Elements of FlashForth.

Mechanical Engineering Report 2013/08
P. A. Jacobs
School of Mechanical and Mining Engineering
The University of Queensland.
June 8, 2013

Abstract

This report is a remix of material from a number Forth tutorials and references, adapted to the FlashForth environment. It provides some examples and explanation of using FlashForth on a PIC18 microcontroller while concentrating on the features of the language rather than the details of the microcontroller hardware. Following an introduction to the FlashForth interpreter, we look at adding our own word definitions to the dictionary and then explore the manipulation of data values on the stack. Flow of program control and more advanced defining words are also explored. These defining words are convenient for making arrays. Finally, strings and formatted numeric output are discussed.

Contents

1 Introduction 3
2 Getting started 3
3 The interpreter 4
4 Extending the dictionary 5
   4.1 Dictionary management 6
5 Stacks and reverse Polish notation 6
   5.1 Manipulating the parameter stack 7
   5.2 The return stack and its uses 8
6 Using memory 9
   6.1 Variables 10
   6.2 Constants 10
   6.3 Values 11
   6.4 Basic tools for allocating memory 11
CONTENTS

7 Comparing and branching 12
8 Comments in Forth code 13
9 Integer arithmetic operations 13
10 A little more on compiling 14
11 Looping and structured programming 14
12 More on defining words 15
   12.1 create ... does> ... 16
   12.2 Creating arrays 17
   12.3 Jump tables 17
13 Strings 18
   13.1 Pictured numeric output 19
14 Forth programming style 19
1 Introduction

Forth is an interesting mix of low-level access tools and language building tools. It is effectively a small toolkit with which you construct a specialized dictionary of words that work together to form your application code. This tutorial will explore and explain the workings of the FlashForth toolkit running on a Microchip PIC18 microcontroller and complements the more hardware-oriented tutorial [1], the FlashForth quick reference [2] and the FlashForth website [3]. Our interest is in using Forth on the microcontroller in an embedded system, such as a special-purpose signal timing device, rather than as part of a general-purpose calculation on a personal computer.

There are quite a number of good introductory tutorials [4, 5], course notes [6], and references [7] for programming in forth on a desktop or laptop computer, however, FlashForth running on a PIC18 microcontroller is a different environment. In the following sections, we will follow closely J. V. Noble’s tutorial [5], reusing as many of his examples and explanations verbatim, while adapting the overall tutorial to FlashForth.

2 Getting started

Although we will be using FlashForth on a PIC18 microcontroller, we communicate with it using a serial terminal program running on a personal computer. On Linux, the gtkterm terminal program is convenient and can be started with the command

```bash
$ sudo gtkterm --port=/dev/ttyUSB0 --speed=38400 --delay=10 --flow=Xon
```

Pressing the `ENTER` key a couple of times should get the display as shown in Figure 1. The `ok<$,ram>` prompt indicates that the current base for representing numbers in hexadecimal format and that the current context for making variables is static RAM, rather than the Flash memory and EEPROM that is also available in the microcontroller.

![Opening screen using gtkterm.](image)

In contrast to Forth on a PC, FlashForth is case sensitive, with most predefined words being spelled with lower case. Also, being intended for use in an embedded system, there
is no command to exit the system. FlashForth only stops when the power is removed or a reset occurs.

3 The interpreter

FlashForth is an interactive programming language consisting of words. Forth words are the equivalent of subroutines or functions in other languages and are executed by naming them. Although FlashForth is interactive at its core, the user doesn’t need to interact with an embedded application if its top-level word is set to automatically execute at power-up.

Here is an example of executing a FlashForth word:

```
decimal ← ok<#,ram>
```

This executes the word that sets the base for representing numbers to 10, a format that you are likely to be more familiar with unless you are a student of mechatronics or computing.

Now, let’s try something a bit more interesting by entering:

```
2 17 + . ← 19 <#,ram>
```

This time FlashForth more clearly shows its interpretive nature. A small program called the outer interpreter continually loops, waiting for input from the serial port. The input is a sequence of text strings (words or numbers) separated from each other by the standard Forth delimiter, one or more ASCII blank characters.

The text strings are interpreted in only three ways: words (subroutine or function names), numbers, or not defined. The outer interpreter tries first to look for the incoming word in the dictionary that contains the already defined words. If it finds the word, it executes the corresponding code.

