math, but some are modified so that they can be typed in from a computer keyboard. Here is a list of those ACTION! supports, each followed by its meaning:

```
        -     unary minus (the negative sign)  Ex: -5
        *     multiplication  Ex: 4*3
        /     integer division Ex: 13/5 (this equals 2,
                          since the remainder is dropped)
        MOD   remainder of integer division  Ex: 13 MOD 5
                            (this equals 3, since 13/5 =2
                                 with a remainder of 3)
        +     addition  Ex: 4+3
        -     subtraction  Ex: 4-3
```

Notice that '=' is not an arithmetic operator. It is used only in relational expressions, certain declarations, and assignment statements.

4.1.2 Bit-wise Operators

Bit-wise operators manipulate numbers in their binary form. This means that you can do operations similar to those the computer does (since it always works with binary numbers). The following list summarizes the operators:

```
        &     bit-wise 'and'
        %     bit-wise 'or'
        !     bit-wise 'exclusive or'
        XOR   same as "!"
        LSH   left shift
        RSH   right shift
        @     address of
```

The first three compare numbers bit by bit and return a result dependent on the operator, as seen below.

```
                                      Bit-wise And
& compares the two bits,      Bit A  Bit B   Result
      returning a value         1      1        1
   based on this table:         Ø      1        Ø
                                Ø      Ø        Ø
                                1      Ø        Ø

   Example: 5 & 39 -- ØØØØØ1Ø1  (equals 5 decimal)
                      ØØ1ØØ111  (equals 39 decimal)
                  & ----------
                      ØØØØØ1Ø1  (result of & is 5)
```

                                         Bit-wise Or
% returns a value depen-   Bit A   Bit B   Result
   dent on this table: 1   1   1
                                  Ø   1   1
                                  Ø   Ø   Ø
                                  1   Ø   1

     Example: 5 % 39 -- ØØØØØ1Ø1 (5)
                        ØØ1ØØ111 (39)
                  % ----------
                        ØØ1ØØ111 (result of % is 39)

                                         Bit-wise XOR
! returns a value depen-   Bit A   Bit B   Result
   dent on this table: 1   1   Ø
                                  1   Ø   1
                                  Ø   Ø   Ø
                                  Ø   1   1

     Example: 5 ! 39 -- ØØØØØ1Ø1 (5)
                        ØØ1ØØ111 (39)
                  ! ----------
                        ØØ1ØØØ1Ø (result of ! is 34)

Both LSH and RSH shift bits. If they operate on two-byte
types (CARD and INT) the shift occurs through both bytes.
In the case of INT, the sign of the number is not preserved
when using RSH or LSH, and may change. Their form is:

        <operand> <operator> <number of shifts>
where
        <operand>   is a numeric constant or
                                variable
        <operator>   is either LSH or RSH
        <number of shifts>   is a numeric constant or
                                variable used to deter-
                                mine the number of single
                                bit shifts to do

Some examples to illustrate both LSH and RSH follow:

   (5)   ØØØØØ1Ø1   (39)   ØØ1ØØ111
   (5 LSH 1 = 1Ø) ØØØØ1Ø1Ø   (39 LSH 1 = 78) Ø1ØØ111Ø
   (5 RSH 1 = 2) ØØØØØØ1Ø   (39 RSH 1 = 19) ØØØ1ØØ11

     operation   MSB   LSB
       ---   Ø1Ø1Ø11Ø   11ØØ1Ø1Ø   ($56CA)
     LSH 1   1Ø1Ø11Ø1   1ØØ1Ø1ØØ   ($56CA LSH 1 =$AD94)
     RSH 1   ØØ1Ø1Ø11   Ø11ØØ1Ø1   ($56CA RSH 1 =$2B65)
     LSH 2   Ø1Ø11Ø11   ØØ1Ø1ØØØ   ($56CA LSH 2 =$5B28)
     RSH 2   ØØØ1Ø1Ø1   1Ø11ØØ1Ø   ($56CA RSH 2 =$15B2)

Notice that a LSH by one is the same as multiplying by two,

and a RSH by one is like division by two (for positive
numbers). In fact, this method of multiplication and
division is faster than using '*2' and '/2' because it is
closer to what the computer understands, so the computer
does not need to translate the expression into its own
binary operation format.

The '@' operator gives the address of the variable to its
right. It cannot be used with numerical constants. '@ctr'
will return the address in memory of the variable 'ctr'.
The '@' operator is very useful when dealing with pointers.

4.1.3 Relational Operators

Relational operators are allowed only in relational
expressions, and relational expressions are allowed only in
IF, WHILE, and UNTIL statements. Relational operators may
not appear anywhere except in these statements. As outlined
in the overview of this section, relational operators test
conditions of equality. A table of the ACTION! relational
operators follows:

       =   tests for equality Ex: 4=7 (this is obvi-
                                     ously false)
       #   tests for inequality Ex: 4#7 (true)
       <>   same as "#"
       >   tests for greater than Ex: 9>2 (true)
       >=   tests for greater than or equal to Ex: 5>=5
                                     (this is true)
       <   tests for less than Ex: 2<9 (true)
       <=   tests for less than or equal to Ex: 5<=5 (this
                                     is true)
       AND logical 'and': see section 4.4
       OR   logical 'or'; see section 4.4

Both 'f' and '<>' mean the same thing to ACTION!, so you
may use the one you prefer. 'AND' and OR are special
relational operators, and are discussed in 'Complex
Relational Expressions', section 4.4.

TECHNICAL NOTE: The ACTION! Compiler does comparisons by
subtracting the two values in question and comparing the
difference to Ø. This method works correctly with one
exception -- if you are comparing a large positive INT
value with a large negative INT value, the outcome could be
wrong (since INTs use the highest bit as a sign bit).

4.1.4 Operator Precedence

Operators require some kind of precedence, a defined order of evaluation, or we would not know how to evaluate expressions like:
4+5*3

Is this equal to (4+5)*3 or 4+(5*3)? Without operator precedence its impossible to tell. ACTION!'s precedence is very precise but can he circumvented by using parentheses, since they have the highest precedence. In the following table the operators are listed in order of highest to lowest precedence. Operators on the same line have equal precedence and are evaluated from left to right in an expression (see examples).

```
( )                   parentheses
- @                   unary minus, address
* / MOD LSH RSH       mult, div, rem, etc...
+ -                   addition, subtraction
= # <> > >= < <=      relational operators
AND &                 logical/bit-wise and
OR %                  logical/bitwise or
XOR !                 bitwise exclusive or
```

According to this table, our earlier example, 4+5*3, would be evaluated as 4+(5*3) because the '*' is of higher precedence than the '+'. What if (4+5)*3 were intended? You would have to include the parentheses, as shown, to override the normal operator precedence. Here are some examples to look over:

```
expression        result      evaluation order
----------        ------      ----------------
4/2*3             6           /,*
5<7               true        <
43 MOD 7*2+19     21          MOD,*,+
-((4+2)/3)        -2          +,/,-
```

4.2 Arithmetic Expressions

An arithmetic expression consists of a group of numerical
constants, variables, and operators ordered in such a way
that there is a numerical result. The order is as follows:

        <operand> <operator> <operand>

where '<operand>' is a numeric constant, numeric variable,
FUNCtion call (see section 6.2.3), or another arithmetic
expression.   The   first   three   possibilities   are
straightforward enough, but the last one is a problem. Here
is an example to show you what we mean:

        starting expression: 3*(4+(22/7)*2)

```
  order    expression    evaluation    simplified exp
  -----    ----------    ----------    --------------
  start  3*(4+(22/7)*2)    ----        -----
    1         (22/7)         3          3*(4+3*2)
    2         (22/7)*2       6          3*(4+6)
    3       (4+(22/7)*2)    1Ø          3*1Ø
    4     3*(4+(22/7)*2)    3Ø          3Ø
```

'order' is   the   order   of   the   expression   evaluation,
'expression' shows which expression is being evaluated,
'evaluation' shows the evaluation of that expression, and
'simplified exp' shows the expression after the evaluation
has taken place.

Notice that expressions 2 through 4 contain another
expression as one of their operands, but that this
"expression as an operand" has already been evaluated,
leaving a number in its place, as seen in 'simplified'.

Some examples follow (all lowercase words are variables
constants):

```
        expression            evaluation order
        ----------            ----------------
        'A*(dog+7)/3              +,*,/
        564                      (none)
        var & 7 MOD 3            MOD,&
        ptr+@xyz                 @,+
```

Arithmetic expressions in ACTION! may involve operands of
differing data types. The result of such mixing is outlined
in the table below. The type at the intersection of any row
and column is the type resulting when the rows and columns
types are mixed:

```
          | BYTE   INT    CARD
    ------+------+------+------
   BYTE | BYTE   INT    CARD
      +
    INT | INT    INT    CARD
      +
   CARD | CARD   CARD   CARD
```

NOTE: using the unary minus (negative sign '-') results in
an implied INT type, and using the address operator, '@',
results in an implied CARD type.

TECHNICAL NOTE: using the '*', '/' or 'MOD' operand results
in an INT type, so processing of very large CARD values (>
32767) will not work properly.

4.3 Simple Relational Expressions

Relational expressions are used in conditional statements
to perform tests to see whether a statement should be
executed (more on conditional statements in section 5.2.1).
Note that they may be used ONLY in conditional statements
(IF, WHILE, UNTIL).

There may be only one relational operator in a simple
relational expression, so tests for multiple conditions
must be handled differently (They are covered in the
following section on complex relational expressions). The
form of a simple relational expression is:

        <arith exp><rel operator><arith exp>

where
        <arith exp>   is an arithmetic expression
        <rel operator>   is a relational operator

Here are some samples of valid relational expressions:

        @cat<=$22A7
        var<>'y
        5932#counter
        (5&7)*8 >= (3*(cat+dog))
        addr/$FF+(@ptr+offset) <> $FØ3D-ptr&offset
        (5+4)*9 > ctr-1

4.4 Complex Relational Expressions

Complex relational expressions allow you to cover a wider
range of tests by including multiple tests. If you want to
do something only on Sundays in July, how do you get the
computer to test whether its Sunday and then test whether
its July? ACTION! allows you to do this kind of multiple
testing with the AND and OR operators (remember how they
were glossed over in section 4.1.3?). The compiler treats
these as special relational operators to test a condition
using simple relational expressions. The form is:

        <rel exp><sp op><rel exp>|:<sp op><rel exp>:|

where
        <rel exp>   is a simple relational expression
        <sp op>   is one of the special operators AND
                       or OR

NOTE: There are no exceptions to this form. If you try
something else, you will usually get the compiler error
'Bad Expression'.

The truth table below shows what each of these operators
will do in a given situation. 'exp 1' and 'exp 2' are the
simple relational expressions on either side of the special
operator; 'true' and 'false' are the possible results of a
relational test.

                  RELATIONALS | RESULTS
                -------------+--------------
                exp 1 | exp 2 | AND | OR
                -------------+--------------
                true | true | true | true
                      +   +   +
                true | false | false | true
                      +   +   +
                false | true | false | true
                      +   +   +
                false | false | false | false

NOTE: You may use parentheses around one segment of a
complex relational expression to insure the order of
evaluation. If you do not do this, the expressions are
evaluated in left to right order. (see Examples)

WARNING: At the writing of this manual, the ACTION!
compiler sees the pairs AND -- &, and OR -- % as synonyms,
and they are evaluated in the same way (bit-wise). If you
follow the rules outlined above using them, you should have
no problems. Also, if you stick to using 'AND' and 'OR'
only in the relational sense, and '&' and '%' only in the

bit-wise sense, your programs will be compatible with possible upgrades of ACTION!.

Here are some samples of valid complex relational expressions:

```
cat<=5 AND dog<>13
(@ptr+7)*3 # $60FF AND @ptr <= $1FFF
x!$F0<>0 OR dog>=100
(8&cat)<10 OR @ptr<>$0D
cat<>0 AND (dog>400 OR dog<-400)
ptr=$D456 OR ptr=$E000 OR ptr=$600
```

Here is a confusing situation:

```
$F0 AND $0F
```

is false because the 'AND' is seen as a bit-wise operator being used in an arithmetic expression, whereas

```
$F0<>0 AND $0F<>0
```

is true because the 'AND' joins two simple relational expressions, and so is a special operator as used in complex relational expressions.

Chapter 5: Statements

A computer program would be useless if it could not
actively operate on data. You would be allowed to declare
variables, constants, etc., but there would be no way to
manipulate them. Statements are the active part of any
computer language, and ACTION! is no exception. Statements
translate an action you want to do into a form which the
computer can understand and execute properly. This is why
statements are sometimes referred to as executable
commands.

There are two classes of statements in ACTION!: simple
statements and structured statements. Simple statements
contain no other statements within themselves, whereas
structured statements are collections of other statements
(either simple or structured) put together following a
certain order. Structured statements may be broken down
into two categories:

        1) Conditional Statements
        2) Looping Statements

Each category is discussed separately in the section on
structured statements.

5.1 Simple Statements

Simple statements are those which do one thing only. They
are the basic building blocks of a program, since any
action the computer performs is a simple statement of one
kind or other. There are two simple statements in ACTION!:

    1) Assignment Statement (including FUNCtion Calls)
    2) PROCedure Calls

PROCedure and FUNCtion calls are discussed in chapter 6,
and the assignment statement follows. There are two
keywords that are also simple statements,

        EXIT   section 5.2.3.2
        RETURN   sections 6.1.2 and 6.2.2

but the last two are used in specific constructs, and so
are discussed where appropriate to their usage.

5.1.1 Assignment Statement

The assignment statement is used to give a value to a
variable. Its most common form is:

        <variable>=<arithmetic expression>

NOTE: <variable> may be a variable of a fundamental data
type, or it can be an array, pointer, or record reference.

NOTE: the expression MUST be arithmetic! If you try to use
a relational expression, you will get an error, because the
ACTION! compiler does not assign a numerical value to the
evaluation of a relational expression.

The assignment operator is '='. It tells the computer that
you want to assign a new value to the given variable. Do
not confuse it with the relational '='. Although they are
the same character, the compiler reads them differently,
each according to its context.

The following examples illustrate the assignment statement.
You will notice a variable declaration section preceding
the examples themselves. It is there because some of the
examples show what happens when you mix types (i.e. the
variable and value being assigned to it are not of the same
data type).

```
BYTE b1,b2,b3,b4
INT i1
CARD c1
```

b3='D   puts the ATASCII code number for 'D' into
                  the byte reserved for 'b3'.

B4=$44   puts the hex number $44 into the byte
                  reserved for the BYTE variable 'b4' ($44
                  is "D" in ATASCII and so 'b3' and 'b4'
                  now contain the same thing).

b1=b4+16   adds 16 to the numerical value of 'b4',
                  and puts the result into the byte
                  reserved for 'b1'.

c1=23439-$Ø7D8   puts the value 21431 ($53B7) in the two
                  bytes reserved for 'c1'.

i1=c1*(-1)   puts the value -21431 ($AC49) in the two
                  bytes reserved for 'i1'.

b2=i1   puts the value $49 (73) into the byte
          reserved for 'b2'. Notice that the
          computer takes the LSB of 'i1' to put
          into 'b2' (the MSB of it is $AC; LSB is
          $49).

b2=b2+1   adds 1 to the current value of 'b2' and
          stores the sum back into 'b2'. 'b2' now
          contains $4A (74).

Notice that the last example's form is:

   <var>=<var><operator><operand>

Since programmers often use the above format, ACTION!
allows the following shorthand form to do the same thing:

   <var>==<operator><operand>

The operator must be either arithmetic or bit-wise. The
operand must be an arithmetic expression. The following are
some examples of this shorthand form:

```
b2==+1          is the same as      b2=b2+1
b2==-b1         is the same as      b2=b2-b1
b2==& $ØF       is the same as      b2=b2 & $ØF
b2==LSH (5+3)   is the same as      b2==b2 LSH (5+3)
```

This shorthand form can save you a lot of typing over the
long method and even generates better machine code in some
instances.

5.2 Structured Statements

If only simple statements were available, you'd be severely
limited in the number of things you could do on a computer:

The only way you could repeat a group of statements a
number of times would be to type them out in the same order
the right number of times. If you wanted to repeat a group
of ten statements ten times, you would end up typing in 1ØØ
statements!

You would not be able to execute a group of statements
conditionally, that is, only execute them if some specified
test is satisfied.

The purpose of structured statements is to solve these and
other problems. Structured statements as a whole are
divided into two separate categories: Conditional
Statements and Looping Statements. We will discuss each of
these categories separately.

5.2.1 Conditional Execution

Conditional execution allows you to test an expression and
execute various statements depending on the outcome of the
test. Since the expression controls conditional execution,
it is called a conditional expression.

Three ACTION! statements allow conditional execution:

               IF   WHILE   UNTIL

WHILE and UNTIL are looping statements and will be dealt
with later, but we will discuss IF immediately after the
rules governing conditional expressions.

5.2.1.1 Conditional Expressions

Since a conditional expression is involved in a test, there
are only two values it may have -- true or false. This does
not mean a conditional expression is a new type of
expression, however. In fact, a conditional expression is
simply either a relational or arithmetic expression. Only
the interpretation is different. The following table shows
what the conditional interpretation is, depending on which
type of expression it is:

Expression Type	Normal Result	Conditional Result
arithmetic	zero (Ø)	false
	non-zero	true
relational	false	false
	true	true

5.2.1.2 IF Statement

The IF statement in ACTION! is much like the 'if'
conditional statement in English. For example:

       "If I have $9 or more, I will buy the steak."

In ACTION! the same statement might be:

```
   BYTE money,
        steak=[9],
        fish=[8],
        chicken=[6],
        hotdog=[2]

   IF money>=9 THEN
       buy(steak,money)
```

NOTE: buy(steak,money) is a procedure call and will be
dealt with in section 6.1.3.

From the above example you can see that the basic form of
the IF statement is:

```
   IF <cond. exp.> THEN
       <statement(s)>
   FI
```

'FI' is not part of "Fe fi fo fum...", but 'IF' spelled backwards, and a keyword to the compiler showing the end of an IF statement. Since IF can work on a list of statements, we need 'FI' to terminate that list. Without this keyword the compiler would not know how many of the statements following the THEN went with the IF statement.

The above is only the basic format. The IF statement has two options, ELSE and ELSEIF. English also has these options, so we will use comparative examples:

> "If I have $9 or more I will buy the steak,
> otherwise I will buy the fish platter."

The ACTION! equivalent of this is:

```
IF money>=9 THEN
   buy(steak,money)
ELSE
   buy(fish,money)
FI
```

ELSEIF is somewhat different:

"If I have $9 or more I will buy the steak. If I have between $8 and $9 I will buy the fried fish. If I have between $6 and $8 I will buy the chicken. Otherwise I will buy the hotdog."

would be:

```
IF money>=9 THEN
   buy(steak,money)
ELSEIF money>=8 THEN
   buy(fish,money)
ELSEIF money>=6 THEN
   buy(chicken, money)
ELSE
   buy(hotdog,money)
FI
```

in ACTION!. Notice that we do not have to check for "money>=8 AND money<9", as in English. We can do this because the computer goes through the list sequentially from top to bottom. If any conditional case is true, the statements it controls are executed, and then the whole rest of the IF statement (including all following ELSEIFs and ELSEs) is skipped. So, if the computer does get to "money>=8", we already know that we have less than $9, because the preceding conditional tested for "money>=9" and found that condition false.

The ACTION! Programming Environment

The ELSEIF option is very useful when you want to test a variable for a number of different conditions, each requiring a different action.

NOTE: there is a relatively obscure bug related to the use of ELSEIF. In particular, statements similar to the form 'ELSEIF a(i) = Ø THEN ...' where 'a' is an ARRAY and 'i' is a CARD OR INT, or statements like 'ELSEIF p^ = Ø THEN' where 'p' is a Pointer, produce incorrect code. The best way around these problems seems to be to code something like:

```
              t = a(i)    ; t is an INTEGER
              ...
              ELSEIF t=Ø THEN ...
```

This works properly.

5.2.2 Null Statement

The null statement is used to do nothing. After showing you some statements that do something, and after stressing the necessity of statements that do something, why a statement that does nothing? There are actually a couple of good uses for a statement that does nothing: Timing Loops and ELSEIF cases.

Since we have not yet discussed loops at all, we will simply say that timing loops are used as a time delay (e. g., if you want to pause between printing lines to the the screen, you just use a timing loop to waste a few moments). You can find an example of a timing loop in section 5.2.4.1.

To illustrate the use of the null statement in ELSEIF cases, here is an example:

Scenario: You are writing a program that allows stock brokers to find out information about certain stocks, using the commands you have made available. The commands you are implementing are: BUY, DOWN?, FIND, QUIT, SELL, and UP?, but you have not implemented FIND yet. All you do is test the first letter of the entered command, so you have to test for B,D,F,Q,S, and U. But FIND is not done, so what do you do when they type 'F'? Easy, you do nothing, hoping that someday (when FIND is ready) you will do something. Here is how the program fragment might look:

```
     IF chr='B THEN
        dobuy()
     ELSEIF chr='D THEN
        dodown()
     ELSEIF chr='F THEN
        ;**** here is the null statement
```

```
    ELSEIF chr='Q THEN
        doquit()
    ELSEIF chr='S THEN
        dosell()
    ELSEIF chr='U THEN
        doup()
    ELSE
        doerror() ;**** no command match
    FI
```

All the 'do---'s are procedures to do the given command. If
you look at the case of "chr='F", you see that nothing is
done. That is the null statement. When FIND is ready, all
you need to do is put the 'dofind()' procedure in where the
null statement now is, and you will have it in the look-up
table and ready for use.

5.2.3 Loops

Loops are used to repeat things, specifically statements.
If, for some strange reason, you wanted to fill the screen
with stars (*) you could either send out each star with a
separate statement, or you could use a loop to do this for
you. All you need to do is tell the loop how many times you
want it to put out a single star, and it will do it (if you
use the proper statement format, of course).

There are two ways to tell a loop how many times you want
it to do something. You can give it an explicit number, or
you can give it a conditional expression and execute the
loop depending on the outcome of that expression. The FOR
statement uses the first method, and both WHILE and UNTIL
use the second.

What happens when you do not tell the loop how many times
it should execute? What happens when the conditional
expression never evaluates to a value that will stop the
loop? You get what is known as an 'Infinite Loop'. There is
only one way to get out of an infinite loop; you have to
push the <SYSTEM RESET> key.

ACTION! approaches loops in the following manner. There is
a basic loop, which, when used alone, is infinite. Then
there are some loop controlling statements (FOR, WHILE,
UNTIL) which limit the number of times this infinite loop
executes. We will follow the same pattern; first a
discussion about the basic loop structure, followed by an
in depth look at the loop controlling statements.

5.2.3.1 DO and OD

'DO' and 'OD' are used to mark the beginning and end,
respectively, of the basic loop . Everything between them
is considered to be part of that loop. As mentioned above,
a loop alone (i.e. without any loop controlling statement)
is an infinite loop, and you must force a break out of it.
Following is a program example to illustrate the DO - OD
loop. Do not worry about the 'PROC' and 'RETURN'
statements; they are just there so that the program will
compile and run properly, and will be discussed in full in
the procedures and functions chapter (6).

Example #1:

```
   PROC timestwo()

      CARD i=[Ø],j

      DO                ;start of DO - OD loop
      i==+1             ;add 1 to
      j=i*2             ;set 'j' equal to i*2
      PrintC(i)                 ;**** See the following
      Print(" times 2 equals ") ;PROGRAMMING NOTE for
      PrintCE(j)                ;an explanation
      OD                ;end of DO - OD loop
   RETURN
```

PROGRAMMING NOTE: the mixed case words (PrintC, Print,
PrintCE) you see in the example above are ACTION! library
functions and procedures. You may learn more about them
(although their jobs here are fairly obvious) in Part VI,
'The ACTION! Library'. You will see library routines used
throughout the rest of this chapter, so do not be alarmed;
they are only there because they do things that make the
examples more visually instructive.

```
Output #1:
      1 times 2 equals 2
      2 times 2 equals 4
      3 times 2 equals 6
      4 times 2 equals 8
      5 times 2 equals 1Ø
      6 times 2 equals 12
      7 times 2 equals 14
      8 times 2 equals 16
              :
              :
```

The colons at the end of the output shows that will go on
forever, or until you press the <SYSTEM RESET> key. On its
own, a DO - OD loop is more or less useless, but when used

in conjunction with the loop controlling statements FOR, WHILE, and UNTIL, it becomes one of the most useful statements available.

NOTE: Hitting the <BREAK> key would also get you out of the loop in example #1, because the loop is doing a lot of I/O. (<BREAK> only works when doing a lot of I/O. See Part IV, 'The ACTION! Compiler', for more information.)

Whenever you see '<DO - OD loop>' in the formats of the loop controlling statements, remember that it means a loop, and that in turn means a DO - OD pair surrounding the loop.

5.2.3.2 EXIT Statement

The EXIT statement is used to hop gracefully out of any loop. This statement will cause program execution to skip to the statement following the next 'OD'. Here is an example:

Example #1:

```
   PROC timestwo()

      CARD i=[Ø],j

      DO               ;start of DO - OD loop
      i==+1            ;add 1 to 'i'
      j=i*2            ;set 'j' equal to i*2
      PrintC(i)
      Print(" times 2 equals ")
      EXIT             ;Here is the EXIT statement
      PrintCE(j)
      OD               ;end of DO - OD loop
      ;**** execution continues here after 'EXIT'
      PrintE("End of Table")

   RETURN
```

Output #1:

```
      1 times 2 equals End of Table
```

As you can see in the output, the statement 'PrintCE(j)' is never executed. The EXIT statement forces execution to hop to the statement 'PrintE("End of Table")'. EXIT is not very useful when utilized alone, but if you use it in conjunction with an IF statement (i.e., make the EXIT into a conditional jump out of the loop), it can be very useful, as the program on the following page shows.

Example #2:

```
   PROC timestwo()

      CARD i=[Ø],j
      DO     ;start of DO - OD loop
      IF i=15 THEN
         EXIT  ;EXIT in an IF conditional
      FI
      i==+1
      j=i*2
      PrintC(i)
      Print(" times 2 equals ")
      PrintCE(j)
      OD     ;end of DO - OD loop
      ;**** execution continues here when i=15
      PrintE("End of Table")
   RETURN
```

Output #2:
```
      1 times 2 equals 2
      2 times 2 equals 4
      3 times 2 equals 6
      4 times 2 equals 8
      5 times 2 equals 1Ø
      6 times 2 equals 12
      7 times 2 equals 14
      8 times 2 equals 16
      9 times 2 equals 18
      1Ø times 2 equals 2Ø
      11 times 2 equals 22
      12 times 2 equals 24
      13 times 2 equals 26
      14 times 2 equals 28
      15 times 2 equals 3Ø
      End of Table
```

This usage turns an infinite loop block into a finite one.
EXIT can control the execution of a loop, but is not
considered a structured loop controlling statement because
it does not stand on its own; that is, it is only useful
when used in conjunction with the structured 'IF'
statement.

5.2.4 Loop Controls

ACTION! has three structured statements that control the
basic DO - OD loop:

                    1) FOR
                    2) WHILE
                    3) UNTIL

By saying that they "control the basic DO - OD loop", we
mean that they limit the number of times the infinite loop
executes, thus making it a finite loop. Controllable loops
are one of the devices that make computers very useful. If
someone told you to write "I will never throw spitwads
again" one thousand times, you would call that punishment,
but if you told the computer to do the same thing (with a
controlled loop, of course), it would think that the task
was easy and mundane.

Now we will take a look at each loop controlling statement
in depth, and then go into a property of all ACTION!
structured statements: nesting.

5.2.4.1 FOR Statement

The FOR statement is used to repeat a loop a given number
of times. It requires its own special variable, commonly
called a counter. In the examples the counter will be
called 'ctr' to remind you of this, but you could call it
anything you like. The format of the FOR element is:

FOR <counter>=<initial value> TO <final value> (STEP <inc>)

        <DO - OD loop>

where

        <counter>   is the variable used to keep
                            track of the number of times the
                            loop has executed
        <initial value>   is the starting value of the
                            counter
        <final value>   is the ending value of the counter
        <inc>   is the amount by which the
                            computer increments the counter
                            after every iteration
        <DO - OD loop>   is a DO - OD infinite loop

NOTE: The 'STEP <inc>' is optional

The ACTION! Programming Environment

Instead of trying to explain this to you using metaphors,
we will throw a few examples at you, because they more or
less speak for themselves. Following each is its output.

Example #1:

```
   PROC hithere()

      BYTE ctr ;counter used in FOR loop

      FOR ctr=1 TO S  ;this FOR loop has no 'STEP', so
         DO           ;an increment of 1 is assumed.
         PrintE("Hi there")
         OD
   RETURN
```

Output #1:
```
      Hi there
      Hi there
      Hi there
      Hi there
      Hi there
```

Example #2:

```
   PROC evens()

      BYTE ctr ;counter used in FOR loop

      FOR ctr=Ø TO 16 STEP 2 ;this FOR loop has a 'STEP'
         DO
         PrintB(ctr)
         Print(" ")
         OD
   RETURN
```

Output #2:
```
      Ø 2 4 6 8 1Ø 12 14 16
```

Look back at the format of the FOR statement. Notice that
we said nothing about using numeric variables as <initial
value>, <final value>, or <inc>. Doing this is legal, and
allows you to make FOR loops execute a variable number of
times.

If you change the value of the variables used as <initial
value>, <final value>, or <inc> in the loop itself, you
will not change the number of times the loop is executed.
This is true because <initial value>, <final value>, and
<inc> are set with a constant value when you enter the
loop. If you do use variables, the value used when setting
these is the value the variable had when the loop was first
entered.

If you change the value of <counter> in the loop, you will change the number of times the loop executes, because <counter> is a variable in the loop. It is variable in the loop because the FOR statement itself must change the value of <counter> every time it goes through the loop (FOR increments <counter> by the STEP value). Following is an example to illustrate changing <initial value>, <final value>, and <counter> in the FOR loop itself:

Example #3:

```
   PROC changeloop()

      BYTE ctr,
           start=[1],
           end=[5Ø]

      FOR ctr=start TO end
         DO
         start=1ØØ ;does not affect number of repetitions
         end=1Ø    ;does not affect number of repetitions
         PrintBE(ctr)
         ctr==*2   ;DOES affect number of repetitions
         OD
   RETURN
```

Output #3:
```
      1
      3
      7
      15
      31
```

Below is table to show what is going on each time through the loop. 'rep' tells which repetition the loop is on, 'inc ctr' shows the result of the FOR loop incrementing the value of the counter, 'Print' shows what is printed out to the screen, and 'ctr==*2' shows how this assignment statement changes the value of the counter.

rep	inc ctr	Print	ctr==*2
1	---	1	2
2	3	3	6
3	7	7	14
4	15	15	3Ø
5	31	31	62

After the fifth loop is through, the counter equals 62. This is greater than <final value> (5Ø), so the loop is exited after only 5 repetitions, not 5Ø. Manipulating the counter within its own loop can lead to very interesting results, some of which might even be useful.

The ACTION! Programming Environment

As promised in section 5.2.2, here is an example of a
timing loop:

```
BYTE ctr

FOR ctr=1 TO 25Ø
    DO
    ;**** here is the null statement
    OD
```

This is just used as a time-waster; something you will use
a lot if you are writing games or other programs which
involve careful timing.

PROGRAMMING NOTE: If you write a FOR loop which continues
to the limit of the data type of the counter (e.g., 'FOR
ctr=Ø TO 255' if ctr is a BYTE, or 'FOR ctr=Ø TO 65535' if
ctr is a CARD), the loop will be infinite because the
counter cannot be incremented to a value greater than the
given <final value>.

5.2.4.2 WHILE Statement

The WHILE statement (and the UNTIL statement, for that
matter) is used when you do not want to execute a loop a
predetermined number of times. WHILE allows you to keep
looping as long as a given conditional expression is
'true'. It has the form:

```
    WHILE <cond exp>
        <DO - OD loop>
```

where
<cond exp>   is the controlling conditional expression
<DO - OD loop> is a DO - OD infinite loop

Since the evaluation of the conditional expression is done
at the start of the loop, '<DO - OD loop>' might not be
executed at all. This is not the case with UNTIL, as you
will see later. Program examples using WHILE start on the
following page.

Example #1:

```
PROC factorials()
;**** This procedure will print out the factorials
;up to some specified number (the variable 'amt')

   CARD fact=[1],    ;the factorial of 'num'
        num=[1],     ;the counter
        amt=[6ØØØ]   ;the upper bound of testing

   Print("Factorials less than ")
   PrintC(amt)        ;prints the upper bound
   PrintE(":")        ;print a     and carriage return
   PutE()             ;prints a carriage return
   WHILE fact*num < amt    ;test next factorial
      DO                    ;start of WHILE loop
      fact==*num
      PrintC(num)          ;print the number
      Print(" factorial is ")
      PrintCE(fact)        ;print number's factorial
      num==+1              ;increment number
      OD                   ;end of WHILE loop
RETURN      ;end of PROC factorials
```

Output#1:

```
Factorials less than 6ØØØ:

1 factorial is 1
2 factorial is 2
3 factorial is 6
4 factorial is 24
5 factorial is 12Ø
6 factorial is 72Ø
7 factorial is 5Ø4Ø
```

PROGRAMMING NOTE: If you go over "Factorials less than 4ØØØØ", you will discover that the compiler does no overflow error checking, because you will see the output 'wrap around'; that is, you will get a number larger than the maximum a CARD allows (65535), and start at zero again. If you got up to, say, 66ØØØ, the output would show 66ØØØ-65536=464 because it went as high as it could go, and then wrapped around. The technical term for this kind of thing is 'overflow', and you can find out more about it in Part IV: The ACTIONI Compiler'.

The ACTION! Programming Environment

Example #2:

```
   PROC guesswhile()
   ;**** This procedure plays a guessing game with
   ;the user, using a WHILE loop to keep the game going

      BYTE num,          ;the number to guess
           guess=[200]   ;guess is initialized to an
                         ;impossible value.

      PrintE("Welcome to the guessing game. I am")
      PrintE("thinking of a number from 0 to 100")
      num=Rand(101) ;gets the number to guess
      WHILE guess<>num
         DO                    ;start of WHILE loop
         Print("What is your guess?  ")
         guess=InputB()        ;get user's guess
         IF guess<num THEN      ;guess too low
            PrintE("Too low, try again")
         ELSEIF guess>num THEN  ;guess too high
            PrintE("Too high, try again")
         ELSE                   ;guess just right
            PrintE("Congratulations!!!!")
            PrintE("You got it")
         FI                    ;end of guess testing
         OD                    ;end of WHILE loop
   RETURN                     ;end of PROC guesswhile
```

Output #2:

```
      Welcome to the guessing game. I am
      thinking of a number from 0 to 100
      What is your guess?  50
      Too low, try again
      What is your guess?  60
      Too high, try again
      What is your guess?  55
      Too low, try again
      What is your guess?  57
      Congratulations!!!!
      You got it
```

Notice how powerful manipulating conditionals like IF
within a loop can be. It allows the computer to take care
of multiple possible outcomes every time it goes through
the loop.

5.2.4.3 UNTIL Statement

In the last section we said that a WHILE loop could execute
zero times because its conditional expression was evaluated
before loop execution began. The form of the UNTIL
statement is such that this loop always executes at least
once. After you see the form you will probably understand
why this is so:

```
DO
<statement>
     :
     :
<statement>
UNTIL <cond exp>
OD
```

This looks like a common DO - OD loop until you get to the
statement just before the 'OD'. This UNTIL controls the
infinite loop using the outcome of the conditional
expression. If <cond exp> is 'true' then execution will
continue at the statement after the 'OD', otherwise it will
loop back up to the 'DO'. Notice that the UNTIL must be the
statement directly before the 'OD'. A program example
should clarify this somewhat:

Example:

```
PROC guessuntil()
;**** This procedure plays a guessing game with
;the user, using an UNTIL loop

   BYTE num,       ;the number to guess
       guess       ;the user's guess

   PrintE("Welcome to the guessing game. I am")
   PrintE("thinking of a number from 0 to 100")
   num=Rand(101) ;get the number to guess
   DO                    ;start of UNTIL loop
   Print("What is your guess?  ")
   guess=InputB()        ;get the user's guess
   IF guess<num THEN     ;guess too low
      PrintE("Too low, try again")
   ELSEIF guess>num THEN  ;guess too high
      PrintE("Too high, try again")
   ELSE                  ;guess just right
      PrintE("Congratulations!!!!")
   FI PrintE("You got it");end of guess testing
   UNTIL guess=num       ;loop control
   OD                    ;end of UNTIL loop
RETURN                   ;end of PROC guessuntil
```

The ACTION! Programming Environment

Output:

        Welcome to the guessing game. I am
        thinking of a number from Ø to 1ØØ
        What is your guess? 5Ø
        Too low, try again
        What is your guess? 6Ø
        Too high, try again
        What is your guess? 55
        Too low, try again
        What is your guess? 57
        Congratulations!!!!
        You got it

This is the same example as in the WHILE section, but this
time implemented using an UNTIL loop. Notice that 'guess'
is not initialized in the variable declaration, as it was
in the WHILE equivalent. We can do this because the
conditional expression 'guess=num' is not evaluated until
we have gotten a guess from the user. This is one of the
advantages of the UNTIL loop, and stems from the fact that
the controlling conditional expression is at the end of the
loop. WHILE requires evaluation of the conditional
expression at the beginning of the loop, and so requires
that 'guess' have a value.

5.2.5 Nesting Structured Statements

As mentioned in the overview of statements, structured
statements are made up of other statements, together with
some execution controlling information particular to a
given structured statement. The statements within the
structured statement may be either simple statements or
other structured statements. Putting one structured
statement inside of another is called nesting (because one
of them is 'nested' inside the other).

In sections 5.2.4.2 (WHILE) and 5.2.4.3 (UNTIL), you can
see examples of nesting an IF statement into WHILE and
UNTIL loops. This type of nesting is very straightforward,
and need not be discussed further. This section will deal
with multiple nesting of the same type of structured
statement (IFs inside IFs, FORs inside FORs, etc...).

When the IF statement is nested inside itself, confusion
might seem to arise when trying to figure out what ELSE
goes with which IF as you go deeper into the nested
statements. The compiler avoids any confusion by IF-FI
pairing. A FI is paired to the first preceding IF that does
not already have a FI paired to it.

For example:
```
  + IF <expA> THEN
  |   + IF <expB> THEN
  |   |     <statements>
  |   |   ELSEIF <expC> THEN ;**** ELSEIF of IF <expB>
  |   |     + IF <expD> THEN
  |   |     |     <statements>
  |   |     | ELSE                ;**** ELSE of IF <expD>
  |   |     |     <statements>
  |   |     + FI                  ;**** end of IF <expD>
  |   + FI               ;**** end of IF <expB>
  |   ELSEIF <expE> THEN   ;**** ELSEIF of IF <expA>
  |           <statements>
  | ELSE                   ;**** ELSE of IF <expA>
  + FI    <statements>     ;**** end of IF <expA>
```

The dashed lines show the IF-FI pairing; the comments show
which IF statement a particular FI or ELSEIF pertains to;
and the indentation shows a change of levels.

The following program example contains nested FORs. This
one even does something worthwhile; it prints out the
multiplication table up to ten times ten.

```
   PROC timestable()

      ;*** This procedure prints out the multiplication
      ;table up to 1Ø times 1Ø

      BYTE c1,          ;*counter for outer FOR loop
           c2           ;*counter for inner FOR loop

      FOR c1=1 TO 1Ø    ;outer loop control
         DO             ;*start of outer loop
         IF c1<1Ø THEN  ;*single digits need a space
           Print(" ")   ;before them in the first
         FI             ;column
         PrintB(c1)     ;*print 1st number in column
         FOR c2=2 TO 1Ø ;*inner loop control
            DO              ;*start of inner loop
            IF c1*c2 < 1Ø THEN ;*single digits need 3
               Print("   ")    ;spaces
            ELSEIF c1*c2 < 1ØØ THEN   ;*double digits
               Print("  ")     ;need 2 spaces
            ELSE               ;*triple digits need 1
               Print(" ")      ;space only
            FI                 ;*end of digit spacing
            PrintB(c1*c2)      ;*print the result
            OD                 ;*end inner loop
         PutE()         ;*put out a carriage return
         OD             ;*end of outer loop
   RETURN    ;*end of PROC timestables
```

The ACTION! Programming Environment

Output:

```
 1    2    3    4    5    6    7    8    9   1Ø
 2    4    6    8   1Ø   12   14   16   18   2Ø
 3    6    9   12   15   18   21   24   27   3Ø
 4    8   12   16   2Ø   24   28   32   36   4Ø
 5   1Ø   15   2Ø   25   3Ø   35   4Ø   45   5Ø
 6   12   18   24   3Ø   36   42   48   54   6Ø
 7   14   21   28   35   42   49   56   63   7Ø
 8   16   24   32   4Ø   48   56   64   72   8Ø
 9   18   27   36   45   54   63   72   81   9Ø
1Ø   2Ø   3Ø   4Ø   5Ø   6Ø   7Ø   8Ø   9Ø  1ØØ
```

As you can see from the above examples, nesting can be used to accomplish quite a bit, if you know what you are doing. Fortunately, "knowing what you are doing" does not take too much time, because the concept of nesting is universal to all structured statements. Once you understand it as applied to one statement, you can apply it to all of them.

Chapter 6: Procedures and Functions

Procedures and functions are used to make your ACTION!
program more readable and usable. Almost everything we do
is a procedure or function in some way or other. For
example, look at this table:

```
  Procedures            Functions
  ----------            ---------
  Washing the car       Balancing your checkbook
  Doing dishes          Looking up a phone number
  Driving to work
  Going to school
```

What makes these procedures and functions? Well, for each
there is

    1) a group of related actions done to accomplish the
        task
    2) an accepted order in which these actions are done

Drying the dishes before you wash them breaks the accepted
order, and taking off your left sock is not an action
related to "Doing the dishes". We know these things from
experience, and have lumped the proper group of actions
done in the proper order into a procedure; one which we
call "Doing the dishes".

