Delta Forth .NET
Version 1.1

World's first Forth compiler for the .NET platform

http://www.dataman.ro/dforth

Reference Guide

Valer BOCAN

May 2002

Chapter 1
Overview

Delta Forth is a non-standard Forth dialect. It has several limitations over traditional standards; however it may be an excellent starting point for beginners. The traditional *compreter* (compiler – interpreter) approach of other implementations did not appeal when the Delta dialect was designed since computers evolved enormously since the original Forth specification was written. Instead, we deal with *compiled* programs and thus several original Forth words – mainly related to chained execution - have lost their meaning (see STATE, COMPILE, IMMEDIATE, a.s.o.). Please consult this document when you need information about the Delta dialect.

It has been stated that a complete traditional Forth environment can be coded by a single person in a three month time. I managed to release the beta 1 version of Delta in half that time. Despite the short time it took to be developed, this tool has a long history, being a continuation of the award-winning **Delta Forth for Java** project that I started back in 1997. At that time, it was the first Forth compiler for Java and was a real surprise when I presented it as my graduation project two years later. The .NET compiler is used to write a part of the software for my Ph.D. thesis.

This software is free of charge. However, I spent hundreds of hours designing and developing it, so if you like it please make a donation to a charity of your choice then drop me a note.

The C# source code has approximately 4000 lines.

Chapter 2
Delta Forth .NET Basics

Overview of a Forth program

As in any Forth dialect, programs contain word definitions. A word is a logical structure and association of items that perform a well defined task. Let's take the well-known example, Hello World:

```
: main
      .”Hello world!”
;
```

Here we define de word **MAIN** and that is the **starting point** of any program. MAIN has to be defined; either we create an executable or a library.

Calling a word is done simply by writing its name:

```
: main
      DisplayText
;
: displaytext
      .”Hello world!”
;
```

We may notice two things: **Forth is not case-sensitive** and **the order in which the words are defined is not important.**

Comments are very important within code, the reason is obvious. There are two types of comments in Forth: multi-line and single-line. Our example becomes:

```
: main                  \ Program starting point
      DisplayText       \ Call the word to display the text
;
(     Word to display the greeting note
      Input:      None
      Output:     None
      Author:     John Doe
)
: DisplayText
      .”Hello world with comments!”
;
```

If you need to spread the program across multiple source files, you have to combine the result to be compiled. For this, we use the **LOAD** directive which is similar to *#include* in C and C++. Here is an example:

```
( Source file: DisplayWords.4th )
: DisplayText
      .”Hello world from another file!”
;
```

```
( Source file: Main.4th)
load DisplayWords.4th

: main
      DisplayText
;
```

If two files include each other, the compiler will issue an error.

Stacks

Forth is a typeless language and it relies on two stacks to perform operations:

- The **Forth Stack** is made of 32-bit integers and it is the main stack. The majority of the Forth words modify this stack in a way or another. The default stack size is 524288 cells, but it may be modified using the /FS option.
- The **Return Stack** is used for brief temporary storage and for holding the current value of the DO-LOOP structure. The default size is 1024, but it may be modified using the /RS option.

Loading a value on the stack is straightforward:

```
: main
      10 .        \ Load 10 on the stack, then display it
;
```

Chapter 3
Primitives

Primitives are built-in words used to perform basic operations.

Memory Operations

Word	Description / Action
@ (addr --- n) "fetch"	Reads the content of a specified memory location and places the result on top of stack
? (addr ---) "question-mark"	Displays the content of a memory location
! (n addr ---) "store"	Stores a value at the specified memory address
+! (n addr ---) "plus store"	Adds a value to the content of a specified memory address

