VTL/2: A Very Tiny Language

By Gary J. Shannon

Gary Shannon
may be reached at:
18411 Vincennes
Northridge, CA 91325

If you assume that in order to run significant programs in a higher level language you need 8-12 k of memory and a BASIC interpreter, you will be interested to learn of Gary Shannon's Very Tiny Language. The interpreter for Gary's language resides in 768 Bytes of ROM and the language is very terse, making it possible for a significant program to fit in a 1k or smaller RAM. The terse notation also pays off in terms of fast execution, and in spite of its size, VTL has some very powerful features.

Of course, by opting for a very compact language, Gary has paid a price–don't expect your VTL programs to be very readable and don't look for lots of diagnostic messages from the interpreter. You too will pay a price if you wish to buy a VTL chip set for the Altair 680–it sells for $114.00 at your local MITS dealer. Just before going to press, we learned that 8800 chip sets are also available at $114.00 for 270X's or $149.00 for 2702's.

If you wish more information, you can get a copy of the 27 page VTL/2 language manual by sending $2.00 to Dick Heiser, The Computer Store, 820 Broadway, Santa Monica, California 90401. Be sure to specify 8800 or 680.

Gary began building home computers out of relays and tubes while in high school. He has been a professional systems programmer for 13 years. At the time this article was written, he was an employee of the Computer Store, and therefore had a commercial interest in VTL.

So you've built your Altair 680, now what? Let's face it, there's not much you can do with only 1K of memory. That's not enough room for BASIC, or even for an assembler. But wait a minute. I know one user who is running a small billing system on his 1K 680. Another, a college professor, is doing complex matrix manipulations on archeological data. These sound like serious applications. Surely no one is taking the 680 seriously!

Well, if that's what you think, then think again. All over the country hundreds of 680 owners are taking their tiny 1K systems very seriously. Their secret is VTL/2, a Very Tiny Language for the Altair 680. This language gives the user the capabilities of BASIC, yet requires only 768 bytes of read-only memory.

Having a language in PROM is the ultimate in convenience. Each time the system is turned on, the interpreter is right there, ready to use, with no time wasted in loading. By way of contrast, 8K of hex format paper tape requires over 20 minutes to load from a slow cassette.

Not only does the interpreter itself use up very little space, but the VTL programs themselves are very compact. You might call it a "smallified BASIC". To begin with, in place of the key-words used in BASIC, VTL uses single character abbreviations for such functions as INPUT, PRINT, GOTO, GOSUB, RETURN, IF, RND, FRE, USR, and for array or string array references.

Variables may be represented by any single alphabetic or special character. Most of these are available for the user to define as he wishes. Some of them, however, have very special meanings. These are called "system variables".

The system variable *pound sign* (#), for example, will always contain the line number of the program line currently being executed. If nothing is done to this

variable, it will advance to the next line number in the program after each line is executed. If, however, the statement changes the value of "#", the next line executed will be the one whose number was placed into "#". In other words, "#=300" means "GOTO 300". The only exception to this rule is that if the result placed in "#" is zero, that value will be ignored, and the next line in the program will be executed. This fact allows us to "calculate" an IF statement in VTL/2. Consider this example:

```
10 X=1              (sets X to 1)
20 #=(x=25)*50      (if X=25
                     goto 50)
30 X=X+1            (add 1 to X)
40 #=20             (goto 20)
50...               (continue)
```

Notice that "X=25" is a *logical expression* which has the value *one* if it is *true* (i.e. if X is equal 25) and *zero* if it is *false* (i.e. if X is not equal 25). When this "logical" value is multiplied by 50, the result must be either zero or 50. If it is 50 the statement causes a "GOTO 50" to be executed. If it is zero, a "GOTO 0", which is a dummy (NOP) statement falls through to the next line in the program.

Every time the value of "#" is changed to some non-zero value, the original value plus one is saved in another system variable, *exclamation point* (!). The variable "#" can be used for both "GOTO" and "GOSUB", since the statement "#=!" means "go back to where you came from plus one line". This is the VTL/2 "RETURN" statement.

The system variable *question mark* (?) represents the user's terminal. It can be either an input, or an output depending on which side of the equal sign it falls. The statement "?=A" means "PRINT A", and the statement "X=?" means "INPUT X". Since "?" is a variable, it may appear anywhere in a statement, so that "R=(?+?+?)/3" will call for three numbers to be input, and will put their average in the variable "R". In response to a request for input, the user may type a number, a variable name (whose value will be the value used), or an entire VTL/2 expression!

For games and simulations, the system variable *apostrophe* (') represents a different random integer between 0 and 65535 every time it is called. if your game (or simulation) program requires a number in some other range, the statement "R=('/(Y-X+1)*0)+%+X" produces a random integer between X and Y. (Notice that the system variable *percent sign* (%) always contains the remainder after the most recent integer divide.)

In addition to decimal input and output, the system variable *dollar sign* ($) is used to input and output ASCII string data. This is accomplished by allowing any variable (or array position) to contain either a numeric value, or a single ASCII character. This dual purpose even allows you to perform computations on character data as if they were numeric. Add one to the character "A" and the result is the character "B". As an example:

```
10 A=65
20 $=A
30 A=A+1
40 #=A<91*20
50 ?=""
```

The above program will print out a continuous string of letters, each of which is one greater than the one before it. Since 65 is the decimal value for the letter "A", and 90 is the letter "Z", the actual output would be "ABCDEFGHIJKLMNOPQRSTUVWXYZ". Statement 50 then prints a carriage return.

Any memory remaining after the end of the program may be used as array storage. The array does not need a name, since there is only one, but it can be divided up as required, and appropriate subscripts calculated. A subscript expression is identical to any other VTL expres-

sion except that it begins with a *colon* (:) and ends with a right parenthesis. This subscript expression may appear anywhere that you would otherwise use a variable name. For example:

```
10 :I)=0      (zero first array loc.)
20 I=1        (set subscript to 1)
30 :I+1)=:I)+1  (put next higher
                 number in next
                 array position)
40 I=I+1      (bump subscript)
50 #=I<101*30   (loop back
                 till I=101)
```

Since subscripts refer to two-byte words in memory, it is possible for large valued subscripts to "wrap around" memory and clobber the VTL/2 source program itself. On the other hand, this also means that clever programs could modify themselves. (Very carefully, of course!)

There are no error messages in VTL/2. If an expression is wrong, the results of executing the instruction will be unpredictable. In other words, VTL expects you to know what you're doing, and will do its best to execute any statement you give it. This gives you wide latitude for trying various programming "tricks", but also leaves you, the user, with the responsibility of verifying program accuracy.

In addition to the features discussed here, VTL/2 has provisions for user defined machine language subroutines, printing string literals, *control-C* (cancel), and *control-A* (suspend). Pointers available to the user as system variables make it possible to compute memory addresses, and "PEEK", or "POKE" to those locations. Similarly, memory sizes and free space may be easily computed from system variables.

The structure of VTL/2 by no means limits you to 1K in your system. The interpreter will handle any size memory up to 63K. (1K must be reserved for PROM.)

How about speed? VTL/2 benchmark programs have run 20-30% faster than 8800 disk extended BASIC. Keep in mind that this speed increase is in spite of the fact that the 8800 system clock runs twice as fast as the 680 clock! The speed of 8800 VTL/2 (yes, it is available for the 8800) is even greater.

By now, most Altair dealers should have the VTL/2 PROMs in stock, so isn't it about time you began to take your 680 seriously?