In computer languages it is the same way. You make a group
of actions that accomplish a single, large task into a
procedure or function, which you then give a name. When you
want to execute this task, all you do is use the procedure
or function name (with some extras we will discuss later).
This is referred to as a procedure or function call. The
procedure or function must have already been defined, just
like in English. (e.g., you would not know what to do if
someone told you "readjust the widget" unless you already
knew the actions required to do this.)

Now, what is the difference between procedures and
functions? They both go through a series of ordered steps
to accomplish a task, so why two names for the same
construct? Because they are not exactly the same construct.
Functions have an added property; they do their task, and
then return a value.

In the table at this section's beginning we see "Balancing
your checkbook" given as a function example. Why? Well,
when you balance your checkbook you go through a series of
steps to bring your records up to date, and come up with a
(hopefully positive) number at the end. This number is

returned and can be used to do other things (like determine the size of your next check).

If we wanted to make "Doing the dishes" a function, we could change the statement to the question "Do the dishes need doing?", hoping that the person would answer the question, and then do the dishes if required. This would get the dishes done (like the procedure), but also return a value (whether the dishes needed doing in the first place), and thus make it a function.

NOTE: Throughout the rest of this manual we will use the word "routine", instead of saying "procedure or function". Doing this makes the concepts easier to follow. When you see "procedure" or "function", it means the concept or idea being discussed is specific to that class of routines and not applicable to both classes.

6.1 PROCedures

Procedures are used to group some statements which accomplish a task into a named block that can be called on to do this task. To utilize procedures in ACTION!, you must learn how to do two things:

        1) declare procedures
        2) call procedures

The following three sections will show you how to do the above and give some examples to let you see procedures in ACTION! (small pun intended).

6.1.1 PROC Declaration

The ACTION! keyword 'PROC' is used to denote the start of a PROCedure declaration. PROCedure construction looks quite like a group of statements with a name and some other information at the beginning, and a funny RETURN statement at the end. Below is a diagram of the construction.

```
   PROC <identifier>{=<addr>}({<parameter list>})
      {<variable decl>}
      {<statement list>}
   RETURN
```

where

PROC   is the keyword denoting a procedure
                  declaration
<identifier>   is the name of the procedure
<addr>   optionally specifies the starting ad-
                  dress of the procedure (See 9.3)
is the list of parameters required by
                  the procedure (see section 6.4 for an
                  explanation of parameters)
<variable decl>   is the list of variables declared local
                  to this procedure (see 3.4.1 for variable
                  declaration and 6.3 for scope of
                  variables)
<statement list> is the list of statements in the
                  procedure
RETURN   denotes the end of the procedure (see
                  next section)

The ACTION! Programming Environment

NOTE: <parameter list>, <variable decl>, and <statement list> are all optional. You will probably use at least some of them, but the following would be a valid procedure declaration:

```
PROC nothing()      ;the parentheses ARE required
RETURN
```

It does nothing, but this type of "empty" procedure is useful when you are writing a program made up of many procedures. If, for example, you have written a program that calls a procedure named "dotest", but you have not yet written "dotest", you could make it an empty procedure so you could test the rest of the program without getting an "Undeclared Variable" error.

Do not worry about '<parameter list>' and 'RETURN' in the format, because they will be discussed later. The rest should look somewhat familiar, so we will give an example:

```
PROC guessuntil()
;**** This procedure plays a guessing game with
;the user, using an UNTIL loop

   BYTE num,        ;the number to guess
        guess       ;the user's guess

   PrintE("Welcome to the guessing game. I am")
   PrintE("thinking of a number from Ø to 1ØØ")
   num=Rand(1Ø1) ;get the number to guess
   DO                   ;start of UNTIL loop
   Print("What is your guess?  ")
   guess=InputB()       ;get the user's guess
   IF guess<num THEN    ;guess too low
      PrintE("Too low, try again")
   ELSEIF guess>num THEN ;guess too high
      PrintE("Too high, try again")
   ELSE                 ;guess just right
      PrintE("Congratulations!!!!")
      PrintE("You got it")
   FI                   ;end of guess testing
   UNTIL guess=num      ;loop control
   OD                   ;end of UNTIL loop
RETURN     ;end of PROC guessuntil
```

This is just the program example from section 5.2.4.3, but now you understand why the PROC statement and the variable declaration section are there. As mentioned in the introduction, an ACTION! program requires a procedure declaration or a function declaration to be compilable. The above example has a procedure declaration, so it is a valid ACTION! program and, as such, may be compiled and run. Its output is the same as that given in the UNTIL section,

namely:

```
Welcome to the guessing game. I am
thinking of a number from Ø to 1ØØ
What is your guess?  5Ø
Too low, try again
What is your guess?  6Ø
Too high, try again
What is your guess?  55
Too low, try again
What is your guess?  57
Congratulations!!!!
You got it
```

If you look back at the above example, you will see
'RETURN' as the last statement. We will now cover why it is
there.

6.1.2 RETURN

RETURN is used to tell the compiler to leave the procedure
and return control to whatever called the procedure. If
your program calls a procedure, execution will continue
with the statement after the procedure call. If you are
compiling a single procedure (or a one procedure program),
control will be returned to the ACTION! monitor.

WARNING: the compiler cannot detect a missing RETURN.
Strange and disastrous things can happen if you leave out
RETURN. This also goes for RETURNs at the end of functions
as well.

There can be more than one RETURN in a procedure. For
example, if your procedure has an IF statement with lots of
ELSEIFs, you might want to RETURN after one or more of the
ELSEIF cases. The example on the following page illustrates
this possibility.

The ACTION! Programming Environment

```
PROC testcommand ()
;**** This procedure tests a command to see if it is
;valid. Valid commands are Ø, 1, 2, and 3. If the
;command is none of these, an error message is
;printed, and control is returned to whatever called
;this procedure

   BYTE cmd

   Print("Command>> ")
   cmd=InputB()
   IF cmd>3 THEN
      PrintE("Command Input ERROR")
      RETURN ; get out before command tests
   ELSEIF cmd=Ø THEN
      <statementØ>
   ELSEIF cmd=1 THEN
      <statement1>
   ELSEIF cmd=2 THEN
      <statement2>
   ELSEIF cmd=3 THEN
      <statement3>
   FI
RETURN
```

Note the 'RETURN' after the first condition, which tests
for illegal input. You do not want to go through all the
command tests if the command is not a valid one, so you
just print your error message and hop out of the procedure
with a RETURN. Voila!

6.1.3 Calling Procedures

You have already seen some procedure calls, although you
probably do not know it. Almost every time we used a
library routine in an example, we were making a procedure
call. The format is simple enough:

        <identifier>({<parameter list>})

where

        <identifier>   is the name of the procedure you
                             want to call
        <parameter list>   contains the values you want to
                             send to the procedure as
                             parameters

Here are a couple of examples (do not worry about the
parameters now, a whole section is devoted to them later):

```
PrintE("Welcome to Joe's Deli, the only")
PrintE("computerized deli in the world.")

factorials()

guessuntil()

BYTE z
CARD add
signoff(add,z)
```

Of course you must already have declared the procedures
'factorials', 'guessuntil', and 'signoff' before using them
here. 'PrintE' is a library procedure, so it is not
declared by you but is declared in the ACTION! library.
Notice that the parentheses are required even when the
procedures have no parameters. When a procedure you call
has parameters, the call must have no more parameters than
the procedure declaration (but it may have fewer). See
section 6.4 for a discussion of parameters.

The ACTION! Programming Environment

6.2 FUNCtions

As mentioned in the overview of procedures and functions,
the fundamental difference between the two is that
functions return a value. This makes the way in which they
are declared and called somewhat different from procedure
declarations and calls. Since functions return a numeric
value, they must be used where a number is valid (e.g., in
arithmetic expressions).

6.2.1 FUNC Declaration

Declaring a function is similar to declaring a procedure,
except that you must be able to show both what type of
number the function returns (BYTE, CARD, or INT) and what
that number is. The format is:

```
   <type> FUNC <identifier>{=<addr>}({<parameter list>})
     {<variable decl>}
     {<statement list>}
   RETURN (<arith exp>)
```

where

<type>   is the fundamental data type of the value
                  the function returns
FUNC   is the keyword denoting a function
                  declaration
<identifier>   is the name of the function
<addr>   optionally specifies the starting address
                  of the function (see 9.3)
is the list of parameters required by the
                  function (see section 6.4 for an
                  explanation of parameters)
<variable decl>   is the list of variables declared local
                  to this function (see section 3.4.1 for
                  variable declaration, and 6.3 for scope
                  of variables)
<statement list> is the list of statements in the function
RETURN   denotes the end of the function
<arith exp>   is the value you wish returned from the
                  function

As in procedure declarations, <parameter list>, <variable
decl>, and <statement list> are all optional. In the case
of procedures, leaving them out was useful only in one
instance.

In functions, doing this sort of thing has another (more worthwhile) use, as the following example shows:

Example #1:

```
CARD FUNC square(CARD x)
RETURN (x*x)
```

This function takes a CARD number and returns its square. Do not worry about the parameter list, as we will discuss it a little later. It was mentioned above that the value returned is in the form of an arithmetic expression. In example 1, you can see this being done in "(x*x)".

In the following example, the arithmetic expression used to return a value is simply a variable name.

Example #2:

```
BYTE FUNC getcommand()
;**** This function reads in a command number, and
;then passes it out if it is 1 through 7 inclusive.
;Otherwise, the function will re-prompt the user.

   BYTE command,   ;this variable holds the command
        error      ;set to 1 if an error is found

   DO
   Print("COMMAND> ")
   command=InputB()
   IF command<1 OR command>7 THEN ;invalid command
      error=1
      PrintE("Command Error: Only 1-7 valid.")
   ELSE            ;valid command
      error=Ø
   FI
   UNTIL error=Ø   ;exit loop if command is valid
   OD
RETURN (command)
```

NOTE: The parentheses around <arith exp> are always required in the RETURN statement.

The above is a simple example: Functions can be used to do quite complicated operations, but even the most convoluted functions must follow the format outlined in this section.

6.2.2 RETURN

As you probably noticed in the format of the FUNCtion declaration, the RETURN is not used in the same way as in PROCedure declarations. In functions it is followed by (<arith exp>). This feature allows a function to return a value. If you tried to put (<arith exp>) after the RETURN in a procedure declaration, you would get an error, because procedures cannot return a value.

Although there are dissimilarities between RETURNs in functions and procedures, there is one convenient similarity: you may have more than one RETURN in both procedures and functions. The following example shows usage of multiple RETURNs in a function:

     Scenario: Example #1 in the function declaration
     section (6.2.1) returned the square of a CARD, but it
     did no checking for overflow. If you squared 256 you
     would get 65536, 1 greater than the maximum CARD
     value allowed. There are two ways to fix this
     problem:

          1) Require that the number being squared be of
             BYTE type, thus making it impossible to enter a
             number greater than 255.
          2) Check for overflow in the function itself

     The following example illustrates the second method:

```
  CARD FUNC square(CARD x)
  ;**** This function tests 'x' for overflow, and
  ;returns its square if valid. IF invalid, the
  ;function prints an error message and returns Ø.

    IF x>255 THEN ;number would cause overflow
      PrintE("Number too big")
      RETURN (Ø) ;return a zero
    FI
RETURN (x*x) ;return 'x' squared
```

See how easy it is? The use of multiple RETURNS can come in very handy when you are testing a lot of different conditions, each requiring that a different value be returned.

NOTE: As mentioned in section 6.1.2, the compiler cannot tell if you leave out a RETURN, so you must make sure you have one.

6.2.3 Calling Functions

You have already seen two examples of function calls. They
can be found in section 5.2.4.2 (WHILE), example #2, and
section 5.2.4.3 (UNTIL), example #1. If you look at those
programs, you will see the lines:

```
        num=Rand(1Ø1)
        guess=InputB()
```

The first is an example of calling a function that requires
parameters, and the second an example of calling one
without parameters. Both 'Rand' and InputB' are library
functions. 'Rand' returns a random number between Ø and the
number you give it (in the above case, 1Ø1) minus one.
'InputB' reads a byte value from the default device (the
screen editor). Notice that both of them return a value.
Because this value must go somewhere or be used somehow,
function calls must be used in arithmetic expressions. In
the above two cases, the arithmetic expressions consist of
the function call only and are used in assignment
statements (a valid use of arithmetic expressions).

Function calls can be used in any arithmetic expression,
with one exception:

> Functions   calls   may   NOT   be   used   in   an
> arithmetic expression when that expression is used
> as   a   parameter   in   a   routine   call   or
> declaration.

> Example:
> ```
>       x=square(2*Rand(5Ø)) ;INVALID
>                 ^~~~~~~^
> ```

Here are some examples of valid function calls:

```
        x=5*Rand(2Ø1)
        c=square(x)-1ØØ/x
        IF ptr<>Peek($8ØØØ)
        chr=uppercase(chr)
```

'Peek' and 'Rand' are library functions, so they need not
be declared by you, but 'square' and 'uppercase' are user
written functions, and so they must be declared before they
are called here.

PROGRAMMING NOTE: although it is not recommended, you can
call functions as though they were procedures. If you do
this, the value returned is ignored.

The ACTION! Programming Environment

6.3 Scope of Variables

The term "Scope of a Variable" is used to express the range
of a variable's legitimacy. To help you understand what
this means, let us apply the concept of "Scope" to a more
familiar situation: the English language.

Below is a table of British English words, followed by
their American English equivalent:

British	American
BONNET	HOOD
LORRY	TRUCK
LIFT	ELEVATOR
FAG	CIGARETTE

Each pair of words means the same, but the words' scopes
are different. "Bonnet" (when used to mean the moveable
cover over an auto's engine) is legitimate only when used
in countries that speak the King's English. "Hood", on the
other hand, is valid only in countries that speak American
English. Hence they have ranges of legitimacy, or Scope.
The words in the left column could be considered "global"
to British English in the sense that any average Brit would
understand what was meant by each word, and the words in
the right column could be considered "global" to American
English because everyone who speaks American English would
associate each word with its intended meaning.

Enough of global scope; now we need to talk about "local"
scope. Scope is local if it is a specific subset of some
global scope. For example, the word "neat" has many
different local scopes within the "global" American English
language:

    1) "Wow, that movie was NEAT!"
    2) "Gertrude keeps the NEATest house I've ever seen."
    3) "Bartender, I'll have my scotch NEAT."

In different situations "neat" can mean different things
(i.e., the meaning is local to the situation), and these
meanings do not overlap.

Variables in ACTION! also have an associated scope. A
variable's scope determines where it may and may not be
used just as, in the above analogy, a word's scope
determines where it may and may not be used.

The following program is a concrete example of variable
scope:

Example #1:

```
    MODULE      ;we are going to declare some variables
                ;as global

    CARD numgames=[Ø], ;number of games played
         goal=[1Ø],    ;number of guesses to beat
         beatgoal=[Ø]  ;number of times you have beaten goal

    PROC intro()
    ;**** This procedure puts the lead-in to the game on
    ;the screen.

       CARD ctr

       PrintE("Welcome to the guessing game. I am")
       PrintE("thinking of a number from Ø to 1ØØ.")
       PrintE("All you have to do is type in your")
       PrintE("guess when I ask you to.")
       PutE()
       PrintE("I will keep track of how many games")
       PrintE("you have played, and tell you how")
       PrintE("many times you have guessed the number")
       PrintE("in fewer tries than your goal, but")
       PrintE("first you have to give me your goal.")
       PutE()
       Print(" Type your goal here --> ")
       goal=Input()
       FOR ctr=Ø to 25ØØ   ;a delay loop, to give the
         DO                ;sense of real-time to the
         OD                ;player.
       Put($7D)            ;clear the screen
    RETURN   ;end of PROC intro


    PROC tally()
    ;**** This procedure prints out the current tally

       Print("You have played ")
       PrintC(numgames)
       PrintE(" games,")
       Print("and in ")
       PrintC(beatgoal)
       PrintE(" of those you")
       PrintE("have beaten your goal of")
       PrintC(goal)
       PrintE(" guesses.")
       PutE()
    RETURN   ;end of PROC tally
```

The ACTION! Programming Environment

```
    PROC playgame()

    CARD numguesses,   ;the number of guesses
         ctr           ;counter used in delay loop

    BYTE num,          ;the number to guess
         guess         ;the user's guess

    PrintE("I am picking my number...")
    FOR ctr=Ø TO 45ØØ   ;delay used to make the user
       DO               ;think the computer is picking
       OD               ;a number
    PutE()
    PrintE(" O.K., here we go!")
    PutE()
    num=Rand(1Ø1)       ;get the number to guess
    numguesses=Ø        ;set number of guesses to Ø
    DO                  ;start of UNTIL loop
    Print("What is your guess?  ")
    guess=InputB()         ;get the user's guess
    numguesses==+1         ;add 1 to number of guesses
    IF guess<num THEN      ;guess too low
       PrintE("Too low, try again")
    ELSEIF guess>num THEN  ;guess too high
       PrintE("Too high, try again")
    ELSE                   ;guess just right
       PrintE("Congratulations 1!!1")
       Print("You got it in ")
       PrintCE(numguesses)
       IF numguesses<goal THEN
          beatgoal==+1
       FI
    FI                     ;end of guess testing
    UNTIL guess=num        ;loop control
    OD                     ;end of UNTIL loop
    RETURN     ;end of PROC playgame

    BYTE FUNC stop()
    ;**** This function finds out if the player wants
    ;to play another game.

       BYTE again

       PrintE("Do you want to play")
       Print("another game? (Y or N)   ")
       again=GetD(1) ;get player's response
                     ;from K: to avoid getting a RETURN as
                     ;the first guess of the next game.
       PutE()
       IF again='N OR again='n THEN   ;does not
          RETURN (1)                  ;want to play
       FI
    RETURN (Ø)   ;end of FUNC stop
```

100

```
PROC main()

    Close(1)         ;just for safety's sake
    Open(1,"K:",4,Ø) ;open K: to read only
    intro()          ;print Out the introduction
    DO
    numgames==+1     ;increment total number of games
    playgame()       ;play the game once
    tally()          ;show tally of games thus far
    UNTIL stop()     ;does not want to play anymore
    OD
    PutE()
    PrintE("Come play again soon")
    close(1)         ;close K,
  RETURN        ;end of PROC main
```

The following table shows how this program uses variables.
It gives the variable name, its scope, its availability and
use in each routine:

      KEY:

          A = variable Available for use in routine
          U = variable Used in routine

VARIABLE NAME	SCOPE	PROC playgame	PROC intro	PROC tally	FUNC stop	PROC main
numgames	global	A	A	A U	A	A U
goal	global	A U	AU	A U	A	A
beatgoal	global	A U	A	A U	A	A
numguesses	local	A U				
num	local	A U				
guess	local	A U				
ctr	local	A U				
again	local				A U	
ctr	local		A U			

You can see that that the global variables are available
for use in every one of the routines, whereas the local
variables are available only in the routine in which they
are declared. Notice that there are two local variables
called 'ctr', one in PROC playgame, and the other in PROC
intro. Although they have the same name, these two
variables are not the same, just as 'neat' meaning "clean"
and 'neat' meaning "undiluted" were not the same earlier.
The two 'ctr's have different local scopes (because they
are declared in two different procedures).

6.4 Parameters

Parameters allow you to pass values into a routine. You may
wonder why this is necessary, since you could use global
variables for passing values into and between routines.
Well, there are two reasons that parameters exist:

        1) They make your routines capable of multi-
            purpose use.
        2) They allow you to manipulate variable values
            within a routine without changing the value
            of any global variable.

we will discuss each of these advantages separately,
following the above order; but first we should give the
format of a parameter list, for those of you who already
know all about parameters.

** Parameters in PROC or FUNC declarations:

        ({<variable decl>}|:,<variable decl>:|)

    where

        <variable decl> is a variable declaration, except
                         that it may not contain the
                         '=<addr> or r<const>]' option

    Examples:

        PROC test(BYTE chr,num,i, CARD x,y)
        INT FUNC docommand(INT cmd, CARD ptr, BYTE offset)
        CARD FUNC square(BYTE x)
        PROC jump()

** Parameters in PROC or FUNC calls:

        ({<arith exp>}|:,<arith exp>:|)

    where
        <arith exp>   is an arithmetic expression

    Examples:

        test(cat,dog,ctr,25ØØ,$8DØØ)
        sqr=square(num)
        jump()
        x=docommand(temp,var,'A)

NOTE: A routine may have up to 8 parameters. Use any and
you will get a compiler error.

We need to do some explaining now. The following example
will show you how to use parameters, and clarify the first
of the two advantages to using parameters.

The following function checks to see if the BYTE variable
'chr' is a lowercase letter. If it is, the function will
return the uppercase of it. Otherwise the function will
simply return 'chr'. Notice that we do not declare 'chr'
anywhere. We will discuss where it should be declared after
the example.

```
    BYTE FUNC lowertoupper()

       IF chr>='a AND chr<='z THEN  ;$2Ø is the offset
          RETURN (chr-$2Ø)          ;between lower and
       FI                           ;upper case in the
                                    ;ATASCII set
    RETURN (chr)
```

Now we must decide where to declare 'chr'. We already know
that we could declare it global, or just local to
'lowertoupper'. If we declare it locally, how will we give
it a value? There seems to be no use to having it local,
because then the function itself would have to give the
variable a value, and that is not what we want the function
to do. We want it to be able to call 'lowertoupper' in a
form similar to

                     chr=lowertoupper()

and have the function test 'chr' and make it uppercase if
necessary. So we will not declare it local. How about
declaring it global? That would do what we wanted, because
now the 'chr' in the function call and the 'chr' in the
function itself would be the same global variable. There is
only one drawback to declaring 'chr' as a global variable:
every time we wanted to use 'lowertoupper', we would get
the uppercase of 'chr'. If we want to uppercase the
variable 'cat', we would have to do the following:

                     chr=cat
                     chr=lowertoupper()
                     cat=chr

This could get very tiresome if you wanted to uppercase a
lot of different variables. Also, if you wanted to use
'lowertoupper' in another program, you would have to
declare a global variable 'chr' there too.

The ACTION! Programming Environment

What if we declared 'chr' as a parameter to the function?
"HOW...?" you ask. Here is how:

```
BYTE FUNC lowertoupper(BYTE chr)   ;<- the parameter
                                   ;declaration
   IF chr>='a AND chr<='z THEN
      RETURN (chr-$2Ø)
   FI
RETURN (chr)
```

"But now how do we call it now?" Easy. All you have to do
is give it the variable you want tested as a parameter.

Examples:

```
          chr=lowertoupper(chr)
          cat=lowertoupper(cat)
          var=lowertoupper('a)
```

Making 'chr' a parameter to the function allows you to use
it for testing any variable in any program, because
'lowertoupper' now stands on its own. It uses no variables
declared elsewhere (i.e., global variables), and yet you
can give it a variable to test. We have overcome the
pitfalls of declaring 'chr' either locally or globally. Tah
dah! This is what we meant by multipurpose".

The second advantage to using parameters is more difficult
to illustrate, but we are going to make it as clear as
possible, again by using an example. The following
procedure takes two CARD type numbers, divides the first by
the second, and prints out the result:

```
PROC division(CARD num,div)

   num==/div       ;changes num to num/div
   PrintC(num)     ;print out num
RETURN
```

And now to use the 'division' procedure in a program:

Example #1:

```
   PROC main()

       CARD ctr,
           number=[713]

       FOR ctr=1 TO 10
          DO
          PrintC(number)
          Print("/")
          PrintC(ctr)
          Print(" = ")
          division(number,ctr)
          PutE()
          OD
   RETURN
```

Output #1:

```
        713/1 = 713
        713/2 = 356
        713/3 = 237
        713/4 = 178
        713/5 = 142
        713/6 = 118
        713/7 = 101
        713/8 = 89
        713/9 = 79
        713/10 = 71
```

Notice that 'number' remains constant, although 'num' changes. The value of 'number' is passed into 'num' when the procedure is called, but the value of 'num' is not passed back into 'number' when the procedure is exited. If the value of 'num' were passed back into 'number', the output would be:

```
        713/1 = 713
        713/2 = 356
        356/3 = 118
        118/4 = 29
        29/5 = 5
        5/6 = 0
        0/7 = 0
        0/8 = 0
        0/9 = 0
        0/10 = 0
```

The flow of information through parameters is one-way. Information can be sent to a routine through parameters,

but information generally may not be sent out using parameters. If you want to send a single value back from a routine, make that routine a function, and then you can send it back in the function RETURN statement. If you want to send out more things, you can use global variables or you can pass pointers as parameters (see 9.5).

A Note On Parameter Pairing:
    When you call a routine that has parameters, the first parameter you give in the call will go into the first variable in the list of parameters in the routine declaration, the second will go into the second, and so on.... You can pass fewer parameters than the routine requires, but no more. For example, if there are 5 parameters in the declaration, you could pass the routine Ø to 5 parameters. This allows you to write routines that require a variable number of parameters, depending on the job it must do. HINT: if you do this, the first parameter should probably be the number of parameters being passed.

A Note On Type Compatibility:
    If the value you pass as a parameter and the value expected by the routine are of different data types, you will not get a compiler error because the ACTION! compiler insures parameter type compatibility. For example, if you pass a CARD when the procedure wants a BYTE, the LSB of the CARD will be put into the BYTE variable, and the procedure will carry on as though you had passed it a BYTE (see Part IV for more info).

A Note On Parameter Variable Types:
    All of the following are valid as parameters:

    1) Fundamental Data Type variables
    2) Array, Pointer, and Record References
    3) Array, Pointer, and Record Names

    In the third case, the names are used as pointers to the first element, the value, or the first field in the named variable.

Chapter 7: Compiler Directives

Compiler directives are different from the standard
language commands in that they are executed at compile-time
rather than run-time. A language command, such as an
assignment statement (see section 5.1.2) is evaluated after
you tell the ACTION! Monitor to RUN your program, when your
program has control over what is being done. A compiler
directive is evaluated when you tell the monitor to COMPILE
your program, so the compiler, not your program, has
control. The ramifications of this will soon become
apparent.

7.1 DEFINE

The DEFINE directive is very similar to the editor's
substitution (<CTRL><SHIFT>S) command, except that it does
the substitution at compile time. To clarify this, we first
need to show the format:

    DEFINE <ident>=<str const>{,<ident>=<str const>}

where

        <ident>   is a valid identifier
        <str const> is a valid ACTION! string constant
                     (That is, with surrounding double
                     quotes)

DEFINEs are not really used in generating any object code
when the program is compiled, but are used to clarify
ACTION! source programs. The compiler substitutes <str
const> for <ident> every time <ident> is used in a program.
For example, when you compile a program with the line

    DEFINE size = "256"

in it, the compiler will replace every occurrence of 'size'
with '256'. This allows for some interesting options (and
problems if misused!). Since DEFINE will replace any
string, you can change the keywords themselves! If you do
not like the keyword CARD, you could change it to, say,
FROG with this command:

    DEFINE FROG = "CARD"

Now whenever you compile the program, every time the
compiler sees 'FROG', it will think to itself, "Oh, he
really means CARD, so I will just put that in instead."

The ACTION! Programming Environment

Here are some more examples to let you become thoroughly familiar with the form:

```
        DEFINE liston = "SET $49A=1"
        DEFINE begin = "DO", end = "OD"
        DEFINE one = "1"
```

NOTE: Do not forget that the string constant must have double quotes around it (see section 3.2).

To better show you what DEFINE does and does not do, here is a table showing the effects of a DEFINE on different program parts.

statement	comments
DEFINE four = " 4 "	the directive
PrintBE(four)	prints '4' with EOL
; four score and	converts 'four' to '4'
; four-score and	does not alter 'four-score'
PrintE("four score")	does not replace inside quotes

7.2 INCLUDE

The INCLUDE directive allows you to include other programs into the program being compiled. Suppose you have a program named 'IOSTUFF.ACT' that does input/output functions and you want to use the I/O routines it offers in some other program you are writing now. All you need to do is put the following command in the program you are writing:

```
        INCLUDE "D1:IOSTUFF.ACT"
```

NOTE: The file specifier must have double quotes around it.

The above statement must come before you use any of the I/O routines in the file 'IOSTUFF.ACT'. Note that this example assumes that the diskette with 'IOSTUFF.ACT' on it is in disk drive #1. If you do not specify a device with your file name, the compiler assumes the device is "D1:". You can INCLUDE files from any readable device (i.e., "P:" is not valid). Here are some more examples:

```
        INCLUDE "D2:IOLIB.ACT"
        INCLUDE "PROG1.DAT"
        INCLUDE "C:"
```

NOTE: Most operating systems require that the file specifiers be in uppercase.

A useful feature of the INCLUDE command is that you can have an INCLUDE in a program which you are already INCLUDEing (i. e., it can he nested). ACTION! allows you to nest it to a maximum of 6 levels, but peripheral devices and the operating system have other limits. When the OS limits are ignored, error # 161 (too many files open) occurs. The cassette limit is 1 INCLUDE, and the disk drive limit is 3 INCLUDEs. If no program is currently in the ACTION! editor buffer, then the maximum number of levels of INCLUDE commands is reduced by one.

7.3 SET

The SET directive is used to modify the computer's RAM (Random Access Memory). SET pokes a new value into a specified memory location at compile time. In most cases, this command is used for changing Editor and Compiler options from a user program, but it can be used to modify user, operating system, and hardware variables as well. The format of the SET command is:

SET <address> = <value>

NOTE: <address> and <value> must be compiler constants.

The result of the SET statement is to set memory location <address> to <value>. If <value> is greater than 255, then memory locations <address> and <address>+1 are assigned <value>. This occurs because 255 ($FF) is the biggest decimal number that can fit into one byte, so any number greater than this requires two bytes for storage.

Examples:
```
        SET $6ØØ=64      ;sets address $6ØØ equal to 64
        SET max=16       ;sets max=16
        SET 1ØØØØ=$FFFF   ;sets 1ØØØØ and 1ØØØ1 to $FFFF
        SET $CFØØ=cat     ;sets $CFØØ and $CFØ1=@cat

        DEFINE add="$7ØØØ"
        SET add=$42
```

The last example shows a DEFINEd numeric constant used in a SET statement. Since DEFINEs are constants at compile time, they are valid in the SET directive. Just make sure you DEFINE the constant before you use it in a SET statement.

NOTES: do not confuse the compile-time effect of SET with the similar run-time effect of Poke and PokeC.

Using a code offset greater than $7FFF (i.e., a negative offset, if you consider it to be of type INT) causes the compiler to generate improper code, which becomes obvious

especially when using the runtime library. To avoid such problems use the relocation program in Appendix J.

7.4 MODULE

MODULE is a very simple directive. Its form is:

        MODULE

It simply tells the compiler that you wish to declare some more global variables. It is useful when you have written a large program in sections, each with its own global variables. If you say MODULE at the beginning of each section, then the compiler will add all the global variables to the global variable table.

A program need not have a MODULE directive, because the compiler assumes one MODULE directive at the beginning of the program, whether you put it there or not.

The declaration of global variables must come either immediately after a 'MODULE', or at the very beginning of the program (which is really right after the 'MODULE' assumed by the compiler).

Chapter 8: Extended Data Types

The extended data types make the ACTION! language more
flexible than many others available on the ATARI. Just as
the structured statements manipulate groups of simple
statements thereby extending the capabilities of the
ACTION! language, the extended types manipulate groups of
fundamental type variables and extend the language
capabilities even more.

The three extended data types in ACTION! are:

        1) Pointers
        2) Arrays
        3) Records

We will discuss each separately, following the order of the
above list.

8.1 POINTERs

Pointer. Sounds like the thing the weatherman uses to show
us a place on his map. Well, it is. In the context of
ACTION!, "pointer" means something very similar.

Pointers contain a memory address, and so point to a memory
location. You can change the value of a pointer and make it
point to a new place, just like moving the weatherman's
pointer to another place on the map. The big difference is
that he points to cities or states, whereas ACTION!
pointers can point to BYTE, CARD, or INT values.

Somehow we have to let the compiler know what type of value
we want a given pointer to point to. The declaration
section will show you how to do this.

After we've gone over the method used to declare a pointer,
we will show you how it can be used. This is done in the
manipulation section through the use of program examples.

8.1.1 Pointer Declaration

The format used to declare a pointer looks quite similar to
the format of fundamental data type variable declarations,
except   that   we   tell   the   compiler   that   the

variable is a pointer, and not just a fundamental data type:

        <type> POINTER <ident>{=<addr>}|:,<ident>{=<addr>}:|
where

        <type>   is the fundamental type of the
                        information the pointer points to.
        POINTER   is the keyword used to show that the
                        variables declared are pointers.
        <ident>   is the name of the pointer variable
        <addr>   tells where in memory you want the
                        pointer to point to initially. It must
                        be a compiler constant.

Because a pointer variable actually contains an address, it must be able to take on values ranging from 0 to 65535 ($0 to $FFFF), since an ATARI with 64k of memory has that many separate memory locations. Pointers are stored as a two-byte unsigned numbers (in LSB, MSB order) to allow this range. That means that they are stored as CARDs, except that they can be interpreted as addresses.

Since the use of pointers is dealt with in the next section, we will just give some sample pointer declarations, instead of whole program examples:

        BYTE POINTER ptr   ;declares ptr as a pointer
                                ;to a BYTE value

        CARD POINTER cpl   ;declares cpl as a pointer
                                ;to a CARD

        INT POINTER ip=$8000   ;declares ip as a pointer ;to
                                an INT, and points it ;to
                                memory location $8000

8.1.2 Pointer Manipulation

Pointers can be used to manipulate a variety of things in ACTION! for the simple reason that they can easily be made to point to different memory locations. This makes cataloging and tabulating information very easy.

The program on the following page is just a simple example to give you an idea of what a pointer actually does. It will introduce the '^' address operator used with pointers; after the example we will discuss the '^' in depth.

Example #1:

```
   PROC pointerusage()

      BYTE num=$EØ,      ;declare and place two
           chr=$E1       ;BYTE variables.

      BYTE POINTER bptr  ;declare a pointer to BYTE type.

      bptr=@num          ;make bptr point to num;
      Print("bptr now points to address ")
      PrintF("$H",bptr)  ;prints out num's address.
      PutE()
      bptr^=255          ;puts 255 into the location bptr
                         ;points to (i;e;, into num).
      Print("num now equals ")
      PrintBE(num)       ;shows that 255 really went into
                         ;num.
      bptr^=Ø            ;puts Ø into num
      Print("num now equals ")
      PrintBE(num)       ;shows that num equals Ø now.
      bptr=@chr          ;makes bptr point to chr now.
      Print("bptr now points to address ")
      PrintF("%H",bptr)  ;prints out chr's address, so we
      PutE()             ;know that bptr really changed.
      bptr^='q           ;puts 'q into the location bptr
                         ;points to (i.e., into chr);
      Print("chr now equals ")
      Put(chr)           ;shows that chr really equals 'q
      PutE()
      bptr^='z           ;changes chr to 'z
      Print("chr now equals ")
      Put(chr)           ;shows that chr is equal to 'z
      PutE()
   RETURN
```

Output #1:

```
      bptr now points to address $EØ
      num now equals 255
      num now equals Ø
      bptr now points to address $E1
      chr now equals q
      chr now equals z
```

Notice that we use the '^' operator when we want to put a
value into the place the pointer points to. So the line
"bptr^=Ø" in the above example is the same as saying
"num=Ø", because 'bptr' is pointing to 'num' at that time.
Although we do not use it in the above example, pointer
references can be used in arithmetic expressions, as
follows:

```
            x=ptr^
```

Also notice that "PrintF("%H", bptr)" is valid. What this
means is that 'bptr' can be accessed as a CARDinal number
as well as an address. This is useful when debugging your
program, because you can find out where the pointer is
pointing easily.

8.2 ARRAYs

Arrays allow you to manipulate lists of variables by making
each variable in the list accessible using only the array
name and a subscript. The variables in the list must be of
the same data type, and only the fundamental data types are
allowed. The array name tells which array you want and the
subscript tells which element of that array you want. The
subscript is just a number, so what you are really saying
when you reference an array element is, "I want the nth
element of array x," where 'n' is the subscript and 'x' is
the array name.

In the following section we will discuss the internal
representation of an array. After that we will show you how
to declare arrays and manipulate them, and then we will
talk about the limitations of arrays in ACTION!.

8.2.1 Array Declaration

Declaring arrays is easy in ACTION!, but that does not mean
that you do not have much control over what is going on.
There are many options you can use to define different
characteristics of the array, including its address, its
size, and even its initial contents. Because of all these
options, the format looks somewhat cluttered, but the
examples should clear up any confusion.

        <type> ARRAY <var init>|:,<var init>:|

where

        <type>   is the fundamental type of the
                    elements of the array.
        ARRAY   is the keyword denoting an array.
        <var init> is the information required to declare
                    one variable as an array of <type>
                    data type elements.

<var init> has the form:

        <ident>{(size)}{=<addr> | [<values>] | <str const>}

where

        <ident>   is the name of the variable
        <size>   is the size of the array, and must be
                      a numeric constant in decimal form.
        <addr>   is the address of the first element of
                      the array, and must be a compiler
                      constant.
        [<values>]   sets the initial values of the
                      elements of the array. Each value must
                      be a numeric constant.
        <str const>   sets the initial values of the
                      elements of the array to the string
                      constant, with the first element being
                      the length of the string.

We warned you that it was cluttered! But now to organize
some of this clutter with some instructive (hopefully)
examples:

        BYTE ARRAY a,b ;declares two arrays with BYTE
                        ;elements without sizes declared

        INT ARRAY x(1Ø) ;declares 'x' as an INT array,
                        ;and dimensions its size

        BYTE ARRAY str="This is a string constant" ;this
                        ;declares 'str' as a BYTE array, and
                        ;fills it with a string constant

        CARD ARRAY junk=$8ØØØ ;declares 'junk' as a CARD
                        ;array, which starts at $8ØØØ in
                        ;memory, without any size implied

        BYTE ARRAY tests=[4 7 18] ;declares 'tests' as a
                        ;BYTE array, and fills in its
                        ;values.

PROGRAMMING NOTES: You should dimension the size of an
array whenever possible, but there are some instances where
you cannot or need not:

    1) When you do not know how big it is going to be
       (i. e., as in a routine parameter, when you do not
       know how big an array is going to be passed).
    2) When you are filling the array in the declaration
       (using either the '[<values>]' or '<str const>'
       construction), and you are not planning to add to
       the array.

The ACTION! Programming Environment

Also, remember that the first byte of a string constant
contains the length of that string. So, to make an string
longer, first you must change the length byte (which is the
zeroth element of the array containing the string).

8.2.2 Internal Representation

The internal representation of an array is very much like
that of a pointer. This is because the array name is
actually a pointer to the first element of the array. The
array itself is simply a contiguous group of cells, each
containing an array element. The size of a cell is
determined by the data type of the elements: one-byte cells
for BYTE type, two-byte cells for both CARD and INT types.
However, having the array name be a pointer leads to some
very interesting ramifications, as shown in examples 2
through 4 of the following section.

8.2.3 Array Manipulation

Using and manipulating arrays is not very difficult once
you know how to declare the array and reference its
elements. You already know how to declare arrays, so now we
will show you how to reference elements:

Example #1:

```
   PROC reftest()
      BYTE x
      BYTE ARRAY nums(1Ø)

      FOR x=Ø TO 9      ;although nums is ten elements
                        ;long, the subscripts run from
                        ;Ø to 9, not 1 to 1Ø
         DO
         nums(x)=x+'A   ;xth element of nums is assigned
                        ;the value x+'A
         Put(nums(x))   ;put out xth element of nums as
                        ;a character
         Print(" ")     ;put a space between the chars.
         OD
      PutE()
   RETURN
```

Output #1:

```
      A B C D E F G H I J
```

There are two array references in the above program -- 'nums(x)' in the assignment statement, and 'nums(x)' as a parameter to the 'Put' library procedure. They, and all other array references, have the form:

```
<ident>(<subscript>)
```

where

```
<identifier>  is the name of the array you want to
              reference.
<subscript>   is the number of the element in that
              array,    and    is    an    arithmetic
              expression.
```

As mentioned in the comment explaining the FOR loop, array subscripts do not start at 1, as you might expect. The first element in array 'cat' is 'cat(Ø)', not 'cat(1)'. This might seem strange, but you get used to it very quickly.

Example F2:

```
PROC changearray()

    BYTE ARRAY barray

    barray="This is string 1."
    PrintC(barray)   ;prints the CARD 'barray' contains
    Print(" ")
    PrintE(barray)   ;prints the string 'barray points
    barray="This is string 2."   ;to (with an EOL)
    PrintC(barray)
    Print(" ")
    PrintE(barray)
RETURN
```

Output #2:

```
    1Ø352 This is string 1.
    1Ø414 This is string 2.
```

EXAMPLE 2 COMMENTS: Notice from the output that the address to which 'barray' is pointing changes. Reassigning the whole array (when doing it using string constants) does not put the new string into the memory space occupied by the old one, but rather allocates new space for the new string, and then changes the value of 'barray' to point to the starting address of the new string. The old string is still in memory, but nothing is pointing to it any more, so it is inaccessible.

The ACTION! Programming Environment

Notice that "PrintE(barray)" is valid, because 'barray'
points to a valid string constant, which is the type of
parameter the PrintE library procedure requires. Pretty
sneaky!!

Example #3:

```
   PROC equatearrays()

      BYTE ARRAY a="This is a string constant",
                 barray

      barray=a
      PrintE(a)
      PrintE(barray)
   RETURN
```

Output #3:

```
       This is a string constant
       This is a string constant
```

EXAMPLE 3 NOTES:
      All this program does is show you that you can equate
      two arrays simply by making them point to the same
      memory location; in this case it is a string constant
      they are both pointing to.

You might have noticed that we have not done anything like

```
   BYTE ARRAY a=['A '  's 't 'r 'i 'n 'g]

   PrintE(a)
```

That is because the above will not work. Remember that
string constants are different from simple strings because
their first byte contains their length, so procedures that
expect a string constant will balk when you attempt to send
them anything else.

And now for a program that uses all the applications of
arrays which we have discussed.