Arithmetic Operations

Word	Description / Action
+ (n1 n2 --- n1+n2)	Addition
- (n1 n2 --- n1-n2)	Subtraction
*** (n1 n2 --- n1*n2)**	Multiplication
/ (n1 n2 --- n1/n2)	Division
MOD (n1 n2 --- n)	Division remainder
/MOD (n1 n2 --- nr nres)	Division remainder and result
***/ (n1 n2 n3 --- n1*n2/n3)**	Scaling operator
***/MOD (n1 n2 n3 --- nr nres)**	Scaling operator
MINUS (n --- -n)	Changes the sign of the number
ABS (n --- |n|)	Absolute value of number
MIN (n1 n2 --- min(n1,n2))	Computes the minimum of two values
MAX (n1 n2 --- max(n1,n2))	Computes the maximum of two values
1+ (n --- n+1)	Increments the top of stack by 1
2+ (n --- n+2)	Increments the top of stack by 2
0= (n --- b)	Test for "zero-equal"
0< (n --- b)	Test for "zero-less"
= (n1 n2 --- b)	Test for "equal"
< (n1 n2 --- b)	Test for "less"
> (n1 n2 --- b)	Test for "greater"
<> (n2 n2 --- b)	Test for "not-equal"

Logical Operations

Word	Description / Action
AND (n1 n2 --- b)	Returns 1 if both initial values are not zero
OR (n1 n2 --- b)	Returns 1 if at least one initial value is not zero
NOT (n1 --- b)	Negates the initial value

Bitwise Operations

Word	Description / Action
~AND (n1 n2 --- n)	Bitwise AND
~OR (n1 n2 --- n)	Bitwise OR
~XOR (n1 n2 --- n)	Bitwise XOR
~NOT (n --- m)	Bitwise NOT

Stack Operations

Word	Description / Action
DUP (n --- n n)	Duplicates the value on the top of the stack
-DUP (n --- n : n n)	Duplicates the value on the top of the stack if different from 0
DROP (n ---)	Drops the element on top of the stack
SWAP (n1 n2 --- n2 n1)	Swaps the two elements of top of the stack
OVER (n1 n2 --- n1 n2 n1)	Duplicates the element before the on top of the stack
ROT (n1 n2 n3 --- n2 n3 n1)	Rotates the last three elements on the top of the stack
SP@	Returns the current position of the parameter stack pointer
RP@	Returns the current position of the return stack pointer
SP!	Flushes the parameter stack
RP!	Flushes the return stack

Return Stack Operations

Word	Description / Action
>R (n ---) "to-R"	Transfers the element to the top of return stack
R> (--- n) "R-from"	Transfers the element to the top of the stack
I (--- n)	Copies the element from the return stack to the parameter stack

Display Operations

Word	Description / Action
EMIT (c ---)	Displays the character with the given ASCII code
CR (---)	Emits the CR and LF characters
SPACE (---)	Displays a space
SPACES (n ---)	Displays a given number of spaces
."<text>" (---)	Displays the text between quotes
TYPE (addr n ---)	Displays a string of specified length from the given address

Keyboard Operations

Word	Description / Action
KEY (--- c)	Places on the stack the ASCII code of the key pressed
EXPECT (addr n ---)	Reads at most n characters and places their codes starting from address addr
QUERY (---)	Awaits at most 80 characters to be typed and places them at the area pointed by TIB

String Operations

Word	Description / Action
FILL (addr n c ---)	Fills n cells from address addr with value c
ERASE (addr n ---)	Fills with 0 n cells from address addr
BLANKS (addr n ---)	Fills with 32 (blank) n cells from address addr
CMOVE (addr1 addr2 n ---)	Moves n cells from address addr1 to address addr2
COUNT (addr --- n)	Counts the number of characters from address addr up to the terminating 0
"<text>" (addr ---)	Places the "text" at address addr

Conversions

Word	Description / Action
STR2INT (--- n)	Converts the string at TIB to an integer and places it on the stack
INT2STR (addr n ---)	Converts the value n to a string and places it to addr

Miscellaneous

Word	Description / Action
EXIT (---)	Leaves the current word unconditionally

System variables

Word	Description / Action
PAD	Points to a 64-cell work area
S0	Parameter stack origin
R0	Return stack origin
TIB	Points to a 80-cell buffer used in I/O operations

Chapter 4
Constants

Delta Forth allows the use of integer and string constants.
Numbers encountered outside words are placed in a virtual stack. Each constant definition "uses" the number on top of stack.

```
10 constant con1              \ Define an integer constant
20 constant con2              \ Define another integer constant
"The sum is" constant text    \ Define a string constant

: main
      tib text                \ Dump the text in 'text' at TIB
      tib dup count type      \ Type the text at TIB
      con1 con2 +             \ Calculate the sum
      .                       \ Display the sum
;
```

When encountered in a word definition, a string constant places the text at the address found on top of stack.

