Forth Bridge - from BASIC to Forth#

adapted Version for VolksForth

original version from the Jupiter ACE Resource Site

Table of Contents

Introduction#

Beginners All-purpose Symbolic Instruction Code - to give BASIC its full title - is, as the name declares, a language for the computer novice. If you have learnt BASIC, then you have served your programming apprenticeship. You should now want to move on to a language which will allow your programming talents freer and fuller development. You are probably already aware that BASIC has distinct limitations, and that those limitations become both more apparent and more inhibiting the more progress you make. To construct a large program in BASIC will inevitably involve you in a bewildering and ungainly maze of GOTOs and GOSUBs; the program will run rather slowly and it will take up too much memory space.

No one who knows will regard BASIC as a particularly good programming language. It caught on early in the development of the Personal Computer and has held its place more through habit than because of any inherent qualities as a computer language.

But even if you feel that, knowing BASIC, you wish to go beyond it there are likely to be questions that will concern you. You may feel that you would rather stick with and build on the language you already know. You may be concerned that to convert to a better language means to convect to a harder language. You will certainly want to know, if you are to learn another language, which will be the best.

To take these points in order, there is little future if you wish to expand your programming, in sticking with BASIC. Sooner or later BASIC will hinder your progress, then it will stop it dead.

The second point is that the quality of a computer language is not a function of its difficulty. Clarity and simplicity are cardinal virtues in good programming, and those will flow more naturally from a language which is itself clear and simple.

The question of which language to choose for Personal Computer applications may seem to be a more complex one. But in fact it is not. There is one language which stands out as being a model of simplicity, clarity and the means through which to acquire the technique of elegant programming. It is also very fast, memory efficient, and emminently suitable for microcomputers.

That language is FORTH.

FORTH offers many advantages. It is no harder than BASIC to learn, but it imposes none of the contrictions which BASIC does. Its greatest virtue for the computer owner who wishes to aquire enhanced programming skills is that it enables him or her to begin from readily learnable words and then advance step by step to programs of greater complexity building word on word. There is no limit to how far FORTH can take you but it will let you proceed securely at your own speed.

Learn FORTH, and you will find that your programming will take off. It provides a fascinating, addictive and infinitely extensible means to learn about real computing.

What follows will aid you in making the step up from BASIC to FORTH. A small step in terms of effort; an enormous step for your programming future. Go from BASIC to FORTH and you won't go back again.

BASIC TO FORTH BRIDGE#

If you are a person who is well-versed in BASIC and you have just come across the language FORTH, you may well want to know if you can translate your favourite programs into FORTH. The short answer is 'yes', and this tutorial will help you to do it. Although it has been writted with volksForth in mind, any remarks about FORTH should apply to other machines as well. Throughout this text, `BASIC` and `FORTH` words are highlighted. FORTH, like BASIC, has a vocabulary of words which you can either type in as commands or group together to make a program. FORTH has a word `WORDS` which prints the dictionary (that is all the words in the vocabulary) on the screen. In BASIC, you form a program by taking a collection of words and putting a line number in front. In FORTH, however, you define new words which then become part of the dictionary, just like the words already there in the volksForth. Like any of the original words, you can execute a new word by typing it in at the keyboard and you can use it in the definitions of other new words. You can think of FORTH words as being like subroutines and your final program as being a list of GOSUB statements- but they don't slow down your program as subroutines do. It is easy to debug programs written in this way because you can test each word separately to check that it does exactly what you want. The most common way of making a new word is the colon definition. Here is a very simple word which prints a message on the screen.

```
: MESSAGE ." This is volksForth " ;
```