Example #4:

> SCENARIO: You have a program that only gives error
> numbers when the user makes an error, and you want it
> to print out error messages as well. You could do
> this using arrays, as in the following program. We
> will discuss how the program works after the program
> itself.

```
PROC doerror(BYTE errnum)
;**** This procedure reads in the error number and
;prints out the related message. See the discussion
;following the program for an explanation of how it
;works.

    BYTE ARRAY errmsg     ;the message printed out

    CARD ARRAY addr(6)    ;holds the addresses of the
                          ;error messages

    addr(Ø)="Illegal command"
    addr(1)="Illegal character" See
    addr(2)="Bad File Name"     EXAMPLE 4
    addr(3)="Number Too Large" NOTES for an
    addr(4)="Wrong Type Of Number" , explanation
    addr(5)="Unknown Error"
    errmsg=addr(errnum) ;puts the error message asso-
    Print("ERROR #")    ;ciated with 'errnum' in
    PrintB(errnum)      ;'errmsg' and prints it
    Print(": ")         ;out after the error
    PrintE(errmsg)      ;number itself
    PutE()
RETURN   ;**** End of procedure doerror

PROC main()
;**** This procedure is just a dummy used to call
;the above procedure, using all valid error numbers,
;to show that the table works.

    BYTE error

    FOR error=Ø TO 5
      DO
      doerror(error)
      OD
RETURN ;**** End of procedure main
```

Output #4:

        ERROR #Ø: Illegal command
        ERROR #1: Illegal character
        ERROR #2: Bad File Name
        ERROR #3: Number Too Large
        ERROR #4: Wrong Type Of Number
        ERROR #5: Unknown Error

EXAMPLE 4 NOTES: The way in which we fill the CARD array in
this example is strange (how can you fill a CARD array
element with a string) but is perfectly valid because the
string constant itself is not being assigned to the array
element; rather its address is. This makes each element of
the array an implicit pointer to a string. All we have to
do is assign the value of the proper array element (i. e.,
the one pointing to the needed error message) to the BYTE
array 'errmsg' thus making 'errmsg' point to the proper
message. Then we just print out the message.

We understand that the above program is very confusing
until you completely understand the concept of arrays and
their internal representation, but it is here so you can
see some of the advanced capabilities of arrays.

8.3 Records

Records are constructions which allow you to group together
some pieces of information, which, although related in some
way, are not of the same type. Your driver's license is an
example of a record. It has your name, photo, address, and
license number all together. These pieces of information
belong together in that they all describe you to some
degree, but they are of different types. Your name is a
character string, your photo is a picture, and your address
is made up of both numbers and characters, as is your
license number. Of course the ACTION! language does not
support all these types. Instead, it groups together the
types of information the compiler understands: the
fundamental data types.

8.3.1 Declaring Records

ACTION! records manipulate the fundamental data types by
creating a new data type composed of one or more of the
fundamental types. Then you declare variables of that type
just as you declare variables of type BYTE, INT, or CARD.
This allows you to declare as many variables of one record
type as you want, without having to re-declare the format
of the record type every time.

The next section (8.3.1.1) shows how to create a record
data type, and section 8.3.1.2 demonstrates how to declare
variables of a predefined record type.

8.3.1.1 The TYPE Declaration

Without further ado we will present the form used to
declare a record data type:

        TYPE <ident>=[<var decls>]

where

        TYPE   is the keyword denoting the definition
                    of a record type.
        <ident>   is the name of that record type.
        <var decls> are valid variable declarations, as in
                    section 3.4.1, except that the
                    '=<init info>' option shown there is
                    forbidden.

At this point, an example would probably help:

    TYPE rec=[BYTE b1,b2   ;two BYTE fields first,
             INT i1   ;then one INT field,
             CARD cl,c2,c3   ;then three CARD fields
             BYTE b3]   ;ending with a BYTE

This needs some explanation so we will go through it piece
by piece:

TYPE rec
        We are defining a new data type called 'rec'

BYTE b1,b2
        The first two fields of this type are of BYTE type,
        and are called 'bl' and 'b2'.

INT i1
        The third field is of type INT, and its name is
        'i1'.

The ACTION! Programming Environment

CARD c1,c2,c3
        The fourth through sixth fields are CARD type, and
        are named c1, c2, and c3, respectively.

BYTE b3
        The seventh and final field of the record type
        'rec' is of BYTE type and is called 'b3'.

Notice that there are no commas between the different
variable declarations (between the CARD and BYTE
declarations, for example). If you do put commas in, the
compiler will try to read the fundamental type words (CARD,
BYTE, INT) as variables, and that will cause a compiler
error.

NOTE: A TYPE declaration will generate a spurious error
whenever the code offset (contents of location $B5) is non-
zero. Presumably only noticed if using the runtime library.
To fix this do all TYPE declarations before changing the
code offset.

8.3.1.2 Declaring Variables

The last section showed you how to declare a record type,
and this section will show you how to declare variables of
a given record type. The format is very similar to that
used when declaring variables of fundamental types, but it
does have its peculiarities:

        <ident> <var>{=<addr>}|:,<var>{=<addr>}:|
where
        <ident>   is the name of the record type.
        <var>   is a variable whose data type is
                      declared to be the record type.
        <init info>   is information used to set some
                      attributes of the variable.
        <addr>   is the address in memory where you
                      want the variable to be located. It
                      must be a numeric constant.

Here is an example using the record type declared in the
previous section. After the example is an explanation of
what is going on.

    TYPE rec=[BYTE b1,b2   ;same TYPE declaration
             INT i1   ;used in the previous
             CARD c1,c2,c3   ;section
             BYTE b]   ;ending with a BYTE

    rec arec,   ;declares arec as data type 'rec'
        brec=$8000   ;declares brec as type 'rec',
                           ;and places it at address $8000

122

EXPLANATION:

rec   Shows that the following variables are of data type
      'rec', just as BYTE, INT, and CARD (when used in
      variable declarations) show that the following
      variables are of those types.

arec Declares 'arec' to be a variable of data type 'rec'.

brec=$8ØØØ Declares 'brec' to be a variable of data type
            'rec', and places it at memory location $8ØØØ.

So now that you know how to declare a record data type, and
then declare variables of that type, it is time to find out
how to reference and manipulate records.

8.3.2 Record Manipulation

To learn how to manipulate records, we first must must
learn how to reference a field within a record. The
following program does just that, using the period ('.')
operator. We will discuss its usage after the program
itself.

Example #1:

```
   PROC recordreference()
   ;**** This procedure reads in some information about
   ;an employee, and then prints it out to let the
   ;employee know it is correct.

      TYPE idinfo=[BYTE level       ;employee's level
                   CARD idnum,      ;his I.D. number
                   entry year]      ;year he started
      idinfo rec                    ;declaring 'rec' as
                                    ;record type 'idinfo'
      Print("What is your I.D. Number?  ")
      rec.idnum=InputC()       ;get his I.D. number
      Print("What is your employment level (A-Z)?  ")
      rec.level=GetD(7)        ;get his employment level
      Print("In what year did you start working here?  ")
      rec.entryyear=InputC()   ;get his entry year
      PrintE("O._K. Here is what I have:")
      PutE()                +
      Print("I.D. # ")      |   Prints
      PrintCE(rec.idnum)    |   out the
      Print("Level: ")      |   information
      Put(rec.level)        |   the
      PutE()                |   employee
      Print("Entry year: ") |   put in
      PrintCE(rec.entry_year) +
   RETURN    ;end of PROC recordreference
```

The ACTION! Programming Environment

Output #1:

        What is your I.D. Number? 4365
        What is your employment level? L
        In what year did you start working here? 1978

O.K. Here is what I have:

        I.D. # 4365
        Level: L
        Entry year: 1978

The '.' is used to notify the compiler that you are making
a record reference (and is only valid in record
references). From the above program example you can see
that the format of a record reference is:

        <record name>.<field name>

Note that <field name> and <record name> are defined in
different declaration statements, as shown in the previous
section. <field name> is defined in the TYPE declaration,
when you define the fields of a record type, whereas
<record name> is defined in a variable declaration, when
you declare the variable to be of a record type.

8.4 Advanced Use of the Extended Types

The extended data types seem to be limited by the fact that
they may only operate on the fundamental types; that is,
you cannot have arrays of records, an array field in a
record, etc. However, there are ways to get around these
limitations, as seen in example 4, section 8.2.3. In that
example we created an array of pointers by using the
elements of a CARD array as pointers, not cardinal numbers.
In this section we will demonstrate some other ways to get
more out of the extended types, including a program using
records with array fields, and another program which uses
an array of records.

"But you just said that was illegal." It is illegal if you
try it directly, but, as we mentioned above, there are ways
around, over, under, and between the literal definition of
the extended types.

The following example will fill a dimensioned array with a
list of records. The way it does this is simple once we
define a "virtual record", because the array is actually a
BYTE array with blocks of bytes being grouped into virtual
records.

A virtual record is not a record in the sense that we declare it as a record type. It is a record only because we access a section of memory as though it were a record, although it is really just a string of bytes. All we do is fill a BYTE array so that it looks like contiguous records, not bytes. This is done by declaring a record data type, and then declaring a pointer to that data type. Then we manipulate the array in blocks the size of one record by making the pointer jump through the array in leaps the size (in bytes) of one record. We will expand on this in the technical discussion following the example itself.

Example #1:
```
   MODULE      ;declaring some global variables

      TYPE idinfo=[CARD idnum,     ;employee's I.D. number
                        codenum    ;his access code
                  BYTE level]      ;his employment level

      BYTE ARRAY idarray(1000)     ;enough space to hold
                                   ;200 records.
      DEFINE recordsize="5"

      CARD reccount=[0]

   PROC fillinfo()
   ;**** This procedure will take some information on a
   ;given employee, put it into an array of records using
   ;a pointer to the record type and indexing that pointer
   ;in the array. This process will continue as long
   ;as the user desires to input more information.

      idinfo POINTER newrecord

      BYTE continue

      DO
      newrecord=idarray+(reccount*recordsize)
      Print("I.D. Number?  ")
      newrecord.idnum=InputC()        ;get I.D. number
      Print("Employment level (A-Z)?  ")
      newrecord.level=GetD(7)         ;employment level
      Print("Access code?  ")
      newrecord.codenum=InputC()      ;get secret code
      reccount==+1
      PutE()
      Print("Input another record (Y or N)?  ")
      continue=GetD(7)
      PutE()
      UNTIL continue='N OR continue='n
      OD
   RETURN
```

PROGRAMMING NOTE: This procedure does not make sure you are within the bounds of the array, nor does ACTION! itself, so you might want to add a boundary checking routine.

EXAMPLE 1 NOTES: There are a couple things this procedure does that require a detailed explanation, including these procedure lines:

```
DEFINE recordsize="5"

idinfo POINTER newrecord

newrecord=idarray+(reccount*recordsize)

newrecord.XXX=xxx

reccount==+1
```

we will go through these one by one. This should not only explain the statements themselves, but should also clarify the concept we are using to accomplish the array of records.

DEFINE recordsize="5"
> This DEFINE is used as the "jump" size when we are going through the array. The record type 'idinfo' is 5 bytes long (2 CARDS and 1 BYTE), so this will allow us to go through the array in 5-byte leaps. Every time we leap like this we will skip over one record, thus eliminating the possibility of writing one record partially on top of another.

idinfo POINTER newrecord
> Here we are defining a pointer to the type 'idinfo'. We can fill fields of a virtual record in the array simply by pointing the pointer to the first field in one of the virtual records, and then using the pointer in a record reference to access a single field.

newrecord=idarray+(reccount*recordsize)
> This assignment makes the pointer point to the current record in the array. It does this by adding the space occupied by all the other records to the starting address of the array. The space occupied by all the other records is simply the number of records ('reccount') times the size of each record ('recordsize').

newrecord.XXX=xxx
> 'XXX' is one of the field names of the record type, and 'xxx' is the corresponding input function used to fill the array. Since we made 'newrecord' point to the end of the array, we can start filling in the new record. We can use the pointer in the record reference because we declared it as a pointer to that record type.

reccount==+1
> Here we are simply incrementing the variable that keeps track of the number of records currently in the array. We do this because we just put another one in.

In example #4 we will use this array we have filled to verify the information typed in by someone trying to gain entrance into a restricted area (by making sure they key in the proper secret code), but we will have to remember to access the array as an array of records, using the same format in which the array was filled, otherwise some strange problems will arise.

Before we go on to show the program that looks into the filled array, let us first modify the records a little bit. We will add one more field which will contain the employee's name in the form:

LastName, FirstName

To do this we must somehow make the field an array. Or must we? Instead, let us simply add a BYTE field to the end of the record type, and then change the DEFINE directive to make the size given each record increase. If we increase it by 2Ø, suddenly we have 25 bytes reserved for 6 bytes of field (2 CARDs and 2 BYTEs). Then we just put the string in the extra space, by accessing the last field (our new BYTE field) and putting in a string instead of a byte. The string cannot be longer than 19 characters (recall the first byte of a string is its length), so we will have to make sure the string is short enough. Without further ado, we will move onto the extended version of the 'idinfo' procedure, complete with strings.

The ACTION! Programming Environment

Example #2:

```
   MODULE     ;declaring some global variables

      TYPE idinfo=[CARD idnum,     ;employee's I.D. number
                        codenum    ;his access code
                  BYTE level,      ;his employment level
                        name]      ;first letter of name

      BYTE ARRAY idarray(1ØØØ)     ;enough space to hold
                                   ;4Ø records.

      DEFINE recordsize="25",
             nameoffset="5"

      CARD reccount=[Ø]

   PROC fillinfo()

;**** This is simply the modified version of the previous
;example.

      idinfo POINTER newrecord

      BYTE POINTER nameptr    ;pointer to 'name' field

      BYTE continue

      DO
      newrecord=idarray+(reccount*recordsize)
      Print("I.D. number?  ")
      newrecord.idnum=InputC()        ;get I.D. number
      Print("Employment level (A-Z)?  ")
      newrecord.level=GetD(7)         ;employment level
      Print("Access code?  ")
      newrecord.codenum=InputC(   )   ;get secret code
      nameptr=newrecord+nameoffset    ;point 'nameptr' to
      PrintE("Employee's name?  ")    ;start of name field
      Print("(form: Last, First) ")
      InputS(nameptr)            ;read name into name field
      reccount==+1
      PutE()
      Print("Input another record (Y or N)?  ")
      continue=GetD(7)
      PutE()
      UNTIL continue='N OR continue='n
      OD
   RETURN
```

EXAMPLE 2 NOTES: As in the previous example, there are
                  some   program   lines   which   need
                  explanation, including:

                      nameoffset="5"

                      BYTE POINTER nameptr

                      nameptr=newrecord+nameoffset

                      InputS(nameptr)

Before discussing the lines individually, let us go
over the method used to put the name into the array of
records. First of all, we need to find where to put
the name once we've read it in, then we need to figure
out a way to read the name in. The explanations of the
above statements show you how we do it:

nameoffset="5"
     This DEFINEs the distance you have to go into a
     single record to get to the first byte of the
     string, and is used when getting the pointer to
     the string to point to the right position.

BYTE POINTER nameptr
     This pointer is used to point to the first byte
     of the 'name' field in a record.

nameptr=newrecord+nameoffset
     Here we are setting the value (i.e., where we
     want the pointer to point) of the pointer
     'nameptr'. Its set by taking the address of the
     start of the record ('newrecord') and adding the
     offset distance to the first byte of the string
     storage location.

InputS(nameptr)
     This is used to read in the name, and uses
     'nameptr' as a pointer to the storage location,
     just as shown in section 8.2.3 (example 2),
     except that we are using a pointer instead of an
     array name (which is just a pointer to the first
     element anyway).

Now that we have a way to put the records into the array,
we need a way to search through the array record by record
when looking for a match. The following is a function
designed to do just that. It will access the array as using
the record format of example 2, and return the address of
the start of the first record with an 'idnum' matching the
one passed in as a parameter. If no match is found, then Ø
is returned as the address. Note that this function uses

The ACTION! Programming Environment

variables declared in the global statement section (i.e.,
after the MODULE) of the previous example.

Example #3:

```
   CARD FUNC findmatch(CARD testidnum)

      idinfo POINTER seeker      ;points to each record
                                 ;in turn to do test

      BYTE ctr ;used as a counter in the FOR loop

      FOR ctr=Ø TO (reccount-1)  ;minus one because we
         DO                      ;start at Ø, not 1
         seeker=idarray+(ctr*recordsize)   ;index record
         IF seeker.idnum=testidnum THEN    ;test for an
            RETURN (seeker)      ;I.D. match and return
         FI                      ;if found
         OD
   RETURN (Ø) ;no match found. End of FUNC findmatch
```

This function needs very little explanation, since it is
straightforward compared to the previous examples. All we
do is go to every record and test its 'idnum' field for a
match with 'testidnum'. Now let us turn the past two
examples into a true program by putting a shell around it.

Example #4:

```
MODULE                 ;declaring some global variables

   TYPE idinfo=[CARD idnum,   ;employee's I.D. number
                     codenum  ;his access code
              BYTE level,     ;his employment level
                     name]    ;first letter of name

   BYTE ARRAY idarray(1ØØØ)  ;enough space for 4Ø records

   DEFINE recordsize="25",
          nameoffset="5"

   CARD reccount=[Ø]
```

```
;*****************************************************
;
;              continued on following page
;
;*****************************************************
```

```
PROC fillinfo()   ;**** Again the array filling procedure

   idinfo POINTER newrecord

   BYTE POINTER nameptr   ;pointer to 'name' field

   BYTE continue
   DO
      newrecord=idarray+(reccount*recordsize)
      Print("I.D. Number?  ")
      newrecord.idnum=InputC()      ;get I.D. number
      Print("Employment level (A-Z)?  ")
      newrecord.level=GetD(7)       ;employment level
      Put(newrecord.level)
      PutE()
      Print("Access code?  ")
      newrecord.codenum=InputC()   ;get secret code
      nameptr=newrecord+nameoffset ;point 'nameptr' to
      PrintE("Employee's name?")   ;start of name field
      Print("(form: Last, First) ")
      InputS(nameptr)              ;read name into name field
      reccount==+1
      PutE()
      Print("Input another record (Y or N)? ")
      continue=GetD(7)
      PutE()
      UNTIL continue='N OR continue='n
   OD
RETURN

CARD FUNC findmatch(CARD testidnum)

   idinfo POINTER seeker   ;points to each record
                           ;in turn to do test
   BYTE ctr                ;used as a counter in the FOR loop

  IF reccoun>Ø THEN     ; prevent endless loop if no
                        ; data have been keyed in already

   FOR ctr=Ø TO (reccount-1)        ;minus one because we
      DO                            ;start at Ø, not 1
         seeker=idarray+(ctr*recordsize) ;index record
         IF seeker.idnum=testidnum THEN ;test for an
            RETURN (seeker)             ;I.D. match and
         FI                             ;return if found
      OD
RETURN (Ø) ;no match found. End of FUNC findmatch

;****************************************************
;
;             continued on following page
;
;****************************************************
```

```
The ACTION! Programming Environment

PROC main()   ;* This procedure controls the whole shebang

   idinfo POINTER recptr        ;pointer to a record
   BYTE POINTER nameptr      ;pointer to 'name' field
   CARD id_num,           ;I.D. number input by user
        code_num,           ;code number input by user
        keyid=[65535]     ;I.D. number allowing loop exit

   BYTE mode                ;controls the operation mode

   Put(125)              ;delete garbage from screen
   PrintE("Startup....")
   PrintE("What operation mode? ")
   PrintE("X = expand list of employees")
   PrintE("A = alert/test input mode")
   Print(">> ")
   mode=GetD(7)          ;read mode
   Put(mode)
   PutE()
   IF mode#'A AND mode#'a THEN  ;anything but A or a will
      fillinfo()                ;go to X mode
   ELSE
      DO      ;loop start for interrogation routine
         Print(" Employee I;D. number >> ")
         id_num=InputC() ;get I.D. number
         IF id_num=keyid THEN ;enables exit from
           EXIT           ;the infinite loop
         ELSE ;a normal I.D. number (i.e., not keyid)
           recptr=findmatch(id_num) ;look for I.D. match
           IF recptr=0 THEN        ;no match
              PrintE("DO NOT PASS")
           ELSE                     ;an I.D. match
              Print(" Code Number >> ")
              code_num=InputC()    ;get access code
              IF recptr.codenum=code_num THEN ;a match
                 nameptr=recptr+nameoffset
                 Print("I.D. #  ")       ; +
                 PrintCE(recptr.idnum)   ; | print
                 Print("Level:  ")       ; | out
                 Put(recptr.level)       ; | known
                 Print("Name:      ")    ; | info.
                 PrintE(nameptr)         ; +
                 PutE()
                 PrintE("O.K. TO PASS")
              ELSE     ;code does not match
                 PrintE("DO NOT PASS")
              FI        ;end of access code testing
           FI            ;end of I.D. number verification
         FI                  ;end of 'keyid' check
      OD                 ;end of infinite loop
   FI                 ;end of 'IF mode=
PrintE("System Shutdown...")
RETURN                ;end of PROC main

132
```

All the main procedure does is go through a series of checks to determine what needs to be done at any given point. The nested IFs are somewhat confusing, but they are lined up (that is, indented the same amount) so you can do IF-FI paring by placing a ruler vertically on the page and sliding it back and forth to change levels of nesting.

Chapter 9: Advanced Concepts

This chapter deals with some techniques the experienced
programmer might find useful. Thus far, we have limited our
discussion of the ACTION! language to a study of the
language with respect to itself; that is, without reference
to the rest of the computer. Most of this chapter is
devoted to interfacing ACTION! to information external to
ACTION! itself, including operating system routines and
system variables.

9.1 Code Blocks

Code blocks allow you to include machine code in your
program. When the compiler sees a code block, it will put
the values in the block into the code generated, just as
though it were code generated by the compiler. No checks
are made, so we do not recommend that you use code block
unless you know quite a bit about assembly and machine
language.

The format for a code block is:

        [<value>|: <value>:|]

where
        <value> is one of the values in the code block. It
must be a compiler constant (see section 3.2). If it is
greater than 255, then it is stored in LSB, MSB order.

Examples:
        [$4Ø $ØD $51 $FØ $6ØØ]

        BYTE b1,b2,b3
        ['A b1 342 b3 4+$A7]

        DEFINE on=1
        [54 on on+'t $FFF8]

Code blocks are useful for including small machine code
routines, but it is too much trouble to insert a large one.
If you want to use a lot of machine code routines, see
section 9.4 for some hints.

9.2 Addressing Variables

In sections 3.4.1, 8.1.1, and 8.2.1 (Fundamental, POINTER,
and ARRAY variable declarations) we showed that a
variable's address could be specified when that variable

was declared, but we did not really make use of that option. We did not even explain the usefulness of doing this.

This option allows you to declare an ACTION! variable which has the same address as any hardware register. Then you can manipulate graphics and sound directly, change operating system characteristics, etc.... To illustrate the advantages of this, we are going to present a graphics program which makes the background color change and scroll. To do this we cannot use the normal (shadow) color registers, because they are only looked at every T.V. frame. Instead, we will directly manipulate the hardware color registers. In this way we can change the background color during one frame. In fact, we can do it 12 times (and so get 12 colors in graphics Ø). We have to make sure that we do not change colors in the middle of a scan line, so we will make use of the hardware variable WSYNC, which tells when a scan line is done, and the next one has not yet begun. The variable VCOUNT tells how many scan lines have been put out, and we use it to time the scrolling.

Example #1:
```
PROC scrollcolors()

   BYTE wsync=54282,    ;the "wait for sync" flag
        vcount=54283,   ;the "scan line count" flag
        clr=53272,      ;hardware register for background
        ctr,chgclr=[Ø], ;a counter and a color changer
        incclr          ;increments color luminance

   Graphics(Ø)          ;set graphics Ø
   PutE()
   FOR ctr=1 TO 23      ;print out demo message
      DO
      PrintE("A DEMO OF SHIFTING BACKGROUND COLORS")
      OD
   Print("A DEMO OF SHIFTING BACKGROUND COLORS")
   DO                   ;start of infinite scrolling loop
   FOR ctr=1 TO 4
      DO
      incclr=chgclr     ;set base color to increment
        DO              ;start of UNTIL loop
       wsync=Ø          ;waits for end of scan line
        clr=incclr      ;change displayed color
        incclr==+1      ;change luminance
      UNTIL vcount&128   ;end of screen test
       OD               ;end of UNTIL loop
      OD                ;end of FOR loop
   chgclr==+1           ;change the base color
   DO                   ;end of infinite scrolling loop
RETURN                  ;end of PROC scrollcolors
```

9.3 Addressing Routines

The concept behind specifying the address of a routine is similar to that of specifying the address of a variable. Only the reason behind the concept changes. In the last section we talked about using ATARI system registers directly by addressing an ACTION! variable to the proper location. Because you can define a routine's address, you can make direct calls to OS and hardware routines directly, and do your own manipulation of I/O. The method used will be discussed in the following section, because this method applies to all machine language routines, whether written by you, resident on the OS, or resident in the ROMs.

9.4 Assembly Language and ACTION!

ACTION! allows you to make calls to machine language routines very easily. There are only two requirements:

     You need to know the starting address of the routine

     The routine must end with an 'RTS' (if you want to
     get back to ACTION!)

For assembly language programmers these are not difficult requirements to fill.

"What about parameters?" "Yes" is the answer. You can send parameters to machine language routines. The compiler stores parameters in this way:

Address	nth byte of parameters
A register	1st
X register	2nd
Y register	3rd
$A3	4th
$A4	5th
:	:
:	:
$AF	16th

And now for an example:

```
PROC CIO=$E456(BYTE areg,xreg)
;**** Declaring the OS procedure CIO. 'xreg' will
;contain the iocb number times 16, and 'areg' is a
;filler, so the number will not go into register A
;(CIO expects it in X reg.)

PROC readchannel2()

;**** This procedure will open channel 2 to the
;given file name, and call CIO to read 'buflen' bytes

   DEFINE buflen="$2ØØØ" ;length of the buffer array

   BYTE ARRAY filename(3Ø),     ;the file name array
              buffer(buflen)    ;the buffer array

   BYTE iocb2cmd=$362           ;iocb 2's command byte

   CARD iocb2buf=$364,  ;iocb 2's buffer start address
        iocb2len=$368   ;iocb 2's buffer length

   PutE()
   Print("File name >> ")
   InputS(filename)     ;get the filename
   Open(2,filename,4,Ø) ;open channel 2 for read only
   iocb2cmd=7           ;'get binary record' command
   iocb2buf=buffer      ;set iocb buffer to our buffer
   iocb2len=buflen      ;set iocb buffer length
   CIO(Ø,$2Ø)           ;***** the call to CIO *****
   Close(2)             ;closing channel 2
RETURN
```

See how easy it is? For those of you with an extensive set of assembly language routines, this ability of ACTION! allows you to use them in a high level language, where building the framework of a program is easy.

9.5 Advanced Use of Parameters

In section 6.4 we discussed parameters and their usage, mentioning that you could not pass a value out of a routine using a parameter. Well, that was a little white lie. You can pass values out through parameters if you use pointers. All you do is create a pointer which points to the variable you really want to pass into a routine, and pass the pointer instead. Then, when you access what the pointer is pointing to, you are really accessing the variable you wanted to pass. You can then change the value of that variable using a pointer reference.

The ACTION! Programming Environment

This method involves some indirection (i. e., using a pointer to a variable instead of the variable itself), but it is very efficient and useful in some cases, as the following example shows.

Example #1:

```
BYTE FUNC substr(BYTE ARRAY str,sub BYTE POINTER errptr,notfound)

;**** This function will search 'str' looking for the sub-
;string 'sub'. If it is found, the function returns the in-
;dex onto the string. If the substring is longer than the
;mainstring an error is returned via pointer. If the sub-
;string is not found, that is returned via another pointer.

  BYTE ARRAY tempstr ;holds temporary substring for test

  BYTE ctr1,   ;outer loop counter
       ctr2    ;inner loop counter

  IF sub(Ø)>str(Ø) THEN    ;substring bigger than string
     errorptr^=1
  ELSE
     FOR ctr1=1 TO str(Ø)  ;loop to check string
        DO
        IF sub(1)=str(ctr1) THEN  ;testing 1st characters
           tempstr(Ø)=sub(Ø)      ;dimension tempstr
           FOR ctr2=1 to sub(Ø)   ;fill tempstr
              DO
              tempstr(ctr2)=str(ctr2+ctr1-1) ;fill tempstr
              OD
           IF SCompare(tempstr,sub)=Ø THEN ;compare 2
                                           ;strings
              RETURN (ctrl) ;return index if equal
           FI
        FI ;end of testing 1st characters
        OD ;end of FOR loop
  FI
  notfound^=1   ;did not RETURN in loop, so no match found
RETURN (Ø)   ;end of FUNC substr
```

Now, when we want to call this we must use the form:

```
<index>=substr(<string>,<substring>,<errptr>,<nofindptr>)
```

where <index>   is the index into <string> where
                  'substring' starts.
       <string)   is the main string
       <subetr>   is the substr we want to find in
                  the main string
       <errptr>   is a pointer to a byte error flag
      <nofindptr> is a pointer to a byte 'substring
                  not found' flag

This kind of parameter manipulation takes some practice if you are not used to the concept of pointers, but is a quick and easy way get more information passed out of a routine without having to resort to using global variables. This means that the routine remains "multipurpose", as discussed in section 6.4.

Part V: The ACTION! Compiler

The ACTION! Programming Environment

                    Part V: The ACTION! Compiler

                       Chapter 1: Introduction

ATARI BASIC offers you great convenience in that you can
write a program in a somewhat English-like language, then
immediately test that program without going through any
other steps. This twofold advantage is gained at the
expense of requiring that each command on each line be
figured out by a special program (called the BASIC
interpreter) at the time of execution.

ACTION! is somewhat more sophisticated. It requires that
your program be figured out by a special program - called a
compiler - before the actual execution of your program.
This requires an intermediate step between your entry of
the program and its execution by the computer. The step is
technically known as "the compile". During the compile, the
ACTION! Compiler analyzes your program on a line-by-line
basis. Your program is converted into a different language
(called machine language) with storage for both global and
local variables. The converted program can then be executed
by your ATARI, running at a speed much greater than that of
the interpreted ATARI BASIC.

1.1 VOCABULARY

This chapter refers to several terms which you first
learned about in Part IV. Those terms are listed here, with
each term briefly defined:

          term   comments
          ----   --------
        <ident>   any valid identifier
        <value>   any valid hex or decimal value
  <compiler constant>   evaluates <ident>'s address
      <address>   memory location

1.2 Compiler Directives

The compiler directives are discussed in depth in part IV,
chapter 7, and little more need be added here. We simply
remind you that the compiler directives are executed at
compile time, not run time), so do not use them when you
want to change an operational parameter while your program
is running.

The ACTION! Programming Environment

Chapter 2: Compiler Operation - Allocating Space

In this chapter we will discuss how the the ACTION!
compiler allocates memory space for your compiled program,
its variables, its routines, and its symbol tables.

When called, the first thing the ACTION! Compiler does is
to decide where to put the code it will generate as it
compiles your ACTION! source program. It does this by
looking at memory location 14. The CARD value this and the
following location contain gives the address of the start
of free memory. This address will vary, depending on the
size of the Editor buffer (see Appendix B). Unless you
specify otherwise, the compiler will put your compiled code
in memory starting with address. To tell the compiler where
you want your program compiled, give the following two
commands to the Monitor right before you compile:

        SET 14=<address>
        SET $491=<address>

where

        <address> is the starting address for the compiled
                   code.

(In case of problems see notes in part IV, chapter 7.3.)

2.1 Comments, SET, DEFINE

Neither comments, the SET directive, nor the DEFINE
directive generate any machine code. This is because they
do not do anything at run time, and so are not required.

2.2 Variable Allocation

Information on variables is stored in two different
locations by the ACTION! Compiler - in the code itself and
in the symbol table. The symbol table is discussed later.

Variables are stored in front of the machine code where
they are used. Some variables are declared before the first
routine is entered. These variables (called global
variables) can be used by any succeeding routine. They need
no additional declaration within the routine.

The allocated variables are assigned space according to the
definition of the basic data types. The following should
help your understanding of data allocation.

data type	allocated	comments
BYTE	1 byte	fundamental type
CHAR	1 byte	fundamental type
CARD	2 bytes	fundamental type
INT	2 bytes	fundamental type
ARRAY	fundamental type size times number of elements	extended type
TYPE	sum of sizes of fundamental types,as given in declaration	extended type
string	all characters in the string plus a preceding byte to note length	each string is allocated separately even if set equal to the same identifier

2.3 Routines

The compiler allocates space for routines (procedures and functions) following that space allocated to the declared global variables. The variables declared local to a given routine precede the executable language statements in that routine. Program text (statements within procedures and/or functions) is evaluated and converted directly into machine code.

2.4 INCLUDEd Programs

Programs can be INCLUDEd at any place in the program. Of course, the INCLUDEd text must not conflict with the text currently being processed. The things to watch out for are conflicting identifiers and out-of-context insertions. When errors are detected in the INCLUDEd text, they are usually displayed in the message area. The error # is always shown in the Monitor's command line and the bell sounds.

2.5 Additional global variables - MODULE

Additional global variables, arrays, and records can be
added, as needed, through the use of the MODULE key word.
The variables are assigned space following the last
previous routine. The identifiers are also included in the
compiler's global symbol table.

2.6 Symbol Tables

The ACTION! Compiler maintains two symbol tables -- one for
the global variables and one for the local variables from
the last-compiled routine. The symbol tables are accessible
from the ACTION! Monitor through the '?', '*', and SET
commands (see Part III). They are also used by the ACTION!
Compiler whenever a variable's address is required.

The Compiler allocates 8 memory pages (2K) for these
tables, located right at the top of available memory.
Because they are placed there, you can wipe them out if you
run a program which changes into a graphics mode which
requires more memory than graphics Ø. This means that you
will not be able to go back to the Monitor during program
execution and look at the values in your variables. The
Compiler will have no record of their existence since you
just overwrote them.

Chapter 3: Using The Options Menu

The options menu offers you several ways to enhance or
alter the performance of the ACTION! compiler. The various
options are discussed here and in part III. The options are
also summarized in Appendix G.

Increasing compiler speed:

    You can gain at least a 30% improvement in compilation
    speed by using the options menu to turn off the screen
    display during both disk I/O and program compilation.
    Simply press 'N<RETURN>' to the 'Screen?' prompt in
    the options menu.

    NOTE: this also turns off the screen for other ACTION!
    system functions, so you should turn the display back
    on after you have finished compiling.

Turning the bell off:

    When you are debugging a new program and have lots of
    errors, such as typographical errors, you might want
    to turn the bell off. Simply press 'N<RETURN>' to the
    'Bell?' prompt in the options menu.

Making the Compiler case sensitive:

    Sometimes, particularly as you get more sophisticated
    in your programming style, you might desire that the
    compiler help you in your programming by reminding you
    whenever you forget to enter an ACTION! key word in
    upper case. You also might wish to benefit from the
    increased flexibility of using different or mixed
    cases in your identifiers. You can do both by pressing
    'Y<RETURN>' to the options menu prompt 'Case
    sensitive?'.

    Use of this option is not necessary to successful
    ACTION! Programming. However, it is useful as an aid
    to documentation and in providing a much greater
    diversity in identifiers.

Listing the compiled code:

    You can command the Compiler to list each program line
    as it is evaluated. This may seem unnecessary because
    most errors which occur are noted and displayed on the
    screen during the compiling process. However, you
    might have a long program which includes routines from
    other sources (remember the INCLUDE command?). If this
    is so, then you might never be able to get the source
    code together for a complete listing otherwise. You
    can get such a listing, and even redirect it to the
    printer (see part VI, section 7.9). To enable the
    listing, press 'Y<RETURN>' to the 'List?' options menu
    prompt.

Chapter 4: Technical Considerations

4.1 Overflow and Underflow

The ACTION! Compiler does no checks for mathematical
overflow or underflow. "What is overflow and underflow
anyway?" They are opposite sides of the same coin.

If you have a BYTE variable which currently equals 255, and
you add 1 to it, you will not get 256 (because a single
byte can only contain values up to 255), you will get Ø.
Similarly, if you are using the decimal system, and only
have two digits of display, you can run into the same
problem if you add 1 to 99. You know that it equals 1ØØ,
but you only have two digits of display, so you see "ØØ".

Underflow is the exact opposite of this. If you subtract 1
from Ø, you get 255.

As mentioned in part IV, section 4.2, some of the
mathematic operators result in a specified type of output,
so you can sometimes avoid the above problems by making use
of these automatic type changes.

Likewise, shift operations can cause overflow and flow. A
shift of the contents of a variable produces similar (but
not identical) results to those achieved by multiplying or
dividing by 2.

4.2 Type Compatibility and Boundary Checking

You must also be careful because the ACTION! compiler
supports no boundary checking of simple variables or
ARRAYs. This is deliberately done in order to allow you
more flexibility in your data manipulation. The price for
this freedom is increased vigilance. You must set up and
maintain your own procedures for checking boundary limits
and the error-handling responses. This is another good
place for a standard set of subroutines which can be
INCLUDEd.

4.3 Channel 7 Restriction

When you enter the ACTION! system, it opens channel 7 for
reading from the keyboard (K:). You may use this channel
for this purpose, but do not alter its attributes by
reOpening or Closing it.

NOTE: if you do make use of channel 7 (and assume that its
already open), your programs will not run without the
ACTION! cartridge.

4.4 Available space

You might be working on a big program and suddenly find
that you are out of space. When this happens, you can do
one of three things, depending on what you are doing at the
moment when the error appears.

If you are Editing:

     Immediately save your file (<CTRL><SHIFT>W), go to the
     Monitor, and reboot the system (BOOT). Then you may go
     back to the Editor and read your file back in.

If you are Compiling:

     Go to the Editor and save your program. Then go back
     to the Monitor, reboot the system, and compile your
     program from the storage device (disk, cassette,
     etc.).

```
                    Part VI:  The ACTION! Library
                    ----------------------------
```

The ACTION! Programming Environment

Part VI: The ACTION! Library

Chapter 1: Introduction

The ACTION! library makes it possible for you to do a lot
of common I/O and graphics routines without having to write
them first. The ACTION! cartridge contains almost 7Ø
prewritten routines which you can call as though they were
routines written by you. This convenience can save you
quite a bit of time and effort whether you are a beginning
or advanced programmer.

1.1 Vocabulary

Most of the vocabulary used in this part has been defined
previously, but there are two terms we will use often which
require some discussion - IOCB and channel.

IOCB stands for "Input Output Control Block". The CIO
(Central I/O) uses IOCBs to perform I/O functions. The
ACTION! library I/O routines set up an IOCB to tell the CIO
what it (the routine) wants done, and then makes a direct
call to CIO.

The IOCBs are numbered (Ø - 7). When you use routines which
require channel numbers, the number is actually the number
of the IOCB which contains the information about a given
peripheral device. That does not mean that certain IOCBs
handle certain peripherals. You must set up one of the
IOCBs so that it will handle the peripheral you want it to.
This is done using the Library routine "Open", and so is
not a difficult task to accomplish.

When you see the term "default channel" it refers to the
IOCB ACTION! sets up and uses for screen display purposes.
This means that routines which do I/O using "default
channel" will get and put information from and to the
screen (device "E:").

NOTE: The default channel is channel Ø.

NOTE: For more information on IOCBs, see your Operating
System reference manual.

The ACTION! Programming Environment

1.2 Library Format

The library routines are presented in a manner which makes it very easy to understand how to use and call them. To show you what we mean, let us take one of the routines and explain what information each part of the presentation format can tell you. The routine we will look at is "Locate".

Example:

5.8 BYTE FUNC Locate

purpose: determine the color or character at a given screen location.

Format: BYTE FUNC Locate(CARD col, BYTE row)

parameters: col - is a column number valid in the current graphics mode.
              row - is a row number valid in the current graphics mode.

description:
This routine retrieves the ATASCII code of the character or the number of the color at the specified location. The registers this routine uses are incremented so as to point to the adjacent horizontal position (the first position in the next line if you Located the last position on a line). All of the Get, Put, Print, and Input routines also use these registers as references for the current cursor location, so you can use this to move to any position and then use another routine to manipulate what is there.

The first thing you see is the section number and name of the routine, including what type of routine it is (in this case a BYTE FUNCtion). This is followed by a short description of the purpose of the routine. The format of the routine itself is then given in the form of a routine declaration. The declaration form is used instead of the form used to call that routine because it tells you more information about the routine in question, including:

        1) the routine's type (PROC or FUNC)
        2) all the parameters
        3) the data type of each parameter

After the format of the routine is given the parameters required by that routine are explained one by one. The last piece of information is a description which discusses the use of the routine in general and its performance in certain special conditions.

                   Chapter 2: Output Routines

The ACTION! Library provides an extremely extensive group
of routines to put both numeric and string data out to any
channel.

The two basic output routines -- Print and Put -- have
options which allow you to direct the output to a specific
channel and/or output an EOL (End of Line, a.k.a. <RETURN>)
following the data. We will go into these options in more
detail in the following sections.

2.1 The Print Procedures

The procedures we are about to discuss all have one thing
in common: they begin with the word "Print". From this
alone you can tell that they print something out somewhere,
but who knows what and where? The answers to these
questions can he found by looking at the option(s) tagged
onto the end of the word "Print".

These options all consist of a single letter, but you can
employ up to three options at one time because different
options control different aspects of the output. "Is this
ever confusing!" It might seem that way, but let us look at
the format of Print to see how these options are grouped:

        Print<data type>{D}{E}(<parameters>)

where
        Print   is the basic function name.

        <data type>   tells what type of data you want to
                      output. The options here are:
                      B   (BYTE type data)
                      C   (CARD type data)
                      I   (INT type data)
                      <nothing> (a string)

        D   stands for "device", and is used when
                      you want to define which device
                      (channel) you want the output to go
                      to.

        E   stands for EOL (End Of Line), and is
                      used to output a <RETURN> after the
                      data.

