APRICOT
A PRogrammable Interactive Calculate O Tron

Gordon Henderson
projects@drogon.net

July 19, 2023

Preface

After looking at VTL-2 for the 6502 and deciding it would be somewhat problematic to port to my own Ruby 6502/65816 SBC systems I decided to try something different, so this project started as a fresh idea to create a minimal programming language for retro computers and ended up looking more like an interactive programmable calculator thanks to the *Harwell WITCH*[1] and fond memories of using my *Casio fx502p* programmable calculator at university.

APRICOT features both interactive and stored program modes. Programs are written using line numbers but these are only to aid the editor - there are no `GOTO` statements and while there is a subroutine mechanism it also doesn't use the line numbers.

You can use an external text editor if you like and the system will accept input from a file with or without line numbers.

The number precision and size is implementation dependant. The reference C, BCPL and 6502 versions all use 32-bit signed integers.

Gordon Henderson, Last updated: July 19, 2023

[1] https://en.wikipedia.org/wiki/Harwell_computer

Contents

Chapter 1

Introduction

1.1 This Document

I've written this document as both a tutorial and as an instruction manual in how to use **APRICOT** It would be best to work through it if you have a working implementation of the **APRICOT** system that you can type the examples directly into.

1.2 Conventions used here

Keeping things simple is the aim here, so there are only really 3 things to look out for[1].

1. Anything you may need to type into a computer running **APRICOT** , or anything it prints will be in this `TYPEWRITER` style font.
2. Anything of some importance, such as a name of an algorithm is emphasised *like this* or sometimes **in bold**.
3. The final thing to look out for is a warning, or just something that may be important to remember. It's represented by a warning exclamation point in a box to the side of the text. As demonstrated in this paragraph.

[1] Well, Possibly four as there may be the occasional footnote at the bottom of a page, so look out for them too.

1.3 Host operating system considerations

APRICOT does not provide any sort of line editing or command history, so hopefully the system you run **APRICOT** on will provide this for you. Either line editing facilities or a way for you to copy and paste commands into the system.

Old calculators didn't have any of that fancy stuff either.

1.4 Starting and startup options

Starting **APRICOT** may be different depending on the system you are using, but generally just typing `apricot` in a terminal window will start it.

There are a few options:

Option	Action
`-s xxx`	Specify the size of the stack used in sections, subroutine calls and pushed variables. The value is in arbitary units where pushing a variable or entering a section requires one unit and calling a subroutine requires 3 units.
`-m xxx`	Set the size of the free RAM available for use. The size is the amount of numbers that can be stored rather than bytes, words, etc.
`-e`	Start directly in editor mode.

Chapter 2

Interactive (or calculator) Mode

APRICOT normally starts in interactive mode. You should see a welcome message followed by the amount of free storage numbers available then the value of the calculators *display register*, then a colon and the cursor, waiting for you to enter something.

```
APRICOT: A PRogrammable Interactive Calculate O Tron
  v1.2. Copyright (c) 2023 by Gordon Henderson.
  Up to 6000 numbers of free store available.
READY
          0:
```

The *display register* (the number zero above), sometimes called an accumulator, is the only register the system has that might resemble a register in a conventional CPU system. This is always displayed in a fixed-width field of usually 11 digits. (10 digits plus an optional minus sign).

Simple calculations are entered like a calculator, so typing `1 + 2` then pressing the `ENTER` key should calculate the expression and print:

```
          0:1 + 2
          3:
```

and it will wait for the next calculation to be entered.

You do not need to (nor should you at this point) press the = key.

Note that there is no command precedence. This is an old fashioned calculator and commands are carried out strictly left to right. There are no brackets to force precedence either.

```
          3: 1 + 2 * 3
```

```
9:
```

will yield the answer of `9` and not `7` as you may expect on some systems with command precedence implemented.

If you want that sort of thing then you have to work the order out yourself. Just like you had to in old calculators or Forth.

2.1 Numbers

Numbers are normally entered as decimal integers optionally preceded with a minus sign for negative numbers. Base 8 (octal), 2 (binary) and 16 (Hexadecimal) are also supported but number printing is always in decimal.

- Octal number are entered with a leading zero. e.g. `0177` is decimal 127.
- Binary numbers are entered with a leading `0b`. e.g. `0b1010` is decimal 10.
- Hexadecimal numbers are entered with a leading `0x`. e.g. `0xA5` which is decimal 165.

The reference implementation of **APRICOT** uses signed 32-bit integers with a range of -2,147,483,648 through 2,147,483,647 or about $\pm$ 2 billion. Other implementations may have different ranges or use floating point.

