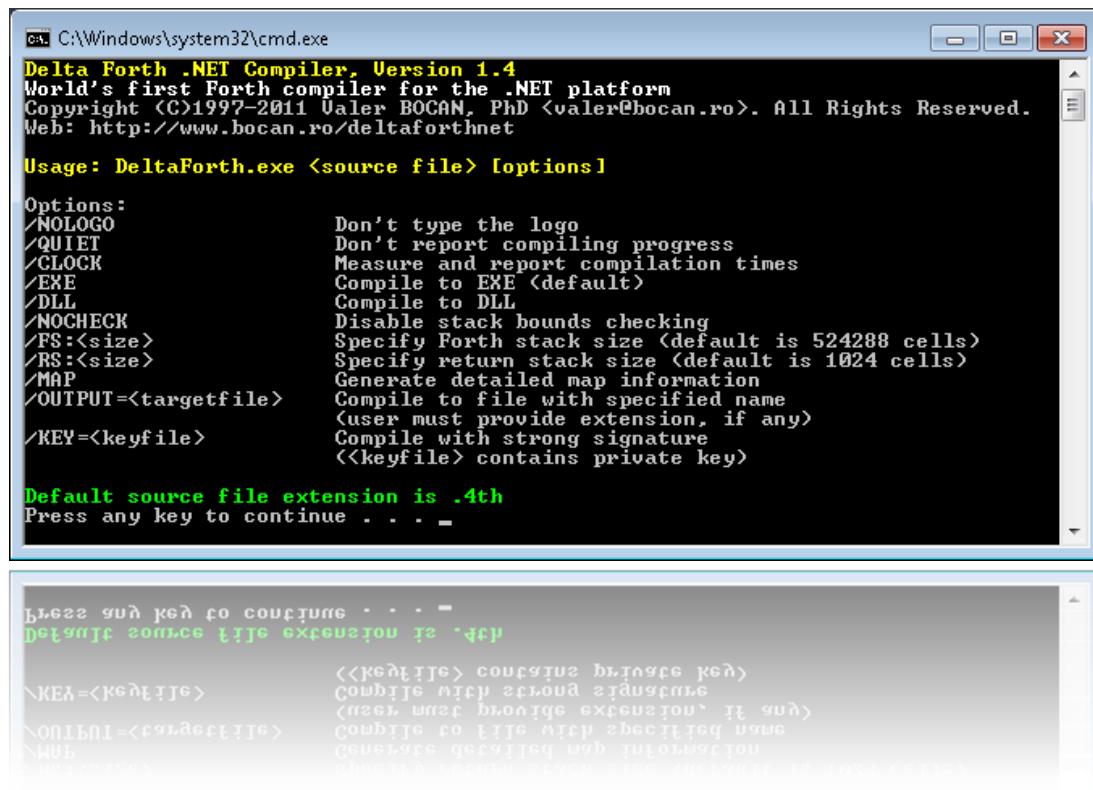


Delta Forth .NET

World's first Forth compiler for the .NET platform



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The text inside the window is as follows:

```
Delta Forth .NET Compiler, Version 1.4
World's first Forth compiler for the .NET platform
Copyright (C)1997-2011 Valer BOCAN, PhD <valer@bocan.ro>. All Rights Reserved.
Web: http://www.bocan.ro/deltaforthnet

Usage: DeltaForth.exe <source file> [options]

Options:
/NOLOGO           Don't type the logo
/QUIET           Don't report compiling progress
/CLOCK           Measure and report compilation times
/EXE             Compile to EXE (default)
/DLL            Compile to DLL
/NOCHECK         Disable stack bounds checking
/FS:<size>       Specify Forth stack size (default is 524288 cells)
/RS:<size>       Specify return stack size (default is 1024 cells)
/MAP            Generate detailed map information
/OUTPUT=<targetfile> Compile to file with specified name
                  (user must provide extension, if any)
/KEY=<keyfile>   Compile with strong signature
                  (<keyfile> contains private key)

Default source file extension is .4th
Press any key to continue . . .
```

The bottom part of the image shows a blurred, mirrored version of the same text, likely a reflection or a second instance of the same content.