        <parameters> are the parameters required by the
                      procedure, and range in number.
NOTE: Both the 'D' and 'E' are optional, but a data type is

always specified (because 'a string' is assumed to be the
type of data output if no type is explicitly given).

From the above format you can see that the following are
all the possible Print routines:

		strings	BYTEs	CARDs	INTs
		-------	-----	-----	----
No Options	-	Print	PrintB	PrintC	PrintI
EOL	-	PrintE	PrintBE	PrintCE	PrintIE
To Device	-	PrintD	PrintBD	PrintCD	PrintID
Both Options	-	PrintDE	PrintBDE	PrintCDE	PrintIDE

Notice that we have grouped the procedures according to the
type of data which they output. This is the way in which we
group them in the following sections, with each section
giving the purpose, format, parameters, and discussion for
each option of the Print procedure basic to that type of
data.

There is one Print procedure not in the above list because
it is a very special case as far as output is concerned.
Its name is PrintF, and it allows you to format output
which contains numbers and strings. A separate section is
devoted to this routine alone.

2.1.1 Printing Strings

There are four string printing procedures, thus making all
the options discussed in the previous section available.

purpose: to print out a string, using some format
             options

formats: PROC Print(<string>)
             PROC PrintE(<string>)
             PROC PrintD(BYTE channel, <string>)
             PROC PrintDE(BYTE channel, <string>)

parameters: <string> - is either a string constant with
                         double quotes or the identifier
                         of a BYTE ARRAY (which you want
                         printed out as a string)
             channel - is a valid channel number (Ø - 7)

description:
These four procedures print out strings, thus:

        Print   outputs the string to the default channel
                 without a <RETURN> at the end.
        PrintE   outputs the string to the default channel
                 with a <RETURN> at the end.
        PrintD   outputs the string to a specified channel
                 without a <RETURN> at the end.
        PrintDE outputs the string to a specified channel
                 with a <RETURN> at the end.

Their usage is very straightforward and simple, but you
must remember that, with the procedures which require a
channel, the channel must first be opened.

2.1.2 Printing BYTE Numbers

The following four procedures are used to print BYTE type
data in decimal format. They start with the 'PrintB' base,
and then add the possible options.

purpose: to output one byte of data as a decimal
              number.

formats: PROC PrintB(BYTE number)
              PROC PrintBE(BYTE number)
              PROC PrintBD(BYTE channel, number)
              PROC PrintBDE(BYTE channel, number)

parameters: number - is an arithmetic expression (re-
                       member that arithmetic expressions
                       can simply be a constant or
                       variable name).

              channel - is a valid channel number ($\emptyset$ - 7)

description:
The above procedures output BYTEs as follows:

        PrintB   outputs the byte to the default channel
                   without a <RETURN> at the end.
        PrintBE   outputs the byte to the default channel
                   with a <RETURN> at the end.
        PrintBD   outputs the byte to a specified channel
                   without a <RETURN> at the end.
        PrintBDE   outputs the byte to a specified channel
                   with a <RETURN> at the end.

2.1.3 Printing CARD Numbers
----

purpose: to output numbers as CARDs in decimal format.

formats: PROC PrintC(CARD number)
              PROC PrintCE(CARD number)
              PROC PrintCD(CARD channel, number)
              PROC PrintCDE(CARD channel, number)

parameters: number - is an arithmetic expression (re-
                       member that arithmetic expressions
                       can simply be a constant or varia-
                       ble name).
              channel - is a valid channel number (Ø - 7)

description:
The above procedures output CARDs as follows:

        PrintC   outputs the CARD to the default channel
                  without a <RETURN> at the end.
        PrintCE   outputs the CARD to the default channel
                  with a <RETURN> at the end.
        PrintCD   outputs the CARD to a specified channel
                  without a <RETURN> at the end.
        PrintCDE outputs the CARD to a specified channel
                  with a <RETURN> at the end.

2.1.4 Printing INT Numbers
----

purpose: to output numbers as INTs in decimal format.

formats: PROC PrintI(INT number)
               PROC PrintIE(INT number)
               PROC PrintID(INT channel, number)
               PROC PrintIDE(INT channel, number)

parameters: number - is an arithmetic expression (re-
                        member that arithmetic expressions
                        can simply be a constant or varia-
                        ble name).
               channel - is a valid channel number (∅ - 7)

description:
The above procedures output INTs as follows:

        PrintI   outputs the INT to the default channel
                    without a <RETURN> at the end.
        PrintIE   outputs the INT to the default channel
                    with a <RETURN> at the end.
        PrintID   outputs the INT to a specified channel
                    without a <RETURN> at the end.
        PrintIDE   outputs the INT to a specified channel
                    with a <RETURN> at the end.

2.1.5 PROC PrintF - Formatted Output

The PrintF procedure allows you to output numbers and
strings on the same line through the use of a "format
control string". This string tells the procedures exactly
how you want the output to look.

purpose: formatted output of data

format: PrintF("<control string>", <data>|:, <data>:|)

arguments: <control string> - the control string is made
                      up of format controls and string
                      text. The text is output directly,
                      and the controls (maximum of 5) give
                      information for outputting the <data>
                      parameters given.
           <data> - is an arithmetic expression, which
                      will be formatted according to its
                      format control. The first control
                      tells how to output the first <data>,
                      the second control tells how to
                      output the second <data>, and so on.

description:
This is a sophisticated procedure enabling you to output
formatted data to the default channel. Up to five different
data elements can be interspersed into a string, each with
its own output format. The format controls are as follows:

   <control>   formatted data type
   ---------   -------------------
     %S   (output data as a string)
     %I   (output data as an INT)
     %U   (output data as an Unsigned CARD)
     %C   (output data as a CHARacter)
     %H   (output data in unsigned hexadecimal)
     %%   (output the '%' character)
     %E   (output an EOL (<RETURN>))

Notice that two of the controls (%E and %%) do not
manipulate or require data elements. They are used to
change the page formatting, not the data element
formatting.

A maximum of five controls are allowed, and each data
element requires its own control.

Characters in the control string which are not themselves
controls are output directly; that is, exactly as they are
in the string.

2.2 The Put Procedures

The "Put" group of library routines are used to output
single characters (i.e., output BYTE type data as an
ATASCII character). These routines use options very similar
to those in "Print", and so the options need not be re-
introduced here.

purpose: to output a single ATASCII character, using
              specified format options.

formats: PROC Put(CHAR character)
              PROC PutE()
              PROC PutD(BYTE channel, CHAR character)
              PROC PutDE(BYTE channel)

parameters: character - is an arithmetic expression
                           (remember that arithmetic
                           expressions can simply be a
                           constant or variable name).
              channel - is a valid channel number(Ø - 7)

description:
These procedures output characters as follows:

        Put   outputs the character to the default channel
                without a <RETURN> at the end.
        PutE   outputs an EOL (<RETURN>) character to the
                default channel.
        PutD   outputs the character to a specified channel
                without a <RETURN> at the end.
        PutDE   outputs a <RETURN> to a specified channel.

                    Chapter 3: Input Routines

In this chapter we discuss the routines which complement
the Print and Put routines; that is, they input data from
somewhere. Similar to the Output routines, the type of data
that is input and where it comes from is defined through
the use of options.

'Input' and 'Get' are the input routines, and each has its
own set of options very similar to those available in the
output routines.

The Input routines are grouped into two categories: those
which input numeric data, and those which input string
data. Each will be dealt with separately.

There is only one Get routine (GetD), and it will be
discussed in the last section of this chapter.

3.1 Numeric Input

The following six functions allow you to input any type of
numeric data from any channel. We have grouped them all
together because they are very easy to understand and so do
not require separate sections, as did the routines used to
output numbers did.

purpose: to input numeric data

formats: BYTE FUNC InputB()
              BYTE FUNC InputBD(BYTE channel)
              CARD FUNC InputC()
              CARD FUNC InputCD(BYTE channel)
              INT FUNC InputI()
              INT FUNC InputID(BYTE channel)

parameters: channel - is a valid channel number (Ø - 7)

description:
The functions input data as follows:

    InputB   inputs a BYTE number from the default channel.

    InputBD inputs a BYTE number from a specified channel.

    InputC   inputs a CARD number from the default channel.

    InputCD inputs a CARD number from a specified channel.

    InputI   inputs an INT number from the default channel.

    InputID inputs an INT number from a specified channel.

The ACTION! Programming Environment

3.2 String Input

String inputting is accomplished by suffixing the "Input"
base with the character "S". There are three such
procedures in the ACTION! Library, and they allow you to
input a string from any channel and/or define the maximum
length of the input string.

purpose: to input string data

formats: PROC InputS(<string>)
             PROC InputSD(BYTE channel, <string>)
             PROC InputMD(BYTE channel, <string>, BYTE max)

parameters: <string> - is the identifier of a BYTE
                         ARRAY.
             channel - is a valid channel number (∅-7)
                         max is the maximum length
                         allowable for the input string.
                         The string is truncated to 'max'
                         length if it is too long.

description:
Here is an outline of what each procedure does:

        InputS   inputs a string of up to 255 characters
                  from the default channel.
        InputSD   inputs a string of up to 255 characters
                  from a specified channel.
        InputMD   inputs a string of up to 'max' characters
                  from a specified channel.

NOTE: When using the string input library functions
(InputS, InputSD, and InputMD), there must be room in the
string for the termination EOL, even though the resulting
string length will not include it.

3.3 CHAR FUNC GetD
----

purpose: to input a single character from a given
             channel.

format: CHAR FUNC GetD(BYTE channel)

parameters: channel - is a valid channel number (∅ - 7)

description:
This function is used to get one character from the device
specified by 'channel'. The character is returned through
the function as its ATASCII character set number.

Chapter 4: File Manipulation Routines

This chapter is devoted to those routines which deal with
external devices (printer, disk drive, cassette, etc.).
With these routines you can open a channel (an IOCB), close
a channel, and do extensive disk file manipulation.

4.1 PROC Open
----

purpose: set up an IOCB channel to allow I/O using a
             peripheral device.

format: PROC open(BYTE channel, <filestring>, BYTE
             mode, aux2)

parameters: channel - is a valid channel number (∅ — 7)
             <filestring> - is the string constant (or
                     array identifier of that string
                     constant) used as the device (D:,
                     P:, S:, etc.) being opened on the
                     given channel (IOCB) number. "D:"
                     files also require a filename.
             mode   - is the number designating the type
                     of I/O, thus:
                                 4 - read only
                                 6 - read directory
                                 8 - write only
                                 9 - write append
                                12 - read/write (update)
             aux2   - a device dependent value (usually
                     zero)

description:
This procedure opens a given channel the device specified
in <filestring>. The I/O mode can be set (see 'mode' above
for the number codes). Any device dependent codes are
passed through 'aux2'.

WARNING: Do not Open channel 7, because it is used by the
ACTION! system to do its own screen input. You can use
channel 7 in your program for getting characters from K:,
but, since that assumes that channel 7 is open, you need
the ACTION! cartridge to run the compiled version of the
program (because ACTION! opens channel 7 to K:).

4.2 PROC Close
----

purpose: to close an IOCB channel to a device

format: PROC Close(BYTE channel)

parameters: channel - is a valid channel number (Ø - 7)

description:
This procedure closes the specified channel. At the end of
a program you should always close any devices you have
opened in the course of that program.

NOTE: DO NOT Close channel 7, as ACTION! uses it.

4.3 PROC XIO
----

purpose:

format: PROC XIO(BYTE chan,Ø,cmd,aux1,aux2,<filestring>

parameters: chan - is a valid channel number (Ø - 7)
            cmd - is the equivalent of the IØCB COMMAND
                   byte (ICCOM in OS/A+ and DOS XL)
            auxl - is the first auxiliary byte in the IOCB
                   (ICAUXl in OS/A+ and DOS XL)
            aux2 - is the second auxiliary byte in the IOCB
                   (ICAUX2 in OS/A+ and DOS XL)
            <filestring> - is a character string specifying
                           a standard device (with a file-
                           name in the case of "D:").

description:
This procedure is a system call designed to provide access
to DOS. Those of you familiar with ATARI BASIC, BASIC A+,
or BASIC XL will recognize XIO as a direct translation of
BASIC's XIO statement.

Rather than give a complete list of all the possible uses
of XIO here, we will refer you to Chapter 8 of either the
OS/A+ or DOS XL manual. The ACTION! XIO procedure can
perform all the system commands listed therein other than
NOTE, POINT, and the various data transfer operations --
all of which are available via other ACTION! Library
routines.

NOTE: The "Ø" given as the second parameter is required.

4.4 PROC Note
----

purpose: to return the current file sector and byte
             offset within that sector on a specified disk
             drive.

format: PROC Note(BYTE chan,
                       CARD POINTER sector,
                       BYTE POINTER offset)

parameters: chan - is a valid channel number (Ø - 7)
             sector - is a pointer to the sector number
                      variable.
             offset - is a pointer to the byte offset
                      variable.

description:
This procedure returns the disk sector and byte offset
within that sector of the next byte to be read or written
(i. e., it returns the value of the disk file pointer).

4.5 PROC Point
----

purpose: to set the disk file pointer (sector and byte
             offset) to allow random file access.

format: PROC Point(BYTE chan,
                        CARD sector,
                        BYTE offset)

parameters: chan - is a valid channel number (Ø - 7)
             sector - is a valid sector number (1 — 72Ø)
             offset - is the byte offset within that
                      sector.

description:
This procedure allows you to set the disk file pointer to
any location within a disk file, thus enabling random
access of information.

NOTE: The disk file must have been Opened mode 12 (update)
for the Point routine to work.

Chapter 5: Graphics and Game Controllers

The ACTION! Library contains quite a few routines designed
specifically to make game writing (using visual and sound
effects) easy and quick. At your fingertips you have the
ability to manipulate bit-map graphics (i.e., the BASIC
graphics modes), the myriad of sounds available on the
ATARI, and get information about the game controllers (both
paddle and joystick).

Since the description of each routine best illustrates its
usage, we will jump right into the routines themselves
without further discussion.

5.1 PROC Graphics
----

purpose: to enable bit-map ATARI graphics.

format: PROC Graphics(BYTE mode)

parameters: mode - is the number of the graphics mode, as
             in the BASIC 'Graphics' routine (see table
             below).

description:
This procedure is exactly equivalent to the BASIC command
of the same name, and allows you access to the many varied
graphics modes available on the ATARI.

The following table gives some information about the 9 base
graphics modes. These modes are all split screen; to get
full screen, add 16 to the base mode number; to preserve
the current screen as you change modes, add 32 to the base
mode number; to get both of these options, add 48 to the
base mode number.

Gr. Mode	Mode Type	Rows	(split) Cols	(full) Cols	Num of Colors
Ø	TEXT	4Ø	N/A	24	2
1	TEXT	2Ø	2Ø	24	5
2	TEXT	2Ø	1Ø	12	5
3	GRAPHICS	4Ø	2Ø	24	4
4	GRAPHICS	8Ø	4Ø	48	2
5	GRAPHICS	8Ø	4Ø	48	4
6	GRAPHICS	16Ø	8Ø	96	2
7	GRAPHICS	16Ø	8Ø	96	4
8	GRAPHICS	32Ø	16Ø	192	1/2

5.2 PROC SetColor
----

purpose: sets the specified color register to the color
             given by 'hue' and 'luminance'.

format: PROC SetColor(BYTE register,hue,luminance)

parameters: register - is one of the five color registers
                         (Ø – 4)
             hue   - is the hue of the color.
             luminance - is the luminance of the color.

description: This routine allows you to set the color of a
specific color register, and so manipulate the colors
displayed in a given mode. The following tables give some
information pertinent to the usage of this procedure.

SetColor hue num.	Color	SetColor hue num.	Color
Ø	Gray	8	Blue
1	Gold	9	Light Blue
2	Orange	1Ø	Turquoise
3	Red-Orange	11	Green-Blue
4	Pink	12	Green
5	Purple	13	Yellow-Green
6	Purple-Blue	14	Orange-Green
7	Blue	15	Light Orange

The above table shows the 16 hues available on the ATARI,
and their numeric code for use as the 'hue' parameter of
the SetColor procedure.

Register	Default Color	Default Luminance	Color
Ø	2	8	Orange
1	12	1Ø	Green
2	9	4	Dark Blue
3	4	6	Pink or Red
4	Ø	Ø	Black

This table shows which colors are the defaults used when
you do not specify your own color for a given SetColor
'register'.

NOTE: Colors may vary depending upon the television or
monitor type, condition, and adjustment.

The luminance value (a measure of the "brightness" of a
color) ranges between Ø and 15, where Ø is darkest and 15
is brightest.

5.3 BYTE color

'color' is not a library routine, but a variable defined in
the library for use with the 'Plot', 'DrawTo', and 'Fill'
library procedures. After you pick your graphics mode
(using 'Graphics') and set up the color registers (using
'SetColor'), you can plot and draw in that mode using any
of the colors you have specified by first using the
assignment:

                        color=<number>

where <number> is related to the color register containing
the color you want to use. The following table shows this
relationship for the different graphics modes. For every
group of related modes, each SetColor 'register' is
followed by its associated 'color' <number>, and some
descriptive comments.

Graphics Mode	SetColor 'register'	Color number	Description and Comments
Ø and all text windows	Ø	N/A	
	1	N/A	Character luminance
	2	N/A	Background
	3	N/A	
	4	N/A	Border
1,2	Ø	N/A	Character
	1	N/A	Character
	2	N/A	Character
	3	N/A	Character
	4	N/A	Background,Border
3,5,7	Ø	1	Graphics Point
	1	2	Graphics Point
	2	3	Graphics Point
	3	--	--
	4	Ø	Gr. Pt.,Border,Backgrd
4,6	Ø	1	Graphics Point
	1	--	--
	2	--	--
	3	--	--
	4	Ø	Gr. Pt.,Border,Backgrd
8	Ø	--	--
	1	1	Gr. Pt., luminance
	2	Ø	Gr. Pt.,Background
	3	--	--
	4	--	Border

5.4 PROC Plot
----

purpose: to   position   the   cursor   at   a   specified
             location, and then display a color using the
             library variable 'Color'.

format: PROC Plot(CARD col,BYTE row)

parameters: col - is the horizontal column number of the
                   point being plotted.
             row - is the vertical row number of the point
                   being plotted.

description:
This procedure is used in graphics modes 3 - 8 to plot a
point on the screen. The size of the point displayed
depends on the graphics mode, and the color of the point
depends on the current value of the library variable
'Color' (see previous section).

5.5 PROC DrawTo
----

purpose: must be proceeded by a 'Plot') to draw a line
             between   the   point   just   Plotted   and   the
             specified position.

format: PROC DrawTo(CARD col,BYTE row)

parameters: col - is the horizontal column number of the
                   end point of the line.
             row - is the vertical row number of the end
                   point of the line.

description:
This procedure is used in graphics modes 3 - 8 to draw a
line from the point just plotted (using 'Plot') and the
position given by the parameters. The color of the line
depends on the current value of the library variable
'Color' (see section 5.3).

5.6 PROC Fill
----

purpose: (must be preceded by a 'Plot') fills a box
              with a color.

format: PROC Fill(CARD col,BYTE row)

parameters: col - is the horizontal column number of the
                  lower left corner of the box being
                  filled.
            row - is the vertical row number of the lower
                  left corner of the box being filled.

description:
This allows you to make boxes of color in graphics modes 3
- 8. The upper left corner of the box is defined by the
position of the 'Plot' immediately before the 'Fill', and
the lower left corner is given by the parameters. The color
used is decided by the contents of the library variable
'Color'.

5.7 PROC Position
----

purpose: to position the cursor anywhere on the screen

format: PROC Position(CARD col),BYTE row)

parameters: col - is the horizontal column number of the
                  position desired.
            row - is the vertical row number of the
                  position desired.

description:
This procedure sets the cursor location to the specified
position in any graphics mode. The library routines Print,
Put, Input, and Get use the cursor registers this command
sets when doing their respective functions.

5.8 BYTE FUNC Locate
----

purpose: determine the color or character at a given
             screen location.

Format: BYTE FUNC Locate(CARD col, BYTE row)

parameters: col - is a column number valid in the current
                  graphics mode.
            row - is a row number valid in the current
                  graphics mode.

description:
This routine retrieves the ATASCII code of the character or
the number of the color at the specified location. The
registers this routine uses are incremented so as to point
to the adjacent horizontal position (the first position in
the next line if you Located the last position on a line).
All of the Get, Put, Print, and Input routines also use
these registers as references for the current cursor
location, so you can use this to move to any position and
then use another routine to manipulate what is there.

The ACTION! Programming Environment

5.9 PROC Sound
----

purpose: to enable the sound capabilities of the ATARI.

format: PROC Sound(BYTE voice,pitch,distortion,volume)

parameters: voice   - is one of the four voices
                          available on the ATARI (Ø - 3).
              pitch   - is the frequency of the sound.
                          The lower the number, the
                          higher the pitch.
              distortion - is a measure of the sound's
                          "fuzziness" (Ø - 14, even
                          values).
              volume   - is the volume of the sound(Ø - 16)

description:
This procedure allows you to control the sound-generating
apparatus on the ATARI, much like the BASIC command of the
same name. Distortion value 1Ø is the only one useful for
making music. The others are useful for airplane, race-car,
etc. sound effects.

Here is a table for various musical notes using distortion
1Ø.

	'pitch'	Note(s)		'pitch'	Note(s)
HIGH	29	C		91	F
NOTES	31	B		96	E
	33	A# or Bb		1Ø2	D# or Eb
	35	A		1Ø8	D
	37	G# or Ab		114	C# or Db
	4Ø	G	MIDDLE C	121	C
	42	F# or Gb		128	B
	45	F		136	A# or Bb
	47	E		144	A
	5Ø	D# or Eb		153	G# or Ab
	53	D		162	G
	57	C# or Db		173	F# or Gb
	6Ø	C		182	F
	64	B	LOW	193	E
	68	A# or Bb	NOTES	2Ø4	D# or Eb
	72	A		217	D
	76	G# or Ab		23Ø	C# or Db
	81	G		243	C
	85	F# or Gb			

5.1Ø PROC SndRst
----

purpose: to reset all the sound voices.

format: PROC SndRst()

parameters: none

description:
This procedure resets all the sound voices to produce no
sound.

5.11 BYTE FUNC Paddle
----

purpose: to return the current numeric value (position)
            of one of the paddles.

format: BYTE FUNC Paddle(BYTE port)

parameters: port - is the port number (Ø - 7) of the
                   desired paddle.

description:
This function returns the current value of the specified
paddle port.

5.12 BYTE FUNC PTrig
----

purpose: to determine whether a paddle trigger has been
            pressed.

format: BYTE FUNC PTrig(BYTE port)

parameters: port - is the port number (Ø - 7) of the
                   desired paddle.

description:
This function returns the current value of the given
paddle's trigger. A value of Ø is returned if the trigger
is pressed, otherwise the value returned is non-zero.

The ACTION! Programming Environment

5.13 BYTE FUNC Stick
----

purpose: to return the current numeric value of a
            specified joystick.

format: BYTE FUNC Stick(BYTE port)

parameters: port - is the port number (Ø - 3) of the
                   desired joystick.

description:
This function returns the current position of the joystick,
using codes as in the following diagram.

```
                14
            1Ø   |   6
              \  |  /
               \ | /
        11 ----- 15 ----- 7
               / | \
              /  |  \
            9    |   5
                13
```
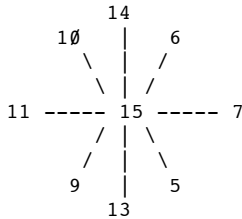
5.14 BYTE FUNC STrig
----

purpose: to determine whether a joystick trigger has
            been pressed.

format: BYTE FUNC STrig(BYTE port)

parameters: port - is the port number (Ø - 3) of the
                   desired joystick.

description:
This function returns the current value of the given
joystick's trigger. A value of Ø is returned if the trigger
is pressed, otherwise the value returned is non-zero.

Chapter 6: String Handling / Conversion

The routines discussed in this chapter allow you to
manipulate strings, change a number to a string, and change
a string into a number. No further discussion is necessary,
since the routine descriptions speak for themselves.

6.1 String Handling Routines

The following four routines make possible some advanced
string manipulation, including string comparison, string
copying, and substring insertion. There is one caution,
however, and that is: remember that the maximum length of a
string is 255 characters, so do not try to use these
routines to create or to manipulate big CHARacter arrays.

6.1.1 INT FUNC SCompare
----

purpose: to compare alphabetically two strings.

format: INT FUNC SCompare(<string1>,<string2>)

parameters: <string1> - is a string with double quotes, or
                        the identifier of a CHAR ARRAY
                        which is a string.
            <string2> - is a string with double quotes, or
                        the identifier of a CHAR ARRAY
                        which is a string.

description:
This function returns a value dependent on the following
table:

            comparison   value returned
            ----------   --------------
      <string1> < <string2>   value < Ø
      <string1> = <string2>   value = Ø
      <string1> > <string2>   value > Ø

The comparison is alphabetic, so this is a good way to
alphabetize a list of strings.

6.1.2 PROC SCopy
----

purpose: to copy one string into another.

format: PROC SCopy(<dest>,<source>)

parameters: <dest>   - is   the   identifier   of   the
                       destination string (CHAR ARRAY for
                       the string copy.
            <source> - is the string with double quotes
                       or identifier of the CHAR ARRAY
                       used as the source string for the
                       copy.

description:
This procedure copies the contents of <source> into <dest>.
Please make sure that <dest> and <source> are compatible in
size.

6.1.3 PROC SCopyS
----

purpose: to copy part of a string into another string.

format: PROC SCopyS(<dest>,<source>, BYTE start,stop)

parameters: <dest>   - is   the   identifier   of   the
                        destination string (CHAR ARRAY)
                        for the string copy.
            <source> - is the string with double quote or
                        identifier of the CHAR ARRAY used
                        as the source string for the copy.
            start   - is the starting point in <source>
                        for the copy.
            stop   - is he stopping point in <source>
                        for the copy. If 'stop' is greater
                        than the length of <source>, it is
                        changed to equal the length of
                        <source>.

description:
This procedure will copy the elements of <source> from
element 'start' to element 'stop' into <dest>. In essence,
this works just like SCopy, but copies only a part of
<source> instead of the whole thing.

6.1.4 PROC SAssign
----

purpose: to copy one string into part of another
            string.

Format: PROC SAssign(<dest>,<source>, BYTE start,stop

parameters: <dest>   - is   the   identifier   of   the
                       destination string (CHAR ARRAY for
                       the string copy.
            <source> - is the string with double quotes
                       or identifier of the CHAR ARRAY
                       used as the source string for the
                       copy.
            start   - is the starting point in <dest>
                       for the copy.
            stop   - is the stopping point in <dest>
                       for the copy. If 'stop' is greater
                       than the length of <source>, it
                       will be changed to the length of
                       <source>.

description:
This procedure is used to copy one string (<source>) into
part of another (<dest>). <source> will be copied starting
at element 'start' of <dest>, and the copying will stop at
element 'stop' of <dest>. If the space allowed (stop-
start+l) in <dest> is greater than the length of <source>,
then 'stop' will be changed to the length of <source>.

The copying this procedure does will overwrite the old
elements of <dest> as it puts in <source>.

NOTE by GBXL: The original text for 'stop' and
'description' could be misleading for non-native readers
and therefore has been changed.

The ACTION! Programming Environment

6.2 Number to String Conversions

The following three procedures convert the number given as
a parameter into a character string. There is one procedure
for each of the numeric data types.

purpose: to change a number into a character string.

format: PROC StrB(BYTE number,<strinq>)
           PROC StrC(CARD number,<string>)
           PROC StrI(INT number,<string>)

parameters: number   - is an arithmetic expression (re-
                        member that arithmetic expressions
                        can simply be a constant or
                        variable name).
            <string> - is the identifier of a CHAR ARRAY.

description:
These procedures turn BYTE, CARD, or INT values into
character strings composed of the digits of the given
number.

6.3 String to Number Conversions
----

purpose: to convert a string composed of digits into
            a number.

format: BYTE FUNC ValB(<string>)
            CARD FUNC ValC(<string>)
            INT FUNC ValI(<string>)

parameters: <source>- is a string with double quotes or
                       identifier of a CHAR ARRAY,
                       composed of digits ("Ø" — "9")
                       only.

description:
These functions will return the numeric value (BYTE, CARD
or INT, depending on the function used) of the given
string.

Chapter 7: Miscellaneous Routines

This chapter contains those routines which do not really fit into any category, but are useful nonetheless. The routines themselves are:

```
    Rand      - a random number generator
    Break     - a routine useful when debugging
    Error     - a system routine you can replace
    Peek      - view a byte of memory
    PeekC     - view two bytes of memory (as a CARD)
    Poke      - put a BYTE value into memory
    PokeC     - put a CARD value into memory
    Zero      - zero out a section of memory
    SetBlock  - fill a block of memory with a value
    MoveBlock - move a block of memory
    Device    - the "default device" variable
    Trace     - controls the 'TRACE' compile option
    List      - controls the 'LIST' compile option
    EOF       - contains EOF status for all channels
```

As you can see, the tasks these routines perform are quite diverse; hence their own chapter.

7.1 BYTE FUNC Rand
----

purpose: to generate a random number.

format: BYTE FUNC Rand(BYTE range)

parameters: range   - is the upper limit for the random
                        number.

description:
This function will return a random number between Ø and ('range'-1). If 'range' is Ø, then a random number between Ø and 255 is returned.

7.2 PROC Break
----

purpose: to stop program execution.

format: PROC Break()

parameters: none

description:
This procedure allows you to stop your program's execution
to examine variables and do other debugging. You can
continue program execution starting with the statement
following the 'Break' routine call by using the 'PROCEED'
monitor command.

7.3 PROC Error

This is the procedure the ACTION! system itself calls when
it (or CIO) encounters an error. If you want to trap your
own errors, you could write a routine to do this, and then
make ACTION! use your error routine instead of its own
simply by having the following statements in your program:

```
   PROC MyError(BYTE errcode)

   ;**** this is your error routine, and the error
   ;code number is passed to it by the ACTION! system.

   ; your error handling routines go here

   RETURN ;end of PROC MyError

   PROC main() ;your main procedure

      CARD temperr    ;holds the address of the system's
                      ;error routine (PROC Error).

      temperr=Error   ;save the address of the system
                      ;error routine.
      Error=MyError   ;make the address of the system
                      ;error routine point to the start
                      ;of your error routine.

      ;the body of your program goes here.

      Error=temperr   ;reset the address of the system
                      ;error routine back to the real
                      ;system error routine, not yours.
   RETURN ;end of program.
```

All you are really doing is changing the pointer to the system error routine so that it points to your error routine instead. You do not have to call this routine because it will be called by the ACTION! system when an error is encountered.

Notice that we saved the original error routine pointer, and then, at the end of the program, we reset that pointer (which was changed to point to your error routine) back to the system error routine. This was done so that the system could again use its error routine after your program finished running.

WARNING: the capability of substituting your error routine for the system's should be used very carefully, because you might to forget to check for something in your routine, and thereby cause the entire system to crash.

7.4 BYTE FUNC Peek and CARD FUNC PeekC
----

purpose: to return the value (BYTE or CARD) at a
            given memory location.

format: BYTE FUNC Peek(CARD address)
            CARD FUNC PeekC(CARD address)

parameters: address - is the address of the memory loca-
                      tion you desire to look at.

description:
These two functions allow you to look at memory during program execution, either as a BYTE or a CARD in LSB, MSB order.

The ACTION! Programming Environment

7.5 PROC Poke and PROC PokeC
----

purpose: to insert new values (BYTE or CARD) into a
            specified memory location.

format: PROC Poke(CARD address, BYTE value)
            PROC PokeC(CARD address, value)

parameters: address - is the address of the memory loca-
                       tion you desire to change.
            value   - is the value you want put into the
                       memory   location   specified   by
                       'address'. When using PokeC, the
                       CARD value is stored in 'address'
                       and 'address'+1 in LSB, MSB order.

description:
These procedures allow you to change the contents of memory
during program execution by changing the given address to
the specified value.

7.6 PROC Zero
----

purpose: to zero out a block of memory.

format: PROC Zero(BYTE POINTER address, CARD size)

parameters: address - is a pointer to the starting
                       address of the block you want
                       zeroed.
            Size   - is the size of the block you want
                       zeroed.

description:
With this procedure you can set all the values of the
memory locations in a block to Ø. This block starts at
'address' and ends at location 'address'+'size'-1.

7.7 PROC SetBlock
----

purpose: to set the memory locations of a memory block
             to a specified value.

Format: SetBlock(BYTE POINTER address, CARD size, BYTE
                                            value)

parameters: address - is a pointer to the starting
                       address of the block you want to
                       set.
            size   - the size of the block you want to
                       set.
            value   - is the value you want the bytes in
                       the block set to.

description:
With this procedure you can set all the values of the
memory locations in a block to 'value'. This block starts
at 'address' and ends at location 'address'+'size'-1.

7.8 PROC MoveBlock
----

purpose: to move the contents of a block of memory.

format: PROC MoveBlock(BYTE POINTER dest,source, CARD size)

parameters: dest   - is a pointer to the start of the
                       destination memory block.
            source   - is a pointer to the start of the
                       source memory block.
            size   - is the size of the block you want
                       to move.

description:
This procedure moves the values in a block starting at
address 'source' and ending at address 'source'+'size'-1 to
a block starting at address 'dest' and ending at address
'dest'+'size-l. If 'dest' is greater than 'source', and
there is not 'size' space between them, then the move will
not work properly because part of the 'source' you are
trying to move is in the 'dest' space.

The ACTION! Programming Environment

7.9 BYTE device

'device' is a variable defined in the ACTION! Library, and
allows you to control the 'default channel' (device) for
I/O. The number contained by 'device' is the channel number
of the default device, so, for example, you send default
output to the printer using the following statements:

        Close(5) ;avoid a 'File already Opened' error
        Open(S,"P:",8)
        device=5

and then reset it to the screen (when you want to) using
the following statements:

        Close(5) ;close "P:"
        device=Ø

7.1Ø BYTE TRACE

This library variable allows you to control the 'TRACE'
compiler option from within your program. You must use it
with the 'SET' compiler directive, and it must come at the
beginning of your program. Setting 'TRACE' to Ø turns off
the option, and setting it to 1 turns it on.

Example: SET TRACE=Ø

7.11 BYTE LIST

This library variable controls the 'LIST' compiler. As with
'TRACE' above, this variable must be used in a 'SET'
directive, and it must come at the beginning of your
program. A Ø turns the listing off, and a 1 turns it on.

7.12 BYTE ARRAY EOF(8)

With this library variable you can found out if you have
reached the End Of File on any channel. Simply give the
number of the channel as the subscript to the EOF array.
For example, if you wanted to find out if you reached the
End of File on channel 1 (the channel must be open), then
you would use:

        IF EOF(1) THEN
             :
             :

EOF equals 1 when the End Of File has been reached,
otherwise it is Ø.

Part VII: The Action! Run Time Package

The ACTION! Programming Environment

Chapter 1: 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 ACTION! 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.

Chapter 2: How ACTION! Works

2.1 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.

2.1.1 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.

2.1.2 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.

2.1.3 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 $7ØØ bytes. Of these
$7ØØ bytes, $4ØØ 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 ($Ø495) 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).

2.2 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 'D1:' prompt by using just its name.

2.3 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.

Chapter 3: 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.

3.1 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.

3.2 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.

The ACTION! Programming Environment

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.

Chapter 4: 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.

4.1 Increasing Your Symbol Table Space

By default, Action! reserves 2K Bytes (2Ø48 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! <u>monitor</u> 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.

4.2 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 51Ø
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.

Chapter 5: 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.

5.1 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 '? $E' command to determine
the bottom of this space. Remember that [$ØE] ([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 '? $BØ'
command. Actually, location $BØ (STBASE) contains a single
BYTE value (so be sure and look at the least significant
byte of the contents of $BØ 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 $5ØØØ, I would put
these two lines as the first two lines of my program:

```
    SET $E=$5ØØØ
    SET $491=$5ØØØ
```

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

5.2 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.

Every time 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=$1ØØØ ; set CODEOFF to 4K Bytes
; assume that APPMHI contains $4ØØØ at this point
PROC P()
...
PROC Q()
P()
...
```

The compiler "knows" that the PROCedure named "P" is
located at address $4ØØØ. Yet, when it compiles PROCedure
"Q" and encounters the reference to "P", it generates the
equivalent of

```
JSR P+[CODEOFF]
  or
JSR P+$1ØØØ
  or
JSR $5ØØØ
```

Since Action! ignores any overflow/carry which results in
adding CODEOFF to an address, we could 'SET $B5=$FØØØ' and
effectively subtract $1ØØØ 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 $7ØØ. Let us further suppose that we are using
Action! with a DOS which causes a LOMEM of $21ØØ. Thus the
initial contents of APPMHI will be approximately $28ØØ
(plus a little). We might, then, start our Action! program
with the following lines:

```
SET $E=$2FØØ ; just to make the setting ...
SET $491=$2FØØ ; ...of CODEOFF easier
SET $B5=$D8ØØ ; equivalent of $B5= —$28ØØ
```

And, lo and behold, if we dumped the compiled code we would
find that we had indeed generated code designed to run at
location $7ØØ.

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 $7ØØ: 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 $7ØØ bytes "lost"
to Action!'s semi-fixed RAM by coding an offset of $F9ØØ.
The space thus gained is not huge (17ØØ bytes or so), but
it may make all the difference to you.

The ACTION! Programming Environment

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 $7ØØ 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.

5.3 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:

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

Action!:

```
  BYTE FUNC LSH3=$3ØØØ (BYTE N)
  BYTE ARRAY MASK(Ø) = $3ØØA
```

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.

Chapter 6: Compiling ROMmable Code

If you have just finished reading Section 5, 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.

6.1 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.

The ACTION! Programming Environment

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 (please see note on page 311):

```
     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! gener-
ates 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 fur-
ther examples and techniques. You will note the BYTE ARRAYs
in the beginning being generated in ROM. These are invari-
ant 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 pa-
rameters 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 exam-
ple, and will generate no actual variable storage.

6.2 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.

6.2.1 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!

The ACTION! Programming Environment

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.

6.2.2 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
desperately 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 = Realflandler
    ...
    ; 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().

6.2.3 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 responsi-
bility 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.

6.2.4 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.

Chapter 7: 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 65Ø2 low/high format. It may, of
course, thereby be an address pointer.

Addr	Label	Description
ØØØE	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).
ØØ9B	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 $5ØØ-$5FF, below.
ØØAØ to ØØAF	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 $AØ and work up. Temporaries start at $AF and work down.
ØØBØ	stBase	High byte of address of start of symbol table.

ØØB1   stGlobal   Location of 512-byte hash table for global
                  symbols.

ØØB3   stLocal   Location of 512-byte hash table for local
                  symbols.

ØØB5   codeOff   Offset between compiled-at address (as de-
                  termined by "code", location $ØE) and com-
                  piled-for address (e. g., when compiling
                  code to be placed in ROM or in place of
                  the DOS code). See Section 4.2.

ØØB7   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.

ØØCE   stG2   Used only if the "big symbol table flag"
                  (bigST, $4C4) is set. This is the address
                  of the 5l2-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.

491   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.

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

495   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.

Ø49A   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.

Ø4AD   frstChar   When a big symbol table is in use (see
                  $Ø4C4), 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".

Ø4CØ   bckgrnd   Background color. Use at your own risk.

Ø4C4   bigST   A flag. If set, the global symbol table is
                 divided into two parts (see also $Ø4AD).
                 Thus you may use a total of 51Ø global
                 symbols. See also Section 3.2 and the file
                 "BIGST.ACT" on the RunTime disk.

Ø4CB   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.

Ø5ØØ   A buffer used by the RunTime library OPEN
 -               routine. When OPEN is passed a filename,
Ø5FF   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.

Part VIII: The ACTION! Toolkit

                                                        215

Chapter 1: Introduction

Welcome to the Programmers' Tool Kit (V. 3). This diskette
contains routines written in ACTION! which extend your
ACTION! programming capabilities. The following is a list
of the files on the disk, together with a short description
of what each file does.

    ABS.ACT   a routine which will return the absolute
                    value of an INT.

    ALLOCATE.ACT   routines which allow dynamic runtime
                    memory manipulation.

    CHARTEST.ACT   routines which perform various tests and
                    functions on characters.

    CIRCLE.ACT   a circle drawing routine using neither
                    Sine nor Cosine.

    CONSOLE.ACT   a routine which both debounces the
                    console keys and allows you to tie
                    routines into them.

    IO.ACT   routines which implement some advanced
                    I/O operations.

    JOYSTIX.ACT   routines which make interpreting joystick
                    input easier.

    PMG.ACT   player/missile graphics routines.

    PRINTF.ACT   an extended version of the ACTION!
                    Library 'PrintF'.

    REAL.ACT   routines which allow you to use floating
                    point numbers.

    SORT.ACT   QuickSort for BYTE, CARD, INT, and string
                    data.

    TURTLE.ACT   an implementation of turtle graphics, ala
                    LOGO.

    GEM.DEM   a four person game written in ACTION!.

    KALSCOPE.DEM   a colorful demo of ACTION!'s speed.

The ACTION! Programming Environment

    MUSIC.DEM   a demo which creates a playable organ.

    SNAILS.DEM   a two person game translated from BASIC to ACTION!.

    WARP.DEM   a one person game which uses many of the advanced constructs and abilities of ACTION!.

There are also some files with the extension '.DMn', where 'n' is a number. These are demos of the routines in a specific file, designed to help you better understand the procedure required to make use of the Tool Kit routines.

NOTE: In most of the ACTION! source files there are global variables and procedures which contain the underline character ('_'). These variables and routines are internal to the Toolkit routines, and should be neither called nor accessed by you unless you are positive you know for what they are used.

<u>To Boot This Disk</u> simply boot your DOS disk with the ACTION! Cartridge inserted, and then put this disk in your drive. THIS D1SKETTE DOES NOT HAVE DOS ON IT AND WILL NOT BOOT DIRECTLY.

NOTE: On the latest version 3, the file ABS.ACT starts with the version number.

Chapter 2: Toolkit Routines

2.1 ABS.ACT

2.1.1 INT FUNC

Purpose: To return the absolute value of an INTeger.