2.2 Variables

There are 26 variables named `A` through `Z`. To store the value of the display register into a variable use the `=` command then the variable letter. You may enter variable names using lower case but note that variable names are capitalised automatically by the system when used in stored program mode.

```
9: 1 + 2 * 3 = A
9:
```

will calculate the result `9` and store it in variable `A`. The display register will contain the last value calculated; `9`.

Simply typing the variable with no preceding command will copy the variable into the display register.

```
 9: A + 5 = B
14:
```

Will transfer `A` (which may have value of 9 above) into the display register, add 5 into it then store it into `B`. The display register will contain the last value calculated; `14`.

2.3 Input and Output

Operator	Action
?	Read a number into the display register
. (Dot)	Print value of display register
@	Output the display register as a single ASCII character
#	Newline

To print the display register use the dot (`.`) command. To print text enclose it in double quotes (`"`). The hash[1] symbol (`#`) will print a new-line.

It may seem somewhat pointless right now as you can see the value in the display register, however printing will become more important in stored program mode.

Try:

```
0: 1 + 2 "The answer is " . #
```

which will give:

```
0: 1 + 2 "The answer is " . #
The Answer is           3
```

The `@` command may be used to output simgle characters. This can be used to implment terminal control codes or just make a beep with ASCII code 7.

```
0: 7 @
```

ASCII code 12 is often clear screen so

```
0: 12 @
```

[1] Pound, Octothorpe, Box, Hashtag symbol - call it what you like

May clear the screen. Or it may not and you will have to consult the manual for the terminal you are using.

For input use the `?` command. This will stop program execution and wait for you to type a number. The number will be read into the display register.

Try:[2]

```
          0: "Number? " ? "Double is " * 2 . #
Number? 42
Double is          84
         84:
```

2.4 Logical values: True/False

Logical Value	Value
TRUE	Non-Zero
FALSE	Zero

`FALSE` is represented by the value of zero and `TRUE` is anything non-zero. This will become easy to remember when we write programs with sections that loop.

2.5 Sections

Command	Action
[	Start Section
]	End Section
\	Else Section
{	Restart Section
}	Break out of Section

A section is a block of commands enclosed in `[` square-brackets `]`.

Sections are conditional and may repeat. They may also have an optional part executed if the entry condition is `FALSE`. Think *ELSE* if you like.

[2] We entered the number 42 here

```
0: 0 [ "Test" # 0]
```

Will print nothing, but:

```
0: 1 [ "Test" # 0]
```

Will print `"Test"`.

So what's going on here...

If, on entry to a section the display register has a `TRUE` value (ie. non-zero) then the section will be executed. If the display register is `FALSE` (ie. Zero) then the section will be skipped (but see later for the *ELSE* bit).

A section will be repeated if, at the end of the section the display register is `TRUE`.

So for the first section line above, we load `0` into the display register which is `FALSE` and so the section is ignored. The second section loads `1` which is `TRUE` and it's executed.

At the end of each section we load `0` into the display register which is `FALSE` so the section is not repeated.

Here is an example which counts to 10:

```
0: 1 = A [ A . # + 1 = A - 11 ]
1
2
3
4
5
6
7
8
9
10
0:
```

Explanation:

- We load `1` into the display register and store it in `A`.
- The section starts with a `TRUE` value in the display register, so the section is executed.
- `A` is loaded into the display register and printed with a newline.

- `1` is added to the display register and its stored back into `A`.
- We then subtract 11 from the display register and the section ends.
- Any number in the display register other than zero will cause the section to repeat, so it's going to repeat until `A` has the value of `11` at which point the section will exit because `11 - 11` is Zero which is `FALSE`.

Another example: Calculate a Factorial:

```
0: "Number? " ? =N=M 1=F [ F * N =F N -1 =N ] M . "! is " F . #
```

This uses variable `N` to hold the starting number and count, `M` is a copy of the starting number so we can print it out when done. The section loops multiplying the count (`N`) and last result (`F`) into `F` and taking one from `N` until `N` is zero.

! Note: There is no check for number overflow - the largest factorial you can generate with signed 32-bit integers is 12! or 479001600.

2.6 ELSE?

As well as looping, sections act like an *IF* statement in other programming languages and in addition to the *IF* part, there is also an *ELSE* statement you can use. It's simply the backslash character (\).

```
0: "Value? " ? . [ " is TRUE" \ " is FALSE" 0] #
```

The `?` command will stop and wait for a number to be typed then enter this number into the display register. The left or right side of the section will then be executed accordingly.