Chapter 5
Global and Local Variables

Global variables are defined outside words and their size is by default 1. This may be modified up to an arbitrary size with the ALLOT primitive:

```
variable X              \ Variable with size 1 cell
variable Y 19 allot     \ Variable with size 1 + 19 = 20 cells
```

When encountered in a word definition, a variable places its address on the stack.

```
variable var
: main
      0 var !           \ Initialize variable
      var dup @ 1+ !    \ Increment variable content
;
```

Unlike global variables, local variables are only visible within the word they have been defined:

```
: word1
      variable locvar
      [ ... ]
;

: main
      locvar @ .        \ Error, locvar is not accessible
;
```

Chapter 6
Libraries

Delta Forth .NET allows you to create libraries that can be later called using reflection. A library is no different from a regular Forth program, just the LIBRARY keyword and the /DLL compiling option.
The name of the .NET class to be created is specified using the LIBRARY keyword. If you don't specify a name, *DeltaForthEngine* is used by default. You are still required to create the function MAIN which in this case can be used to initialize the Forth environment (variables, settings, etc.)

```
library MathOp

: main      \ Library entry point
      \ Your code here
;

: addition
      [ ... ]
;

: subtraction
      [ ... ]
;
```

Now compile the above code using the /DLL option and the result will be a DLL file that may be invoked from other languages using reflection.

To invoke the words defined in the MathOp library, we can use the following sequence:

```
extern addword mathop.dll MathOp.addition

: main
      addword
;
```

We assume that the library is in file MathOp.dll. The **EXTERN** keyword defines the external word **addword**, which at runtime calls the method *addition* of the class MathOp, in the file mathop.dll.

Chapter 7
Control Structures

IF-ELSE-THEN

The conditional structure is used to take decisions based on some condition. In Forth, the condition is true if the top of stack is non-zero and false if otherwise.

<condition> IF <branch for true> THEN

<condition> IF <branch for true> ELSE <branch for false> THEN

```
: main
      10 30       \ We've got two number
      >           \ Compare them
      if
            ."Branch for true"
      else
            ."Branch for false"
;
```

The number of nested IF structures is not limited.

DO-LOOP

This is similar to the FOR statements of other languages. This structure is used when the number of wanted iterations is known in advance.

fv iv DO *[....]* LOOP

fv iv DO *[....]* +LOOP **fv – final value, iv – initial value**

The DO statement transfers the initial and the final value to the return stack and begins execution of statements after DO. LOOP peeks the stack and if the initial value is equal to or greater than the final value, exits the loop, otherwise it increments the current value by 1. If the value needs to be incremented by something else than 1, you may use +LOOP to end the structure.

```
: main      \ Display numbers from 0 to 100
      100 0
      do
            I . space
      loop
;

: main      \ Display numbers from 0 to 100 step 2
      100 0
      do
            I . space
```

```
      2     \ Step
      +loop
;
```

If for any reason the loop needs to be left, use the LEAVE statement, which forces early termination of the structure.

BEGIN-AGAIN

This is an infinite loop.

BEGIN *[...]* AGAIN

```
: main
      begin
      ."Blah..."
      again
;
```

BEGIN-UNTIL

This is similar to final-test constructs in other languages. The sequence between BEGIN and UNTIL is executed until a condition is met.