Syntax: INT FUNC Abs(INT n)

Params: n - the INTeger whose absolute value is returned.

Description: This function will return the absolute value
of the INT passed to it.

2.2 ALLOCATE.ACT

The routines in this file allow you to allocate and free
blocks of memory at runtime. If you want to use this
capability, you must first call the AllocInit routine.
AllocInit expects a global CARD variable called EndProg to
contain the address of the end of your program. To do this,
compile your program, and then type the following in the
monitor immediately after compiling:

        SET EndProg=* [RETURN]

Now you can run your program.

TECHNICAL NOTE: The Alloc and Free routines operate on a
'free list'. This list gives the location and size of every
free memory block. Alloc simply removes a block from the
free list, and Free puts a block back into the list.

2.2.1 PROC AllocInit

Purpose: To set up the free list and initialize the allo-
         cation routines.

Syntax: PROC AllocInit(CARD p)

Params: p - the address of the first free memory location
         in memory.

Description: This routine is used to create the free list
so that Alloc and Free may be used. See the introduction to
this section for instructions on its use.

NOTE: If you are planning to use P/M graphics and/or bit-map graphics, you should enable the P/Ms and be in the most memory intensive graphics mode you plan to use when you call AllocInit, since it considers all memory up to MEMHI ($2E5) to be free space. (Alternatively, you can merely change the value of MEMHI.)

2.2.2 CARD FUNC Alloc

Purpose: To allocate a block of memory of a specified size, returning the address of that block.

Syntax: CARD FUNC Alloc(CARD nBytes)

Params: nBytes - the size in bytes of the block to be allocated.

Description: This routine allows you to reserve a block of memory 'nBytes' long. The starting address of the block is returned, so, for example, you could use it to allocate space for a large array at runtime, after you've determined the size array you need:

```
    PROC Test()
      CARD size
      BYTE ARRAY bigarray

      Print("Size of Array>> ")
      size=InputC()
      bigarray=Alloc(size)
    RETURN
```

NOTE: the smallest block you can allocate is 3 bytes.

2.2.3 PROC Free

Purpose: To free a block of memory which has previously been reserved using the Alloc function.

Syntax: PROC Free(CARD target,nBytes)

Params: target - starting address of the block to free.
         nBytes - length in bytes of the block to free.

Description: This procedure allows you to return a block of memory used by Alloc to the free list.

2.2.4 PROC PrintFreeList

Purpose: To print out the free list.

Syntax: PROC PrintFreeList()

Params: none

Description: This procedure will print out the current free
list, and should be used mostly for diagnostic debugging
reasons.

2.3 CHARTEST.ACT

The routines in this file are very diverse, including:

        IsAlpha - a character test
        IsUpper - a character test
        IsLower - a character test
        IsDigit - a character test
        ToUpper - a character manipulation
        ToLower - a character manipulation

2.3.1 BYTE FUNC IsAlpha

Purpose: To test a single character to see if it is a let-
         ter.

Syntax: BYTE FUNC IsAlpha(BYTE c)

Params: c - the character to be tested.

Description: This function checks c to see if it is an
alphabetic character. If it is, a 1 is returned; otherwise
a Ø is returned.

2.3.2 BYTE FUNC IsUpper

Purpose: To test a single character to see if it is an
         uppercase letter.

Syntax: BYTE FUNC IsUpper(BYTE c)

Params: c - the character to be tested.

Description: This function checks c to see if it is an
uppercase alphabetic character. If it is, a 1 is returned;
otherwise a Ø is returned.

2.3.3 BYTE FUNC IsLower

Purpose: To test a single character to see if it is a
         lowercase letter.

Syntax: BYTE FUNC IsLower(BYTE c)

Params: c - the character to be tested.

Description: This function checks c to see if it is a
lowercase alphabetic character. If it is, a 1 is returned;
otherwise a Ø is returned.

2.3.4 BYTE FUNC IsDigit

Purpose: To test a single character to see if it is a
         digit.

Syntax: BYTE FUNC IsDigit(BYTE c)

Params: c - the character to be tested.

Description: This function checks c to see if it is a digit
(Ø - 9). If it is, a 1 is returned; otherwise a Ø is
returned.

2.3.5 BYTE FUNC ToUpper

Purpose: To Change lowercase letters to uppercase.

Syntax: BYTE FUNC ToUpper(BYTE c)

Params: c - the character to be tested.

Description: This function will return the uppercase of the
character passed to it. If the character is already
uppercase, or is not alphabetic, then the character is
returned unchanged.

2.3.6 BYTE FUNC ToLower

Purpose: To change uppercase letters to lowercase.

Syntax: BYTE FUNC ToLower(BYTE c)

Params: c - the character to put into lowercase.

Description: This function will return the lowercase of the
character passed to it. If the character is already

lowercase, or is not alphabetic, then the character is returned unchanged.

2.4 CIRCLE.ACT

The circle drawing routine in this file is somewhat special, since it does not need to compute Sine or Cosine, and so is very fast. One caveat, however, this routine does no screen bounds checking, so either make sure your circle will fit on the screen, or add your own bounds checking.

2.4.1 PROC Circle

Purpose: To draw a circle of specified center, radius, and color.

Syntax: PROC Circle(INT x BYTE y,r,c)

Params: x - the horizontal position of the center of the circle to be drawn.
         y - the vertical position of the center of the circle to be drawn.
         r - the radius of the circle.
         c - the color of the circle.

Description: This procedure allows you to draw a circle of specified center, radius, and color. The system variable color is set to c, so c should not be the actual color number as used in the SetColor procedure, but rather the 'color' value in the current graphics mode which corresponds to the SetColor register which contains the color you want to use.

The ACTION! Programming Environment

2.5 IO.ACT

The routines in this file allow you to do advanced disk
file manipulation from an ACTION! program. Operations
implemented are:

        Rename a file   Format a diskette
        Erase a file   Block Get of data from disk
        Protect a file   Block Put of data to disk
        Unprotect a file

NOTE: The first four of the above operations (those
involving a disk fill) require that the file name have the
'Dn:' (n=1-8) device specifier prepended to the actual file
name; otherwise you will get a 'Nonexistent Device' error.

2.5.1 PROC Rename

Purpose: To rename a disk file.

Syntax: PROC Rename(BYTE ARRAY filename)

Params: filename - the old and new file names.

Description: This routine will rename the specified disk
file, and should be used as follows:

        Rename("D1:TEMP1.ACT TEMP.ACT")

This example will rename TEMP1.ACT on drive 1 to TEMP.ACT.
Notice that the new name follows the old name in the file
name string, with only a space or comma separating the two.
Note that the new name may NOT have a device specifier.

2.5.2 PROC Erase

Purpose: To erase a disk file.

Syntax: PROC Erase(BYTE ARRAY filename)

Params: filename - the file to erase.

Description: This procedure will erase a disk file and
should be used as follows:

        Erase("D2:JUNK.ACT")

This example will erase JUNK.ACT on drive 2.

2.5.3 PROC Protect

Purpose: To protect a disk file.

Syntax: PROC Protect(BYTE ARRAY filename)

Params: filename - the file to protect.

Description: This will protect a disk file, and should be used as follows:

        Protect("D:*.*")

This example will protect all files on the current drive, mostly 1.

2.5.4 PROC UnProtect

Purpose: To unprotect a disk file.

Syntax: PROC UnProtect(BYTE ARRAY filename)

Params: filename - the file to unprotect.

Description: This procedure will unprotect a file which has been protected using either the Protect routine above, or the DOS XL PRO command. It is used in the same way as Protect above.

2.5.5 PROC Format

Purpose: To format a diskette.

Syntax: PROC Format(BYTE ARRAY DriveSpec)

Params: DriveSpec - the drive containing the disk to be initialized.

Description: This routine allows you to initialize disks, and should be used as follows:

        Format("D2:")

This example will format whatever disk is in drive 2 (unless of course it has a write protect tab on it).

2.5.6 CARD FUNC Bget

Purpose: To read a block of binary or text data from a
         specified device.

Syntax: CARD FUNC Bget(BYTE chan CARD addr,len)

Params: chan – the channel.
         addr – the address at which to put the data.
         len - the number of bytes of data.

Inscription: This function allows you to read a block of
data, returning the actual number of data bytes read (this
will be different from len if End-Of-File was reached
before 'len' bytes were read).

2.5.7 PROC BPut

Purpose: To write a block of binary or text data to a
         specified device.

Syntax: PROC BPut(BYTE chan CARD addr,len)

Params: chan – the channel
         addr – the address from which to get the data.
         len - the number of bytes of data.

Description: This procedure allows you to write a block of
data, and is the complement to BGet.

2.6 JOYSTIX.ACT

2.6.1 INT FUNC HStick(BYTE port)

Purpose: To return the horizontal reading of a specified
         joystick.

Syntax:

Params: port - the port of the joystick whose horizontal
                reading is desired.

Description: This routine reads the value of a joystick and
returns the following values:

        -1 - horizontal movement left
         Ø - no horizontal movement
         1 - horizontal movement right

This routine is much easier to use than the Stick function
in the ACTION! Library.

2.6.2 INT FUNC VStick

Purpose: To return the vertical reading of a specified
         joystick.

Syntax: INT FUNC VStick(BYTE port)

Params: port - the port of the joystick whose vertical
                reading is desired.

Description: This routine reads the value of a joystick and
returns the following values:

        -1 - vertical movement up
         Ø - no vertical movement
         1 - vertical movement down

This routine is much easier to use than the Stick function
in the ACTION! Library.

2.7 PMG.ACT

The routines in this file allow you easy implementation of
the ATARI's player/missile (thereafter called P/M's)
graphics capabilities. To give you a sense of the extent of
this implementation, we'll give a quick synopsis of the
routines before going into them in detail:

        PMGraphics - Set up P/M graphics
        PMSetColor - Set a P/M's color
        PMAdr   - Give the address of a P/M
        PMClear   - Erase a P/M
        PMMOVE   - Move a P/M
        PMCreate   - Create a P/M
        PMHit   - Test the P/M collision registers
        PMHitClr   - Reset the collision registers
        PMHPos   - Horizontal positions of P/Ms
        PMVPos   - Vertical positions of P/Ms
        Graphics   - A modified Graphics

Introductory Notes: In several of the routines in this
section you will see the parameter num, referring to the
number of the player/missile. This number is assigned
values as follows:

        Ø - player Ø   4 - missile Ø
        1 - player 1   5 - missile 1
        2 - player 2   6 - missile 2
        3 - player 3   7 - missile 3

In some cases only the values Ø - 3 will be valid or make
sense.

2.7.1 PROC PMGraphics

Purpose: To turn P/M graphics on or off.

Syntax: PROC PMGraphics(BYTE mode)

Params: mode - determines which P/M mode.

Description: This procedure is very much like the Graphics
routine in the ACTION! Library, except that this one
controls player/missile graphics. The mode values are as
follows:

        Ø - turn off P/Ms
        1 - single line resolution P/Ms
        2 - double line resolution P/Ms

NOTE: This procedure moves all the players and missiles off
the screen, but does not erase the P/M memory. To erase it,
use PMClear.

2.7.2 PROC PMSetColor

Purpose: To set the hue and luminance of a player and its
         associated missile.

Syntax: PROC PMSetColor(BYTE num,hue,lum)

Params: num - the player number (Ø-3).
         hue - the hue for the player.
         lum - the luminance for the player.

Description: This procedure is very much like the SetColor
Library routine. In fact the colors corresponding to hue
and lum art exactly as shown in the ACTION! manual under
SetColor. The difference is that it allows you to set the
color of a P/M, not a playfield.

2.7.3 CARD FUNC PMAdr

Purpose: To return the address of a given P/M's memory
         block.

Syntax: CARD FUNC PMAdr(BYTE nm)

Params: num - the P/M number.

Description: This function returns the starting address of
the memory block allotted to the P/M specified by num.
Since the missiles all occupy the same block of memory, num
values 4 - 7 will all return the same address.

2.7.4 PROC PMClear

Purpose: To clear out the memory block of a specified P/M.

Syntax: PROC PMClear(BYTE num)

Params: num - the P/M number.

Description: This procedure zeroes all the bytes in the
memory block of the P/M given by num. If it is a missile,
only that part of the block allotted to the missile will be
zeroed.

2.7.5 PROC PMMove

Purpose: To move a specified P/M.

Syntax: PROC PMMove(BYTE num,x,y)

Params: num - the P/M number.
         x - horizontal position to which to move the P/M.
         y - vertical position to which to move the P/M.

Description: This procedure allows you to move P/Ms easily
and quickly. You simply need to specify the P/M number and
the x,y position to which you want it moved.

2.7.6 PROC PMCreate

Purpose: To allow easy creation of a P/M.

Syntax: PROC PMCreate(BYTE num BYTE ARRAY pm BYTE len,
                                          width,x,y)

Params: num - the P/M number.
         pm - the array which contains the P/M's shape data.
         len - the length of the array pm.
         width - the width of the player.
         x - the starting horizontal position of the P/M.
         y - the starting vertical position of the P/M.

Description: This routine allows you to create a P/M. You
need to pass it the P/M number, the name of the array which
contains its shape, the length of that array,the P/M's
width (1=single, 2=double, 4=quadruple), and the starting
x,y position of the P/M.

2.7.7 BYTE FUNC PMHit

Purpose: To determine whether a specified, P/M has collided
         with a specified player or playfield.

Syntax: BYTE FUNC PMHit(BYTE num,cnum)

Params: num - the P/M number.
         cnum - the player or playfield to test for a
                collision.

Description: This function allows you to see if a given P/M
has collided with a specified player or play field,
returning a 1 if there is a collision, a Ø otherwise. The
num values are described in the beginning of this section,
but the cnum values need to be explained:

```
Ø - player Ø    8 - playfield Ø
1 - player 1    9 - playfield 1
2 - player 2   1Ø - playfield 2
3 - player 3   11 - playfield 3
```

The playfield numbers Ø-3 are the same as those used in the
SetColor Library routine to set playfield colors.

2.7.8 BYTE PMHitClr

Purpose: To clear the P/M collision registers.

Syntax: BYTE PMHitClr

Params: Not Applicable.

Description: By using the statement: PMHitClr=Ø you can
clear the P/M collision registers. You should do this just
before you do something which might result in a collision
(such as PMMove), or you may have information from previous
collisions still in the registers.

2.7.9 BYTE ARRAY PMHPos

Purpose: To keep track of the current horizontal positions
         of the P/Ms.

Syntax: BYTE ARRAY PMHPos(Ø)

Params: The element number of the array (same as P/M
         number).

Description: By accessing an element of this array you can
find out the current horizontal position of any P/M. Simply
use the P/M number as the array element (e.g. PMHPos(3)
will give the horizontal position of player 3).The values
in this array should not be changed by you.

2.7.1Ø BYTE ARRAY PMVPos

Purpose: To keep track of the current vertical positions of
         the P/Ms.

Syntax: BYTE ARRAY PMVPos(Ø)

Params: The element number of the array (same as P/M
         number).

Description: By accessing an element of this array you can
find out the current vertical position of any P/M. Simply

use the P/M number as the array element (e.g. PMVPos(5) will give the vertical position of missile 1).The values in this array should not be changed by you.

2.7.11 PROC Graphics

Purpose: To turn off P/M graphics whenever changing bit-map graphics modes.

Syntax: PROC Graphics(BYTE mode)

Params: mode - same as in the Graphics Library routine.

Description: This procedure simply turns off the P/M graphics every time you change bit-map graphics modes, and replaces the normal Graphics Library routine. This routine is necessary, for the P/M graphics memory is allocated just below screen memory, so changing screen modes could wipe out part of the P/M space. If you are changing between graphics modes which use the same amount of memory, you can comment out this procedure from the source listing and so keep Graphics just the way it was.

2.8 PRINTF.ACT

The following two procedures are extensions of the library
PrintF routine, and allow you to control field size and
justification as well as the type of data output.

The following routines are internal to the PRINTF routines,
and should not be used by you unless you are sure of their
function:

        BYTE FUNC PF_ToLower
        BYTE FUNC PF_IsDigit
        CARD FUNC PF_Nbase

2.8.1 PROC PrintF

Purpose: To allow formatted output of data.

Syntax: PROC PrintF(BYTE ARRAY control CARD c1,c2,c3,c4,c5,c6)

Params: control - the string which determines the format
                   of the following data.

         c1 thru c6 - the data to be output

Description: This procedure is an upgrade to the PrintF
routine in the ACTION! Library. The difference lies in the
controls available and the modifications which can be made
to the controls. The controls themselves are:

        %D - Decimal Notation
        %O - Octal Notation
        %H - Hexadecimal Notation
        %U - Unsigned CARD Notation
        %C - Character
        %S - String (BYTE ARRAY)
        %E - Carriage Return/End-of-Line
        %% - the '%' character

So far this looks very similar to the 'normal' PrintF
routine. However, the best is yet to come. Between the '%'
and the control character (except 'E' and '%') you may
insert some field size and justification information as
follows:

A minus sign: this indicates left justification of the data
within its field (right justification is the default).

A number: determines the minimum field size for the data.
The data will be printed in a field at least number wide,
and wider if the data is too long. If the data is shorter

than the field size it will be right justified in the field
unless the '-' modifier has been used.

A '.' followed by a number: indicates the maximum number of
characters of data to print into the field.

NOTE by GBXL: Make sure the bug fix for this routine has
been applied to the respective toolkit file.

Example: The following list of control strings show how the
different modifiers affect the printing of the string
"ACTION!" (we have placed broken bars to show the field
size):

```
        %S        |ACTION!|
        %5S       |ACTION!|
        %1ØS      |   ACTION!|
        %-1ØS     |ACTION!   |
        %1Ø.4S    |      ACTI|
        %-1Ø.4S   |ACTI      |
        %.4S      |ACTI|
```

2.8.2 PROC PrintFD

Purpose: To allow formatted output of data to a specified
         channel.

Syntax: PROC PrintFD(BYTE chan BYTE ARRAY control CARD
                                      c1,c2,c3,c4,c5,c6)

Params: chan - the channel number (Ø-7)
         control - same as PrintF
         c1 thru c6 - same as PrintF

Description: This procedure is exactly like the above
PrintF, except that it allows you to direct the output to a
specific channel (device).

2.9 REAL.ACT

This file contains routines which allow you to access the
ROM floating point routines from ACTION!, thus making the
ACTION! language more useful when writing numerically
oriented programs.

To use the floating point routines (hereafter called the
Real routines), you must declare variables of the type
REAL, for example:

        REAL x,y,z

The type REAL is actually a record type, so the name of the
variable is a pointer to the record itself. This makes it
very similar to an array.

You cannot use the assignment statement to assign a value
to a real, since the ACTION! Compiler does not internally
understand reals. You must instead use RealAssign, ValR,
IntToReal, InputR, or InputRD.

Also included in this file are some mathematical routines
to manipulate reals, as well as routines to print out
reals.

Following each routine's description section are some
examples of that routine's usage. For these examples,
assume the following declarations:

        REAL xreal,yreal,zreal
        BYTE ARRAY astring
        INT xint,yint,zint
        BYTE channel

The following routines are internal to the ACTION! real
routines, and should not be used by you:

        PROC ROM_AFP   PROC ROM_FASC
        PROC ROM_IFP   PROC ROM_FPI
        PROC ROM_FSUB   PROC ROM_FADD
        PROC ROM_FMULT PROC ROM_FDIV
        PROC ROM_EXP   PROC ROM_EXP1Ø
        PROC ROM_LOG   PROC ROM_LOG1Ø
        PROC ROM_INIT

NOTE: You will often see the type REAL POINTER in the
declaration of the parameters of a routine. This simply
means that you should use the name (identifier) of the
real, since the name alone is a pointer to the real.

2.9.1 REAL Conversion Routines

2.9.1.1 PROC IntoReal

Purpose: To put an INI value into a REAL variable.

Syntax: PROC IntoReal(INT i REAL POINTER r)

Params: i - the INT value to be assigned to the REAL.
         r - the REAL to which the INT value is assigned.

Description: This procedure allows you to assign the value
of an INT to a REAL variable. If the ACTION! compiler could
manipulate reals, this routine would be the equivalent of:
r=i.

Examples: xint=453
          IntToReal(xint,xreal) ;xreal now equals 453
          IntToReal(2534,yreal) ;yreal now equals 2534

2.9.1.2 INT FUNC RealToInt

Purpose: To return the INT value of a REAL variable.

Syntax: INT FUNC RealToInt(REAL POINTER r)

Params: r — the REAL variable

Description: This function will return the INT value of the
REAL passed to it as a parameter.

Example:xint=RealToInt(xreal) ;xint now equals the INT
                                 value of xreal

2.9.1.3 PROC StrR

Purpose: To convert a REAL to a string.

Syntax: PROC StrR(REAL POINTER r BYTE ARRAY s)

Params: r - the REAL to convert.
         s - the string in which to store the character
             representation of the REAL.

Description: This procedure converts a REAL into its
character representation.

Examples: IntToReal(3926,xreal) ;xreal = 3926
          StrR(xreal,astring) ;astring now contains "3926"

2.9.1.4 PROC ValR

Purpose: To convert a string to a REAL.

Syntax: PROC ValR(BYTE ARRAY s REAL POINTER r)

Params: s - the string to convert.
         R - the REAL to which the value of s will be
             assigned.

Description: This procedure will convert as much of the
string as possible into a REAL variable (i. e., if the
string is "abcde", this routine will put Ø into the REAL).

Examples: astring="45.276"
          ValR(astring,xreal) ;same as xreal="45.276"
          ValR("2.7E-4",yreall ;same as xreal=2.7*1Ø^-4
          ValR("7Ø.2agr",zreal) ;same as zreal=7Ø.2

2.9.2 REAL Mathematical Routines

2.9.2.1 PROC RealAssign

Purpose: To assign the value of one REAL variable to
         another.

Syntax: PROC RealAssign(REAL POINTER a,b)

Params: a - the REAL value to assign.
         b - the REAL to which the value a is assigned.

Description: This procedure allows you to assign the value
of one REAL to another one. If the ACTION! Compiler could
manipulate reals, the equivalent would be: b=a.

Examples: RealAssign(xreal,yreal) ;same as yreal=xreal
          RealAssign(zreal,yreal) ;same as zreal=yreal

2.9.2.2 PROC RealAdd

Purpose: To add two REALs

Syntax: PROC RealAdd(REAL POINTER a,b,c)

Params: a - an addend
         b - an addend
         c - the sum

The ACTION! Programming Environment

Description: This procedure allows you to add two REALs. If the ACTION! Compiler could manipulate reals, this routine would equivalent to: c=a+b.

Example: RealAdd(xreal,yreal,zreal)
                    ;same as zreal=xreal+yreal

2.9.2.3 PROC RealSub

Purpose: To subtract two REALS

Syntax: PROC RealSub(REAL POINTER a,b,c)

Params: a - the subtrahend
         b - the minuend
         c - the difference

Description: This procedure allows you to subtract two REALs. If the ACTION' Compiler could manipulate reals, this routine would equivalent to: c=a-b.

Example: RealSub(xreal,yreal,zreal)
                    ;same as zreal=xreal-yreal

2.9.2.4 PROC RealMult

Purpose: To multiply two REALS

Syntax: PROC RealMult(REAL POINTER a,b,c)

Params: a - the multiplicand
         b - the multiplier
         c - the product

Description: This procedure allows you to multiply two REALs. If the ACTION! Compiler could manipulate reals, this routine would equivalent to: c=a*b.

Example: RealSub(xreal,yreal,zreal)
                    ;same as zreal=xreal*yreal

2.9.2.5 PROC RealDiv

Purpose: To divide two REALS

Syntax: PROC RealDiv(REAL POINTER a,b,c)

Params: a - the dividend
         b - the divisor
         c - the quotient

Description: This procedure allows you to divide two REALs.
If the ACTION! Compiler could manipulate reals, this
routine would equivalent to: c=a/b.

Example: RealDiv(xreal,yreal,zreal)
                  ;same as zreal=xreal/yreal

2.9.2.6 PROC Exp

Purpose: To raise e to the a power.

Syntax: PROC Exp(REAL POINTER a,b)

Params: a - the power to which to raise e.
         b - the result of raising e to the a power.

Description: This procedure allows you to get the base e
exponential of a REAL. The equivalent of this is: b=e^a.

Example: Exp(xreal,yreal) ;same as yreal=e^xreal

2.9.2.7 PROC Exp1Ø

Purpose: To raise 1Ø to the a power.

Syntax: PROC Exp1Ø(REAL POINTER a,b)

Params: a - the power to which to raise 1Ø.
         b - the result of raising 1Ø to the a power.

Description: This procedure. allows you to compute the base
1Ø exponential of a REAL. Its equivalent is: b=1Ø^a.

Example: Exp1Ø(xreal,yreal) ;same as yreal=1Ø^xreal

2.9.2.8 PROC Power

Purpose: To raise REAL to a REAL power.

Syntax: PROC Power(REAL POINTER a,b,c)

Params: a - the base of the power.
         b - the power to which to raise a.
         c - the result of raising a to the b power.

Description: This routine allows you to raise one REAL to a
power specified by another REAL, and is equivalent to:
c=a^b.

Example: Power(xreal,yreal,zreal) ;same as zreal=xreal^yreal

239

2.9.2.9 PROC Ln

Purpose: To take the natural logarithm of a REAL.

Syntax: PROC Ln(REAL POINTER a,b)

Params: a - the REAL whose natural log is taken.
         b - the result of taking the natural log of a.

Description: This procedure allows you to take the natural
(base e) logarithm of a REAL, and is equivalent to:
b=ln(a).

Example: Ln(xreal,yreal) ;same as yreal=ln(xreal)

2.9.2.1Ø PROC Log1Ø

Purpose: To take the common (base 1Ø) logarithm of a REAL.

Syntax: PROC Log1Ø(REAL POINTER a,b)

Params: a - the REAL whose common log is taken.
         b — the result of taking the common log of a.

Description: This procedure allows you to take the common
(base 1Ø) logarithm of a REAL. and is equivalent to:
b=log(a).

Example: Log1Ø(xreal,yreal) ;same as yreal=log(xreal)

2.9.3 I/O Routines

2.9.3.1 PROC PrintR

Purpose: To output a REAL to the default device.

Syntax: PROC PrintR(REAL POINTER a)

Params: a - the REAL to be output.

Description: This procedure outputs a real number to the
default device without a RETURN.

2.9.3.2 PROC PrintRD

Purpose: To output a REAL to a specified channel (device).

Syntax: PROC PrintRD(BYTE channel REAL POINTER a)

Params: channel - the output channel
         a - the REAL to be output.

Description: This procedure outputs a real number to the
device specified by channel without a RETURN.

2.9.3.3 PROC PrintRE

Purpose: To output a REAL to the default device with a
         RETURN.

Syntax: PROC PrintRE(REAL POINTER a)

Params: a - the REAL to be output.

Description: This procedure outputs I real number to the
default device with a RETURN.

2.9.3.4 PROC PrintRDE

Purpose: To output a REAL to a specified channel (device)
         with a RETURN.

Syntax: PROC PrintRDE(BYTE channel REAL POINTER a)

Params: channel - the output channel.
         a - the REAL to be output.

Description: This procedure outputs a real number to the
specified device with a RETURN.

2.9.3.5 PROC InputR

Purpose: To input a REAL from the default device.

Syntax: PROC InputR(REAL POINTER a)

Params: a - the REAL variable in which to store the input
             value.

Description: This procedure inputs a real number from the
default device and stores it in the specified REAL
variable.

241

The ACTION! Programming Environment

2.9.3.6 PROC InputRD

Purpose: To input a REAL from a specified channel (device).

Syntax: PROC InputRD(BYTE channel REAL POINTER a)

Params: channel - the input channel.
         a - the REAL variable in which to store the input
             value.

Description: This procedure inputs a real number from the
specified device and stores it in the given REAL variable.

2.1Ø SORT.ACT

The following four sort routines all us the QuickSort
algorithm. This algorithm was used because it is very fast
(order N log N). In the best case QuickSort is, in fact,
among the fastest sorting algorithms known. For comparison,
both the Bubble and the Shell algorithms are of order N^2.
The QuickSort can deteriorate to this speed when sorting
presorted data.

If you take a look at the SORT.ACT source you will see that
you can create your own routines to sort REALs or complex
record TYPEs simply by writing your own Compare and Swap
routines.

Usage note: Before using any of these routines you should
first change the source line which reads

        DEFINE SortMax="1ØØØØ"

to the maximum size of the data array you expect to
encounter. An alternative is to change the sort routines so
that they INCLUDE ALLOC.ACT and dynamically create the
'list' array.

2.1Ø.1 PROC SortB

Purpose: To sort one-byte data in either ascending or
         descending order.

Syntax: PROC SortB(BYTE ARRAY data CARD len BYTE order)

Params: data - the array containing the data to be sorted.
         len - the length of the data array.
         order - determines order of sort (Ø=ascending,
                  1=descending)

Description: This procedure allows you to sort one-byte
data very quickly.

2.1Ø.2 PROC SortC

Purpose: To sort two-byte unsigned data in either ascending
         or descending order.

Syntax: PROC SortC(CARD ARRAY data CARD len BYTE order)

Params: data - the array containing the data to be
                sorted.
         len - the length of the data array.
         Order - determines order of sort (Ø=ascending,
                 1=descending)

Description: This procedure allows you to sort two-byte
unsigned data very quickly.

2.1Ø.3 PROC SortI

Purpose: To sort two-byte signed data in either ascending or
         descending order.

Syntax: PROC SortI(INT ARRAY data CARD len BYTE order)

Params: data - the array containing the data to be
                sorted.
         len - the length of the data array.
         order - determines order of sort (Ø=ascending,
                 1=descending)

Description: This procedure allows you to sort two-byte
signed data very quickly.

2.1Ø.4 PROC SortS

Purpose: To sort string data in either ascending or
         descending order.

Syntax: PROC SortS(CARD ARRAY data CARD len BYTE order)

Params: data - the array containing the addresses of the
                strings to be sorted.
         len -   the length of the data array.
         order - determines order of sort (Ø=ascending,
                 1=descending)

Description: This procedure allows you to sort strings very
quickly. Notice that the "addresses" of the strings to be
sorted must be the elements of the CARD ARRAY data.

2.11 TURTLE.ACT

The routines in this file implement turtle graphics ala
LOGO.

These routines require that the screen be in a bit-map
graphics mode in which Plot and DrawTo are useable. Also,
the length of a line drawn depends on the graphics mode,
and there is no screen bounds checking.

Also, the color of the line drawn by the turtle depends
entirely upon the current value of the system variable
color, so you should use SetColor and color to choose the
color you want.

The following routines are internal to the turtle graphics
and should not be called by you

        CARD FUNC TG_ISin
        CARD FUNC TG_ICos

2.11.1 PROC Right

Purpose: To turn the turtle right (clockwise) theta degrees.

Syntax: PROC Right(INT theta)

Params: theta - the angle to turn the turtle clockwise.

Description: This procedure allows you to turn the turtle
clockwise a specified number of degrees.

2.11.2 PROC Left

Purpose: To turn the turtle left (counterclockwise) theta
         degrees.

Syntax: PROC Left(INT theta)

Params: theta - the angle to turn the turtle counter-
                 clockwise.

Description: This procedure allows you to turn the turtle
counterclockwise a specified number of degrees.

The ACTION! Programming Environment

2.11.3 PROC Turn

Purpose: To turn the turtle either clockwise or counter-
         clockwise.

Syntax: PROC Turn(INT theta)

Params: theta - the angle to turn the turtle.

Description: This routine allows you to turn the turtle
either clockwise or counterclockwise, depending on the sign
of the angle. If theta is positive, the turtle will turn
counterclockwise, otherwise it will turn clockwise.

2.11.4 PROC Forward

Purpose: To move the turtle forward a specified length.

Syntax: PROC Forward(INT length)

Params: length - the length to move forward.

Description: This procedure allows you to move the turtle
forward a specified length. This length depends entirely
upon the current graphics mode.

2.11.5 PROC SetTurtle

Purpose: To move the turtle to a specified x,y position at a
         given angle.

Syntax: PROC SetTurtle(INT x,y,theta)

Params: x - the horizontal position at which to set the
             turtle.
         y - the vertical position at which to set the
             turtle.
         theta - the angle at which to set the turtle.

Description: This procedure allows you to move the turtle
to an absolute x,y position and point it in a specific
direction. At theta=0° the turtle points right, at 90° it
points up, at 180° it points left, and at 270° it points
down. In essence, increasing positive values of theta turn
the turtle counterclockwise. And increasing negative theta
values turn it clockwise.

Chapter 3: Demonstrations

3.1 GEM.DEM

Gem is a game which was written by Joel Gluck after having
the ACTION! cartridge for only 2 days. If you look at the
code, you will notice how similar it is to BASIC. This
reflects Joel's previous programming experience (BASIC
only) and is not due to its being originally written in
BASIC (which it was not). Enough of its history, Gem is
designed for 1 to 4 players, each using a joystick. The
object is to steal the gem in the center of the screen and
return to your home base before one of the robots or other
players zaps you. However, before the game itself begins,
you are prompted for some information, specifically:

How many points to win the game?

How many robots in the final round?

Winning Points - to win a point, you must get the Gem and
return with it to your corner.

Robots - the number of robots increase each round. Although
they seem to die off, whenever the gem is picked up they
all come back. If one of the robots is destroyed while one
of the players is carrying the gem, it is reincarnated
immediately.

Zapping - to zap a robot or another player, press the
joystick trigger while pointing the joystick in the
direction of the target. While you are zapping you cannot
move. You can also zap by running into the target, but this
also zaps you, so only use this method when on a Kamikaze
run to keep another player from getting the gem home.

Getting Zapped - when you get zapped, you are reincarnated
back at your home base and the gem is taken from you if you
are carrying it. There is no limit to the number of times
you can be reincarnated.

Winning - when one of the players has accumulated the
required number of points, he wins the gem, and you may
either play again, or quit and go to the ACTION! Monitor.

Technical notes - to use this game, do not read it into the
ACTION! Editor and then compile it, since there is not
enough memory to do both. Instead, RUN it from the ACTION!
Monitor directly from disk. (This note does not apply if
you are using DOS XL, since you have more memory and can
have the game in the Editor while compiling it).

The ACTION! Programming Environment

The maximum recommended number of robots is 1ØØ. Bugs will
appear in the program if you use many more, but do not
fret. The most robots survived to date is 45 in a one
player game.

3.2 KALSCOPE.DEM

This demo program uses advanced math and display list
algorithms to achieve the effect of a kaleidoscope on your
TV. When you run it you will be amazed by its speed. You
can even change the kaleidoscope's speed and persistence
(amount of time a point remains on the screen) by moving
joystick Ø vertically or horizontally, respectively. After
playing with it a while you will be surprised by the number
and variety of the different patterns it can create.

P.S. - you can freeze the picture by pressing the trigger.

3.3 MUSIC.DEM

This demo program uses a couple of the Toolkit utilities
and knowledge of the Atari's keyboard matrix to produce an
organ which will play as you press the keys.

The letters on the keyboard represent the notes, and the
letters above and below the keyboard represent the actual
computer keys you must press to get the note. By pressing
<SHIFT><note> you can access the middle octave, and by
pressing <CONTROL><note> you can access the high octave.

This organ is special (for Atari's) in that it only plays a
note as long as you keep the key depressed. Few people know
how to determine how long a key is pressed (unless they've
deciphered the "Type-a-Tune" demo in the BASIC reference
manual, or waded through the hardware manual), so if you
look at the source code you can discover something useful
(possibly).

3.4 SNAILS.DEM

Games similar to "SNAILS' TRAILS"·have been around for a
long time. A version called "SURROUND" was one of the first
games available for the Atari 26ØØ. But, in the tradition
of the video game industry, we present a storyline:

You are a giant, mutant snail. Wherever you travel, you
leave a trail of radioactive slime behind. So poisonous and

248

impenetrable is this slime that should any being (including you yourself) touch it, it dies instantly. (Yes, yes. If it's that poisonous, how could you lay the trail in the first place? How should we know … YOU are the mutant.)

Further, the scientists of far off H'tra-E have discovered your kind and have imprisoned you and another of your race in a large rectangular arena. Unfortunately, both of you are neither male or female. Instead, you are each a S'ti, specially bred to do battle until death! You don't know the meaning of the word "STOP".

So, as the scientists release you from stasis (you hear three bells as the stasis field is lifted), you begin by charging straight toward your opponent. But wait! A bit of intelligence enters your crazed brain. If your slime trail is so deadly, perhaps you can entice your enemy to run into it, thus killing the other S'ti without damage to yourself. Great strategy!

What's this, though? Your opponent has developed the same strategy. Now you and the other S'ti must race around the arena, with the strategic goal of forcing each other to touch a poisonous trail or to run into the electrified outer fence. (Well, we had to keep you in the arena somehow, didn't we?). But tactics can be important as well. Look, you are running straight across the arena. At the last second, you veer in front of your enemy! He can't avoid your trail in time! He's going to … Oops. You forgot about the wall. Too bad. R.I.P.

To make a long story into a short game, you and another human opponent must each use a joystick (plugged into ports 1 and 2) to control your snail. The first snail to run into a slime trail or a wall loses, and the other snail scores a point. The first snail to score 1Ø points wins the game. Also, if both snails die at the same instant, neither scores a point. Good Luck!

P.S. This game was converted from BASIC XL to ACTION! in about two hours. The original BASIC XL version is in Chapter 29 of "Thirty Days to Understanding BASIC XL" and is on the BASIC XL Toolkit diskette.

3.5 WARP.DEM

Warp Attack is a game for only the most daring interstellar pilots. You have been chosen as one of this special breed and are sent on a surface patrol over the planet Stripes. You can move your ship left or right and you can dive or climb as in an airplane, but your on-board navigational

equipment won't allow you to crash into the planet surface. As you are flying along minding your own business, an Hospites (your sworn enemy) Stellar Fortress warps right into your path,and she's armed to the teeth with Seeker Plasma Balls. One touch of them and you're dust. And all you have are puny pulse cannons.

Now you know why only the best were chosen for this assignment: very few know how to destroy a Stellar Fortress, and you are one of them. You first must destroy its right engine (on your left as it approaches), then its left engine, and finally its main engine, and each must be a direct hit. While completing this feat of precision marksmanship you must remember to avoid those Plasma Balls. Piece of Cake!

Technical Notes: Warp Attack uses quite a few advanced programming techniques, including a modified display list, display list interrupts, vertical blank interrupts, and a block fastdraw. The DLI and VBI together create the scrolling planet surface, and the fastdraw is used to move the Stellar Fortress (it's not a player!).

Note from bugsheet #3: this file can only be run when compiled from disk (unless you are using DOS XL to gain extra memory). WARP.DEM is just too big for ACTION! to hold both the source and object in memory at one time.

NOTE by GBXL: SpartaDOS X 4.47 will do as well, when configured properly.