If no dictionary entry exists, the interpreter tries to read the input as a number. If the string satisfies the rules for defining a number, it is converted to a number in the microcontroller’s internal representation, and stored in a special memory location, called the top of stack (TOS).

In the example above, FlashForth interpreted 2 and 17 as numbers, and pushed them onto the stack. “+” is a predefined word, as is “.”, so they are looked up and executed. The “+” (plus) word removed 2 and 17 from the stack, added them together, and left the result 19 on the stack. The word “.” (dot) removed 19 from the stack and sent it on the standard output device, the serial port for FlashForth. Here is a picture of the stack through the process.

```
word executed      | 2 17 + .
stack result       | TOS    NOS
                  | 2 17   2 19
```

We might also work in hexadecimal, which was the default at power on:

```
hex 0a 14 * . ← c8 <$,ram>
```
This default is probably convenient for most embedded systems work, where setting and monitoring bit patterns forms a large part of the code. If you want to explicitly indicate the base of a number, you can prepend a sigil to the digits of the number. For example, $10, #16 and %10000 all represent the decimal value sixteen.

If the incoming text cannot be located in the dictionary nor interpreted as a number, FlashForth issues an error message.

```
OA ← OA ?
thing ← thing ?
```

Note that the apparent hexadecimal number OA was not interpreted as such because of the case sensitivity of FlashForth. Other error messages that you might see include SP ?, for a stack pointer error, and CO ?, for a context error. If the word * was to be executed without there being at least two numbers sitting on the stack, the interpreter would abort, issuing the SP error message, and then wait for new input.

Finally, here is the classic Hello World! program.

```
: hi ." Hello, World!" ; ← ok<$,ram>
```

Forth lets you output text using the word ." while the words : and ; begin and end the definition of your own word hi. Note that blank characters are used to delimit each of these words. Now, type in hi and see what happens.

```
hi ← Hello, World! ok<$,ram>
```

## 4 Extending the dictionary

Forth belongs to the class of Threaded Interpretive Languages. This means that it can interpret commands typed at the console, as well as compile new subroutines and programs. The Forth compiler is part of the language and special words are used to make new dictionary entries (i.e. words). The most important are : (start a new definition) and ; (terminate the definition). Let’s try this out by typing:

```
: ** * + ; ← ok<$,ram>
```

What happened? The action of “:” is to create a new dictionary entry named ** and switch from interpret 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, FlashForth builds the number into the dictionary space allotted for the new word, following special code that puts the stored number onto the stack whenever the word is executed. The run-time action of ** is thus to execute sequentially 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 the code that returns control to the next outer level of the interpreter and, second, it switched back from compile mode to interpret mode.

Now, try out your new word:
5 Stacks and reverse Polish notation

The stack is the Forth analog of a pile of cards with numbers written on them. The numbers are always added to the top of the pile, and removed from the top of the pile. FlashForth incorporates 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 ← ok<#,ram>
```

leaves the parameter stack in the state

<table>
<thead>
<tr>
<th>cell</th>
<th>contents</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-16</td>
<td>TOS (Top Of Stack)</td>
</tr>
<tr>
<td>1</td>
<td>73</td>
<td>NOS (Next On Stack)</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

We will usually employ 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 right-most number becomes TOS and the left-most number sits at the bottom of the stack.
Suppose that we followed the original input line with the line

```
+ - * . ← xxx ok<#,ram>
```

What would the `xxx` be? The operations would produce the successive stacks:

<table>
<thead>
<tr>
<th>word executed</th>
<th>+ - * .</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack result</td>
<td>TOS</td>
</tr>
<tr>
<td></td>
<td>-16 57 -52 -104</td>
</tr>
</tbody>
</table>

After both lines, the gtkterm console shows

