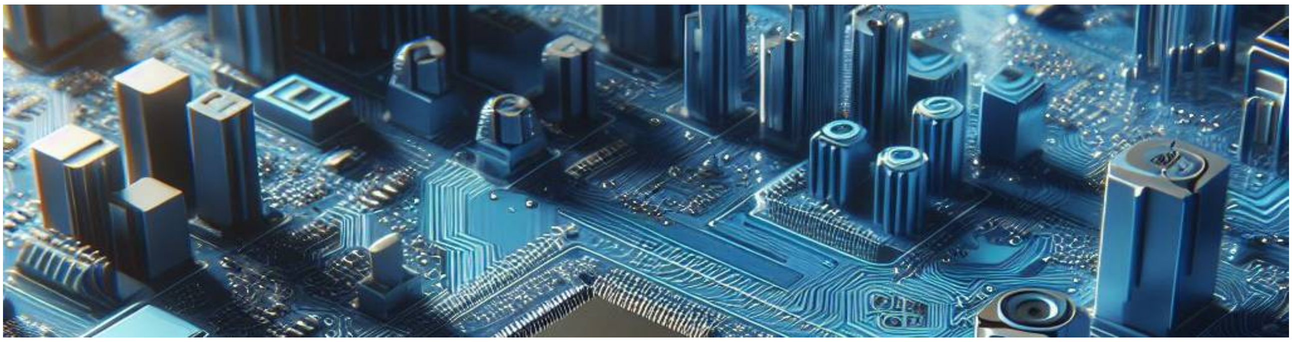
The great book for eFORTH Windows

version 1.0 - 8 novembre 2024



Author

- Marc PETREMANN

Contents

Dictionary / Stack / Variables / Constants

Expand the dictionary

Forth belongs to the class of woven interpreter languages. This means that it can interpret commands typed on the console, as well as compile new subroutines and programs.

The Forth compiler is part of the language and special words are used to create new dictionary entries (i.e. words). The most important ones are `:` (start a new definition) and `;` (completes the definition). Let's try this by typing:

```
: *+ * + ;
```

What happened? The action of : is to create a new dictionary entry named `*+` and switch from interpreter mode to compile mode. In compile mode, the interpreter looks up words and, rather than executing them, installs pointers to their code. If the text is a number, instead of pushing it onto the stack, eForth Windows builds the number in the dictionary in the space allocated for the new word, following special code that pushes the stored number onto the stack each time the word is executed. The action of executing `*+` is therefore to sequentially execute the previously defined words `*` and `+` .

The word `;` is special. It is an immediate word and is always executed, even if the system is in compile mode. What `;` does is twofold. First, it installs code that returns control to the next external level of the interpreter, and second, it returns from compile mode to interpret mode.
Now try your new word:

```
decimal 5 6 7 *+ . \ displays 47
```

This example illustrates two main working activities in Forth: adding a new word to the dictionary, and trying it out once it has been defined.

Stacks and Reverse Polish Notation

Forth has an explicitly visible stack that is used to pass numbers between words (commands). Using Forth effectively requires you to think in terms of a stack. This can be difficult at first, but as with anything, it gets much easier with practice.

In FORTH, the stack is analogous to a stack of cards with numbers written on them. Numbers are always added to the top of the stack and removed from the top of the stack. eForth Windows integrates two stacks: the parameter stack and the return stack, each consisting of a number of cells that can hold 16-bit numbers.