Appendix A: ACTION! Language Syntax

The following is the syntax of the ACTION! language in
Backus-Naur form. This form has a couple of special
characters:

         Symbol   Meaning
         ------   -------
           ::=   "is defined as"
           | "or"
           {}   "optional"

The appendix is set up to allow you easy access to the
particular information you want, with subsections as
follows:

A.1 ACTION! Constants

Numeric Constant

```
<num const> ::= <dec num> | <hex num> | <char>
<dec num> ::= <dec num><digit> | <digit>
<hex num> ::= <hexnum><hex digit> | $<hex digit>
<char> ::= '<any printable character>
<hex digit> ::= <digit> | A | B | C | D | E | F
<d ig i t> ::= Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

String Constant

```
<str const> ::= "<string>"
<string> ::= <string><str char> | <str char>
<str char> ::= <all printable characters, except " >
```

Compiler Constant

```
<comp const> ::= <comp const>+<base comp const> |
                                      <base comp const>

<base comp const> ::= <identifier> | <num const> |
                                      <ptr ref> | *
```


A.2 Operators and Fundamental Data Types

Operators

```
<special op> ::= AND | OR | & | %
<rel op> ::= XOR | ! | = | # | <> | < | <= | > | >=
<add op> ::= + | -
<mult op> ::= * | / | MOD | LSH | RSH
<unary op> ::= @ | -
```


Fundamental Data Types

```
<fund type> ::= CARD | CHAR | BYTE | INT
```


A.3 ACTION! Program Structure

ACTION! Program

```
<program> ::= <program> MODULE <prog module> |
                              {MODULE} <prog module>
<prog module> ::= {<system decls>} <routine list>
```

```
A.4 Declarations
----------------


System Declarations
-------------------
<system decls> ::= <DEFINE decl> | <TYPE decl> | <var decl>

DEFINE Directive
----------------
<DEFINE decl> ::= <DEFINE> <def list>
<def list> ::= <def list>,<def> | <def>
<def> ::= <identifier>=<str const>

TYPE Declaration (for records)
------------------------------
<TYPE decl> ::= TYPE <rec ident list>
<rec ident list> ::= <rec ident list> <rec ident> | <rec ident>
<rec ident> ::= <rec name>=[<field init>]
<rec name> ::= <identifier>
<field init> ::= <fund var decl>

Variable Declarations
---------------------
<var decl> ::= <var decl> <base var decl> | <base var decl>
<base var decl> ::= <fund decl> | <POINTER decl> |
                            <ARRAY decl> | <record decl>

Variable Declaration for Fundamental Data Types
-----------------------------------------------
<fund decl> ::= <fund decl> <base fund decl> |
                                        <base fund decl>
<base fund decl> ::= <fund type> <fund ident list>
<fund type> ::= CARD | CHAR | BYTE | INT
<fund ident list> ::= <fund ident list>,<fund ident> |
                                            <fund ident>
<fund ident> ::= <identifier>{=<init opts>}
<init opts> ::= <addr> | [<value>]
<addr> ::= <comp const>
<value> ::= <num const>

Variable Declaration for Pointers
---------------------------------
<POINTER decl> ::= <ptr type> POINTER <ptr ident list>
<ptr type> ::= <fund type> | <rec name>
<ptr ident list> ::= <ptr ident list>,<ptr ident> |
                                            <ptr ident>
<ptr ident> ::= <identifier>{=<value>}
```

```
Variable Declaration for Arrays
-------------------------------
<ARRAY decl> ::= <fund type> ARRAY <arr ident list>
<arr ident list> ::= <arr ident list>,<arr ident> |
                                            <arr ident>
<arr ident> ::= <identifier>{(<dim>)}{=<arr init opts>}
<dim> ::= <num conat>
<arr init opts> ::= <addr> | [<value>] | <str const>
<addr> ::= <comp const>
<value list> ::= <value list><value> | <value>
<value> ::= <comp const>


Variable Declaration for Records
--------------------------------
<record decl> ::= <identifier> <rec ident list>
<rec ident list> ::= <rec ident list>,<rec ident> |
                                            <rec ident>
<rec ident> ::= <identifier>{=<address>}
<address> ::= <comp const>



A.5 Variable References
-----------------------

Memory References
-----------------
<mem reference> ::= <mem contents> | @<identifier>
<mem contents> ::= <fund ref> | <arr ref> | <ptr ref> |
                                            <rec ref>
<fund ref> ::= <identifier>
<arr ref> ::= <identifier>(<arith exp>)
<ptr ref> ::= <identifier>^
<rec ref> ::= <identifier>.<identifier>



A.6 ACTION! Routines
--------------------

Routines
--------
<routine list> ::= <routine list> <routine> | <routine>
<routine> ::= <proc routine> | <func routine>

Procedure Structure
-------------------
<proc routine> ::= <PROC decl> {<system decls>}
                                    {<stmt list>}{RETURN}
<proc decl> ::= PROC <identifier>{=<addr>}({<param decl>})
<addr> ::= <comp const>
```

```
Function Structure
------------------
<func routine> ::= <FUNC decl> {<system decls>}
     {<stmt list>}{RETURN (<arith exp>)}
<FUNC decl> ::= <fund type> FUNC <identifier>{.<addr>}
          ({<param decl>})
<addr> ::= <comp const>

Routine calls
-------------
<routine call> ::= <FUNC call> | <PROC call>
<FUNC call> ::= <identifier>({<params>})
<PROC call> ::= <identifier>({<params>})

Parameters
----------
<param decl> ::= <var decl>

NOTE: max. of 8 parameters allowed


A.7 Statements
--------------
<stmt list> ::= <stmt list> <stmt> | <stmt>
<stmt> ::= <simp stmt> | <struc stmt> | <code block>
<simp stmt> ::= <assign stmt> | <EXIT stmt> | <routine call>
<struc stmt> ::= <IF stmt> | <DO loop> | <WHILE loop> |
                                        <FOR loop>

Assignment Statement
--------------------
<assign stmt> ::= <mem contents>=<arith exp>

EXIT Statement
--------------
<EXIT stmt> ::= EXIT

IF Statement
------------
<IF stmt> ::= IF <cond exp> THEN {stmt list}
                         {|:ELSEIF exten:|}{ELSE exten} FI
<ELSEIF exten> ::= ELSEIF <cond exp> THEN {stmt list}
<ELSE exten> ::= ELSE {stmt list}

DO - OD Loop
------------
<DO loop> ::= DO {<stmt list>} {<UNTIL stmt>} OD

UNTIL Statement
---------------
<UNTIL stmt> ::= UNTIL <cond exp>
```

```
WHILE Loop
----------
<WHILE loop> ::= WHILE <cond exp> <DO loop>


FOR Loop
--------
<FOR loop> ::= FOR <identifier>=<start> TO <finish>
                                 {STEP <inc>}<DO loop>
<start> ::= <arith exp>
<finish> ::= <arith exp>
<inc> ::= <arith exp>


Code Blocks
-----------
<code block> ::= [<comp const list>]
<comp const list> ::= <comp const list> <comp const> |
                                            <comp const>



A.8 Expressions

Relational Expressions
----------------------
<complex rel> ::= <complex rel><special op><simp rel exp> |
                  <simp rel exp><special op><simp rel exp>
<simple rel exp> ::= <arith exp><rel op><arith exp>

Arithmetic Expressions
----------------------
<arith exp> ::= <arith exp><add op><mult exp> | <mult exp>
<mult exp> ::= <mult exp><mult op><value> | <value>
<value> ::= <num const> | <mem reference> | (<arith exp>)
```

```
              Appendix B: ACTION! Memory Map
              -----------------------------


      $ØØ  +--------------------------+
           |      OS and ACTION!      |
           |        Variables         |
      $CA  +--------------------------+
           |       Free Memory        |
      $CE  +--------------------------+
           |     ACTION! Variables    |
      $D4  +--------------------------+
           |   ATARI Floating Point   |
           |        Registers         |
     $1ØØ  +--------------------------+
           |     Operating System     |
     $48Ø  +--------------------------+
           |     ACTION! Variables    |
     $58Ø  +--------------------------+
           |   ATARI Floating Point   |
           |          Buffer          |
     $6ØØ  +--------------------------+
           |     Operating System     |
    MEMLO  +--------------------------+
           |  ACTION! Compiler Stacks |
  LO+$2ØØ  +--------------------------+
           |    ACTION! Editor Line   |
           |          Buffer          |
  LO+$3ØØ  +--------------------------+
           |    ACTION! Hash Tables   |
  LO+$75Ø  +--------------------------+
           |    ACTION! Editor Text   |
           |          Buffer          |
           +- - - - - - - - - - - - -+
           |  ACTION! Compiler Code   |
           |          Space           |
  TOP-$8ØØ +--------------------------+
           |  ACTION! Compiler Symbol |
           |          Table           |
   MEMTOP  +--------------------------+
           |      Screen Memory       |
    $AØØØ  +--------------------------+
           |     ACTION! Cartridge    |
    $CØØØ  +--------------------------+
           |      OS, ROMs, etc.      |
    $FFFF  +--------------------------+
```

NOTE: the Compiler Code Space starts wherever the Editor
Text Buffer ends. This makes both the Editor Buffer and the
Compiler Buffer dynamic in memory. For more information on
this, see part V, chapter 2.

Appendix C: Error Code Explanation

In this appendix we will describe the meaning of each of
the error numbers you could encounter while programming in
ACTION!. Included are those errors which the ACTION!
system itself discovers, but not those which the operating
system discovers (errors 128 - 255).

Error Code	Explanation
Ø	Out of system memory. See Part II, section 4.3, and Part V, section 4.4, to find out how to remedy this error.
1	Missing " (double quote) in a string.
2	Nested DEFINEs. You can not nest the DEFINE directive.
3	Global variable symbol table full.
4	Local variable symbol table full.
5	SET directive syntax error.
6	Declaration error. You used the wrong declaration format when declaring something.
7	Invalid argument list. You gave a statement or routine too many arguments.
8	Variable not declared. Remember, you must declare your variables before you use them.
9	Not a constant. You used a variable where a constant of some kind was required.
1Ø	Illegal assignment. You are trying to do some sort of assignment that is not allowed (e.g., var=5>7 is illegal).
11	Unknown error. You have somehow impaired the ACTION! system error routines, so it cannot tell you which error you have made.
12	Missing THEN
13	Missing FI

```
Error Code   Explanation
----------   -----------

    14       Out of code space. See Part V, section 4.4,
             for more information.

    15       Missing DO

    16       Missing TO

    17       Bad Expression. You have used an illegal
             expression format.

    18       Unmatched parentheses.

    19       Missing OD

    2Ø       Cannot allocate memory. You have impaired
             the ACTION! system, and it is unable to
             allocate any more memory.

    21       Illegal array reference

    22       The input file is too large. You need to
             break it into smaller pieces.

    23       Illegal Conditional Expression

    24       Illegal FOR statement syntax

    25       Illegal EXIT. There is no DO - OD loop for
             the EXIT to exit out of.

    26       Nesting too deep (16 levels maximum).

    27       Illegal TYPE syntax.

    28       Illegal RETURN.

    61       Out of Symbol Table space. See Part IV for
             more information.

   128       <BREAK> key was used to stop program
             execution.
```

Appendix D: Bibliography and References

D.1 ATARI Hardware Systems

ATARI Publications:

  ATARI Operating System Manuals
  ATARI User and Hardware Manuals

Other ATARI References:

  Poole, McNiff, Cook. Your ATARI Computer
  ABBUC e.V., Das ATARI Profibuch, ABBUC-Edition (German)
  SpartaDOS X User Guide as of V. 4.47

D.2 Magazine Articles

ANALOG

No. 16, p. 54, Moriarty, Brian 1984:
               A New Language for the ATARI!
No. 17, p. 58, Parker, Clinton 1984:
               Introduction to Action! - Part 1
No. 18, p. 91, Parker, Clinton 1984:
               Introduction to Action! - Part 2
No. 2Ø, p. 82, Glover, Donald E. 1984:
               Stars 3-D in Action!
No. 2Ø, p. 86, Plotkin, David 1984:
               Bounce in Action!
No. 26, p. 79, Gluck, Joel 1985:
               PuLse in Action!
No. 27, p. 43, Gluck, Joel 1985:
               More Fun with Bounce! (in Action!)
No. 28, p. 42, Bullok, Dan 1985:
               Demon Birds
No. 31, p. 24, Stortz, Mike 1985:
               R.O.T.O.
No. 32, p. 23, Wetmore, Russ 1985:
               Getting in on the Action! - Part 1
No. 32, p. 35, Guber, Sol 1985:
               Color the Shapes
No. 35, p. 97, Wetmore, Russ 1985:
               Getting in on the Action! - Part 2
No. 36, p. 33, Plotkin, David 1985:
               Sneak Attack
No. 38, p. 59, Page, Chris 1986:
               Air Hockey

No. 44, p. 23, Yates, Steven 1986: D:
            CHECK in Action!
No. 5Ø, p. 61, Garlow, Kevin R. 1987:
            Trails in Action!
No. 54, p. 31, Stortz, Mike 1987:
            Zero Free
No. 6Ø, p. 6Ø, Knaus, Gregg 1988:
            Cloudhopper
No. 62, p. 13, Plotkin, David 1988:
            ANALOG Man
No. 67, p. 38, McCarty, Monty 1988:
            Action! Graphics Toolkit
No. 69, p. 11, Knauss, Gregg 1989:
            Trial by Fire
No. 74, p. 8, Arlington, Dave 1989:
            Character Set Display Utility

ANTIC

Vol. 3 No. 3, p. 31,   Plotkin, David 1984:
            Interrupt with Action!
Vol. 3 No. 7, p. 7,   Abbot, Brian 1984:
            Demo 'Pretty'
Vol. 3 No. 12, p. 43, Chabot, Paul 1985:
            SPLASH in Action!
Vol. 4 No. 1, p. 55,   Plotkin, David 1985:
            Amazing
Vol. 4 No. 2, p. 38,   Chabot, Paul 1985:
            View 3D
Vol. 4 No. 3, p. 31,   Mitchell, Michael 1985:
            Darkstar
Vol. 4 No. 4, p. 48,   Oblad, Dave 1985:
            Display Master
Vol. 4 No. 5, p. 4Ø,   Oblad, Dave 1985:
            8 Queens Action!
Vol. 5 No. 6, p. 37,   Burchill, Lloyd 1986:
            Video Stretch
Vol. 6 No. 1Ø, p. 9,   Knauss, Gregg 1988:
            Killer Chess
Vol. 6 No. 1Ø, p. 13, Knauss, Gregg 1988:
            Reardoor
Vol. 6 No. 1Ø, p. 13, Knauss, Gregg 1988:
            Frog
Vol. 7 No. 6, p. 31,   Sherratt, Kevin 1988:
            Action! Toolbox
Vol. 7 No. 11, p. 6,   Peterson, Jon 1988:
            Demon Racer
Vol. 8 No. 4, p. 2Ø,   Skrecky, Douglas 1989:
            Superhop Action!

Hi-Res magazine

Vol. 1 No. 4, p. 72,   Laporte, Leo G. 1984:
                       Lights, Camera, Action!

ROM

No. 9, p. 8, Gregg, Kevin 1984:
              Action! Corner

S.P.A.C.E. Newsletter

January 1995 — March 1996, Serflaten, Larry:
                            Larry's Action! Tutorial

```
                    Appendix E: Editor Commands Summary
                    ----------------------------------


E.1 I/O Commands
----------------

     Read a File      position cursor, <CTRL><SHIFT> R,
                      enter filespec
     Disk Directory   <CTRL><SHIFT> R ?n:*.* (n = device num)
     Write a File     <CTRL><SHIFT> W, enter filespec
     List to Printer  <CTRL><SHIFT> W, enter P:


E.2 Cursor Movement within Window
---------------------------------

     Up               <CTRL><up arrow>
     Down             <CTRL><down arrow>
     Right            <CTRL><right arrow>
     Left             <CTRL><left arrow>
     Start of Line    <CTRL><SHIFT> <
     End of Line      <CTRL><SHIFT> >
     Next Line        <RETURN>
     Tab              <TAB>


E.3 Tab Handling
----------------

     Set Tab          <SHIFT><SET TAB>
     Clear Tab        <CTRL><CLR TAB>


E.4 Window Movement
-------------------

     Start of File    <CTRL><SHIFT>H
     End of File      <CTRL><SHIFT>E
     Up one Screen    <CTRL><SHIFT> <up arrow>
     Down one Screen  <CTRL><SHIFT> <down arrow>
     Left 1 Char.     <CTRL><SHIFT> ]
     Right 1 Char.    <CTRL><SHIFT> [


E.5 Text Entry
--------------

     Enter Program    enter text
     Next Line        <RETURN>
     Control Chars.   precede each character with <ESC>
```

```
E.6 Delete Text
---------------

     Back 1 Char.    <BACK S>
     Cursor          <CTRL><DELETE>
     Delete Line     position cursor on line,
                                     <SHIFT><DELETE>


E.7 Insert / Replace Text
-------------------------

     Toggle Modes    <CTRL><SHIFT> I
     Insert Line     <SHIFT><INSERT>


E.8 Restore Altered Line
------------------------

     Restore Line    do not move cursor, <CTRL><SHIFT> U
     Recall Line     do not move cursor, <CTRL><SHIFT> P


E.9 Text Blocks
---------------

     Load Block      position cursor, <SHIFT><DELETE>
                     until done
     Paste Block     position cursor, <CTRL><SHIFT> P


E.1Ø Searches / Substitutions
-----------------------------

     Find String     <CTRL><SHIFT> F, enter string
     Substitute      <CTRL><SHIFT> S, enter new string,
                     <RETURN>, enter old string


E.11 Breaking s Combining Lines
-------------------------------

     Break Line      position cursor, <CTRL><SHIFT> <RETURN
     Combine Line    put cursor at front of second line,
                     <CTRL><SHIFT> <BACK S>


E.12 Leaving the Editor
-----------------------

     Leave Editor    <CTRL><SHIFT> M
```

```
         Appendix F: Summary of ACTION! Monitor Commands
         ----------------------------------------------


B                     restart ACTION! system

C {"<filespec>"}      compile an ACTION! program

D                     call DOS

E                     go to the ACTION! Editor

O                     go to the ACTION! Options Menu

P                     proceed from program halt

R {"<filespec>"}      run an ACTION! program

SET <address> = <value>
                      sets a value in a specified
                      memory location

W {"<filespec>"}      save a compiled program to disk

X <statement>|:, <statement>:|
                      execute ACTION! language statement(s)

? <address>           display value of an address (or
                      compiler constant)

* <address>           display values of all addresses
                      starting  at  an address (or
                      compiler constant)
```

    prompt   default   range
    ------   -------   -----

    Display on   Y   Y or N
         Controls the screen during compile & device I/O

    Bell off   N   Y or N
         Controls bell response.

    Case insensitive N   Y or N
         Controls the compiler check for upper case key words
         in the language and the case distinction in variable
         names.

    Trace on   N   Y or N
         Controls compiler setup of programs so that the
         program, during execution, notes entry into any
         PROCedure or FUNCtion.

    List on   N   Y or N
         Controls compiler listing of program lines to screen
         during compile process.

    Window size   18   5 to 18
         Controls window 1 size. Window 1 and window 2,
         combined, use 23 lines.

    Line size   12Ø   1 to 24Ø
         Controls line length.

    Left margin   2   Ø to 39
         Controls left margin in window; set as low as you find
         comfortable.

    EOL character $9B   any ATASCII character
         Change the End-Of-Line character to aid visualization
         of program.

```
                    Appendix H: "PRIMES" Benchmark
                    -----------------------------
This is the benchmark test from September, 1981 BYTE
Magazine, pp. 180-198, as implemented in ACTION! Here is a
table of our times to compare with those in the magazine:

                  Mode          Time
                  ----          ----
             Compilation    ~.25 sec.
             Display off    12.2 sec.
             Display on     17.9 sec.

    DEFINE size = "8190",
           ON = "1",
           OFF = "0"

    BYTE ARRAY flags(size+1)

    CARD count, i, k, prime

    BYTE DISPLAY=$22F,
         iter,
         tick=20,
         tock=19

PROC Primes()
    DISPLAY = 0 ;comment this line to leave display on
    tick = 0
    tock=0
    FOR iter=1 TO 10
        DO
        count = 0
        ; turn flags on (non-zero)
        SetBlock(flags, size, ON)
        FOR i = 0 TO size
            DO
            IF flags(i) THEN
                prime = i+i+3
                ;PrintCE(prime) ;Uncomment to print primes
                k = prime + i
                WHILE k <= size
                    DO
                    flags(k) = OFF
                    k ==+ prime
                    OD
                count ==+ 1
            FI
            OD
        OD
    i=tick+256*tock
    DISPLAY = $22 ;turn display back on
    PrintF("%U Primes done in %U ticks %E", count, i)
```

RETURN

```
  Appendix I: Converting BASIC Concepts to ACTION! Programs
  ---------------------------------------------------------

This appendix presents several BASIC functions routines,
statements, etc. For each BASIC example given, a
corresponding ACTION! example is also given.

In the BASIC examples given, no line numbers are shown
unless necessary for illustration purposes. You should
assume the existence of appropriate line numbers in most
cases.

In the ACTION! examples shown, assume the following
variable declarations:

        INT i,j,k
        CARD c,d,e
        BYTE a,b
        BYTE ARRAY s,t,aa,ba
        CARD ARRAY ca,da,ea
        INT ARRAY ia,ja,ka


BASIC statements               ACTION! equivalents
----------------               -------------------

C=D+I*A                        c = d+ i* a

IF A<>Ø THEN B=1               IF a<>Ø THEN b=1 FI

1Ø IF A=Ø THEN 3Ø             IF a<>Ø THEN
2Ø B=1 : C=A*2                    b=1   c=a*2
3Ø EM                          FI

1Ø IF A=Ø THEN B=1 GOTO 3Ø    IF a=Ø THEN b=1
2Ø B=7                         ELSE b=7
3Ø REM                         FI

FOR I=1 TO 1ØØ ...            FOR i = 1 TO 1ØØ DO ...
       NEXT I                  OD

PRINT "HELLO"                  PrintE("HELLO")

PRINT "HELLO";                 Print("HELLO")

PRINT #5;"HELLO"               PrintDE(5,"HELLO")

PRINT #5;"HELLO";              PrintD(5,"HELLO")

PRINT I                        PrintIE(i)
```

```
PRINT "I=";I                 PrintF("I=AIAE", i)
                                   or
                             Print("I=") PrintIE(i)

PRINT #3; B*3;               PrintBD(3, b*3)

INPUT I                      Put('?) : i=InputI()
```

    Note the use of the optional colon (:) in the ACTION!
    example. Colons are ignored by ACTION! and so may used
    as statement separators.

```
INPUT B$                     Put('?) : InputS(ba)

PUT #0,65                    Put('A)
                                or
                             Put(65)
                                or
                             Put($41)

GET #C,B                     b = GetD(c)

OPEN=1,4,0,"K:"              Open(1, "K:", 4, 0)

CLOSE #3                     Close(3)

NOTE #1,C,B                  Note (1, @c, @b)

POINT #1,C,B                 Point(1, c, b)

XIO 18,#6,0,0,"S:"           XIO(6,0,18,0,0,"S:")
                                 or see also the Fill
                                 library routine

B=PEEK(C)                    b = Peek(c)
                               or, in better ACTION! form,
                             ba = c    b = ba"

POKE C,B                     Poke(c,b)
                               or, in better ACTION! form,
                             ba - c : ba" = b

GRAPHICS 8                   Graphics(8)

COLOR 3                      color = 3
                                 Note: color is a system
                                 library variable and is
                                 predefined by ACTION!

DRAWTO C,D                   DraWTo(c,d)
```

```
LOCATE C,D,B                  b = Locate(c,d)

PLOT C,D                      Plot(c,d)

POSITION C,D                  Position(c,d)

SETCOLOR Ø,1,C                SetColor(Ø,1,c)

GRAPHICS 24 : COLOR C :       Graphics(24) : color = c
  PLOT 2ØØ,15Ø :              Plot(2ØØ,15Ø)
  DRAWTO 12Ø,2Ø :             DrawTo(12Ø,2Ø
  POSITION 4Ø,15Ø :           Fill(4Ø,15Ø)
  POKE 765,C :
  XIO 18,#6,Ø,Ø,"S:"

SOUND Ø,121,1Ø,6              Sound(Ø,121,1Ø,6)

C=PADDLE(B)                   c = Paddle(b)

C=STRIG(B)                    c = Ptrig(b)

C=STICK(B)                    c = Stick(b)

C=STRIG(B)                    c = Strig(b)

B$=S$                         SCopy(ba, s)

B$=S$(3,5)                    SCopyS(ba, s, 3, 5)

B$(3,5)=S$                    SAssign(ba, s, 3, 5)

B=INT(6*RND(Ø))+1             b = Rand(6) + 1

FOR C=4ØØØ TO 5ØØØ :          Zero(4ØØØ, 1ØØ1)
    POKE C,Ø : NEXT C

STOP                          Break()

B$=STR$(I)                    StrI(i, ba)

I=VAL(S$)                     i = ValI(s)
```

The Runtime was disassembled by Erhard. The process is not finished, but
this is the current status. The interdependency between some of the
routines are marked by dotted underlines. For the writing of a custom
runtime these need to be obeyed.

The table scheme is noted below.

Addres s	Routine
Sourcecode	Sourcecode

$6ØØØ	MODULE ; SYS.ACT
	; (c) 1983,1984 ACS
	; Copyright (c) 1983,1984 ; by Action Computer Services (ACS) ; All rights reserved. ; ; version 1.4 ; last modified March 27, 1984
	DEFINE STRING="CHAR ARRAY" DEFINE EOL="$9B" DEFINE OpenBuf = "$Ø5ØØ" DEFINE OpenBufL = "$ØØ" DEFINE OpenBufH = "$Ø5"
	STRING copy_right(Ø) = "(c)1983 Action Computer Services"

;	*= $6ØØØ .BYTE $Ø1,$2Ø	

$6ØØ2	;Primitive IO routines PROC Clos=*(BYTE d) [$FFA2$A686$CAØ$ADØ]

P_CLOS	LDX #$FF ; A2 FF STX LØØA6 ; 86 A6 LDY #$ØC ; AØ ØC BNE L6Ø14 ; DØ ØA	

$6ØØA	PROC Output=*(BYTE d,STRING s) [$A684$BAØ$4DØ]

P_OUTPUT	STY LØØA6 ; 84 A6	

	LDY #$ØB	; AØ ØB		
	BNE L6Ø14	; DØ Ø4		

$6Ø1Ø	PROC In=*(BYTE d,STRING s)
	[$A684$5AØ$A586$A2$Ø$A386]

P_IN	STY LØØA6	; 84 A6		
	LDY #$Ø5	; AØ Ø5		
L6Ø14	STX LØØA5	; 86 A5		
	LDX #$ØØ	; A2 ØØ		
	STX LØØA3	; 86 A3		

$6Ø1A	PROC XIOstr=*(BYTE d,x,c,a1,a2,STRING s)
	[$AØA$AØA$98AA$9D342A3A5$AFØ$9D$34A$A4A5$9D$34B$A9$Ø$9DA8$349$A5B1$
	9D$348$12FØ18A5A5$169$9D344A6A5$69$Ø$9D$345$4C$E456$6Ø]

P_XIOSTR	ASL A	; ØA		STA ICBLH,X	; 9D 49 Ø3
	ASL A	; ØA		LDA (LØØA5),Y	; B1 A5
	ASL A	; ØA		STA ICBLL,X	; 9D 48 Ø3
	ASL A	; ØA		BEQ L6Ø4E	; FØ 12
	TAX	; AA		CLC	; 18
	TYA	; 98		LDA LØØA5	; A5 A5
	STA ICCOM,X	; 9D 42 Ø3		ADC #$Ø1	; 69 Ø1
	LDA LØØA3	; A5 A3		STA ICBAL,X	; 9D 44 Ø3
	BEQ L6Ø31	; FØ ØA		LDA LØØA6	; A5 A6
	STA ICAX1,X	; 9D 4A Ø3		ADC #$ØØ	; 69 ØØ
	LDA LØØA4	; A5 A4		STA ICBAH,X	; 9D 45 Ø3
	STA ICAX2,X	; 9D 4B Ø3		JMP CIOV	; 4C 56 E4
	LDA #$ØØ	; A9 ØØ	L6Ø4E	RTS	; 6Ø
L6Ø31	TAY	; A8			

$6Ø4F	PROC Opn=*(BYTE d,STRING s,BYTE m,o)
	[$A586$A684$3AØ$4CXIOstr]

P_OPN	STX LØØA5	; 86 A5		
	STY LØØA6	; 84 A6		
	LDY #$Ø3	; AØ Ø3		
	JMP P_XIOSTR	; 4C 1A 6Ø		

$6Ø58	PROC Prt=*(BYTE d,STRING s)
	[$A586$A684$A2$Ø$A386$9AØ$2ØXIOstr$ADØ$BA9$9D$342$9BA9$4C$E456$6Ø]

P_PRT	STX LØØA5	; 86 A5		BNE L6Ø71	; DØ ØA
	STY LØØA6	; 84 A6		LDA #$ØB	; A9 ØB
	LDX #$ØØ	; A2 ØØ		STA ICCOM,X	; 9D 42 Ø3
	STX LØØA3	; 86 A3		LDA #$9B	; A9 9B
	LDY #$Ø9	; AØ Ø9		JMP CIOV	; 4C 56 E4
	JSR P_XIOSTR	; 2Ø 1A 6Ø	L6Ø71	RTS	; 6Ø

	PROC Error(BYTE err) [$6C$A0$1113$8301]	
L6072 P_ERROR() L6076	CLC ; 18 JMP L6076 ; 4C 76 60 STA L6072 ; 8D 72 60 JMP (DOSVEC) ; 6C 0A 00 .BYTE $13,$11,$01,$83	

$6080	PROC Break=*() [BA8E$4C1$80A0$98$4C Error]	
P_BREAK	TSX ; BA STX L04C1 ; 8E C1 04 LDY #$80 ; A0 80 TYA ; 98 JMP P_ERROR() ; 4C 73 60	

$608A	;math library routines PROC LShift=*() [$84A4$AF0$8586$A$8526$88$FAD0$85A6$60]		
P_LSHIFT L6090	LDY L0084 ; A4 84 BEQ L6098 ; F0 0A STX L0085 ; 86 85 ASL A ; 0A ROL L0085 ; 26 85	 L6098	DEY ; 88 BNE L6090 ; D0 FA LDX L0085 ; A6 85 RTS ; 60

$6099	PROC RShift=*() [$84A4$AF0$8586$8546$6A$88$FAD0$85A6$60]		
P_RSHIFT L609F	LDY L0084 ; A4 84 BEQ L60A7 ; F0 0A STX L0085 ; 86 85 LSR L0085 ; 46 85 ROR A ; 6A	 L60A7	DEY ; 88 BNE L609F ; D0 FA LDX L0085 ; A6 85 RTS ; 60

$60A8 $60AC	PROC SetSign=*() [$D3A4$1010] PROC SS1=*() [$8685$8786$38$A9$0$86E5$A8$A9$0$87E5AA98$60]		
P_SETSIGN P_SS1	LDY L00D3 ; A4 D3 BPL L60BC ; 10 10 STA L0086 ; 85 86 STX L0087 ; 86 87 SEC ; 38 LDA #$00 ; A9 00 SBC L0086 ; E5 86	 L60BC	TAY ; A8 LDA #$00 ; A9 00 SBC L0087 ; E5 87 TAX ; AA TYA ; 98 RTS ; 60

$6ØBD	PROC SMOps=*()
	[$D386$EØ$Ø$31Ø$2ØSS1$8285$8386$85A5$E1Ø$AA$D345$D385$84A5$2ØSS1$8485$8586$A9$Ø$8785$6Ø]

P_SMOPS	STX LØØD3	; 86 D3		EOR LØØD3	; 45 D3
	CPX #$ØØ	; EØ ØØ		STA LØØD3	; 85 D3
	BPL L6ØC6	; 1Ø Ø3		LDA LØØ84	; A5 84
	JSR P_SS1	; 2Ø AC 6Ø		JSR P_SS1	; 2Ø AC 6Ø
L6ØC6	STA LØØ82	; 85 82		STA LØØ84	; 85 84
	STX LØØ83	; 86 83		STX LØØ85	; 86 85
	LDA LØØ85	; A5 85	L6ØDC	LDA #$ØØ	; A9 ØØ
	BPL L6ØDC	; 1Ø ØE		STA LØØ87	; 85 87
	TAX	; AA		RTS	; 6Ø

$6ØE1	PROC MultB=*()
	[$1BFØ$CA$C786$AA$15FØ$C686$A9$Ø$8A2$A$C6Ø6$29Ø$C765$CA$F6DØ$18$8765$8785$86A5$87A6$6Ø]

P_MULTB	BEQ L6ØFE	; FØ 1B		BCC L6ØF6	; 9Ø Ø2
	DEX	; CA		ADC LØØC7	; 65 C7
	STX LØØC7	; 86 C7	L6ØF6	DEX	; CA
	TAX	; AA		BNE L6ØEF	; DØ F6
	BEQ L6ØFE	; FØ 15		CLC	; 18
	STX LØØC6	; 86 C6		ADC LØØ87	; 65 87
	LDA #$ØØ	; A9 ØØ		STA LØØ87	; 85 87
	LDX #$Ø8	; A2 Ø8	L6ØFE	LDA LØØ86	; A5 86
L6ØEF	ASL A	; ØA		LDX LØØ87	; A6 87
	ASL LØØC6	; Ø6 C6		RTS	; 6Ø

$61Ø3	PROC MultI=*()
	[$2ØSMOps$82A6$1BFØ$C686$84A6$15FØCAC786$8A2$A8726C6Ø6$69Ø$C765$29Ø$87E6CAFØDØ$8685$82A5$85A6$2ØMultB$83A5$84A6$2ØMultB$4CSetSign]

P_MULTI	JSR P_SMOPS	; 2Ø BD 6Ø		ADC LØØC7	; 65 C7
	LDX LØØ82	; A6 82		BCC L6122	; 9Ø Ø2
	BEQ L6125	; FØ 1B		INC LØØ87	; E6 87
	STX LØØC6	; 86 C6	L6122	DEX	; CA
	LDX LØØ84	; A6 84		BNE L6115	; DØ FØ
	BEQ L6125	; FØ 15	L6125	STA LØØ86	; 85 86
	DEX	; CA		LDA LØØ82	; A5 82
	STX LØØC7	; 86 C7		LDX LØØ85	; A6 85
	LDX #$Ø8	; A2 Ø8		JSR P_MULTB	; 2Ø E1 6Ø
L6115	ASL A	; ØA		LDA LØØ83	; A5 83
	ROL LØØ87	; 26 87		LDX LØØ84	; A6 84
	ASL LØØC6	; Ø6 C6		JSR P_MULTB	; 2Ø E1 6Ø
	BCC L6122	; 9Ø Ø6		JMP P_SETSIGN	; 4C A8 6Ø

$6138	PROC DivI=*()
	[$2ØSMOps$85A5$27FØ$8A2$8226$8326$8726$38$83A5$84E5$A8$87A5$85E5$49Ø$
	8785$8384$CA$E7DØ$82A5$2A$A2$Ø$83A4$8684$4CSetSign$1ØA2$8226$8326$2A$
	4BØ$84C5$39Ø$84E5$38CAEFDØ$8226$8326$8685$82A5$83A6$4CSetSign]

P_DIVI	JSR P_SMOPS	; 2Ø BD 6Ø		LDY LØØ83	; A4 83
	LDA LØØ85	; A5 85		STY LØØ86	; 84 86
	BEQ L6166	; FØ 27		JMP P_SETSIGN	; 4C A8 6Ø
	LDX #$Ø8	; A2 Ø8	L6166	LDX #$1Ø	; A2 1Ø
L6141	ROL LØØ82	; 26 82	L6168	ROL LØØ82	; 26 82
	ROL LØØ83	; 26 83		ROL LØØ83	; 26 83
	ROL LØØ87	; 26 87		ROL A	; 2A
	SEC	; 38		BCS L6173	; BØ Ø4
	LDA LØØ83	; A5 83		CMP LØØ84	; C5 84
	SBC LØØ84	; E5 84		BCC L6176	; 9Ø Ø3
	TAY	; A8	L6173	SBC LØØ84	; E5 84
	LDA LØØ87	; A5 87		SEC	; 38
	SBC LØØ85	; E5 85	L6176	DEX	; CA
	BCC L6157	; 9Ø Ø4		BNE L6168	; DØ EF
	STA LØØ87	; 85 87		ROL LØØ82	; 26 82
	STY LØØ83	; 84 83		ROL LØØ83	; 26 83
L6157	DEX	; CA		STA LØØ86	; 85 86
	BNE L6141	; DØ E7		LDA LØØ82	; A5 82
	LDA LØØ82	; A5 82		LDX LØØ83	; A6 83
	ROL A	; 2A		JMP P_SETSIGN	; 4C A8 6Ø
	LDX #$ØØ	; A2 ØØ			

$6186	PROC RemI=*()
	[$2Ø DivI$86A5$87A6$6Ø]

P_REMI	JSR P_DIVI	; 2Ø 38 61
	LDA LØØ86	; A5 86
	LDX LØØ87	; A6 87
	RTS	; 6Ø

$618E	PROC SArgs=*()
	[$AØ85$A186$A284$18$68$8485$369$A8$68$8585$69$Ø$48$98$48$1AØ$84B1$828
	5$C8$84B1$8385$C8$84B1$A8$B9$AØ$Ø$8291$88$F81Ø$11A5$FDØ$11E6$4C
	Break$63Ø8$11Ø9$1819$2113$3323$6Ø]

P_SARGS	STA FRET	; 85 AØ		STA LØØ83	; 85 83
	STX FRET+1	; 86 A1		INY	; C8
	STY BPTR2	; 84 A2		LDA (LØØ84),Y	; B1 84
	CLC	; 18		TAY	; A8
	PLA	; 68	L61B2	LDA FRET,Y	; B9 AØ ØØ
	STA LØØ84	; 85 84		STA (LØØ82),Y	; 91 82
	ADC #$Ø3	; 69 Ø3		DEY	; 88
	TAY	; A8		BPL L61B2	; 1Ø F8
	PLA	; 68		LDA BRKKEY	; A5 11
	STA LØØ85	; 85 85		BNE L61CD	; DØ ØF
	ADC #$ØØ	; 69 ØØ		INC BRKKEY	; E6 11
	PHA	; 48		JMP P_BREAK	; 4C 8Ø 6Ø
	TYA	; 98		.WORD $63Ø8	
	PHA	; 48		.WORD $11Ø9	
	LDY #$Ø1	; AØ Ø1		.WORD $1819	
	LDA (LØØ84),Y	; B1 84		.WORD $2113	
	STA LØØ82	; 85 82		.WORD $3323	
	INY	; C8	L61CD	RTS	; 6Ø
	LDA (LØØ84),Y	; B1 84			

$61CE	SET $4E4=LShift
	SET $4E6=RShift
	SET $4E8=MultI
	SET $4EA=DivI
	SET $4EC=RemI
	SET $4EE=SArgs
	PROC ChkErr=*(BYTE r,b,eC)
	[$161Ø$88CØ$8FØ$98$8ØCØ$12FØ$4C Error$8A$4A4A$4A4A$98AA$9D EOF$6Ø]

P_CHKERR	BPL L61E6	; 1Ø 16		LSR A	; 4A
	CPY #$88	; CØ 88		LSR A	; 4A
	BEQ L61DC	; FØ Ø8		LSR A	; 4A
	TYA	; 98		LSR A	; 4A
	CPY #$8Ø	; CØ 8Ø		TAX	; AA
	BEQ L61EB	; FØ 12		TYA	; 98
	JMP P_ERROR()	; 4C 73 6Ø		STA EOF,X	; 9D CØ Ø5
L61DC	TXA	; 8A	L61E6	RTS	; 6Ø

$61E7	PROC Break1=*(BYTE err)
	[$1A2$1186$48$2Ø Break68A8$6Ø]

P_BREAK1	LDX #$Ø1	; A2 Ø1		PLA	; 68
	STX BRKKEY	; 86 11		TAY	; A8
L61EB	PHA	; 48		RTS	; 6Ø
	JSR P_BREAK	; 2Ø 8Ø 6Ø			

$61F2	PROC Open=*(BYTE d,STRING f,BYTE m,a2) [48A186$A284$A8$A9$Ø$99 EOF$A8$A1B1$8D OpenBuf $A8$C8$9BA9$2DØ$A1B1$99 OpenBuf 88F8DØ68A2 OpenBufL $AØ OpenBufH $2ØOpn$4C ChkErr]

P_OPEN	PHA	; 48		LDA #$9B	; A9 9B
	STX FRET+1	; 86 A1		BNE L62ØB	; DØ Ø2
	STY BPTR2	; 84 A2	L62Ø9	LDA (FRET+1),Y	; B1 A1
	TAY	; A8	L62ØB	STA OPENBUF,Y	; 99 ØØ Ø5
	LDA #$ØØ	; A9 ØØ		DEY	; 88
	STA EOF,Y	; 99 CØ Ø5		BNE L62Ø9	; DØ F8
	TAY	; A8		PLA	; 68
	LDA (FRET+1),	; B1 A1		LDX # <OPENBUF	; A2 ØØ
	STA OPENBUF	; 8D ØØ Ø5		LDY # >OPENBUF	; AØ Ø5
	TAY	; A8		JSR P_OPN	; 2Ø 4F 6Ø
	INY	; C8		JMP P_CHKERR	; 4C CE 61

$621C	PROC PrintE=*(STRING s) [$A186$AA$A1A4$A5device] PROC PrintDE=*(BYTE d,STRING s) [$2Ø Prt$4C ChkErr]

P_PRINTE	STX FRET+1	; 86 A1	P_PRINTDE	LDA DEVICE	; A5 B7
	TAX	; AA		JSR P_PRT	; 2Ø 58 6Ø
	LDY FRET+1	; A4 A1		JMP P_CHKERR	; 4C CE 61

$6229	PROC Close=*(BYTE d) [$2Ø Clos$4C ChkErr]

P_CLOSE	JSR P_CLOS	; 2Ø Ø2 6Ø
	JMP P_CHKERR	; 4C CE 61

$622F	PROC Print=*(STRING s) [$A186$AA$A1A4$A5device]
$6236	PROC PrintD=*(BYTE d,STRING s) [$2ØOutput$4C ChkErr]

P_PRINT	STX FRET+1	; 86 A1	P_PRINTD	JSR P_OUTPUTQ	; 2Ø ØA 6Ø
	TAX	; AA		JMP P_CHKERR	; 4C CE 61
	LDY FRET+1	; A4 A1			
	LDA DEVICE	; A5 B7			

$623C	PROC InS=*() [$2ØIn$AØ84BD348$3FØ$38$1E9$AØ$Ø$A591$AØA4$6Ø]

P_INS	JSR P_IN	; 2Ø 1Ø 6Ø		SBC #$Ø1	; E9 Ø1
	STY FRET	; 84 AØ	L6249	LDY #$ØØ	; AØ ØØ
	LDA ICBLL,X	; BD 48 Ø3		STA (LØØA5),Y	; 91 A5
	BEQ L6249	; FØ Ø3		LDY FRET	; A4 AØ
	SEC	; 38		RTS	; 6Ø

```
$6250    PROC InputS=*(STRING s)
         [$A286$AA$A2A4$A5device]
$6257    PROC InputSD=*(BYTE d,STRING s)
         [$48$FFA9$A385$68]
$625D    PROC InputMD=*(BYTE d,STRING s,BYTE m)
         [$48$A186$A284$A0$0$A3A5$A191$68$A2A4]
$626B    PROC InputD=*(BYTE d,STRING s)
         [$20InS$4C ChkErr]
```

P_INPUTS	STX BPTR2	; 86 A2		STX FRET+1	; 86 A1
	TAX	; AA		STY BPTR2	; 84 A2
	LDY BPTR2	; A4 A2		LDY #$00	; A0 00
	LDA DEVICE	; A5 B7		LDA L00A3	; A5 A3
P_INPUTSD	PHA	; 48		STA (FRET+1),Y	; 91 A1
	LDA #$FF	; A9 FF		PLA	; 68
	STA L00A3	; 85 A3		LDY BPTR2	; A4 A2
	PLA	; 68	P_INPUTD	JSR P_INS	; 20 3C 62
P_INPUTMD	PHA	; 48		JMP P_CHKERR	; 4C CE 61

```
$6271    CHAR FUNC GetD=*(BYTE d)
         [$7A2]
$6273    PROC CCIO=*()
         [$A386$A0A$A0A$AA$A3A5$9D$342$A9$0$9D$348$9D$349$98$20$E456$A085$4C
         ChkErr]
```

F_GETD	LDX #$07	; A2 07		STA ICCOM,X	; 9D 42 03
P_CCIO	STX L00A3	; 86 A3		LDA #$00	; A9 00
	ASL A	; 0A		STA ICBLL,X	; 9D 48 03
	ASL A	; 0A		STA ICBLH,X	; 9D 49 03
	ASL A	; 0A		TYA	; 98
	ASL A	; 0A		JSR CIOV	; 20 56 E4
	TAX	; AA		STA FRET	; 85 A0
	LDA L00A3	; A5 A3		JMP P_CHKERR	; 4C CE 61