```
decimal 2 5 73 -16 ok<#,ram>2 5 73 65520
+ - * . -104 ok<#,ram>
```

Note that FlashForth conveniently displays the stack elements on interpreting each line and that the value of -16 is displayed as the 16-bit unsigned integer 65520. Also, the word “.” consumes the -104 data value, leaving the stack empty. If we execute “.” on the now-empty stack, the outer interpreter aborts with a stack pointer error (SP ?).

The programming notation where the operands appear first, followed by the operator(s) is called reverse Polish notation (RPN). It will be familiar to students who own RPN calculators made by Hewlett-Packard.

### 5.1 Manipulating the parameter stack

Being a stack-based system, FlashForth must provide ways to put numbers onto the stack, to remove them and to rearrange their order. We’ve already seen that we can put numbers onto the stack by simply typing the number. We can also incorporate the number into the definition of a Forth word.

The word `drop` removes a number from the TOS thus making NOS the new TOS. The word `swap` exchanges the top 2 numbers. `dup` copies the TOS into NOS, pushing all of the other numbers down. `rot` rotates the top 3 numbers. These actions are shown below.

<table>
<thead>
<tr>
<th>word executed</th>
<th>drop swap rot dup</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack result</td>
<td>TOS</td>
</tr>
<tr>
<td></td>
<td>-16 73 5 2 2</td>
</tr>
<tr>
<td></td>
<td>5 2 2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

FlashForth also includes the words `over`, `tuck` and `pick` that act as shown below. Note that `pick` must be preceded by an integer that (gets put on the stack briefly and) says where on the stack an element gets picked.
From these actions, we can see that 0 \text{pick} is the same as \text{dup}, 1 \text{pick} is a synonym for \text{over}. The word \text{pick} is mainly useful for dealing with deep stacks, however, you should avoid making the stack deeper than 3 or 4 elements. If you are finding that you often have to reason about deeper stacks, consider how you might refactor your program.

Double length (32-bit) numbers can also be handled in FlashForth. A double number will sit on the stack as a pair of 16-bit cells, with the cell containing the least-significant 16-bits sitting below the cell containing the most-significant 16-bits. The words for manipulating pairs of cells on the parameter stack are \text{2dup}, \text{2swap}, \text{2over} and \text{2drop}. For example, we can put a double value onto the stack by including a period in the number literal.

hex 23. \text{ok}$\langle$,ram$>23$ 0

Note that memory on the PIC18 microcontrollers is limited and, for FlashForth, the parameter stack is limited to 48 cells.

5.2 The return stack and its uses

During compilation of a new word, FlashForth establishes links from the calling word to the previously-defined words that are to be invoked by execution of the new word. This linkage mechanism, during execution, uses the return stack (rstack). The address of the next word to be invoked is placed on the rstack so that, when the current word is done executing, the system knows where to jump to the next word. Since words can be nested, there needs to be a stack of these return addresses.

In addition to serving as the reservoir of return addresses, the return stack is where the counter for the \text{for} \text{...} \text{next} construct is placed. (See section 11.) The user can also store to and retrieve from the rstack but this must be done carefully because the rstack is critical to program execution. If you use the rstack for temporary storage, you must return it to its original state, or else you will probably crash the FlashForth system. Despite the danger, there are times when use of the rstack as temporary storage can make your code less complex.

To store to the rstack, use $\triangleright$r to move TOS from the parameter stack to the top of the rstack. To retrieve a value, $\triangleright$r moves the top value from the rstack to the parameter stack TOS. To simply remove a value from the top of the rstack there is the word \text{rdrop}.
The word \texttt{r@} copies the top of the rstack to the parameter stack TOS and is used to get a copy of the loop counter in a \texttt{for} loop discussed in Section 11.

6 Using memory

As well as static RAM, the PIC18 microcontroller has program memory, or Flash memory, and also EEPROM. Static RAM is usually quite limited on PIC18 controllers and the data stored there is lost if the MCU loses power. The key attribute of RAM is that it has an unlimited endurance for being rewritten. The Flash program memory is usually quite a bit larger and is retained even with the power off. It does, however, have a very limited number of erase-write cycles that it can endure. EEPROM is also available, in even smaller amounts than static RAM and is non-volatile. It has a much better endurance than Flash, but any particular cell is still limited to about 100000 rewrites. It is a good place to put variables that you change occasionally but that want to retain even when the power is off. Calibration or configuration data may be an example of the type of data that could be stored in EEPROM. The registers that configure, control and monitor the microcontroller’s peripheral devices appear as particular locations in the static RAM memory.

In FlashForth, 16-bit numbers are fetched from memory to the stack by the word \texttt{@} (fetch) and stored from TOS to memory by the word \texttt{!} (store). \texttt{@} expects an address on the stack and replaces the address by its contents. \texttt{!} expects a number (NOS) and an address (TOS) to store it in. It places the number in the memory location referred to by the address, consuming both parameters in the process.

Unsigned numbers that represent 8-bit (byte) values can be placed in character-sized cells of memory using \texttt{c@} and \texttt{c!}. This is convenient for operations with strings of text, but is especially useful for handling the microcontroller’s peripheral devices via their special-function file registers. For example, data-latch register for port B digital input-output is located at address \$ff8a and the corresponding tristate-control register at address \$ff93. We can set pin RB0 as an output pin by setting the corresponding bit in the tristate control register to zero.

\[
%11111110 \ $ff93 \ c! \quad \longleftrightarrow \quad \text{ok<$\$,ram>}
\]

and then set the pin to a digital-high value by writing a 1 to the port’s latch register

\[
1 \ $ff8a \ c! \quad \longleftrightarrow \quad \text{ok<$\$,ram>}
\]

If we had a light-emitting diode attached to this pin, via a current-limiting resistor, we should now see it light up as in the companion hardware tutorial [1]. Here is what the gtkterm window contains after turning the LED on and off a couple of times.

\[
\text{FlashForth V3.8}
\]

\[
%11111110 \ $ff93 \ c! \quad \text{ok<$\$,ram>}
\]

1 $ff8a \ c! \quad \text{ok<$\$,ram>}
0 $ff8a \ c! \quad \text{ok<$\$,ram>}
1 $ff8a \ c! \quad \text{ok<$\$,ram>}
0 $ff8a \ c! \quad \text{ok<$\$,ram>}
\]
6 USING MEMORY

FlashForth allows you to very easily play with the hardware.

6.1 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  ← ok<$,ram>
```