The colon at the start says the next word is the name of a new word to be compiled into the dictionary and that everything following, as far as the semicolon, is the instructions to be executed when you use the new word. Here the name of the new word is MESSAGE and the instructions is ." (pronounced 'dot quote') which says 'print the following characters up to " on the screen'. When you type in MESSAGE it will print "This is volksForth". It will also print 'OK' afterwards to show that it has executed your commands without any problems. `WORDS` will now show that MESSAGE has been added to the dictionary. It also shows :, ." and ; which are all FORTH words in the volksForth Language. You can write the same thing in BASIC, like this

```
10 PRINT "This isn't volksForth"
```

and you can execute it by saying RUN, or RUN 10, or GOTO 10. If you want to use it more than once, you can add `20 RETURN` and say `GOSUB 10` to execute it. Some versions of BASIC allow you to put more than one statement in a line, in which case you would probably write `10 PRINT "This isn't volksForth" : RETURN` One of the advantages of the word MESSAGE over the set of line 10 is that MESSAGE is an English word and can convey an idea of what the word actually does - it prints a message. `GOSUB 10` doesn't mean anything in English, so you either have to remember what it does or list it. Putting in a comment, e.g. `5 REM message` helps you recognise what it does from the listing, but not when you are referring to it by a line number. You can put comments in FORTH words, too, by putting the text inside parentheses - `(` is a FORTH word that says 'ignore what follows till you come to)'.

With the volksForth decompiler, you can list any word you have defined, just as you can list any part of a BASIC program. `SEE MESSAGE` will write

```
SEE MESSAGE
  : MESSAGE
```

```
(." THIS IS VOLKSFORTH" EXIT ; -2 allot  OK
```

on the screen. (if you get an error message "SEE MESSAGE WHAT?", you need to load the SEE command with `INCLUDE" D:SEE.F"`)

The next important feature of FORTH is the way it handles numbers. When you type in a number, it is put on the stack. The stack is like a pile of cards with the numbers written on them. You can put more cards on top of the pile or take them away, and the last ones put on are normally the first ones to go. Almost all computer languages use a stack, but it is often hidden from the programmer. (In fact, FORTH has two stacks, called the data stack and the return stack. The data stack is the one with your numbers on it; the return stack is used by the machine, although there are a few words which allow you to take advantage of what volksForth does with the return stack.)

In FORTH, most words communicate via the (data) stack. They take their operands off the stack and leave their results on it. Having a stack for numbers makes it very convenient for the arithmetic words to use reverse polish or postfix notation (used on most Hewlett Packard calculators). This means that the operators go after the operands.

BASIC uses infix notation. Instead of writing the operators in between the operands, 2+3 in FORTH you would say `2 3 +`. 2 and 3 are both numbers and so are put on the stack and + is a FORTH word which takes two numbers off the stack and puts back their sum, in this case 5. This is just like a recipe where you list the ingredients and then put the instructions. It may seem strange to expect you to think about arithmetic in a different way - after all why can't the computer do it for you? But it does have its benefits. For instance, you don't need any parantheses in your calculations because there is never any ambiguity about which operation to do first. The operator just takes the operands it needs off the stack and puts the results back afterwards.

So `(2+4)*3` becomes `2 4 + *` in postfix notation. There is a very common way of showing what words do to the stack. You list the operands that the word requires on the stack starting with the one lowest down and ending with the top. So with +, this is

```
   (n,m - n+m)
operands  result
```

A word can have any number of operands and leave any number of results. This makes it very easy to define your own functions. You don't have to declare a list of variables, you just write the definition bearing in mind that the numbers you want will be on the stack. e.g. to define a function which squares a number, in BASIC you would say `DEF FN s(x)=x*x : REM x squared` in FORTH, this becomes `: SQUARE ( n - n squared) DUP * ;`

DUP (n - n,n) makes a copy of the top number on the stack and puts that on the stack as well. FORTH has other stack-manipulation words for getting the numbers into the order you want. They are

- ?DUP duplicates the top of the stack if it is non-zero.
- DROP drops the top number from the stack.
- OVER makes a copy of the second number down.
- PICK makes a copy of a given number down.
- ROLL moves a given number down to the top.
- ROT moves the third number down to the top.
- SWAP swaps the top two numbers.
- * (n - n,n*m) takes the top two numbers of the stack and puts back their product.