BEGIN *[...]* <condition> UNTIL

```
: main
      variable cnt      \ Local variable
      0 cnt !           \ Initialize 'cnt' to 0
      begin
            cnt @ 1+ cnt !    \ Increment variable by 1
            cnt ? space       \ Display the counter value
            cnt @ 25 >        \ Test if the counter is less than 25
      until
;
```

BEGIN-WHILE-REPEAT

This is similar to initial-test constructs in other languages. The execution begins with the sequence between BEGIN and WHILE. The WHILE statement checks the top of stack and if the value is true (non-zero), execution continues until REPEAT, then the process starts again. If the value is false, structure execution is aborted.

BEGIN *[...]* <condition>WHILE *[...]* REPEAT

```
:  main
        variable cnt          \ Local variable
        0 cnt !               \ Initialize 'cnt' to 0
        begin
            cnt ? space       \ Display the counter value
            cnt @ 25 <        \ Test if the counter is less than 25
```

```
        while
            cnt @ 1+ cnt !    \ Increment variable by 1
        repeat
;
```

CASE-ENDCASE

The selector structure has the following general structure:

<selector value>
CASE
<case_value> OF *[…]* ENDOF
<case_value> OF *[…]* ENDOF

[…]

<case_value> OF *[…]* ENDOF
ENDCASE

```
: main
  1 test cr       \ Test for 1
  2 test cr       \ Test for 2
  3 test cr       \ Test for 3
  4 test cr       \ Test for 4
;
: test
  case
  1 of ."One"   endof
  2 of ."Two"   endof
  3 of ."Three" endof
  ."Something else"
  endcase
;
```

Chapter 8
Compiler Error Messages

***Identifier* should be declared outside words.**
- The specified identifier should be declared outside words. Such examples are CONSTANT, ALLOT etc.

***Identifier* should be declared inside words.**
- The specified identifier should be declared inside words. Such examples are IF, DO, WHILE etc.

***Identifier* is a reserved identifier.**
- You cannot redeclare the meaning of the built-in identifiers.

***Identifier* is an invalid identifier.**
- The identifier is not properly defined (should not begin with a figure and be less than 31 characters in length.

Unable to define constant. Number or string required before CONSTANT.
- The "virtual" stack is empty. You need to specify at least an integer or a string before you can define constants.

Unable to allot variable space. Number needed before ALLOT.
- You did not specify the amount by which to increase the size of the variable.

Unexpected end of file.
- The file ended before the compiler found certain expected constructs.

Wrong constant type specified for ALLOT. Use an integer.
- You specified a string where an integer was needed.

Nested words are not allowed.
- You must end a word definition before you start another one.

Malformed control structure.
- See the definition of the control structures and follow the specifications.

Control structures must be terminated before ';'
- Control structures cannot spread across multiple words.

Program starting point is missing. Please define the word MAIN.
- You must define a function MAIN in a Delta Forth program.

Constant redefines an already defined constant or variable.
Variable redefines an already defined constant or variable.
- Global variables and constants share the same name space, thus the names must be unique. You cannot have a variable and a constant with the same name.

Chapter 9
Compiler Command-Line Options

The Delta Forth .NET compiler has a few options that you can set when you compile programs.

Command-line syntax:

DeltaForth.exe <source file> [options]

<source file>
- Represents the Forth source file to be compiled.

/NOLOGO
- Disables the display of copyright logo

/QUIET
- Disable the display of any messages on the screen, except for the compiling error message

/CLOCK
- Displays the timings for various compiling stages, as well as the total amount of time

/EXE
- Generates EXE files (this option is default)

/DLL
- Generates DLL files

/NOCHECK
- Disables the generation of stack bounds checking code. Any operation that causes a stack overflow or underflow will throw an exception

/FS:<size>
- Specifies the Forth stack size at runtime. Default is 524288 cells.

/RS:<size>
- Specifies the return stack size at runtime. Default is 1024 cells.

/MAP
- Generates map information (address and size of variables, type and value of constants, external words, etc.)

/OUTPUT=<target file>
- Sets the name and directory of the target file, in case the default is not suitable