creates a named storage location, x, which executes by leaving the address of its storage location as TOS:

```
x  ← ok<$,ram>f170
```

We can then fetch from or store to this address as described in the previous section.

FlashForth V3.8

```
marker -play ok<$,ram>
variable x ok<$,ram>
3 x ! ok<$,ram>
x @ . 3 ok<$,ram>
```

For FlashForth, the dictionary entry, x, is in the Flash memory of the microcontroller but the storage location for the number is in static RAM (in this instance).

FlashForth provides the words \texttt{ram}, \texttt{flash} and \texttt{eeprom} to change the memory context of the storage location. Being able to conveniently handle data spaces in different memory types is a major feature of FlashForth. To make another variable in EEPROM, try

```
eeprom variable y  ← ok<$,eeprom>
```

We can access this new (nonvolatile) variable as we did for the RAM variable x, but y retains its value, even when we turn off and on the power to the microcontroller.

```
4 y ! ok<$,eeprom>
y @ . 4 ok<$,eeprom>
x @ . 3 ok<$,eeprom>
FlashForth V3.8
```

```
y @ ok<$,ram>4
x @ ok<$,ram>4 0
```

6.2 Constants

A constant is a number that you would not want to change during a program’s execution. The addresses of the microcontroller’s special-function registers are a good example of use and, because the constant numbers are stored in nonvolatile Flash memory, they are available even after a power cycle. Execution of the word associated with a constant results in the data value being left on the stack.
6 USING MEMORY

$ff93 con trisb ok<$,ram>
$ff8a con latb ok<$,ram>
%11111110 trisb c! ok<$,ram>
0 latb c! ok<$,ram>
1 latb c! ok<$,ram>
0 latb c! ok<$,ram>
FlashForth V3.8

trisb ok<$,ram>ff93
%11111110 trisb c! ok<$,ram>ff93
0 latb c! ok<$,ram>ff93
1 latb c! ok<$,ram>ff93

6.3 Values

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

decimal ok<#,ram>
13 value thirteen ok<#,ram>
thirteen ok<#,ram>13
47 to thirteen ok<#,ram>13
thirteen ok<#,ram>13 47

The word to also works within word definitions, replacing the value that follows it with whatever is currently in TOS. You must be careful that to is followed by a value and not something else.

6.4 Basic tools for allocating memory

The words create and allot are the basic tools for setting aside memory and attaching a convenient label to it. For example, the following transcript shows a new dictionary entry x being created and an extra 16 bytes of memory being allotted to it.

create x ok<$,ram>
x u. f178 ok<$,ram>
here u. f178 ok<$,ram>
10 allot ok<$,ram>
here u. f188 ok<$,ram>

The word u. prints an unsigned representation of a number and the word here returns the address of the next available space in memory. In the example above, it starts with the same value as x but is incremented by (decimal) sixteen when we allotted the memory.