Conventionally, FORTH works only with integers, usually two bytes long. There is also same double-length arithmetic which uses numbers four bytes long. With many programs, it is very easy to scale

all the numbers up to integers for the calculations and then scale them back again. For instance, if you were dealing with money you would work in pence and convert back to pounds afterwards.

Basic	volksForth	Description	
ABS	ABS	returns absolute value	
AND	AND	This is a bitwise Boolean AND (BASIC varies) so, e.g. 42 23 AND leaves 2 on the stack.	
ASC	ASCII	get ASCII Code of Character	
POS	ATXY	set Cursos Position	
CHR$	EMIT	Prints out the character whose ASCII value is on the stack. e.g. `PRINT CHR$(32)` or `32 EMIT` prints a space	
DATA		see Arrays below	
Procedure	:	Functions are defined as words (see the introduction)	
DIM		See section on arrays below	
FOR n= x TO y NEXT	y+1 x DO ..... LOOP		
FOR n = x TO y STEP z NEXT	y+1 x DO ..... z +LOOP		

Notice how in FORTH, the limit of the loop is one more than in BASIC. This means that if you want to execute the loop n times starting at x then the limit is x+n, but in BASIC it is x+n-1. e.g.

Basic:

```
10 REM character set
20 FOR n = 0 TO 255
30 PRINT CHR$ n;
40 NEXT M
```

Forth:

```
: CHARS
   256 0 DO
     I EMIT
   LOOP ;
```

The word I copies the current value of the loop counter to the stack, then EMIT prints the character with that ASCII value.

Basic	volksForth	Description
GOSUB		Subroutines are replaced by words in Forth. To call one, you type in its name
GOTO		FORTH doesn't have explicit GOTO statements but several constructions have them implicitly. The most important is IF ... ELSE ... THEN - see IF. The following also contain GOTO's BEGIN ... n UNTIL which repeats until n is non-zero. BEGIN...n WHILE ... REPEAT which repeats while n is non-zero. DO ... LOOP and DO ... +LOOP - see FOR
IF ... THEN	IF ... ELSE ... THEN	

IF takes a number (condition) off the stack and if it is non-zero (true) it executes the part between IF and ELSE and then jumps to THEN, otherwise if the condition is zero (false) it jumps to ELSE and executes the ELSE ... THEN part. You can omit ELSE when there is nothing to be done if the condition is false. Notice how THEN doesn't come in the same place as in BASIC. The way to think of it in FORTH is if the condition is true, do something THEN get on with the rest of the program. e.g.

Basic:

```
10 REM balance
20 PRINT ABS (bal);
30 IF bal >=0 THEN GOTO 100
35 REM balance negative
40 PRINT "debit"
50 GOTO 120
100 REM balance positive
110 PRINT "credit"
120 RETURN
```

Forth:

```
: BALANCE ( balance - )
   DUP ABS .
     0 IF
        ( if balance negative)
        ." debit"
      ELSE
        ( if balance positive)
        ." credit"
      THEN ;
```

Basic	volksForth	Description	
INKEY$	KEY? / KEY	KEY? Reads the keyboard and puts 0 on the stack if no key (or more than one key) was pressed. KEY read the ASCII value of the key pressed.	
INPUT		FORTH does not have one word which translates INPUT. Instead there are several words which cover all the different uses of INPUT. QUERY clears the input buffer then waits for you to type in things. WORD takes text out of the input buffer up as far as an ASCII delimitor. NUMBER takes a number out of the input buffer.	
LET		FORTH has variables just as BASIC does but you have to declare them before using them (like DIM and arrays). The word which does this is.... VARIABLE which puts the variable name in the dictionary along with space for a 2-byte number. e.g. VARIABLE SCORE Sets up a variable called 'SCORE'. You update its value with the word ! (pronounced 'store'). e.g. `100 SCORE !` makes 100 the value in STORE. You can put the current value of the variable on the stack using the word @ (pronounced 'Fetch') e.g. `SCORE @` puts 100 on the stack. See also the section on arrays for use of the FORTH word CREATE.	
LIST	SEE <word>	Lists a word you have defined in terms of its component words.	
USR	CALL		