```
$6290    PROC PutE=*()
         [$A9$9B]
$6292    PROC Put=*(CHAR c)
         [$AA$A5device]
$6295    PROC PutD=*(BYTE d,CHAR c)
         [$A186$A1A4]
$6299    PROC PutD1=*()
         [$BA2$4C CCIO]
```

P_PUTE	LDA #$9B	; A9 9B	P_PUTD	STX FRET+1	; 86 A1
P_PUT	TAX	; AA		LDY FRET+1	; A4 A1
	LDA DEVICE	; A5 B7	P_PUTD1	LDX #$0B	; A2 0B
				JMP P_CCIO	; 4C 73 62

```
$629E    PROC PutDE=*(BYTE dev)
         [$A0$9B$F7D0]
```

P_PUTDE	LDY #$9B	; A0 9B	
	BNE P_PUTD1	; D0 F7	

$62A2	PROC XIO=*(BYTE d,f,c,a1,a2,STRING s) [$2ØXIOstr$4C ChkErr]	
P_XIO	JSR P_XIOSTR ; 2Ø 1A 6Ø JMP P_CHKERR ; 4C CE 61	

$62A8	PROC CToStr=*() [$D485$D586$2Ø$D9AA$2Ø$D8E6$FFAØ$A2$Ø$C8$E8$F3B1$9D$55Ø$F71Ø$8Ø49$9D$ 55Ø$8E$55Ø$6Ø]	
P_CTOSTR L62B6	STA LØØD4 ; 85 D4 STX LØØD5 ; 86 D5 JSR IFP ; 2Ø AA D9 JSR FASC ; 2Ø E6 D8 LDY #$FF ; AØ FF LDX #$ØØ ; A2 ØØ INY ; C8 INX ; E8	LDA (LØØF3),Y ; B1 F3 STA LØ55Ø,X ; 9D 5Ø Ø5 BPL L62B6 ; 1Ø F7 EOR #$8Ø ; 49 8Ø STA LØ55Ø,X ; 9D 5Ø Ø5 STX LØ55Ø ; 8E 5Ø Ø5 RTS ; 6Ø

$62C8 $62CA $62CF	PROC PrintB=*(BYTE n) [$A2$Ø] PROC PrintC=*(CARD n) [$2Ø CToStr$A5device] PROC PNum=*() [$5ØA2$5AØ$2Ø Output$4C ChkErr]	
P_PRINTB P_PRINTC	LDX #$ØØ ; A2 ØØ JSR P_CTOSTR ; 2Ø A8 62 LDA DEVICE ; A5 B7	P_PNUM	LDX #$5Ø ; A2 5Ø LDY #$Ø5 ; AØ Ø5 JSR P_OUTPUTQ ; 2Ø ØA 6Ø JMP P_CHKERR ; 4C CE 61

(merged table below)

$62C8 $62CA $62CF	PROC PrintB=*(BYTE n) [$A2$Ø] PROC PrintC=*(CARD n) [$2Ø CToStr$A5device] PROC PNum=*() [$5ØA2$5AØ$2Ø Output$4C ChkErr]
P_PRINTB LDX #$ØØ ; A2 ØØ P_PRINTC JSR P_CTOSTR ; 2Ø A8 62 LDA DEVICE ; A5 B7	P_PNUM LDX #$5Ø ; A2 5Ø LDY #$Ø5 ; AØ Ø5 JSR P_OUTPUTQ ; 2Ø ØA 6Ø JMP P_CHKERR ; 4C CE 61

$62D9 $62DB	PROC PrintBE=*(BYTE n) [$A2$Ø] PROC PrintCE=*(CARD n) [$2ØPrintC$4CPutE]	
P_PRINTBE P_PRINTCE	LDX #$ØØ ; A2 ØØ JSR P_PRINTC ; 2Ø CA 62 JMP P_PUTE ; 4C 9Ø 62	

$62E1 $62E3	PROC PrintBD=*(BYTE d, n) [$AØ$Ø] PROC PrintCD=*(BYTE d, CARD n) [$AØ85$8A$A284$8A$A2A6$2Ø CToStr$AØA5$4CPNum]	
P_PRINTBD P_PRINTCD	LDY #$ØØ ; AØ ØØ STA FRET ; 85 AØ TXA ; 8A STY BPTR2 ; 84 A2	LDX BPTR2 ; A6 A2 JSR P_CTOSTR ; 2Ø A8 62 LDA FRET ; A5 AØ JMP P_PNUM ; 4C CF 62

$62F2	PROC PrintBDE=*(BYTE d,n)
	[$AØ$Ø]
$62F4	PROC PrintCDE=*(BYTE d,CARD n)
	[$2ØPrintCD$AØA5$4CPutDE]

```
P_PRINTBDE  LDY #$ØØ      ; AØ ØØ
P_PRINTCDE  JSR P_PRINTCD ; 2Ø E3 62
            LDA FRET      ; A5 AØ
            JMP P_PUTDE   ; 4C 9E 62
```

$62FC	PROC PrintI=*(INT n)
	[$A286$AA$A2A4$A5device]
$63Ø3	PROC PrintID=*(BYTE d,INT n)
	[$CØ$Ø$161Ø$48$A186$A284$2DAØ$2ØPutD1$38$A9$Ø$A1E5AAA9$Ø$A2E5$A8$68
	$4CPrintCD]

```
P_PRINTI    STX BPTR2   ; 86 A2        JSR P_PUTD1   ; 2Ø 99 62
            TAX         ; AA           SEC           ; 38
            LDY BPTR2   ; A4 A2        LDA #$ØØ      ; A9 ØØ
            LDA DEVICE  ; A5 B7        SBC FRET+1    ; E5 A1
P_PRINTID   CPY #$ØØ    ; CØ ØØ        TAX           ; AA
            BPL L631D   ; 1Ø 16        LDA #$ØØ      ; A9 ØØ
            PHA         ; 48           SBC BPTR2     ; E5 A2
            STX FRET+1  ; 86 A1        TAY           ; A8
            STY BPTR2   ; 84 A2        PLA           ; 68
            LDY #$2D    ; AØ 2D  L631D  JMP P_PRINTCD ; 4C E3 62
```

| $632Ø | PROC PrintIE=*(INT n) |
| | [$2ØPrintI$4CPutE] |

```
P_PRINTIE   JSR P_PRINTI  ; 2Ø FC 62
            JMP P_PUTE    ; 4C 9Ø 62
```

| $6326 | PROC PrintIDE=*(BYTE d,INT n) |
| | [$2ØPrintID$AØA5$4CPutDE] |

```
P_PRINTIDE  JSR P_PRINTID ; 2Ø Ø3 63
            LDA FRET      ; A5 AØ
            JMP P_PUTDE   ; 4C 9E 62
```

$632E	PROC StrB=*(BYTE n, STRING s)
	[$A286$A384$A2$Ø$A2A4]
$6336	PROC StrC=*(CARD n, STRING s)
	[$A284$2Ø CToStr$C8$B9$55Ø$A291$88$F81Ø$6Ø]
$6345	PROC StrI=*(INT n, STRING s)
	[$EØ$Ø$ED1Ø$AØ85$A186$A284$38$A9$Ø$AØE5$A8$A9$Ø$A1E5AA98$2Ø
	CToStr$E8$8A$A8$B9$54F$A291$88$F8DØ$8A$A291$C8$2DA9$A291$6Ø]

```
P_STRB    STX BPTR2      ; 86 A2              TAY            ; A8
          STY LØØA3      ; 84 A3              LDA #$ØØ       ; A9 ØØ
          LDX #$ØØ       ; A2 ØØ              SBC FRET+1     ; E5 A1
          LDY BPTR2      ; A4 A2              TAX            ; AA
P_STRC    STY BPTR2      ; 84 A2              TYA            ; 98
          JSR P_CTOSTR   ; 2Ø A8 62           JSR P_CTOSTR   ; 2Ø A8 62
          INY            ; C8                 INX            ; E8
L633C     LDA LØ55Ø,Y    ; B9 5Ø Ø5           TXA            ; 8A
          STA (BPTR2),Y  ; 91 A2              TAY            ; A8
          DEY            ; 88        L6361     LDA LØ54F,Y    ; B9 4F Ø5
          BPL L633C      ; 1Ø F8              STA (BPTR2),Y  ; 91 A2
          RTS            ; 6Ø                 DEY            ; 88
P_STRI    CPX #$ØØ       ; EØ ØØ              BNE L6361      ; DØ F8
          BPL P_STRC     ; 1Ø ED              TXA            ; 8A
          STA FRET       ; 85 AØ              STA (BPTR2),Y  ; 91 A2
          STX FRET+1     ; 86 A1              INY            ; C8
          STY BPTR2      ; 84 A2              LDA #$2D       ; A9 2D
          SEC            ; 38                 STA (BPTR2),Y  ; 91 A2
          LDA #$ØØ       ; A9 ØØ              RTS            ; 6Ø
          SBC FRET       ; E5 AØ
```

```
$6372   BYTE FUNC InputB=*()
        CARD FUNC InputC=*()
$6374   INT FUNC InputI=*()
        [$A5 device]
$6384   BYTE FUNC InputBD=*(BYTE d)
        CARD FUNC InputCD=*(BYTE d)
        INT FUNC InputID=*(BYTE d)
        [$13A2$8E$55Ø$5ØA2$5AØ$2ØInputD$5ØA9$5A2]
        BYTE FUNC ValB=*(STRING s)
        CARD FUNC ValC=*(STRING s)
        INT FUNC ValI=*(STRING s)
        [$A485$A586$AØ$Ø$AØ84$A184$A284$A4B1$A385$A3E6$2ØA9$C8$A4D1$5DØ$C8$A3
        C4$F73Ø$A4B1$2DC9$3DØ$A285$C8$A3C4$361Ø$A4B1$3ØC9$3Ø3Ø$3AC9$2C1Ø$38$3
        ØE9$AA$A1A5$48$AØA5$A$A126$A$A126$18$AØ65$AØ85$68$A165$A185$AØØ6$A126
        $18$8A$AØ65$AØ85$29Ø$A1E6$C8$A3C4$CA3Ø$A2A5$DFØ$38$A9$Ø$AØE5$AØ85$A9$
        Ø$A1E5$A185$6Ø]
```

```
F_INPUTBCI    LDA DEVICE     ; A5 B7              SBC #$3Ø        ; E9 3Ø
F_INPUTBCID   LDX #$13       ; A2 13              TAX             ; AA
              STX LØ55Ø      ; 8E 5Ø Ø5           LDA FRET+1      ; A5 A1
              LDX # <LØ55Ø   ; A2 5Ø              PHA             ; 48
              LDY # >LØ55Ø   ; AØ Ø5              LDA FRET        ; A5 AØ
              JSR P_INPUTD   ; 2Ø 6B 62           ASL A           ; ØA
              LDA # <LØ55Ø   ; A9 5Ø              ROL FRET+1      ; 26 A1
              LDX # >LØ55Ø   ; A2 Ø5              ASL A           ; ØA
F_VALBCI      STA LØØA4      ; 85 A4              ROL FRET+1      ; 26 A1
              STX LØØA5      ; 86 A5              CLC             ; 18
              LDY #$ØØ       ; AØ ØØ              ADC FRET        ; 65 AØ
              STY FRET       ; 84 AØ              STA FRET        ; 85 AØ
              STY FRET+1     ; 84 A1              PLA             ; 68
              STY BPTR2      ; 84 A2              ADC FRET+1      ; 65 A1
              LDA (LØØA4),Y  ; B1 A4              STA FRET+1      ; 85 A1
              STA LØØA3      ; 85 A3              ASL FRET        ; Ø6 AØ
              INC LØØA3      ; E6 A3              ROL FRET+1      ; 26 A1
              LDA #$2Ø       ; A9 2Ø              CLC             ; 18
              INY            ; C8                 TXA             ; 8A
L6399         CMP (LØØA4),Y  ; D1 A4              ADC FRET        ; 65 AØ
              BNE L63A2      ; DØ Ø5              STA FRET        ; 85 AØ
              INY            ; C8                 BCC L63EØ       ; 9Ø Ø2
              CPY LØØA3      ; C4 A3              INC FRET+1      ; E6 A1
              BMI L6399      ; 3Ø F7   L63EØ      INY             ; C8
L63A2         LDA (LØØA4),Y  ; B1 A4              CPY LØØA3       ; C4 A3
              CMP #$2D       ; C9 2D              BMI L63AF       ; 3Ø CA
              BNE L63AB      ; DØ Ø3   L63E5      LDA BPTR2       ; A5 A2
              STA BPTR2      ; 85 A2              BEQ L63F6       ; FØ ØD
              INY            ; C8                 SEC             ; 38
L63AB         CPY LØØA3      ; C4 A3              LDA #$ØØ        ; A9 ØØ
              BPL L63E5      ; 1Ø 36              SBC FRET        ; E5 AØ
L63AF         LDA (LØØA4),Y  ; B1 A4              STA FRET        ; 85 AØ
              CMP #$3Ø       ; C9 3Ø              LDA #$ØØ        ; A9 ØØ
              BMI L63E5      ; 3Ø 3Ø              SBC FRET+1      ; E5 A1
              CMP #$3A       ; C9 3A              STA FRET+1      ; 85 A1
              BPL L63E5      ; 1Ø 2C   L63F6      RTS             ; 6Ø
              SEC            ; 38
```

$63F7	PROC PrintH=*(CARD n) [$A485$A586$4A9$A685$24A9$2ØPut$A9$Ø$4A2$A4Ø6$A526$2ACAF8DØ$3Ø69$3A C9$23Ø$669$2ØPut$A6C6$E5DØ$6Ø]

```
P_PRINTH   STA LØØA4   ; 85 A4              DEX         ; CA
           STX LØØA5   ; 86 A5              BNE L64Ø8   ; DØ F8
           LDA #$Ø4    ; A9 Ø4              ADC #$3Ø    ; 69 3Ø
           STA LØØA6   ; 85 A6              CMP #$3A    ; C9 3A
           LDA #$24    ; A9 24              BMI L6418   ; 3Ø Ø2
           JSR P_PUT   ; 2Ø 92 62           ADC #$Ø6    ; 69 Ø6
L64Ø4      LDA #$ØØ    ; A9 ØØ      L6418   JSR P_PUT   ; 2Ø 92 62
           LDX #$Ø4    ; A2 Ø4              DEC LØØA6   ; C6 A6
L64Ø8      ASL LØØA4   ; Ø6 A4              BNE L64Ø4   ; DØ E5
           ROL LØØA5   ; 26 A5      L641F   RTS         ; 6Ø
           ROL A       ; 2A
```

$642Ø	PROC PrintF=*(STRING f, CARD a1,a2,a3,a4,a5) [$CØ85$C186$8C$5FØ$AØ$Ø$CØB1$C285$C2E6$DA2$A2B5$9D$5FØ$CA$F8DØ$8B86$8 A86]
$643D	PROC PF2=*() [$8AE6$8AA4$C2C4$DABØ$CØB1$25C9$FDØ$8AE6$C8$CØB1$25C9$6FØ$45C9$8DØ$9B A9$2ØPut$4CPF2$8BA4$8BE6$8BE6$AØ85$B9$5FØ$BE$5F1$AØA4$43CØ$E6FØ$53CØ$ 6DØ$2ØPrint$4CPF2$49CØ$6DØ$2ØPrintI$4CPF2$48CØ$6DØ$2ØPrintH$4CPF2$2ØP rintC$4CPF2]

```
P_PRINTF   STA LØØCØ    ; 85 CØ              BNE L646Ø    ; DØ Ø8
           STX LØØC1    ; 86 C1              LDA #$9B     ; A9 9B
           STY LØ5FØ    ; 8C FØ Ø5   L645A   JSR P_PUT    ; 2Ø 92 62
           LDY #$ØØ     ; AØ ØØ              JMP P_PF2    ; 4C 3D 64
           LDA (LØØCØ),Y ; B1 CØ     L646Ø   LDY LØØ8B    ; A4 8B
           STA LØØC2    ; 85 C2              INC LØØ8B    ; E6 8B
           INC LØØC2    ; E6 C2              INC LØØ8B    ; E6 8B
           LDX #$ØD     ; A2 ØD              STA FRET     ; 85 AØ
L6431      LDA BPTR2,X  ; B5 A2              LDA LØ5FØ,Y  ; B9 FØ Ø5
           STA LØ5FØ,X  ; 9D FØ Ø5           LDX LØ5F1,Y  ; BE F1 Ø5
           DEX          ; CA                 LDY FRET     ; A4 AØ
           BNE L6431    ; DØ F8              CPY #$43     ; CØ 43
           STX LØØ8B    ; 86 8B              BEQ L645A    ; FØ E6
           STX LØØ8A    ; 86 8A              CPY #$53     ; CØ 53
P_PF2      INC LØØ8A    ; E6 8A              BNE L647E    ; DØ Ø6
           LDY LØØ8A    ; A4 8A              JSR P_PRINT  ; 2Ø 2F 62
           CPY LØØC2    ; C4 C2              JMP P_PF2    ; 4C 3D 64
           BCS L641F    ; BØ DA      L647E   CPY #$49     ; CØ 49
           LDA (LØØCØ),Y ; B1 CØ             BNE L6488    ; DØ Ø6
           CMP #$25     ; C9 25              JSR P_PRINTI ; 2Ø FC 62
           BNE L645A    ; DØ ØF              JMP P_PF2    ; 4C 3D 64
           INC LØØ8A    ; E6 8A      L6488   CPY #$48     ; CØ 48
           INY          ; C8                 BNE L6492    ; DØ Ø6
           LDA (LØØCØ),Y ; B1 CØ             JSR P_PRINTH ; 2Ø F7 63
           CMP #$25     ; C9 25              JMP P_PF2    ; 4C 3D 64
           BEQ L645A    ; FØ Ø6      L6492   JSR P_PRINTC ; 2Ø CA 62
           CMP #$45     ; C9 45              JMP P_PF2    ; 4C 3D 64
```

$6498	PROC Note=*(BYTE d,CARD POINTER s,BYTE POINTER o) [$A186$A284$AØ$AØAA26A9$9D$342$2Ø$E456$2Ø ChkErr$AØ$Ø$BD$34E$A391BD34C$A191$BD$34D$C8$A191$6Ø]

```
P_NOTE    STX FRET+1    ; 86 A1         JSR P_CHKERR    ; 2Ø CE 61
          STY BPTR2     ; 84 A2         LDY #$ØØ        ; AØ ØØ
          ASL A         ; ØA            LDA ICAX5,X     ; BD 4E Ø3
          ASL A         ; ØA            STA (LØØA3),Y   ; 91 A3
          ASL A         ; ØA            LDA ICAX3,X     ; BD 4C Ø3
          ASL A         ; ØA            STA (FRET+1),Y  ; 91 A1
          TAX           ; AA            LDA ICAX4,X     ; BD 4D Ø3
          LDA #$26      ; A9 26         INY             ; C8
          STA ICCOM,X   ; 9D 42 Ø3      STA (FRET+1),Y  ; 91 A1
          JSR CIOV      ; 2Ø 56 E4      RTS             ; 6Ø
```

$64BF	PROC Point=*(BYTE d,CARD s,BYTE o) [$A186$AØ$AØ$98AA$9D$34D$A1A5$9D$34C$A3A5$9D$34E$25A9$9D$342$2Ø$E45 6$4C ChkErr]

```
P_POINT   STX FRET+1    ; 86 A1         LDA FRET+1      ; A5 A1
          ASL A         ; ØA            STA ICAX3,X     ; 9D 4C Ø3
          ASL A         ; ØA            LDA LØØA3       ; A5 A3
          ASL A         ; ØA            STA ICAX5,X     ; 9D 4E Ø3
          ASL A         ; ØA            LDA #$25        ; A9 25
          TAX           ; AA            STA ICCOM,X     ; 9D 42 Ø3
          TYA           ; 98            JSR CIOV        ; 2Ø 56 E4
          STA ICAX4,X   ; 9D 4D Ø3      JMP P_CHKERR    ; 4C CE 61
```

$64DF	MODULE ; GRAPHIC ROUTINES
$64E9	STRING dev_S="S:", dev_E="E:" PROC Graphics=*(BYTE m) [48A9$Ø$2Ø Close$CA9$A385$A9$Ø$AEdev_E$ACdev_E+1$2ØOpen$6A9$2Ø Close$68$A485$3Ø29$1C49$A385$6A9$AEdev_S$ACdev_S+1$4COpen]

```
DEV_S      .BYTE $Ø2,$53,$3A          LDY L64E8       ; AC E8 64
L64E2      .BYTE $DF                  JSR P_OPEN      ; 2Ø F2 61
L64E3      .BYTE $64                  LDA #$Ø6        ; A9 Ø6
DEV_E      .BYTE $Ø2,$45,$3A          JSR P_CLOSE     ; 2Ø 29 62
L64E7      .BYTE $E4                  PLA             ; 68
L64E8      .BYTE $64                  STA LØØA4       ; 85 A4
P_GRAPHICS PHA             ; 48       AND #$3Ø        ; 29 3Ø
           LDA #$ØØ        ; A9 ØØ    EOR #$1C        ; 49 1C
           JSR P_CLOSE     ; 2Ø 29 62 STA LØØA3       ; 85 A3
           LDA #$ØC        ; A9 ØC    LDA #$Ø6        ; A9 Ø6
           STA LØØA3       ; 85 A3    LDX L64E2       ; AE E2 64
           LDA #$ØØ        ; A9 ØØ    LDY L64E3       ; AC E3 64
           LDX L64E7       ; AE E7 64 JMP P_OPEN      ; 4C F2 61
```

```
$6517    PROC Position=*(CARD c,BYTE r)
         [$5B85$5C86$5A84]
$651D    PROC Pos1=*()
         [$5585$5686$5484$6Ø]
```

```
P_POSITION  STA OLDCOL    ; 85 5B      P_POS1   STA COLCRS     ; 85 55
            STX OLDCOL+1  ; 86 5C               STX COLCRS+1   ; 86 56
            STY OLDROW    ; 84 5A               STY ROWCRS     ; 84 54
                                               RTS            ; 6Ø
```

```
$6524    PROC GrIO=*()
         [$2ØPos1$AD$2FD$8D$2FB$ADdev_S$A585$ADdev_S+1$A685$A9$Ø$A385$A485$6A9
         $6Ø]
```

```
P_GRIO    JSR P_POS1    ; 2Ø 1D 65              STA LØØA6      ; 85 A6
          LDA FILDAT    ; AD FD Ø2              LDA #$ØØ       ; A9 ØØ
          STA ATACHR    ; 8D FB Ø2              STA LØØA3      ; 85 A3
          LDA L64E2     ; AD E2 64              STA LØØA4      ; 85 A4
          STA LØØA5     ; 85 A5                 LDA #$Ø6       ; A9 Ø6
          LDA L64E3     ; AD E3 64              RTS            ; 6Ø
```

```
$654Ø    PROC DrawTo=*(CARD c,BYTE r)
         [$2ØGrIO$11AØ$4CXIO]
```

```
P_DRAWTO  JSR P_GRIO    ; 2Ø 24 65
          LDY #$11      ; AØ 11
          JMP P_XIO     ; 4C A2 62
```

```
$6548    BYTE FUNC Locate=*(CARD c,BYTE r)
         [$2ØPosition$6A9$4CGetD]
```

```
F_LOCATE  JSR P_POSITION ; 2Ø 17 65
          LDA #$Ø6       ; A9 Ø6
          JMP F_GETD     ; 4C 71 62
```

```
$655Ø    PROC Plot=*(CARD c,BYTE r)
         [$2ØPos1$6A9$AE$2FD$4CPutD]
```

```
P_PLOT    JSR P_POS1    ; 2Ø 1D 65
          LDA #$Ø6      ; A9 Ø6
          LDX FILDAT    ; AE FD Ø2
          JMP P_PUTD    ; 4C 95 62
```

$655B	PROC SetColor=*(BYTE reg,hue,lum)
	[$5C9$161Ø$A85$98$F29$A285$8A$AØA$AØA$A2Ø5$AØA6$9D$2C4$9D$DØ16$6Ø]

P_SETCOLOR	CMP #$Ø5	; C9 Ø5		ASL A	; ØA
	BPL L6575	; 1Ø 16		ASL A	; ØA
	STA FRET	; 85 AØ		ASL A	; ØA
	TYA	; 98		ORA BPTR2	; Ø5 A2
	AND #$ØF	; 29 ØF		LDX FRET	; A6 AØ
	STA BPTR2	; 85 A2		STA COLORØ,X	; 9D C4 Ø2
	TXA	; 8A		STA COLPFØ,X	; 9D 16 DØ
	ASL A	; ØA	L6575	RTS	; 6Ø

$6576	PROC Fill=*(CARD c,BYTE r)
	[$2ØGrIO$12AØ$4CXIO]

P_FILL	JSR P_GRIO	; 2Ø 24 65
	LDY #$12	; AØ 12
	JMP P_XIO	; 4C A2 62

$657E	BYTE FUNC Rand=*(BYTE r)
	[AED2ØA$C9$Ø$9FØ$8486$A2$Ø$8586$2ØMultI$AØ86$6Ø]

F_RAND	LDX RANDOM	; AE ØA D2		STX LØØ85	; 86 85
	CMP #$ØØ	; C9 ØØ		JSR P_MULTI	; 2Ø Ø3 61
	BEQ L658E	; FØ Ø9	L658E	STX FRET	; 86 AØ
	STX LØØ84	; 86 84		RTS	; 6Ø
	LDX #$ØØ	; A2 ØØ			

$6591	PROC Sound=*(BYTE v, p, d, vol)
	[AA284$A8$7C9$53Ø$64AØ$2Ø
	Error$998A$D2ØØ$A2A5$AØA$AØA$A3Ø5$99$D2Ø1$6Ø]

P_SOUND	ASL A	; ØA		LDA BPTR2	; A5 A2
	STY BPTR2	; 84 A2		ASL A	; ØA
	TAY	; A8		ASL A	; ØA
	CMP #$Ø7	; C9 Ø7		ASL A	; ØA
	BMI L659E	; 3Ø Ø5		ASL A	; ØA
	LDY #$64	; AØ 64		ORA LØØA3	; Ø5 A3
	JSR P_ERROR()	; 2Ø 73 6Ø		STA AUDC1,Y	; 99 Ø1 D2
L659E	TXA	; 8A		RTS	; 6Ø
	STA AUDF1,Y	; 99 ØØ D2			

$65AE	PROC SndRst=*()
	[AD232$EF29$8D$232$8D$D2ØF$A9$Ø$8A2$9D$D2ØØCAFA1Ø$6Ø]

P_SNDRST	LDA SSKCTL	; AD 32 Ø2		LDX #$Ø8	; A2 Ø8
	AND #$EF	; 29 EF	L65BD	STA AUDF1,X	; 9D ØØ D2
	STA SSKCTL	; 8D 32 Ø2		DEX	; CA
	STA SKCTL	; 8D ØF D2		BPL L65BD	; 1Ø FA
	LDA #$ØØ	; A9 ØØ		RTS	; 6Ø

$65C4	BYTE FUNC Paddle=*(BYTE p) [$BDAA$27Ø$AØ85$6Ø]	
F_PADDLE	TAX ; AA LDA PADDLØ,X ; BD 7Ø Ø2 STA FRET ; 85 AØ RTS ; 6Ø	

$65CB	BYTE FUNC PTrig=*(BYTE p) [$A2$Ø$4C9$33Ø$E8$329$A8$BD$D3ØØ$39*+5$AØ85$6Ø$8Ø4$8Ø4Ø]	
F_PTRIG	LDX #$ØØ ; A2 ØØ CMP #$Ø4 ; C9 Ø4 BMI L65D4 ; 3Ø Ø3 INX ; E8 AND #$Ø3 ; 29 Ø3	LDA PORTA,X ; BD ØØ D3 AND L65DE,Y ; 39 DE 65 STA FRET ; 85 AØ RTS ; 6Ø L65DE .BYTE $Ø4,$Ø8,$4Ø,$8Ø
L65D4	TAY ; A8	

$65E2	BYTE FUNC Stick=*(BYTE p) [$A2$Ø$2C9$33Ø$E8$129$BDA8$D3ØØ$88$4DØ$4A4A$4A4A$F29$AØ85$6Ø]	
F_STICK	LDX #$ØØ ; A2 ØØ CMP #$Ø2 ; C9 Ø2 BMI L65EB ; 3Ø Ø3 INX ; E8 AND #$Ø1 ; 29 Ø1	BNE L65F6 ; DØ Ø4 LSR A ; 4A LSR A ; 4A LSR A ; 4A LSR A ; 4A
L65EB	TAY ; A8 LDA PORTA,X ; BD ØØ D3 DEY ; 88	L65F6 AND #$ØF ; 29 ØF STA FRET ; 85 AØ RTS ; 6Ø

$65FB	BYTE FUNC STrig=*(BYTE p) [$BDAA$DØ1Ø$AØ85$6Ø]	
F_STRIG	TAX ; AA LDA TRIGØ,X ; BD 1Ø DØ STA FRET ; 85 AØ RTS ; 6Ø	

$66Ø2	BYTE FUNC Peek=*(CARD a) CARD FUNC PeekC=*(CARD a) [$A285$A386$AØ$Ø$A2B1$AØ85$C8$A2B1$A185$6Ø]	
F_PEEKC	STA BPTR2 ; 85 A2 STX LØØA3 ; 86 A3 LDY #$ØØ ; AØ ØØ LDA (BPTR2),Y ; B1 A2 STA FRET ; 85 AØ	INY ; C8 LDA (BPTR2),Y ; B1 A2 STA FRET+1 ; 85 A1 RTS ; 6Ø

$6612	PROC Poke=*(CARD a,BYTE v) [$A0 85$A1 86$A0 98$91 00$60 A0]		
P_POKE	STA FRET ; 85 A0	LDY #$00 ; A0 00	
	STX FRET+1 ; 86 A1	STA (FRET),Y ; 91 A0	
	TYA ; 98	RTS ; 60	

$661C	PROC PokeC=*(CARD a,v) [$20 Poke$A5 C8$91 A3$60 A0]
P_POKEC	JSR P_POKE ; 20 12 66
	INY ; C8
	LDA L00A3 ; A5 A3
	STA (FRET),Y ; 91 A0
	RTS ; 60

$6625	PROC Zero=*(BYTE POINTER a,CARD s) [48A9 0A4 85$68]		
$662B	PROC SetBlock=*(BYTE POINTER a,CARD s,BYTE v) [$A0 85$A1 86$A2 84$A0 0A4 A5$A3 A6$10 F0$A0 91$C8$FB D0$A1 E6$A3 C6$F5 D0$3F F0$ A0 91$C8$A2 C4$F9 D0$60]		
P_ZERO	PHA ; 48		INY ; C8
	LDA #$00 ; A9 00		BNE L6639 ; D0 FB
	STA L00A4 ; 85 A4		INC FRET+1 ; E6 A1
	PLA ; 68		DEC L00A3 ; C6 A3
P_SETBLOCK	STA FRET ; 85 A0		BNE L6639 ; D0 F5
	STX FRET+1 ; 86 A1		BEQ L6649 ; F0 03
	STY BPTR2 ; 84 A2	L6646	STA (FRET),Y ; 91 A0
	LDY #$00 ; A0 00		INY ; C8
	LDA L00A4 ; A5 A4	L6649	CPY BPTR2 ; C4 A2
	LDX L00A3 ; A6 A3		BNE L6646 ; D0 F9
	BEQ L6649 ; F0 10		RTS ; 60
L6639	STA (FRET),Y ; 91 A0		

$664E	PROC MoveBlock=*(BYTE POINTER d,s,CARD sz) [$A0 85$A1 86$A2 84$A0 0A5 A5$16 F0$A2 B1$A0 91$C8$F9 D0$A1 E6$A3 E6$A5 C6$F1 D0 $5F F0$A2 B1$A0 91$C8$A4 C4$F7 D0$60]		
P_MOVEBLOCK	STA FRET ; 85 A0		INC L00A3 ; E6 A3
	STX FRET+1 ; 86 A1		DEC L00A5 ; C6 A5
	STY BPTR2 ; 84 A2		BNE L665A ; D0 F1
	LDY #$00 ; A0 00		BEQ L6670 ; F0 05
	LDA L00A5 ; A5 A5	L666B	LDA (BPTR2),Y ; B1 A2
	BEQ L6670 ; F0 16		STA (FRET),Y ; 91 A0
L665A	LDA (BPTR2),Y ; B1 A2		INY ; C8
	STA (FRET),Y ; 91 A0	L6670	CPY L00A4 ; C4 A4
	INY ; C8		BNE L666B ; D0 F7
	BNE L665A ; D0 F9		RTS ; 60
	INC FRET+1 ; E6 A1		

$6675	INT FUNC SCompare=*(STRING a,b)
	[$A485$A586$A284$AØ$Ø$AØ84$A184$A4B1$A2D1$3FØ$2Ø*+21$C9$Ø$1DØ$6Ø$685
	$C8$A4B1$A2D1$5DØ$A6C4$F59Ø$6Ø$FFA2$AØ86$39Ø$A2B1$E8$A186$6Ø]

```
F_SCOMPARE  STA LØØA4     ; 85 A4    L6691       INY           ; C8
            STX LØØA5     ; 86 A5                LDA (LØØA4),Y ; B1 A4
            STY BPTR2     ; 84 A2                CMP (BPTR2),Y ; D1 A2
            LDY #$ØØ      ; AØ ØØ                BNE L669D     ; DØ Ø5
            STY FRET      ; 84 AØ                CPY LØØA6     ; C4 A6
            STY FRET+1    ; 84 A1                BCC L6691     ; 9Ø F5
            LDA (LØØA4),Y ; B1 A4                RTS           ; 6Ø
            CMP (BPTR2),Y ; D1 A2    L669D       LDX #$FF      ; A2 FF
            BEQ L668A     ; FØ Ø3                STX FRET      ; 86 AØ
            JSR L669D     ; 2Ø 9D 66             BCC L66A6     ; 9Ø Ø3
L668A       CMP #$ØØ      ; C9 ØØ                LDA (BPTR2),Y ; B1 A2
            BNE L668F     ; DØ Ø1                INX           ; E8
            RTS           ; 6Ø      L66A6        STX FRET+1    ; 86 A1
L668F       STA LØØA6     ; 85 A6                RTS           ; 6Ø
```

$66A9	PROC SCopy=*(STRING d,s)
	[$AØ85$A186$A284$AØ$Ø$A2B1$AØ91$8FØ$A8$A2B1$AØ91$88$F9DØ$6Ø]

```
P_SCOPY     STA FRET      ; 85 AØ    L66B7       TAY           ; A8
            STX FRET+1    ; 86 A1    L66B8       LDA (BPTR2),Y ; B1 A2
            STY BPTR2     ; 84 A2                STA (FRET),Y  ; 91 AØ
            LDY #$ØØ      ; AØ ØØ                DEY           ; 88
            LDA (BPTR2),Y ; B1 A2                BNE L66B8     ; DØ F9
L66B3       STA (FRET),Y  ; 91 AØ    L66BF       RTS           ; 6Ø
            BEQ L66BF     ; FØ Ø8
```

$66CØ	PROC SCopyS=*(STRING d,s,BYTE b,e)
	[$AØ85$A186$A284$AØ$Ø$A2B1$A5C5$2BØ$A585$A4C6$18$A2A5$A465$A285$29Ø$A
	3E6$38$A5A5$A4E5$2BØ$A9$Ø$4CSCopy+1Ø]

```
P_SCOPYS    STA FRET      ; 85 AØ                ADC LØØA4     ; 65 A4
            STX FRET+1    ; 86 A1                STA BPTR2     ; 85 A2
            STY BPTR2     ; 84 A2                BCC L66DD     ; 9Ø Ø2
            LDY #$ØØ      ; AØ ØØ                INC LØØA3     ; E6 A3
            LDA (BPTR2),Y ; B1 A2    L66DD       SEC           ; 38
            CMP LØØA5     ; C5 A5                LDA LØØA5     ; A5 A5
            BCS L66DØ     ; BØ Ø2                SBC LØØA4     ; E5 A4
            STA LØØA5     ; 85 A5                BCS L66E6     ; BØ Ø2
L66DØ       DEC LØØA4     ; C6 A4                LDA #$ØØ      ; A9 ØØ
            CLC           ; 18      L66E6        JMP L66B3     ; 4C B3 66
            LDA BPTR2     ; A5 A2
```

$66E9	PROC SAssign=*(STRING d,s,BYTE b,e) [$AØ85$A186$A284$AØ$Ø$A2B1$DFØ$A685$A4C6$38$A5A5$A4E5$2FØ$1BØ$AA6Ø$A6 C5$89Ø$18$A6A5$AA$A465$A585$A5A5$AØD1$39Ø$AØ91$18$AØA5$A465$AØ85$29Ø$ A1E6$4C8ASCopy+14]

P_SASSIGN	STA FRET	; 85 AØ			LDA LØØA6	; A5 A6	
	STX FRET+1	; 86 A1			TAX	; AA	
	STY BPTR2	; 84 A2			ADC LØØA4	; 65 A4	
	LDY #$ØØ	; AØ ØØ			STA LØØA5	; 85 A5	
	LDA (BPTR2),Y	; B1 A2		L671Ø	LDA LØØA5	; A5 A5	
	BEQ L67Ø2	; FØ ØD			CMP (FRET),Y	; D1 AØ	
	STA LØØA6	; 85 A6			BCC L6719	; 9Ø Ø3	
	DEC LØØA4	; C6 A4			STA (FRET),Y	; 91 AØ	
	SEC	; 38			CLC	; 18	
	LDA LØØA5	; A5 A5		L6719	LDA FRET	; A5 AØ	
	SBC LØØA4	; E5 A4			ADC LØØA4	; 65 A4	
	BEQ L67Ø2	; FØ Ø2			STA FRET	; 85 AØ	
	BCS L67Ø3	; BØ Ø1			BCC L6723	; 9Ø Ø2	
L67Ø2	RTS	; 6Ø			INC FRET+1	; E6 A1	
L67Ø3	TAX	; AA		L6723	TXA	; 8A	
	CMP LØØA6	; C5 A6			JMP L66B7	; 4C B7 66	
	BCC L671Ø	; 9Ø Ø8			RTS	; 6Ø	
	CLC	; 18					

	MODULE ; for user

;	

```
                    Appendix K: ACTION! BUG SHEET #3
                    -------------------------------
```

This document supersedes the previous two bug sheets published for ACTION! - November 6, 1984

NOTE: Bugs affecting the latest ACTION! versions are already applied to this manual. It is recommended to change the cartridge ROMs to V. 3.6, the Run Time Package to V. 1.4, and the Toolkit to V. 3.0.

```
                   PART 1 - GENERAL INFORMATION
                   ----------------------------
```

Before getting to the bad stuff (the bugs), here are some goodies about ACTION! which we would like to pass on to you:

Tips on Temps

A magazine article titled "Lights, Camera, ACTION!" by Dave Plotkin, which appeared in the July 1984 issue of ANTIC, featured a set of routines to facilitate writing ACTION!-based interrupt handlers.

The article gave the listings for two routines (more properly, two DEFINEs) named "SaveTemps" and "GetTemps". These routines are adequate only if no math beyond addition and subtraction is performed in the interrupt service routine. The following versions of these two routines will work properly in the more general case:

Make the following DEFINEs in your program before you declare your interrupt routine (comments may be omitted--they exist only for clarification):

```
DEFINE SaveTemps=
  "[
      $A2 $07     ;          LDX #7
      $B5 $C0     ; LOOP     LDA $C0,X
      $48         ;          PHA
      $B5 $A0     ;          LDA $A0,X
      $48         ;          PHA
      $B5 $80     ;          LDA $80,X
      $48         ;          PHA
      $B5 $A8     ;          LDA $A8,X
      $48         ;          PHA
      $CA         ;          DEX
      $10 $F1     ;          BPL LOOP
      $A5 $D3     ;          LDA $D3
      $48         ;          PHA
  ]"
```

```
DEFINE GetTemps=
  "[
    $68           ;       PLA
    $85 $D3       ;       STA $D3
    $A2 $00       ;       LDX #0
    $68           ; LOOP  PLA
    $95 $A8       ;       STA $A8,X
    $68           ;       PLA
    $95 $80       ;       STA $80,X
    $68           ;       PLA
    $95 $A0       ;       STA $A0,X
    $68           ;       PLA
    $95 $C0       ;       STA $C0,X
    $E8           ;       INX
    $E0 $08       ;       CPX #8
    $D0 $EF       ;       BNE LOOP
  ]"
```

Use these routines inside your interrupt routine as follows:

```
   ; Your interrupt routine.
   PROC InterruptRoutine()
   ; Local declarations, if any.
      BYTE a, b, c, etc.
      ; First line of code within
      ; procedure SaveTemps

      ; Your interrupt
      ; code goes here.

      GetTemps            ; Last line of code
                          ; within procedure.
   [$6C OldVBI]           ; A special way to
                          ; end for VBIs- see
                          ; below.
```

For example, the following program will set up the routine ChangeColor as a vertical blank interrupt routine (hit the <START> key to exit the program):

```
DEFINE SaveTemps=
  "[ $A2 $07 $B5 $C0 $48
     $B5 $A0 $48 $B5 $80
     $48 $B5 $A8 $48 $CA
     $10 $F1 $A5 $D3 $48 ]"

DEFINE GetTemps=
  "[ $68 $85 $D3 $A2 $00 $68
     $95 $A8 $68 $95 $80 $68
     $95 $A0 $68 $95 $C0 $E8
     $E0 $08 $D0 $EF ]"
```

```
CARD OldVBI    ; Will hold previous
               ; contents of vertical
               ; blank interrupt
               ; vector.

; This procedure will change the
; background color to random values.
; The main routine will set up this
; code to operate during the
; deferred vertical blank interrupt.
  PROC ChangeColor()
     BYTE hue, lum

     SaveTemps
     hue = Rand( 16 )
     lum = Rand( 16 )
     SetColor(2,hue,lum)
     GetTemps

  [ $6C OldVBI ]  ; Vertical blank
                  ; interrupts must end
                  ; like this ($6C is a
                  ; 6502 indirect jump
                  ; instruction).