We can now access the memory allotted to x using the fetch and store words discussed earlier, in Section 6. To compute the address of the third byte allotted to x we could say x 2 +, remembering that indices start at 0.
We will discuss a way to neatly package the snippets of code required to do the address calculation later, in Section 12.2. Finally, note that the memory context for this example has been the static RAM, however, (as shown for variables in Section 6.1) the context for allotting the memory can be changed.

7 Comparing and branching

FlashForth lets you compare two numbers on the stack, using the relational operators $>\text{ }, <\text{ }$, and $=\text{ }$.

\[
\begin{align*}
2 & \text{ 3 } \text{ = ok<ram>0} \\
2 & \text{ 3 } \text{ > ok<ram>0 0} \\
2 & \text{ 3 } \text{ < ok<ram>0 0 ffff} \\
-1 & \text{ ok<ram>0 0}
\end{align*}
\]

These operators consume both arguments and leave a flag, to represent the boolean result. Above, we see that “2 is equal to 3” is false (value 0), “2 is greater than 3” is also false, while “2 is less than 3” is true. The true flag has all bits set to 1, hence the 16-bit hexadecimal representation $\text{ffff}$ and the corresponding signed representation $-1$.

FlashForth also provides the unary relational operators $0=\text{ }$ and $0<\text{ }$ which test if the TOS is zero and negative, respectively.

The relational words are used for branching and control. For example,

\[
\begin{align*}
: \text{test 0= invert if cr ." Not zero!" then ; ok<ram>} \\
0 \text{ test ok<ram>} \\
-14 \text{ test} \\
\text{Not zero! ok<ram>}
\end{align*}
\]

The TOS is compared with zero and the invert operator (ones complement) flips all of the bits in the resulting flag. If TOS is nonzero, the word if consumes the flag and executes all of the words between itself and the terminating then. If TOS is zero, execution jumps to the word following the then. The word cr issues a carriage return (newline).

The word else can be used to provide an alternate path of execution as shown here.

\[
\begin{align*}
: \text{truth 0= if ." false" else ." true" then ; ok<ram>} \\
1 \text{ truth true ok<ram>} \\
0 \text{ truth false ok<ram>}
\end{align*}
\]

A nonzero TOS causes words between the if and else to be executed, and the words between else and then to be skipped. A zero value produces the opposite behaviour.
8 Comments in Forth code

The word ( – a left parenthesis followed by a space – says “disregard all following text until the next right parenthesis in the input stream”. Thus we can add explanatory comments to colon definitions.

Stack comments are a particular form of parenthesized remark which describes the effect of a word on the stack. For example the comment ( x -- x x) could be used as the stack-effect comment for the word dup. The comment indicates that the word will make a copy of TOS and add it to the stack, leaving the original value, now as NOS.

The word \ (backslash followed by a space) is known as drop-line and is also available as a method of including longer comments. Upon executing, it drops everything from the input stream until the next carriage-return. Instructions to the user, clarifications of usage examples can be conveniently expressed in a block of text, with each line started by a backslash.

9 Integer arithmetic operations

With FlashForth having 16-bit cells, the standard arithmetic operators shown in Table 1 operate on 16-bit signed integers, in the range -32768 to +32767 (decimal). Note that the word u*/mod (scale) uses a 32-bit intermediate result. FlashForth also provides arithmetic operators for double numbers (32-bit integers), signed and unsigned. See the companion reference sheet [2] for a more complete list.

<table>
<thead>
<tr>
<th>word</th>
<th>effect</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>( n1 n2 -- n1+n2)</td>
<td>sum</td>
</tr>
<tr>
<td>-</td>
<td>( n1 n2 -- n1-n2)</td>
<td>difference</td>
</tr>
<tr>
<td>*</td>
<td>( n1 n2 -- n1*n2)</td>
<td>product</td>
</tr>
<tr>
<td>/</td>
<td>( n1 n2 -- n1/n2)</td>
<td>quotient</td>
</tr>
<tr>
<td>u/</td>
<td>( u1 u2 -- u1/u2)</td>
<td>unsigned quotient</td>
</tr>
<tr>
<td>u*/mod</td>
<td>( u1 u2 -- rem quot)</td>
<td>remainder and quotient</td>
</tr>
<tr>
<td>u*/mod</td>
<td>( u1 u2 u3 -- rem quot)</td>
<td>scaled remainder and quotient</td>
</tr>
</tbody>
</table>

Table 1: Arithmetic operators for single (16-bit) numbers.