The FORTH input line:

```
decimal 2 5 73 -16
```

leaves the parameter stack as is

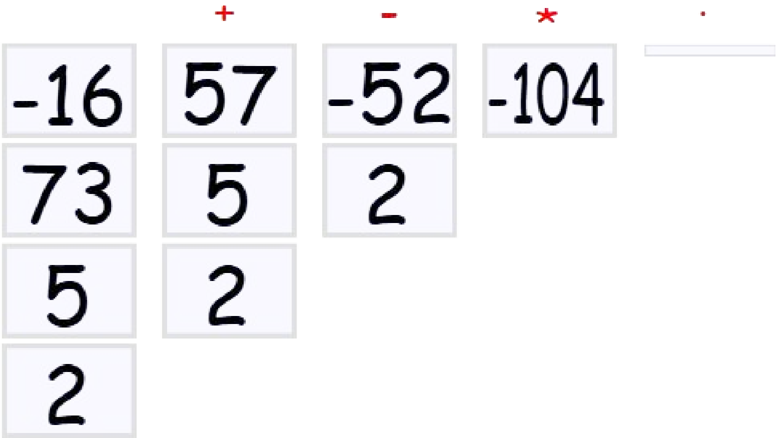
Cell	content	comment
0	-16	(TOS) Top stack
1	73	(NOS) Next in the stack
2	5	
3	2	

We will typically use zero-based relative numbering in Forth data structures such as stacks, arrays, and tables. Note that when a sequence of numbers is entered like this, the rightmost number becomes *TOS* and the leftmost number is at the bottom of the stack.

Suppose we follow the original input line with the line

```
+ - * .
```

The operations would produce the successive stack operations:



After the two lines, the console displays:

```
decimal 2 5 73 -16      \ displays: 2 5 73 -16 ok
+ - * .                 \ displays: -104 ok
```

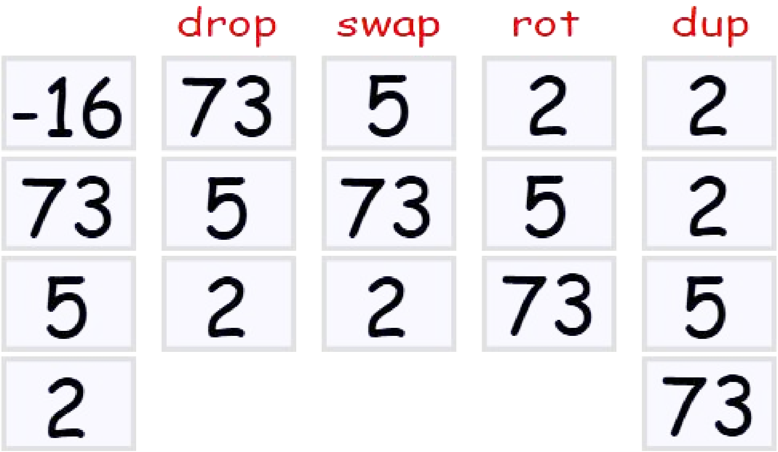
Note that eForth Windows conveniently displays the stack elements as it interprets each line, and the value of -16 is displayed as a 32-bit unsigned integer. Also, the word . consumes the data value -104, leaving the stack empty. If we execute . on the now empty stack, the external interpreter aborts with a stack pointer error STACK UNDERFLOW ERROR.

The programming notation where the operands appear first, followed by the operator(s) is called Reverse Polish Notation (RPN).

Handling the parameter stack

Being a stack-based system, eForth Windows must provide ways to put numbers on the stack, to remove them, and to rearrange their order. We have already seen that we can put numbers on the stack simply by typing them. We can also integrate the numbers into the definition of a FORTH word.

The word `drop` removes a number from the top of the stack, putting the next number on top. The word `swap` swaps the first 2 numbers. `dup` copies the number on top, pushing all the other numbers down. `rot` rotates the first 3 numbers. These actions are shown below.



The Return Stack and Its Uses

When compiling a new word, eForth Windows establishes links between the calling word and previously defined words that are to be invoked by the execution of the new word. This linking mechanism, at runtime, uses the return stack (rstack). The address of the next word to be invoked is placed on the return stack so that when the current word is completed during execution, the system knows where to jump to the next word. Since words can be nested, there must be a stack of these return addresses.