Version 1.4

Copyright 1997-2011 © Valer BOCAN, PhD

valer@bocan.ro

This page is intentionally left blank

Contents

Overview	4
Legal Stuff	4
Delta Forth .NET Basics	5
Overview of a Forth program.....	5
A Word on Stacks (not a Forth Word though)	6
Primitive Forth Words.....	6
Memory Operations.....	6
Arithmetic Operations	7
Logical Operations	7
Bitwise Operations.....	7
Stack Operations	7
Return Stack Operations	8
Display Operations.....	8
Keyboard Operations	8
String Operations	8
Conversions.....	8
Miscellaneous	9
System Variables	9
Constants	9
Global and Local Variables	9
Libraries.....	10
Control Structures.....	10
IF-ELSE-THEN.....	10
DO-LOOP	11
BEGIN-AGAIN	11
BEGIN-UNTIL	12
BEGIN-WHILE-REPEAT.....	12
CASE-ENDCASE.....	12
Compiler Error Messages.....	13
Identifier should be declared outside words.	13
Identifier should be declared inside words.	13
Identifier is a reserved identifier.....	13

Identifier is an invalid identifier.....	13
Unable to define constant. Number or string required before CONSTANT.	13
Unable to allot variable space. Number needed before ALLOT.	13
Unexpected end of file.....	13
Wrong constant type specified for ALLOT. Use an integer.....	14
Nested words are not allowed.....	14
Malformed control structure.....	14
Program starting point is missing. Please define the word MAIN.	14
{Constant Variable } redefines an already defined constant or variable.	14
Compiler Command-Line Options.....	14

Overview

Delta Forth is a non-standard Forth dialect. It has several limitations over traditional standards but it is an excellent starting point for beginners and enthusiast. The traditional **compreter** (compiler – interpreter) approach of other implementations does not fit in the .NET environment as we deal with compiled programs. Several of the words in the original Forth specification have lost their meaning (see STATE, COMPILE, IMMEDIATE, etc.) but I think the compiled approach of this flavor is still appealing and useful. The true power of Forth is in its ability to stay close to the machine assembly language while it has some incredibly powerful control structures.

Delta Forth generates true .NET code in the form of console executables (.EXE) and libraries (.DLL). The code can be executed on any .NET platform, such as Microsoft .NET on Windows or Mono on Linux. Beginning with version Delta Forth 1.2 the code generated by the compiler can be strongly-signed using a regular signature file generated by the *sn.exe* tool.

Forth literature states 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.

Legal Stuff

This software is free to use for any purpose. You are free to experiment with the compiler, its source code, compiled Forth programs, etc. Branching this code is not allowed, that is you may not create your own flavor of Forth based on the code provided here. If you have done anything fancy and you think that would be useful for other Forth enthusiast, please send me a note and I will incorporate your change in Delta Forth.

Delta Forth .NET Basics

Overview of a Forth program

Forth programs contain word definitions. Words are to Forth what methods or procedures are to other imperative languages, so a word is a logical structure and association of items that perform a well-defined task. This is how Hello World looks like in Delta Forth:

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

The starting point of any Delta Forth program is the word *main* and it has to be defined whether we create an executable or a library.

A word is invoked simply by mentioning its name. As parameters are transmitted through the evaluation stack there is no need for word parameters and that is one of the strengths of Forth (and also one of the most difficult things to understand for beginners).

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

You may note that Forth is not case-sensitive (you can mix the case of word definitions and word calls) and the order in which the words are defined is not important.