For an example of using arithmetic operators, consider the conversion of temperature values from Celsius to Fahrenheit using the formula \(n2 = (n1\times9/5 + 32)\).

decimal ok<#,ram>
: to-f-1 ( n1 -- n2) 9 * 5 / #32 + ; ok<#,ram>
0 to-f-1 . 32 ok<#,ram>
100 to-f-1 . 212 ok<#,ram>
500 to-f-1 . 932 ok<#,ram>
5000 to-f-1 . -4075 ok<#,ram>

This simple function works fine, up until the intermediate result \((n1\times9)\) overflows the 16-bit cell. With a bit more bother, we can make use of the scaled operator to avoid
overflow of the intermediate result. Again, the following function computes expression
\((u1*9/5 + 32)\) but now uses the scale operator, producing remainder and quotient for
unsigned numbers.

: to-f-2 ( u1 -- u2) 9 5 u*/mod swap drop #32 + ; ok<#,ram>
0 to-f-2 . 32 ok<#,ram>
100 to-f-2 . 212 ok<#,ram>
500 to-f-2 . 932 ok<#,ram>
5000 to-f-2 . 9032 ok<#,ram>

The phrase \(\text{swap drop}\) is used to discard the remainder the was left on the the stack by
\(u*/mod\).

10 A little more on compiling

While compiling, it is possible to temporarily switch to interpreter mode with the word
\([\text{ and switch back into compile mode with }\]\. The following example defines the word
\texttt{feet} that converts a number representing a length in feet to an equivalent number of
millimetres. The intermediate result is in tenths of a millimetre so that precision is
retained and, to make the conversion clear, the numeric conversion factor is computed as
we compile the word.

: feet ( u1 -- u2)
  [ #254 #12 * ] literal #10 u*/mod
  swap drop ; ok<#,ram>
10 feet ok<#,ram>3048

The word \texttt{literal} is used to compile the data value in TOS into the definition of \texttt{feet}.
At run-time, that data value will be placed onto the stack.

11 Looping and structured programming

The control words available for structured programming are shown in Table 2, where \(xxx\)
and \(yyy\) denote sequences of words and \texttt{cond} denotes a boolean flag value. Within the
body of a \texttt{for} loop, you may get the loop counter with the word \texttt{r@}. It counts from \(u-1\)
down to 0. If you \texttt{exit} from a \texttt{for} loop, you must drop the loop count with \texttt{rdrop}.

Here are a couple of examples of counted loops, one constructed from the generic
\texttt{begin...until} construct, and the other using the dedicated \texttt{for...next} construct. Note
the difference in counter ranges.

-countdown ok<#,ram>
marker -countdown ok<#,ram>
: countdown1 ( n --)
  begin cr dup . 1- dup 0= until
  drop ; ok<#,ram>
5 countdown1
Table 2: Flow control in FlashForth.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cond if xxx else yyy then</code></td>
<td>Conditional execution.</td>
</tr>
<tr>
<td><code>begin xxx again</code></td>
<td>Infinite loop.</td>
</tr>
<tr>
<td><code>begin xxx cond until</code></td>
<td>Loop until <code>cond</code> is true.</td>
</tr>
<tr>
<td><code>begin xxx cond while yyy repeat</code></td>
<td>Loop while <code>cond</code> is true, <code>yyy</code> is not executed</td>
</tr>
<tr>
<td><code>u for xxx next</code></td>
<td>Loop <code>u</code> times.</td>
</tr>
<tr>
<td><code>leave</code></td>
<td>Sets r@ to zero so that we leave the loop when</td>
</tr>
<tr>
<td></td>
<td>next is encountered.</td>
</tr>
<tr>
<td><code>exit</code></td>
<td>Exit from a word.</td>
</tr>
</tbody>
</table>

5
4
3
2
1  ok<#,ram>:
countdown2 ( n --)
   for cr r@ . next ;  ok<#,ram>
5  countdown2
4
3
2
1
0  ok<#,ram>

It was convenient, when setting up these examples, to put the source code into a little file that could be reloaded easily each time the source text was changed.