  PROC Test()      ; Main routine
     BYTE critic=$42,     ; Critical I/O flag
          console=$D01F   ; Console key
                          ; hardware location
     CARD VBIvec=$224     ; Deferred vertical
                          ; blank interrupt vector

     ; You must install a VBI
     ; routine like this:
     critic = 1    OldVBI = VBIvec
     VBIvec = ChangeColor
     critic = 0

     ; ChangeColor is now running
     ; as the vertical blank interrupt
     ; routine-- since our mainline
     ; code has nothing more to do,
     ; we just go into a loop waiting
     ; for the START key to be
     ; pressed.#    WHILE console&1
     DO
     OD

     ; Now turn off the VBI routine.
     critic = 1
     VBIvec = OldVBI
     critic = 0
  RETURN
```

This method of saving and restoring ACTION zero page
variables may also be used to write BASIC machine language
subroutines in ACTION! Your main ACTION routine should then
have SaveTemps as the first executable line, and GetTemps
as the last executable line before the RETURN statement.

           PART 2 - BUGS IN THE ACTION! CARTRIDGES

The following is a list of all bugs we currently know exist
in the ACTION! cartridge. We list these bugs separately
from those in the runtime library and/or the Programmer's
Aid Disk (PAD) or Toolkit, which occur in following pages.
Each bug is described in detail and, when possible, bug
fixes are given.   Many of these bugs deal only with
specific versions of ACTION!. To find out which version of
ACTION! you own, type the following from the ACTION!
monitor:

     ?$BØØØ [RETURN]

Below is an actual copy of what printed following that
command for one of our cartridges.

     45Ø55,$BØØØ = Ø $Ø73Ø 48 184Ø
                   ^

To find out the version number, look at the character to
the right of the equals sign (here printed with a caret
under it). The "Ø" in this case implies that the cartridge
is version 3.Ø. If yours has a "6", you own version 3.6,
etc. As of the date of this bug sheet, the current
cartridge version is 3.6.

1. Offsets with TYPE Declaration

Using a TYPE declaration will generate a spurious error
whenever the code offset (contents of location $B5) is non-
zero.

Affects: All versions of the cartridge to date.
(Presumably only noticed if using runtime disk, though.)

Fix: Make all TYPE declarations before changing the code
offset.

Example:

   ; Beginning of program --
   ; First, declare TYPEs
   TYPE IOCB = [BYTE Id, Devnum,
                 Command, Status]
   ; Then, if desired,

```
    ; change offset
    SET $B5 = $1ØØØ
    ; example: offset=4Ø96
```

2. Offsets

Using a code offset greater than $7FFF (i.e., a negative
offset, if you consider it to be of type INT) causes the
compiler to generate improper code.

Affects: All versions, especially when used with the
runtime disk.

Fix: No direct fix, but you may use the relocator program
described later in this document (which is also usable with
assembly language).

3. ATARI DOS

Exiting to ATARI DOS from ACTION! can cause a system crash
if DUP.SYS is not present on the disk in drive 1.

Affects: All versions, but only when used with ATARI DOS.

Fix: Use DOS XL (or be careful when exiting to DOS).

4. Arrays and ELSEIF

We have just learned that there is a relatively obscure bug
in ACTION! related to the use of ELSEIF. In particular,
statements similar to the form

```
    ELSEIF a(i) = Ø THEN ...
```

where 'a' is an ARRAY and 'i' is a CARD OR INT, or
statements like

```
    ELSEIF p^ = Ø THEN
```

where 'p' is a Pointer, produce incorrect code.

Affects: All versions.

Fix: There is no direct fix at this time. The best way
around the problem seems to be to code something like this:

```
    t = a(i)    ; t is an INTEGER
    ...
    ELSEIF t=Ø THEN ...
```

This works properly.

5. Writing Object Files

If a monitor Write command fails because of a disk error
(e.g., disk full, 162, or device done, 144), the IOCB is
not properly closed. If the disk is changed before another
disk operation is performed, the new disk can have invalid
data written to it.

Affects: All versions.

Fix: If you get an error when writing an ACTION! object
file, type the following command to the monitor:

    X Close(1) [RETURN]

You can then erase the file which caused the error.

6. Hex Array Sizes

Hexadecimal values as array dimensions cause incorrect code
to be generated.

Affects: All versions.

Fix: Use decimal array dimensions.

7. TYPE Pointer Arguments

PROC or FUNC declarations with record pointer arguments
other than the first do not compile correctly. For example,
the following code generates an error 7 (invalid argument
list):

    TYPE REC=[...]
        ...
     PROC Test(BYTE x, REC POINTER p)

Affects: All versions.

Fix: Omit the comma in the argument list for the PROC/FUNC,
as in:

    PROC Test(BYTE x
              REC POINTER p)

As this is just a temporary fix, it may not work in future
versions, but the correct declaration (with the comma)
will.

8. Monitor Lockup

Typing the following command from the monitor will lock up
the system:

    R* <RETURN>

Affects: All versions.

Fix: Do not do it! If you do type that command, hit
<RESET>.

9. Paddle Function

The Paddle function does not work properly in all versions
of the ACTION! cartridge.

Affects: Versions 3.0 to 3.5.

Fix: Make the following declaration in your program:

    BYTE ARRAY Paddle(4) = 624

10. Sound on Channels 3 and 4

If you use a Sound() procedure call after having done any
disk I/O, sound channels 3 and 4 will remain silent. This
is because ATARI's OS does not reset some of the serial
control registers completely.

Affects: Versions 3.0 to 3.5.

Fix: Type in and use the following procedure. You should
call this before doing any Sound() calls and/or in place of
any SndRst() calls:

```
   ; Contributed by Michael Ross
   PROC SoundOff()
   BYTE AudCtl = $D208,
        SSKCtl = $232,
        SKCtl  = $D20F
        SSKCtl = 3
        SKCtl  = 3
        AudCtl = 0
        SndRst()
   RETURN
```

11. TYPE Fields as Parameters

Using fields of TYPEs as parameters to PROCs or FUNCs
generates incorrect code. For example,

```
   MoveBlock( rec.addr1,
              rec.addr2, length )
```

Affects: Versions 3.0 to 3.5.

Fix: Assign the TYPE field to a temporary variable and pass
that as a parameter:

```
     temp1 = rec.addr1
     temp2 = rec.addr2
     MoveBlock(temp1,temp2,length)
```

12. SAssign Problems

SAssign does not work properly when the source string has a
length of zero.

Affects: Versions 3.0 to 3.5.

Fix: No fix available at this time.

13. CARD FIELDS IN TYPES

Accessing CARD fields of TYPEs generates incorrect code.

Affects: Versions 3.0 to 3.2.

Fix: No fix available at this time.

14. MoveBlock Problems

MoveBlock does not move more than 256 bytes of data.

Affects: Versions 3.0 to 3.2.

Fix: No fix at this time. You could write an ACTION!
routine to do the equivalent.
15. Editor Command <CONTROL><SHIFT><RETURN>

Using <CONTROL><SHIFT><RETURN> to split a line into two
lines generates garbage in the second line.

Affects: Versions 3.0 and 3.1.

Fix: No fix available, but not a disastrous problem.

16. Division Errors

On old cartridges, neither the "/" operator nor the "MOD"
operator works properly under certain conditions.

Affects: Versions 3.Ø and 3.1.

Fix: Insert the following code into your program before any
of your own PROCedure or FUNCtion declarations (this can be
done easily using INCLUDE):

```
    ; Copyright (c) 1983 by
    ; Action Computer Services
    ;
    ; Permission is granted to
    ; duplicate and/or distribute
    ; the contents of this file
    ; to ACTION! users.  Copies of
    ; this file may not be sold or
    ; used for monetary gain.

    PROC DivI=*()
    [$2Ø $AØ6C $85 $86 $A2 $1Ø
     $26 $82 $26 $83 $26 $86 $26
     $87 $38 $A5 $86 $E5 $84 $A8
     $A5 $87 $E5 $85 $9Ø $Ø4 $85
     $87 $84 $86 $CA $DØ $E5 $A5
     $82 $2A $26 $83 $A6 $83
     $4C $AØ32]

    PROC RemI=*()
    [$2Ø DivI $86A5 $87A6 $6Ø]

    SET $4EA=DivI
    SET $4EC=RemI
```

17. Error Routine Not Initialized

The address of the Error PROCedure is not restored by
ACTION! if a user program has changed it.

Affects: Versions 3.Ø and 3.1.

Fix: Make sure to restore the original Error vector upon
exiting a program, if you changed it.

18. Complex Expressions in UNTIL

Complex relational expressions in an UNTIL statement
generate incorrect code. For example,

```
    DO
       ...
       UNTIL a>Ø AND b=3
    OD
```

Affects: Versions 3.Ø and 3.1.

Fix: Assign the expression to a temporary variable and test
that variable, instead:

```
    DO
       ...
       temp = a>Ø AND b=3
       UNTIL temp
    OD
```

19. Bank Switch Bug

When loading and running compiled ACTION! object files from
DOS, the system can crash when using older cartridges. This
is because the ACTION! library is not accessible.

Affects: Version 3.Ø only.

Fix: Put the following program lines at the VERY BEGINNING
of your main procedure (i.e., the last procedure in your
program):

```
    BYTE bank = $D5ØØ
    ; This declares the variable
    ; 'bank' to reside at $D5ØØ.

    bank = Ø ; This must be the
    ; first executable statement.
```

2Ø. '.COM' Programs

Running compiled ACTION! programs as '.COM' files under
OS/A+ causes those programs to execute twice.

Affects: All versions, but only when using a version of
OS/A+. DOS XL is not affected.

Fix: Insert the following as the first global variable you
declare:

```
    BYTE RTS=[$6Ø]
    ; This MUST be the first
    ; line in your program,
    ; aside from comments and
    ; SET commands.
```

21. PROC Addressing

Under certain conditions, specifying the address of a
procedure (e.g., to interface to a machine code routine)
causes ACTION! to generate incorrect code which could cause
your program to "hang".

Affects: Versions 3.1 and 3.4.

Fix: Insert an empty code block after the declaration of a
procedure whose address is specified. For example:

```
   PROC CIO = $E456() []
   ; An empty code block!
```

22. Error #3

If you get an ERROR 3 during a compile, the system hangs
when you return to the editor.

Affects: All versions.

Fix: Do not go to the editor until you type the following
line to the monitor. This command resets the ACTION! memory
pointer.

```
    SET $E=$491^
```

23. String Input

When using the string input library functions (InputS,
InputSD, and InputMD), there must be room in the string for
the termination EOL, even though the resulting string
length will not include it.

Affects: All versions.

Fix: Adjust your declaration appropriately.

PART 3 - BUGS IN THE ACTION! RUNTIME LIBRARY

We have found a few bugs in the original version(s) of the
Runtime Library Disk. Fortunately, they are all easy to
fix. The runtime library is independent of the cartridge,
so bugs affect all versions.

In the fixes given below, the portion to be changed (to
implement the fix) is underlined. The rest of the line
remains the same. To make the fixes, simply load the
library file containing the affected PROCedure, edit, and
save it back to disk.

1. Hex numbers are printed incorrectly by PrintH and the %H
parameter of PrintF.

Fix: Change second line of CCIO:

   PROC CCIO=*()
   [$A386$A0A$A0A$AA$A3A5$9D$342 ...
    ---   ---

2. PrintBDE can cause a spurious compile time error.#

Fix: Change first line of PrintBDE:

   PROC PrintBDE =*(BYTE d,n)[$A0$0]

3. A minor error exists in ChkErr.#

Fix: Change second line of ChkErr:

   PROC ChkErr=*(BYTE r,b,eC)
       [$1610$88C0$8F0
         $98$80C0$12F0 ...
                 ---

4. If your program redefines a library procedure (e.g., one
which declares its own version of PROC Graphics), it will
compile with no errors using the cartridge only (because
declared procedures take precedence over built-in ones).
However, since the RunTime library uses this same
precedence trick to include its own definitions of library
procedures, your program will generate Error 6 (doubly
defined name) if you do not delete the appropriate
PROCedure (or FUNCtion) from the RunTime library before
INCLUDEing it.

Fix: Make a custom version of the RunTime library on a COPY
(please, only on a copy) of your RunTime disk which does
not contain the routines you wish to replace.

5. On page 17 of the Reference Guide for the Runtime
Package (p. 2Ø8 here), the DEFINE for ROM will cause
incorrect code if you use local variables. (Not verified
yet).

Fix: Use the following form of definition, instead:

```
DEFINE ROM = "BYTE ZZQQJUNK
              -------------
                SET $68Ø = $E^
                SET $B5  = $58ØØ
                SET $E   = $682^"
```

        PART 4(a) - PROBLEMS WITH PROGRAMMER'S AID DISK

We will list the problems (and solutions) regarding the
Programmer's Aid Disk (PAD) here in reasonably compact
form.

1. IO.ACT - BGet/BPut Problems

The BGet and BPut routines in the IO.ACT file do not work
properly under certain conditions. To fix this bug, replace
the BGet and BPut routines with the following ACTION! code:

```
   ;*************************************
   ;Burst (Block) I/O routines to do quick
   ;disk I/O, utilizing a call to CIO
   ;*************************************

   PROC CIO=$E456( BYTE areg, xreg )
   ;*******************************


      CARD FUNC Burst( BYTE chan, mode,
                       CARD addr, buflen )

      TYPE IOCB=[BYTE id,num,cmd,stat
                 CARD badr,padr,blen
                 BYTE a1,a2,a3,a4,a5,a6]

      IOCB POINTER iptr

      chan ==& $Ø7
      iptr = $34Ø+(chan LSH 4)
      iptr.cmd  = mode
      iptr.blen = buflen
      iptr.badr = addr
      CIO( Ø, chan LSH 4 )
   RETURN( iptr.blen )
   ;*******************************
```

```
    CARD FUNC BGet( BYTE chan,
                    CARD addr, len )

    CARD temp

       temp = Burst(chan,7,addr,len)
    RETURN( temp )
    ;*******************************

    PROC BPut(BYTE chan,
              CARD addr,len)
       Burst( chan, 11, addr, len )
    RETURN
```

2. PRINTF.ACT - Bug

This routine has a bug which was reported and fixed in the
Spring, 1984 newsletter. In the file PRINTF.ACT, use the
ACTION! editor to find

```
     args ==+ s
```

and change it to

```
     args ==+ 2
```

3. PMG.ACT - Player/Missile Base Address

Because S: uses some memory just below the display list
(undocumented), our method of finding the base address for
Player/Missile Graphics needs a slight revision. Use the
ACTION! editor with the file PMG.ACT to find

```
   PM_BaseAdr=(HiMem-PM_MemSize(mode))&PM_AdrMask(mode)
```

and change it to

```
PM_BaseAdr=(HiMem-PM_MemSize(mode)-$8Ø)&PM_AdrMask(mode)
```

4. PMG.ACT — PMMove

If you use the PMMove procedure and specify a vertical
movement of zero, the horizontal movement does not take
place (it should). To fix this, change the lines in PMG.ACT
which read

```
   IF deltaY=Ø THEN
        RETURN ; do nothing
   FI
```

to the following:

```
    IF deltaY=Ø THEN
       ; do horizontal anyway
       PMHpos(n)=x
       RETURN
    FI
```

5. PMG.ACT - PMHpos

The documentation for PMG.ACT states that you may read the
contents of PMHpos to find the horizontal position of a
player or missile. This is simply not true. PMHpos is a set
of write-only hardware registers. (Note that in the ToolKit
we have added a shadow array and changed the name of the
hardware registers, so this works correctly. If you wish,
you could consider doing something similar on your PAD.)

6. REAL.ACT - Documentation

There are two discrepancies in PROCedure names in the
REAL.ACT library as compared to the REAL.DOC documentation,
as follow:

```
        Name in .DOC      Name in .ACT
        ------------      ------------
           StrR             RealToStr
           ValR             StrToReal
```

We suggest that you change the source code in REAL.ACT to
reflect the names given in the documentation (rather than
vice versa), since this makes the names appear compatible
with the library's other number-string conversion routines.

7. REAL.ACT - RealToStr

In that same area, the routine RealToStr (or should that be
StrR?) needs to change the line which reads

```
    ptr=LBuff
```

to the following:

```
    ptr=InBuff
```

8. ALLOC.ACT — Changes Necessary

The free list pointer may not be set up properly. Also,
when freeing a block, right adjacency is not handled
properly if left adjacency has already been found. Fix
these problems as follows:

In the PROCedure Free, after the line reading:

```
last.size==+nBytes
```

insert the line:

```
target=last
```

Also, in the same procedure, change the line reading:

```
IF target+nBytes=current THEN
```

to read:

```
IF target+target.size=current THEN
```

In the PROCedure AllocInit, replace the line reading:

```
FreeList.next=p
```

with the following lines:

```
FreeList=p
p==+4
FreeList.next=p
```

PART 4(b) - TOOLKIT TROUBLES

It is hard to believe that a product as new as the ACTION!
ToolKit can already have bug reports. Sigh. Anyway, there
are already three versions of the ToolKit. Version 1 has 31
free sectors (when you list its directory). Version 2 has
fewer free sectors and the second line of the file
MUSIC.DEM reads ";Version 2". On version 3, the file
ABS.ACT starts with the version number. This last
convention will be followed in future versions. The
comments here are organized by affected version(s).

VERSION 1 ONLY
----

1. I/O ROUTINES

The manual describes a routine called Format (in the IO.ACT
library), but no such procedure exists on the disk.
However, the routine is there--it is just called Init
instead. You should change your disk to match your manual.

2. MUSIC.DEM

The program called MUSIC.DEM will not work as is on older
400/800 machines. This is because it uses a call to
Graphics(15), which is only available on XL machines. You
may change the program to use Graphics(8) with no effect
except that the true colors of mode 15 become artifact
colors in mode 8 instead.

VERSIONS 1 AND 2
----

1. REAL ROUTINES

There are two discrepancies in PROCedure names in the
REAL.ACT library as compared to the REAL.DOC documentation,
as follow:

Name in .DOC	Name in .ACT
StrR	RealToStr
ValR	StrToReal

We suggest that you change the source code in REAL.ACT to
reflect the names given in the documentation (rather than
vice versa), since this makes the names appear compatible
with the library's other number-string conversion routines.

2. SORT ROUTINES

There are four discrepancies in PROCecure names in the
SORT.ACT library as compared to the SORT.ACT documentation,
as follows:

```
        Name in .DOC    Name in .ACT
        ------------    ------------
          SortB           BSort
          SortC           CSort
          SortI           ISort
          SortS           SSort
```

Please change your disk file to agree with your manual.

3. PRINTF

The PRINTF routine has a bug which was reported and fixed
in the Sprint, 1984 newsletter. In the file PRINTF.ACT, use
the ACTION! editor to find

```
        args ==+ s
```

    and change it to

```
        args ==+ 2
```


VERSIONS 1, 2, AND 3
----

1. ALLOC ROUTINES

The manual indicates that the procedure AllocInit requires
that you pass it the address of the first free byte of
memory (because Alloc "dispenses" memory from the first
free byte through the top of memory, as correctly described
in the manual).   However, since you MUST follow the
procedure described in the introduction to ALLOCATE.ACT
(that is, you must declare in your program a CARD called
EndProg and use the command

```
        SET EndProg=*
```

after compiling), the parameter to AllocInit is not really
needed and so has been eliminated. (AllocInit uses EndProg
just as Alloc does). If you pass a parameter to AllocInit,
it will be ignored.

2. WARP.DEM

No mention is made in the Toolkit manual that this file can
only be run when compiled from disk (unless you are using
DOS XL to gain extra memory). WARP.DEM is just too big for
ACTION! to hold both the source and object in memory at one
time.

3. ALLOCATE.ACT

The free list pointer may not be set up properly. Also,
when freeing a block, right adjacency is not handled
properly if left adjacency has already been found. Fix
these problems as follows:

In the PROCedure Free, after the line reading:

     last.size==+nBytes

   insert the line:

     target=last

Also, in the same procedure, change the line reading:

     IF target+nBytes=current THEN

   to read:

     IF target+target.size=current THEN
In the PROCedure AllocInit, replace the line reading:

     p=EndProg

   with the following lines:

     FreeList=EndProg
     p=EndProg+4

4. PRINTF

The PrintF routine on the Action! Toolkit works great
unless you try to print a CARD value greater than 32767, or
try to print the INT value -32768. The reason these
problems occur is that the PROC PF_NBase in the PRINTF.ACT
file uses the "/" and "MOD" operators, which call the
cartridge divide routine. The divide routine is a SIGNED
divide, so it does not work for large card values. The
solution is to insert an UNSIGED divide routine into the
PRINTF.ACT file and use it, instead. First, insert the
following code at the beginning of PRINTF.ACT:

```
   CARD Quotient, Remainder

   PROC UDiv(CARD a, divisor)
       DEFINE GETCARRY="~[$2E carry]"
       BYTE carry, i
       CARD temp
       Remainder = Ø
       FOR i = 1 TO 16
         DO
           Remainder ==LSH 1
           Quotient ==LSH 1
           IF (a&$8ØØØ)#Ø THEN
               Remainder ==% 1
           FI
           a ==LSH 1
           temp = Remainder - divisor
           GETCARRY
           IF (carry&1)#Ø THEN
               Remainder = temp
               Quotient ==+ 1
           FI
         OD
   RETURN
```

Some code in the PROCedure PF_NBase must also be changed.
Find the section of code that reads as follows:

```
    WHILE n>Ø
      DO
      d=n MOD base       <-
      IF d<1Ø THEN
        d==+'Ø
      ELSE
        d==+55
      FI
      s(ptr)=d
      ptr==-1
      length==+1
      n=n/base           <-
      OD
```

And change the two lines indicated so the code reads like
this:

```
    WHILE n>Ø
      DO
      UDiv( n, base )    <-
      d=Remainder        <-
      IF d<1Ø THEN
        d==+'Ø
      ELSE
        d==+55
      FI
      s(ptr)=d
```

```
    ptr==-1
    length==+1
    n=Quotient        <-
    OD
```

The resulting PrintF routine will work properly for all CARD and INTeger numbers.

```
                    Part 5 - ACTION MANUAL ERRATA
                    -----------------------------
```

First of all, you need to know which version of the manual
you have. If Part III is the Language, then you have the
first version of the manual. Otherwise, you have the second
(newest) version.   Unfortunately, both manuals contain
content as well as typographical errors. we will skip the
typos and concentrate on the content errors, since typos do
not impair your understanding of the language (although you
may wonder where we learned to spell).

VERSION 1 ERRATA

Page 2

In the last paragraph, it says that the library is on the
disk. This is not true. It is in your cartridge.

Page 23

Under the description of <BACK-S>, the comparison with the
ATARI screen editor is exactly reversed. If you are in
REPLACE mode, this key works as in the ATARI editor.

Page 26

Under <CTRL><SHIFT>T, it says you may not use lower-case
characters as tags. This is untrue.

Page 48

In the NOTE preceeding 4.3, you should add "The *, /, and
MOD operators result in an implied INT type. For this
reason, multiplication, division, and modulus of large CARD
numbers does not always work properly."

Page 48

Section 4.4 says that you may only have one special
operator in a complex relational expression.   This is
untrue. For example, the following is perfectly legal:

     (x=7 AND y#1Ø) OR z<1ØØ

Page 82

Section 6.2.3 implies that you may not use a function as a
procedure. This is not true. You may call a function as
though it were a procedure, but the value returned from the
function is ignored.

Page 97

Section 8.1.1 states that you may either initialize a
POINTER to an address or give it a value. Only the second
is possible, and you should use this form:

     BYTE POINTER x=<value>

Not this:

     BYTE POINTER x=[<value>]

Page 48

In example #1 there are two PrintF statements which have
"ptr" as one parameter. These should be "bptr", not "ptr".

Page 1Ø1

In the last example of ARRAY declaration (BYTE ARRAY
tests(5)...), the   dimension   is   overruled   by   the
initialization options, and so its dimension is only three.
To fill only the first 3 of 5 elements, do the following:

     BYTE ARRAY tests(5)=[4 7 18 Ø Ø]

Page 1Ø4

In example #3 you see the program line "PrintE(b)". This
should read "PrintE(barray)".

Page 1Ø8

Section 8.3.1.2 states that you can initialize the fields
of a record when you declare it. This is untrue; you may
only initialize its address.

Page 11Ø

The   program   line   "rec.level=InputB()"   should   read
"rec.level=GetD(7)".

Page 112

Same as previous error.

Page 115

Same as previous error.

Page 112

The   program   line   "continue=InputB()"   should   read
"continue=GetD(7)".

```
Page 120
--------
The   program   line   "mode=InputB()"   should   read
"mode=GetD(7)", and the program line "PrintE(name)" should
read "PrintE(nameptr)".

Page 115
--------
Same as previous error.

Page 122
--------
The    program    line    "incctr=chgclr"    should    read
"incclr=chgclr".

Page 142
--------
Section 5.3 states that you should not use channel 7.
ACTION! uses this channel to get characters from the
keyboard, and you may use it to do this also. However, do
not close this channel or alter its configuration in any
way.

Page 153
--------
The example of declaring an ACTION! procedure at an address
is wrong!  If you do this, the internal pointer to the
procedure will point to the specified address, but the code
generated by the procedure will not be there. Instead, it
will be in with your main code.  Use procedure and function
addressing ONLY to call machine language routines.

Page 161
--------
Where the table of contents lists the routines in section
2.3, it should read:

          PrintBD  NOT  PrintDB
          PrintCD  NOT  PrintDC
          PrintID  NOT  PrintDI

Page 162
--------
Where the table of contents lists the routines in sections
6.7 and 6.8, it should read:

          PeekC  NOT  CPeek
          PokeC  NOT  CPoke

Page 165
--------
Error in section 2.3.   See changes for pg. 161 and make
similar corrections.
```

Page 179

Section 6.4 states some information concerning the results
of misusing the SCopy routine, detailing that the
routine does string trucation, etc., to make the procedure
work.   This is not true.   You must make sure that the
strings are compatible in size.

Page 181

Section 6.8 states that the parameters to Poke and PokeC
consist only of an address. Instead, they consist of an
address and a value, as follows:

        Poke(<address>,<BYTE value>)
        PokeC(<address>,<CARD value>)

Page 182

Section 6.11. MoveBlock will move a maximum block of 256
bytes in versions 3.Ø to 3.4 of ACTION! Versions 3.5 and
up will move any number of bytes.

Page 191

Some error numbers are wrong. The corrections are:

        14 Out of Space
        15 Missing DO
        19 Missing OD
        24 Illegal FOR statement
        26 Nesting Too Deep
        27 Illegal TYPE reference
        28 Illegal RETURN
        128 BREAK key abort

Also, error 62 is error 61, and 54 & 56 do not exist.

Page 197

In the PrintF statement, %D should be changed to %U.

VERSION 2 ERRATA

Page 38

Section 2.7, paragraph 3. The last sentence states that
you can RUN compiled ACTION! programs from disk. This is
untrue. The RUN command will only compile and run ACTION!
source files. Use DOS to run compiled object files.

Page 39

The last RUN example (RUN PrintE()) will not work, since
RUN expects a file name. Use the "Xecute" command instead.

Page 63

In the TECHNICAL NOTE preceeding section 4.3, "*" should be
changed to "*, /, or MOD".

Page 126

The last assignment on the page makes newrecord point to
the current record in the array, not the end of the array.

Page 132

The program line "mode=InputB()" should be changed to
"mode=GetD(7)".

Page 138

The program line "IF sub(1)=str(ctr)" should read "IF
sub(1)=str(ctrl)".

Page 163

The PutDE procedure requires only a channel as a parameter,
and does not put out both a character and a <RETURN>.
Rather, it puts out a <RETURN> only.

Page 172

In graphics mode Ø and all text windows, color 1 is the
character luminance, color 2 is the background color, and
color 3 is unused.

Page 174

In section 5.6, references to the "lower right corner"
should instead be "lower left corner".

Page 18Ø

Section 6.1.2 states some information concerning the
results of misusing the SCopy routine, detailing that the
routine does string truncating, etc. This is not true.
You must make sure that the strings are compatible in size.

Page 182

Section 6.11. MoveBlock will move a maximum block of 256
bytes in versions 3.Ø to 3.4 of ACTION! Versions 3.5 and
up will move any number of bytes.

Part 6 - ACTION OBJECT CODE RELOCATION PROGRAM

The program SIMPLREL.ACT on this BBS may be used to cause
an ACTION! program to load and run at a different address
than that address at which it was compiled. The same
program will also work for assembly language object files,
providing you also follow the given instructions.

The program takes two object files as input and produces a
third file which will load and run at a desired address.
The relocating program prompts the user for the two input
files, which must have been compiled one page (256 bytes)
apart.   It then prompts for an output file name (the
relocated file), the page number of the starting address of
the first file, and the page number of the desired
destination address. Both page numbers must be decimal
values. For example, specifying 32 as the destination page
will cause the output file to load at address 32*256
($2ØØØ), not $32ØØ.

See part V, "The ACTION! Compiler", chapter 2, page 144,
for information on compiling programs to a specified
address (Used to compile the two object files one page
apart).

In order to use the relocating program, download
SIMPLEREL.ACT and read the instructions therein.

NOTE: I was not able to identify a source where from it
might be obtainable. Support welcome!

Alternatively, there are programs available written by John
DeMar which provide such functionality.

Instructions for:

  RELGEN.ACT:== Relocation Generator
& RELOC.ACT:== Run-time Relocator

These programs were intended to create a self-relocating
object file from either an ACTION! compiled program or an
Assembled program. The original object file must be a
single-stage boot with only one origin except for the
trailing run or init address. The following instructions
detail the steps to make the target object file. This file
may be appended to other binary load files and may have
other binary files appended to it. The program will load at
the next possible page boundary (increment of 256) after
MEMLO.

Because RELGEN compares two versions of you object file, you may want to init all variables to zero to keep the relocation table at a minimum. Stray data in the uninitialized variables may be interpreted as machine code that needs relocating.

1) Compile (or Assemble) your code at a convenient area but not conflicting with DOS. In ACTION!, use the following commands to force the program's origin to a specified value ($3000 for example):

        SET 14=$3000
        SET $491=$3000

2) Re-Compile your code at $100 higher than the first. For the above example, this would be at $3100.

3) From the ACTION! monitor, RUN the program RELGEN.ACT. It will prompt you for the filenames for the two object code files that you compiled above. Remember to give the Dn: prefix to the filenames. The program will compare the two object files and note their differences as offsets into the file. This information is saved in ACTION! form in a file with the original name and a ".GEN" extention. This will be used in the next step. Also, the program creates an object file image of the original but with an origin of zero. This is done to make the relocation process easier and this file, with a ".REL" extention will be used in step 5.

NOTE: RELGEN.ACT requires four open DOS files simultaneously. By default, DOS usually has buffers for only 3. You must use the command:

        SET $709=4

in the ACTION! monitor and type D for DOS. Rewrite DOS to the disk and reboot. Now, DOS will allow the four files to be opened.

4) Now, Read the program RELOC.ACT into the ACTION! Editor. This is a "generic" run-time relocator. The file generated with the RELGEN.ACT program (with the ".GEN" extention) must be merged into this program with the editor Read function. Position the cursor where instructed and read in the file. Compile this code but be sure that it is SET to compile above the expected end of YOUR program's target location. Save this object code to disk and go to DOS.

5) Using the DOS Copy command, append the ".REL" file generated in RELGEN.ACT, to the merged relocator file saved in step 4. For example:

```
       C
       Copy from,to:
       TEST.REL,AUTORUN.SYS/A
```

This assumes that you saved the file in Step 4 as AUTORUN.SYS.

6) Finally, the appended file can be loaded from DOS or named AUTORUN.SYS as above for permanent applications.

If you have question, send E-Mail to:
     John DeMar 71066,337 on Compuserve
     or leave a message on the ACE-BASE
     BBS at (315)451-7747. Good Luck!

Please see the latest known versions:

```
MODULE ;RELGEN2.ACT

;COPYRIGHT 1984, QMI, JS DeMar
;REV. 1.1, March 20, 1984

;OBJECT CODE RELOCATION GENERATOR for
;ACTION! compiled binary-load files.

;WARNING!!! This program requires
;four OPEN files simultaneously.
;Be sure that DOS is configured for
;this. With DOS 2.0, set $709 equal
;to at least 4, rewrite DOS and
;reboot.

;Requires the second file compiled
;at any even page increment higher
;than the first file, for example:
;$3000 and $3100.

;Generates a table of the locations
;that require relocating and saves
;it in a ".GEN" file in ACTION!.
;  The ".REL" file is the original
;object code with an origin of "0".
;The actual relocator is compiled
;from the generic relocator source
;called "RELOC.ACT" merged with the
;".GEN" file generated here. Append
;".REL" file to that code and it
;will load and relocate to MEMLO.

DEFINE in1="1",
       in2="2",
       out1="3",
       out2="4"
BYTE abrt
```

```
;-------------------------------

PROC MyError(BYTE a,x,y)

IF y=17Ø THEN
  PrintE("ERROR File not found!")
ELSE
  Print("ERROR! ")
  PrintBE(y)
FI
abrt=1
RETURN
;-------------------------------

PROC Ferror()

BYTE t,clock=$14

PrintE("ERROR in Output filespec!")
t=clock-$8Ø
DO
UNTIL t=clock
OD
RETURN
;-------------------------------

PROC EndIt()

Close(in1)
Close(in2)
Close(out1)
Close(out2)
RETURN
;-------------------------------

PROC Main()

CARD start1,start2,end1,end2
CARD offsets,offsete,i,count,hits
CARD test1,test2,old1,old2,old3,oldØ
BYTE x,z,j,wnum,d1,d2,
     sthigh

BYTE ARRAY fname1(18),fname2(18),
           fnameout1(18),fnameout2(18)

DO
PrintE("}    Relocation Code Generator II ")
PrintE("#          JS DeMar, 8/84 ")
PutE()
PrintE(" Requires two code files compiled")
PrintE("     with an offset of $Ø1ØØ.")
PutE()
```

```
Print("Filespec for code A >")
InputMD(device,fname1,18)
PutE()
Print("Filespec for code B >")
InputMD(device,fname2,18)
PutE()

Scopy(fnameout1,fname1)
SCopy(fnameout2,fnameout1)
j=1
IF fnameout1(1)#'D
      OR fnameout1(Ø)<4 THEN
  Ferror()
ELSEIF fnameout1(2)=': THEN
  z=Ø
ELSEIF fnameout1(3)=': THEN
  z=1
FI
DO
  x=fnameout1(j)
  j==+1
  IF x=$2Ø THEN
    EXIT
  ELSEIF x='. THEN
    EXIT
  ELSEIF j>fnameout1(Ø) THEN
    j==+1
    EXIT
  ELSEIF j>11+z THEN
    Ferror()
  FI
OD

fnameout1(j-1)='.
fnameout1(j)='G
fnameout1(j+1)='E
fnameout1(j+2)='N
fnameout1(Ø)=j+2

j=1
IF fnameout2(1)#'D
      OR fnameout2(Ø)<4 THEN
  Ferror()
ELSEIF fnameout2(2)=': THEN
  z=Ø
  EXIT
ELSEIF fnameout2(3)=': THEN
  z=1
  EXIT
FI
OD
DO
  x=fnameout2(j)
  j==+1
```

```
  IF x=$2Ø THEN
    EXIT
  ELSEIF x='. THEN
    EXIT
  ELSEIF j>fnameout2(Ø) THEN
    j==+1
    EXIT
  ELSEIF j>11+z THEN
    Ferror()
    EXIT
  FI
OD

fnameout2(j-1)='.
fnameout2(j)='R
fnameout2(j+1)='E
fnameout2(j+2)='L
fnameout2(Ø)=j+2

Print("Generation file = ")
PrintE(fnameout1)
Print("Relocation file = ")
PrintE(fnameout2)

Error=MyError
abrt=Ø
Close(in1)
Close(in2)
Close(out1)
Close(out2)
Open(in1,fname1,4)
Open(in2,fname2,4)
IF abrt=1 THEN
  Close(1)
  Close(2)
  RETURN
FI
Open(out1,fnameout1,8)
Open(out2,fnameout2,8)

x=GetD(in1) ;throw away two $FF's.
x=GetD(in1)
PutD(out2,$FF)
PutD(out2,$FF)
x=GetD(in1)
PutD(out2,x)
start1=x     ;start addr of file1.
x=GetD(in1)
PutD(out2,x)
start1==+(x*256)
x=GetD(in1)
PutD(out2,x)
end1=x
x=GetD(in1)
```

```
        PutD(out2,x)
        end1==+(x*256) ;end addr of file1.

        x=GetD(in2) ;throw away two $FF's.
        x=GetD(in2)
        x=GetD(in2)
        start2=x    ;start addr of file2.
        x=GetD(in2)
        start2==+(x*256)
        x=GetD(in2)
        end2=x
        x=GetD(in2)
        end2==+(x*256) ;end addr of file2.

        offsets=start2-start1
        sthigh=start1/256
        offsete=end2-end1


        PrintDE(out1,"MODULE")
        PrintD(out1,";For file ")
        PrintDE(out1,fnameout2)
        PrintDE(out1,"")
        Print("Code starts at ")
        PrintD(out1,"CARD start=[")
        PrintCE(start1)
        PrintCD(out1,start1)
        PrintDE(out1,"]")
        Print("   and ends at ")
        PrintD(out1,"CARD finish=[")
        PrintCE(end1)
        PrintCD(out1,end1)
        PrintDE(out1,"]")
        Print("Compile offset was ")
        PrintCE(offsets)

        IF offsete#offsets THEN
          PrintE("Diferrent size files!")
          PrintE("ABORTED!")
          EndIt()
          RETURN
        FI
        PrintDE(out1,"")
        PrintD(out1,"CARD ARRAY otable=[")
        wnum=Ø
        hits=Ø
        count=Ø
        FOR i=start1 TO end1
        DO
          d1=GetD(in1)
          d2=GetD(in2)
          IF d1#d2 THEN
            hits==+1
            IF wnum=Ø THEN
```

```
          PrintD(out1,"  ")
          Print(" ")
        ElSE
          PrintD(out1," ")
          Print(" ")
        FI
        PrintCD(out1,count)
        Print(" ")
        PrintC(count)
        wnum==+1
        IF wnum>4 THEN
          PrintDE(out1,"")
          PrintE("")
          wnum=0
        FI
        d1==-sthigh
      FI
      PutD(out2,d1)
      count==+1
  OD
  FOR i=0 TO 2
  DO
    d1=GetD(in1)
    d2=GetD(in1)
  OD
  test1=d1
  test1==+(d2*256)
  IF test1>=start1 AND test1<=end1 THEN
    PrintDE(out1,"]")
    PrintE("]")
    PrintD(out1,"CARD hits=[")
    PrintCD(out1,hits)
    PrintDE(out1,"]")
    PrintDE(out1,"")
    Print("CARD hits=[")
    PrintC(hits)
    PrintE("]")
    PrintE("")
    PrintD(out1,"CARD runaddr=[")
    Print("CARD runaddr=[")
    test1==-start1
    PrintCD(out1,test1)
    PrintC(test1)
    PrintDE(out1,"]")
    PrintE("]")
  ELSE
    PrintE("No Run Address! - ABORTED!")
  FI
  PrintE("")
  PrintDE(out1,"")
  EndIt()
  PrintE("Finished!}")
  RETURN
```

```
MODULE ;RELOCATE.ACT

;Run-time Relocator Code.
;For use with RELGEN.ACT
;COPYRIGHT 1984, JS DeMar
;Rev. 2.Ø, August 17,1984
;-------------------------------

SET 14=$6ØØØ
SET $Ø491=$6ØØØ
;-------------------------------
;The beginning of the relocator
;table and code should be higher
;than the end of the original
;compiled program. But, there must
;be enough space left for the table
;and the relocator code itself!
;-------------------------------




;-------------------------------
;Read the ".GEN" file above here.
;-------------------------------
;Compile this after reading in the
;".GEN" file above. Then append this
;code to the ".REL" file using the
;DOS C (COPY) command with /A after
;the filenames:
; PROGRAM.OBJ,PROGRAM.REL/A
;Then rename the ".REL" file to
;AUTORUN.SYS to run at boot-time.
;-------------------------------

PROC Relocate()

BYTE offset,memlohi=$Ø2E8,x,y
CARD memlo=$Ø2E7,i,j,top,entry
CARD POINTER p
BYTE ARRAY newplace

[$8E y $4E y $4E y $4E y $4E y]

newplace=memlo
newplace==&$FFØØ
offset=memlohi
i=memlo&$ØØFF
IF i#Ø THEN
  newplace==+$Ø1ØØ
  offset==+1
FI

j=Ø
FOR i=start TO finish
```

```
DO
  p=i
  x=p^
  newplace(j)=x
  j==+1
OD

FOR i=∅ TO hits-1
DO
  entry=otable(i)
  newplace(entry)==+offset
OD
runaddr==+newplace
[$6C runaddr]
```

ACTION!
The Best Complete
Software Development System

The Fastest, High Level Language Available for the ATARI®: A versatile, structured language that runs at almost assembly language speeds (100+ times faster than BASIC).

Best Structured Language: Incorporates features found in Pascal, C, ALGOL, and ADA, yet has many of the same commands familiar to ATARI BASIC programmers.

Has Everything You Need:

THE EDITOR: Many advanced features for easily creating and modifying source text...two separate program windows, each allowing up to 240 characters per line...fast horizontal and vertical scrolling...move and copy text...string find and replace...and much more!

THE MONITOR: Selects compilation options, saves compiled programs, examines variable values, and memory locations...and even traces the execution of your programs.

THE COMPILER: Super fast compilation into machine code, accepting source from the Editor or from tape or disk.

THE LIBRARY: A built in collection of useful subroutines for you to use in your programs including string manipulation...print procedures and formatting...I/O routines...and, graphics and game controller routines.

THE RUNTIME: Everything needed to run compiled programs without the cartridge ... and to write extrinsic DOS commands.

THE TOOLKIT: A collection of routines to extend programming capabilities ... and example programs to show how it works.

GBXL

2022