In addition to serving as a reservoir of return addresses, the user can also store and retrieve from the return stack, but this must be done carefully because the return stack is essential to program execution. If you use the return stack for temporary storage, you must return it to its original state, otherwise you will likely crash the eForth Windows system. Despite the danger, there are times when using the return stack as temporary storage can make your code less complex.

To store on the stack, use `>r` to move the top of the parameter stack to the top of the return stack. To retrieve a value, `r>` moves the top value of the return stack to the top of the parameter stack. To simply remove a value from the top of the stack, there is the word `rdrop` . The word `r@` copies the top of the return stack to the parameter stack.

Memory Usage

In eForth Windows, 32-bit numbers are fetched from memory to the stack by the word `@` (fetch) and stored from the top to memory by the word `!` (store). `@` expects an address on the stack and replaces the address with its contents. `!` expects a number and an address to store it. It places the number in the memory location referenced by the address, consuming both parameters in the process.

Unsigned numbers that represent 8-bit (byte) values can be placed in character-sized memory cells using `c@` and `c!` .

```
create testVar
cell allot
$f7 testVar c!
testVar c@ .          \ displays 247
```

Variables

A variable is a named location in memory that can store a number, such as the intermediate result of a calculation, off the stack. For example:

```
variable x
```

creates a storage location named, `x` , which executes by leaving the address of its storage location on top of the stack:

```
x .            \ displays the address
```

We can then collect or store at this address:

```
variable x
3 x !
x @ .          \ displays: 3
```

Constants

A constant is a number that you would not want to change while a program is running. The result of executing the word associated with a constant is the value of the data remaining on the stack.

```
\ defines extrem values for alpha channel
255 constant SDL_ALPHA_OPAQUE
0   constant SDL_ALPHA_TRANSPARENT
```

Pseudo-constant values

A value defined with value is a hybrid type of variable and constant. We define and initialize a value and it is invoked as we would a constant. We can also change a value as we can change a variable.

```
decimal
```

```
13 value thirteen
thirteen .           \ display: 13
47 to thirteen
thirteen .           \ display: 47
```

The word `to` also works in word definitions, replacing the value following it with whatever is currently on top of the stack. You have to be careful that `to` is followed by a value defined by `value` and not something else.

Basic tools for memory allocation

The words `create` and `allot` are the basic tools for reserving memory space and attaching a label to it. For example, the following transcription shows a new graphic-array dictionary entry :

```
create graphic-array ( --- addr )
%00000000 c,
%00000010 c,
%00000100 c,
%00001000 c,
%00010000 c,
%00100000 c,
%01000000 c,
%10000000 c,
```

When executed, the `graphic-array` word will push the address of the first entry.

We can now access the memory allocated to `graphic-array` using the fetch and store words explained earlier. To calculate the address of the third byte allocated to `graphic-array` we can write `graphic-array 2 +` , remembering that indices start at 0.

```
30 graphic-array 2 + c!
graphic-array 2 + c@ .          \ displays 30
```

Ressources

English

- **ESP32forth** page maintained by Brad NELSON, the creator of ESP32forth. You will find all versions there (ESP32, Windows, Web, Linux...)
  https://esp32forth.appspot.com/ESP32forth.html

French

- **eForth** two-language site (French, English) with lots of examples
  https://eforthwin.arduino-forth.com/

GitHub

- **Ueforth** resources maintained by Brad NELSON. Contains all Forth and C source files for ESP32forth and ueForth Windows, Linux and web.
  https://github.com/flagxor/ueforth
- **eForth Windows** source codes and documentation for eForth Windows. Resources maintained by Marc PETREMANN
  https://github.com/MPETREMANN11/eForth-Windows
- **eForth SDL2 project** for eForth Windows
  https://github.com/MPETREMANN11/SDL2-eForth-windows

Facebook

- **Eforth** group for eForth Windows
  https://www.facebook.com/groups/785868495783000

Index lexical