`-countdown` marker `+countdown`
: countdown1 ( n --)
   begin cr dup . 1- dup 0= until
   drop ;
5  countdown1
: countdown2 ( n --)
   for cr r@ . next ;
5  countdown2

12 More on defining words

The compiler word `create` makes a new dictionary entry using the next name in the input stream and does nothing else. It can be used to build other defining words. For example, we can make our own variation of word `variable` as
: make-var create 1 cells allot ; ok<#,ram>

Here `make-var` can be used to make an uninitialized variable that can hold a single number. When `make-var` is executed, the first word within its definition (create) sets up a dictionary entry with the name coming from the next text in the input stream, the number 1 is pushed onto the stack, the word `cells` converts the TOS to the appropriate number of bytes (2 in this case) and the word `allot` increments the pointer to the next available space in memory by this number of bytes. This allots one cell to the newly created `child` word.

```
make-var alpha ok<#,ram>
13 alpha ! ok<#,ram>
alpha @ . 13 ok<#,ram>
```

At run time for the newly created word, `alpha` leaves its data-space address on the stack and we may store to or fetch from this address as shown above.

As a second example, we can also build a defining word for making initialized variables.

```
: make-zero-var create 0 , ; ok<#,ram>
```

Instead of just allotting space for the data, the word `,` (comma) puts TOS into the next cell of memory and increments the memory-space pointer by appropriate number of bytes. Run time use of the newly defined variable is the same as for any other variable.

```
make-zero-var beta ok<#,ram>
beta @ . 0 ok<#,ram>
```

### 12.1 `create ... does> ...`

The word `does>` is used to specify a run-time action for the child words of a defining word. We can make our own variant of `constant` and test it.

```
: make-con create , does> @ ; ok<#,ram>
53 make-con prime ok<#,ram>
```

At run time for the word `make-con`, create sets up the new dictionary entry with the next text in the input stream (`prime`), the word `,` (comma) compiles TOS (53 in this example) into the memory and `does>` stores the following words up to the terminating semicolon (only `@` in this case), such that they will be executed at the run time of words defined by `make-con`. Thus, when `prime` is executed, the address of its data memory is put onto the stack and the word `@` executed.

```
prime . 53 ok<#,ram>
```

Although only one word is stored as the run time code for `prime` in this example, it could be arbitrarily complex.
12.2 Creating arrays

The `create...does>` pair can be used to define some convenient array-defining words. For an array of bytes, it is straight-forward to manually allot the memory and address it at run time.

```
create my-array 10 allot ok<#,ram>
my-array 10 $ff fill ok<#,ram>
my-array @ . -1 ok<#,ram>
```

Here, `my-array` is first allotted 10 bytes. At run time for `my-array`, the address of the start of the 10 bytes is left on the stack, and we use the `fill` word to completely fill the array of bytes with ones. Accessing the first cell (2 bytes) and printing it (to get -1 sent to the serial port) confirms that all bits are 1s.

It is also straight-forward to create an array with particular data compiled in.

```
create my-cell-array 100 , 340 , 5 , ok<#,ram>
```

Remember to use comma to compile in every data element, not leaving off the last one. At run time, `my-cell-array` puts the address of the start of the associated memory on the stack and the address of the desired cell (index 2, in this case) needs to be calculated (2 `cell` +) before executing `@` (fetch).

```
my-cell-array 2 `cell` + @ . 5 ok<#,ram>
```

This address calculation code can be added into the defining word with `does>` such that the subsequently defined `my-cells` array will have the snippet of code (`swap `cell` +`) executed at run time to convert from a cell index to an address.

```
\ A defining word for creating arrays. 
: mk-cell-array ( u --)
create cells allot
does> swap `cell` + ; ok<#,ram>
```

The `swap` is used to get the index as TOS with the array address as NOS, `cell` scales the index to the offset as a number of bytes and + adds the offset to the array address. The newly computed cell address is left as TOS for use by the words `!` (store) and `@` (fetch).

```
\ Make an array and access it. 
5 mk-cell-array my-cells ok<#,ram>
3000 0 my-cells ! ok<#,ram>
3001 1 my-cells ! ok<#,ram>
3002 2 my-cells ! ok<#,ram>
1 my-cells @ . 3001 ok<#,ram>
```