2.7 Break and Restart

As with other languages that implement loops, there is also the means to break out of a section before reaching the end or jump back to the start. The commands to do this are; { for restart (or continue) and } for break.

These commands are also conditional, so you need to make sure the value of the display register is set accordingly. `1}` will force a break out of the section and `1{` will force a restart.

2.8 Nesting Sections

Sections may be nested inside each other. The limit to the number of sections you can next inside each other is implementation dependant. The storage used for nesting (sometimes called a stack) is shared with the means to store the return point for sub-routines as well as storing the value of variables you push down to temporarily save them.

2.9 Arithmetic, Logic, and Test operators

Operator	Action
$+$	Add
$-$	Subtract
$*$	Multiply
$/$	Divide
$\%$	Modulo (Remainder after division)
$	MulDiv
&	Logical AND
$\vert$	Logical OR
^	Logical Exclusive OR
$<$	Test for less-than
$>$	Test for greater-than

The $ or MulDiv operator performs an operation on three numbers; x, y, and z in the form:

$$\frac{x.y}{z}$$

Where the intermediate product $x.y$ is evaluated with 64-bit precision (in a 32-bit system) then divided by z. This can be used to aid scaled fixed point arithmetic for exmaple. To use, the z value is in the display register, then the $ operator then 2 named variables.

e.g. to multiply 80,000 by 40,000 then divide by 2 - if you were to do the multiplication as normal then it would overflow - ie.

```
0: 80000 * 40000 / 2
```

```
-547483648:
```

But using the $ operator then:

```
          0: 80000 = A
      80000: 40000 = B
      40000: 2 $ AB
 1600000000:
```

The 2 test operators; $<$ and $>$ test the display register against the next variable or number and set the display register to either TRUE or FALSE accordingly.

Note that there is no equality test (or not equal). To test for equality you need to perform a subtraction operation on the 2 values.

Test the less-than, greater-than and equality constructs:

```
0: "Enter A: " ? = A "Enter B: " ? = B
0: A < B [ "A is less than B" 0]
0: A > B [ "A is greater than B" 0]
0: A - B [ \ "A = B" 0]
```

As in the equality test above, use the ELSE (\) command to test for equality in the subtraction example above and to generate less than or equal ($\leq$) or greater than or equal ($\geq$) tests:

```
0: A < B [ \ "A is >= B" 0]
```

2.10 Indirect Variables

In addition to the 26 variables A through Z **APRICOT** gives access to unused or free memory in the computer via an indirect mechanism. This is sometimes called vectors or arrays in other programming languages.

Free memory starts at location zero and goes up to some pre-determined number. This is announced when **APRICOT** starts.

The indirection command is the exclamation mark (!)[3].

You use it by storing the address of the memory location you wish to store data into or read from into an ordinary variable then...

[3]It has many names, some call it an “eek” mark or symbol, some may say ’“bang’, or even “shreik”, or possibly even “pling”. Call it what you like. I like to say “Fetch at M” for !M and “Store at M” for = !M.

To store use the command form:

```
0: 12 = M 42 = !M
```

This stores the value 42 into the memory location that variable `M` points to - in this case location `12`.

and to read into the display register:

```
0: !M
```

Essentially a variable prefixed with the `!` symbol means to use the memory location that variable points to rather than the value of the variable itself.

! Free memory is not initialised and may contain random data.

Example: Fill the first 500 locations (locations 0 through 500 inclusive, so 501 locations in total) with an ascending sequence of 0 through 500:

```
0: 0 = M 1[ M = !M + 1 = M < 501 ]
```

And to sum each of those 501 numbers:

```
    0: 0 = M = S 1[ !M + S = S M + 1 = M < 501 ] S .#
125250
```

Testing with the formula:

$$Sn = n(\tfrac{A1+An}{2})$$

or:

$$Sn = 501(\tfrac{0+500}{2})$$

which looks like this in **APRICOT** :

```
    0: 0 + 500 / 2 * 501
125250
```

2.11 Subroutines

Subroutines aren't that useful in interactive mode, but you can call a stored subroutine from immediate mode if needed. e.g. you could load in a library of common subroutines and use them from interactive mode.

Subroutines have a single letter name A through Z. The name has nothing to do with the variables of the same letter.

Operator	Action
;x	Call Subroutine x
(x	Define Subroutine ”x”
)	Return from subroutine

Other than using variables and the display register there is no way to pass parameters into subroutines. Normally you'd return a value (if any) in the display register.

Subroutines can call other subroutines and themselves (recursion). The limit to the number of calls is implementation dependant.