As in many other programming languages, there are two types of comments in Forth: multi-line and single-line. Let's enrich our small Forth example with comments:

```
: 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 the code grows beyond a manageable size or if you want to semantically group Forth words, you may want to spread the program across multiple source files. This is done by **LOADing** files, a technique 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
;
```

The compiler will be invoked with the source file that contains the word *main* and that in turn will automatically include all word, constant and variable definitions in files specified by the LOAD directives. Invoking the compiler with a source file that does not have the word *main* will result in an error. The compiler also keeps track of the files already processed so any circular dependency will be detected and signaled as such.

A Word on Stacks (not a Forth Word though)

Forth is a special breed among programming languages and it is essentially a typeless language. In order to transmit parameters it relies on two stacks, that is the evaluation stack (or the Forth stack) and the return stack.

- The Forth Stack consists of 32-bit integers and it is used to transmit parameters from one word to another. You may think about it as a large collection of processor registers that can do virtually anything. The majority of the Forth words modify this stack in one way or another and the default size is 524288 cells. The default size can be modified using the /FS compiler option.
- The return stack is used for brief storage and for holding the current enumerator value of the DO-LOOP/+LOOP structure. The default size is 1024 and it can be modified using the /RS compiler option.

When the compiler encounters an integer which is not part of a constant, variable or array definition it will simply add the respective value to the evaluation stack. See the sample code below:

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

Primitive Forth Words

The words that are part of the Delta Forth dialect are listed in the tables below. Words are categorized by their purpose.

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 ---) "questionmark"	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 one 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 top element from the return stack to the parameter stack
I' (--- n)	Copies the second top element from the return stack to the parameter stack
J (--- n)	Copies the second top 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 64cell work area
S0	Parameter stack origin
R0	Return stack origin
TIB	Points to a 80-cell buffer used in I/O operations

Constants

Delta Forth allows the use of integer and string constants. Numbers encountered outside word definitions are placed in a temporary “virtual” stack from which they are extracted as constant definitions are encountered. See example below:

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

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

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 to create arrays. See example below:

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

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

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

```
: word1
    variable locvar
    \ Some more code here
;
: main
    locvar @ . \ Error, locvar is not accessible
```

```
;
```

Local variables are always static, therefore the values are preserved between recursive and non-recursive calls.

Libraries

Delta Forth allows you to create libraries that can be later called from other .NET languages. A library is no different from a regular Forth program, except for the `LIBRARY` keyword and the `/DLL` compiler option. The name of the .NET class that will 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 word `MAIN` which will be used to initialize the Forth environment (variables, settings, etc.)

```
library MathOp
: main \ Library entry point
    \ Your code here
;
: addition
    \ Perform addition
;
: subtraction
    \ Perform subtraction
;
```

If we compile the previous code using the `/DLL` option, the result will be a .NET library (as DLL) that can be consumed by any CLR-compliant language. In Delta Forth we would call the methods in the library like so:

```
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*.

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 numbers
> \ 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 in other imperative languages. This structure is used when the number of iterations is known in advance.

fviv DO [....] LOOP fviv 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 it exits the loop, otherwise it increments the current value by 1. If the value needs to be incremented by something else than 1, you can 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
;

```

The LEAVE statement inside a DO-LOOP structure terminates the iteration early and returns at the first statement after LOOP or +LOOP.

BEGIN-AGAIN

This is the infinite loop.

BEGIN [...] AGAIN

```

: main
  begin
  ."Test-"
  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 and the process starts again. If the value is false, execution jumps to the first statement after REPEAT.

```
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
;

```

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 should 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.
- See the definition of the control structures and follow the specifications.
- Control structures cannot spread across multiple words.

Malformed control structure.

- Control structures must be terminated before ';'.

Program starting point is missing. Please define the word MAIN.

- You must define the word MAIN in a Delta Forth program (it's the starting point or the library initializer).

{Constant | 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.

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 compilation error messages.
- **/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=<targetfile>** - Sets the name and directory of the target file, in case the default is not suitable.
- **/KEY=<keyfile>** - Signs the code with the key specified in the key file. A new key file can be generated with the standard .NET tool *sn.exe*.