		executes the machine code starting at the address on the stack, so $E477 CALL is what the CPU does when it is switched on. (on an Atari 8bit)	
NEXT		See FOR FORTH LOOP +LOOP	
NOT	0=	This takes the top number off the stack and leaves 1 if it was zero and 0 otherwise. Some versions of FORTH also have a word NOT which is identical to 0=.	
OR	OR	Bitwise Boolean OR	
PEEK	C@	Fetches the byte stored in the address on the stack. C@ stands for 'character fetch' as it was designed for reading 1-byte ASCII codes instead of 2-bytes variables @ (see LET) fetches the value stored in the 2 bytes starting at the address on the stack.	
POKE	C!	Stores the second number on the stack in the byte at the address at the top of the stack. ! Stores the second number on the stack in the pair of bytes starting at the address at the top of the stack. (See LET too.)	
PRINT	.	Prints the number at the top of the stack on the screen. `."` Prints the subsequent characters on the screen. See introduction. e.g. `PRINT "Hello"` is in Forth `." Hello"` ; TYPE Prints out a given number of bytes starting at a given address as ASCII characters.	
REM	(	Treats text up to) as a comment and ignores it.	
RESTORE		See below	

RETURN		Not needed in FORTH	
RUN		There is no direct equivalent in FORTH - you just name the word you want to run.	
SAVE	SAVESYSTEM	SAVESYSTEM Saves the current dictionary as a dictionary file	

Arrays#

FORTH doesn't have any array handling words of its own but it does allow you to set aide space in the dictionary for your own data. There is a word CREATE which puts a word name in the dictionary but nothing else. So CREATE ROW makes a word called ROW and when you type in 'ROW' it puts on the stack the address of the first byte in the dictionary after the definition of ROW. This may seem pointless, but there is another FORTH word, ALLOT which tags a specified number of bytes onto the end of the dictionary. This gives you a data field for your empty name, so if you now say 6 ALLOT you have made a 1-d array called ROW which is 6 bytes long. `3 5 ROW + 1- C!` stores 3 in the 5th element of ROW and `5 ROW + 1- C@` reads it back again. (You need the 1- because the 1st element is at ROW+0.)

A two dimensional array is just a row of elements but the elements themselves are rows so you have to allot the number of rows times the length of each row (the number of columns). This is exactly the way your BASIC computer will do it, but it doesn't let on.

Forth does not have READ, DATA and RESTORE (although you could mimic them if you wanted) but it does have something very much simpler which does just as well, namely the stack. Instead of a DATA statement, you have a word which puts all the data on the stack. e.g. `: DATA 5 0 7 2 1 4 ;`

In BASIC this would be

```
110 DATA 5,0,7,2,1,4
```

One of the main uses for READ, DATA & RESTORE is for initialising arrays, so you can now define a word which will set up ROW (from the section on arrays)

```
: INIT
  DATA -1 5 DO
       ROW I + C!
    -1 +LOOP ;
```

or in BASIC

```
120 FOR I = 1 to 6
130 READ R(I)
140 NEXT I
```

This starts at the end of the row and moves backwards along it in the data. (It goes backwards because the last piece of data written is the first one read.) If you want to reinitialise the row you don't need RESTORE, you just type in INIT.

Strings#

FORTH has a word TYPE which types out a number of characters from a given address. So an easy way of referring to a string of characters is the address of the start of the string and the length. The volksForth manual contains words for setting up strings in this manner and also for string and comparisons. You can have string arrays by making an array containing ASCII codes. If you want a word to print a string you can use ." (see introduction and PRINT).

Penny Vickers 1983 OCRed and scanned with Omnipage Pro 15 September 2005 Jupiter Preservation Project The Jupiter Ace Archive Project

(volksForth adaption 2006 by Carsten Strotmann)