2.12 Temporary storage - Push and Pull

Operator	Action
, (comma)	Push the display register into the storage stack
’ (apostrophe)	Pull the display register from the storage stack

You can “push” and ‘pull” the display register into and from a temporary storage stack with these commands.

You can use this to implement local variables in subroutines if you think they might be used elsewhere.

So, if your subroutine wants to use variables A and B then

```
  A , B ,
.... some code
  ’=B ’=A )
```

in the subroutine will work and preserve A and B at the start and restore them at the end.

! There is no automatic restore. It's up to you do do the restore and do it in the right order which is the reverse of the order you pushed them in.

! If you should not push a variable outside a section then pull it inside that section. This is because sections and the push/pull operations use the same stack.

You can push variables then return from a subroutine without pulling them. In which case the pushed values will be lost.

2.13 Comments

The back-tick (`) can be used to start a comment. The comment ends at the end of the input line.

Chapter 3

Commands in Interactive Mode

When in interactive mode **APRICOT** has some additional commands. These commands are all prefixed with the colon (`:`) character which must be at the start of the line.

Command	Action
?	Print a list of commands.
Q	Quit or return to the underlying operating system, if possible.
E	Enter Program Editor mode.
P	Print (List) the current program
L	Load a program from the filing system.
R	Run a loaded program.
C	Clear the calculator - Set the display register and all `A` to `Z` variables to zero with the exception of variable `R` which is initialised to a random value.
V	Display all `A` to `Z` variables.

Example:

```
 -547483648: :c
+++ Clear
          0: :V
Display Register: 0
| A:          11 | B:           0 | C:           0 | D:           0 |
| E:           0 | F:           0 | G:           0 | H:           0 |
```

```
| I:          0 | J:          0 | K:          0 | L:          0 |
| M:          0 | N:          0 | O:          0 | P:          0 |
| Q:          0 | R: 1804289383 | S:          0 | T:          0 |
| U:          0 | V:          0 | W:          0 | X:          0 |
| Y:          0 | Z:          0 |
         0:
```

Chapter 4

Stored Program Mode

APRICOT lets you enter and store program lines from the keyboard in a manner similar to that of *BASIC* with line numbers. Lines are inserted in ascending order and lines may be deleted by entering a blank line after the line number.

There are also additional commands you can use to help manipulate the stored program and allow for transfers to and from the systems filing system.

Note that in stored program mode, **APRICOT** does not show the value of the display register. You need to use the printing commands to show results.

4.1 Changing Mode

From *Interactive Mode*, use the `:E` key combination to enter *Editing Mode*. To get back to *Interactive Mode*, just type the command `I`.

In editing mode, the prompt will change to an > symbol.

There are a series of commands (described later) you can use, but the primary function is to enter and run programs.

Chapter 5

Commands in Editor (Stored Program) Mode

When in editor mode **APRICOT** has some additional commands. The commands are all a single letter and some are similar to those in interactive mode:

Command	Action
?	Print a list of commands.
I	Back to Interactive/Calculator mode.
P	Print (List) the current program.
R	Run the current program.
N	ReNumber the current program.
E	Erase the current program.
V	Display all `A` to `Z` variables.
E	Erase the current program.
L	Load a new program
S	Save the current program

A few notes:

- The renumber command renumbers the program starting at 100 and incrementing by 10.
- The save command doesn't save line numbers and the file may be edited outside **APRICOT** by any text editor you may usually use.
- Loaded programs should not have line numbers - they will be numbered by 10,

starting at line number 100.

5.1 Entering lines of code

When in the editor (stored program) mode, any line that starts with a number is considered a line of code and will be stored - providing there is at least one space between the line number and the remainder of the line.

If you enter a line with the same number as an existing line, then the existing line will be overwritten with the new line.

If you enter a line with just a number, then that line will be deleted from the stored program.

5.2 Examples

Print “Hello, World!” 10 times: (Note “p” and “r” commands here):

```
> p
 100 10       ‘ times to run loop
 110 [         ‘ Start section (DR is not zero)
 120    "Hello, world!" #
 130    - 1
 140 ] _

> r
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
+++ Program END
>
```

To convert it to a subroutine that takes a parameter as the number of times to print:

```
100 (H
140 ] )
```

Note that we change line 140 to replace the end program symbol (underscore) with end subroutine.

To call this, we could return to interactive mode and use the semi-colon command, or add a couple more lines like:

```
10 "How many times? " ?
20 ;H _
```

and run with the “r” command.

Study the example programs supplied with the distribution for more programming examples. Look for filenames that end with the `.cot` suffix.