12.3 Jump tables

Sometimes we want to execute one of a large selection of words, depending on the outcome of a calculation. It is possible to set up a sequence of tests and branches (as introduced in Section 7), however, FlashForth allows a neater solution in the form of a *jump table*. A
word in FlashForth can be executed by feeding its execution token to the word execute. If you have a table of execution tokens, then you need only look up the one corresponding to a given index, fetch it and say execute. The following transcript shows such a jump-table for a collection of four functions, the word do-action to execute the appropriate word, given an index, and a sample of trials that shows the jump-table mechanism in action.

\ Set up the words that we want to selectively execute.
: ring ( -- ) " ring ring" ; ok<#,ram>
: open ( -- ) " opening" ; ok<#,ram>
: laugh ( -- ) " ha ha" ; ok<#,ram>
: cry ( -- ) " crying" ; ok<#,ram>

\ Store the execution tokens in a table that allots into flash memory.
flash ok<#,flash>
create actions ' ring , ' open , ' laugh , ' cry , ok<#,flash>
ram ok<#,ram>

: do-action ( n --)
   0 max 3 min
cells actions + @ execute ; ok<#,ram>

\ Call up the actions.
3 do-action crying ok<#,ram>
0 do-action ring ring ok<#,ram>
2 do-action ha ha ok<#,ram>
5 do-action crying ok<#,ram>

The word ' (tick) finds the following name in the dictionary and puts its execution token (xt) on the stack. The word , (comma) compiles the xt into the table. Note that we are compiling these tokens into the flash memory of the microcontroller so that the jump table continues to work, even after a power break. In do-action, the words 0 max 3 min limit the incoming index value to the valid range for looking up a token in the jump-table. The token is fetched to TOS and then execute called. The final line of the transcript shows that the word cry is executed for the invalid index value of 5. You may what to handle incorrect input differently in your application.

13 Strings

The Hello World! program, back in Section 3 could have been written a little differently, as the following transcript shows.

: hi2 s" Hello World!" cr type ; ok<#,ram>
hi2
Hello World! ok<#,ram>
The word s" compiles the string into Flash memory and, at run time, leaves the address of the string and the number of characters in the string on the stack. From these data, the word type will send the characters to the standard output.

### 13.1 Pictured numeric output

To get numbers output in a particular format, FlashForth provides the basic words #, <#, #s, #>, sign and hold. These words are intended for use within word definitions. Here is an example of their use.

: (d.2) ( d -- caddr u)  
  tuck dabs <# # # [char] . hold #s rot sign #> ; ok<#,ram>

A double number sits on the stack, with it most significant bits, including its sign, in TOS. First, tuck copies TOS (with the sign bit) to below the double number and then the absolute value is converted to a string of characters representing the unsigned value. Starting with the least significant digits, there will be two to the right of a decimal point. The phrase [char] . hold adds the decimal point to the string. In this phrase, [char] . builds in the representation of the decimal point as a numeric literal (ASCII code 46) and hold then adds it to the string under construction. After adding the decimal point, the word #s converts as many remaining digits as required. The word rot is used to bring the copy of the original most-significant cell to TOS and the word sign adds a negative sign to the string if required. Finally, word #> finishes the conversion, leaving the character-address of the resultant string and the number of characters in it on the top of the stack.

437658. (d.2) type 4376.58 ok<#,ram>  
-437699. (d.2) type -4376.99 ok<#,ram>  
45. (d.2) type 0.45 ok<#,ram>

Note that, with FlashForth, double integers must be entered as literals with the decimal point as the last character.

### 14 Forth programming style

There is much written on the style of Forth programming and, indeed, there is book called “Thinking Forth” [8]. Here are a number of recurring statements on programming in Forth that are relevant to sections of this tutorial:

- Build your application from the bottom up, testing new words as you go.

- Choose simple and meaningful names, so that the intent of each word clear and your code is easily read, almost as statements you would make to another person.

- Always provide stack-effect comments.

- As you build your application, refactor your code aggressively so that you don’t need complicated stack manipulations to access your data.
• Clean up after yourself and don’t leave rubbish on the stack.
• Use the return-stack for temporary storage when it makes your code cleaner but be very careful to clean up when doing so.

References


