MinForth V3.4

a MINimalistic but complete FORTH system in C

© Andreas Kochenburger 1994 - 2021

... beware, it's all Genglish down there! ☺

What is Forth?

Forth is a language for direct communication between human beings and machines. Using natural language diction and machine-oriented syntax, Forth provides an economical, productive environment for interactive compilation and execution of programs. Forth also provides low-level access to computer-controlled hardware, and the ability to extend the language itself. This extensibility allows the language to be quickly expanded and adapted to special needs and different hardware platforms.

Essential elements of the Forth programming language are standardized in ISO/IEC 15145:1997 or ANSI X3.215-1994 publications. Newer refinements are in progress by the Forth200x committee. Information and useful links are available at the web site https://forth-standard.org/.

What is MinForth?

The MinForth interpreter is a nearly complete implementation of ISO/ANS Standard Forth. It passes the Forth-2012 test suite and thus provides a reliable system to run own Forth software.

Building MinForth

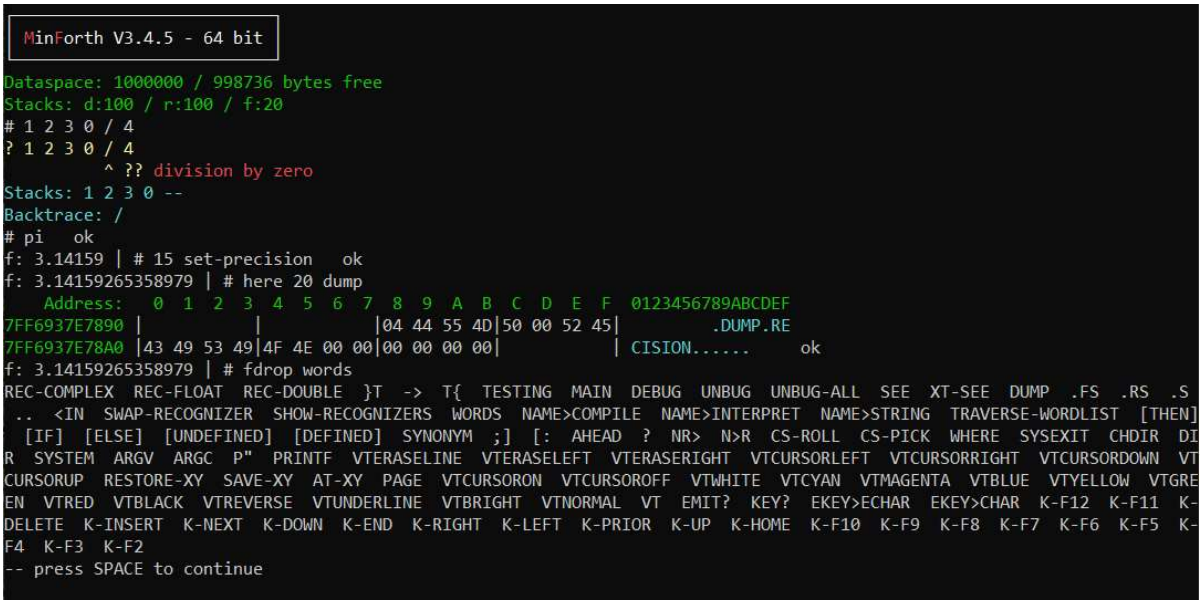
Its attribute "minimalistic" indicates that it requires very little development resources: a text editor and a plain C compiler are sufficient. MinForth can be compiled as 32-bit or 64-bit version alike on Windows and Linux operating systems.

MinForth source code is a seamless mixture of C and Forth definitions. The build process goes first through a unique Forth to C source code transpiler. In a second step it is compiled to fast machine code through the C compiler. In this way portability to other platforms and easy code maintenance is ensured.

Using MinForth

Main target application is the interactive MinForth interpreter which can compile and execute high-level Forth definitions without C compiler support.

Using it for highly interactive testing of numeric algorithms, embedding it into other C programs, or scripting in Forth language are other interesting application areas.



1 MinForth Design

1.1 Introduction

MinForth calls itself minimalistic. Some call it fat. Decide by yourself. Anyhow, by design MinForth has just minimal requirements. A normal C compiler and a text editor are sufficient to build MinForth from a Windows or Linux command line.

So the 'min' in MinForth means resource minimalism (MinForth itself is a rather complete although compact standard Forth system). Its historic roots ('closed source' and under a different name) reach back to early DOS times; its application field was remote diagnostics and service of measurement and control systems. MinForth V3.4 is a very modified version adapted to PCs operating with a Windows or Linux OS. A number of words and wordsets were added to follow modern Forth standards.

From a User point-of-view, MinForth provides a small Forth system very close to ISO/ANS Forth standard, even with most newer Forth-2012 functions like quotations. It is operated "à la classique" i.e. through command-line input. A graphical IDE is not provided.

From a developer or system adapter viewpoint, MinForth's speciality is a Forth-to-C transpiler (mf2c), a little C program that translates MinForth source text input to commented C source text output. The transpiled code can be compiled by any decent C99 compiler to executable machine code.

The most important feature is that Forth code and C code can be mixed seamlessly even within definitions, which brings great flexibility to software development f.ex. for being able to include external C libraries without hassle (the transpiler will pass C code just through). A good example is the optional arbitrary-length floating-point number word set, outlined in § 2.7.14.

Development cycles are fast. A complete rebuild takes only seconds (using the Tiny C compiler by the make32 or make64 scripts about 1 second on a modern desktop PC).

1.2 Quick Start

When you read this document, you probably have already downloaded a copy of all MinForth files. It is assumed that you know how to use the command shell of your OS, or at least a file manager.

Create a new minforth folder (name is not important, MF34 is recommended) somewhere in your home directory and copy all files into this new directory while preserving MinForth's file names and (sub)directory structure.

Move to the `./src` directory to check if your C compiler works. On Windows the `./src/tcc` directory should contain a copy of a 64-bit Tiny C compiler (if you are on a 32-bit OS vou'll need the 32-bit version, see §6.2.5). From the `./src/windows` directory, copy `make*.bat` and `tc*.bat` script files to the `./src` directory. On Linux you should have already gcc or clang installed.

From your OS command line
- on Windows enter `tc32 hello`
- on Linux enter `cc hello.c -o hello`

to compile the famous 'Hello World' program. If something goes wrong, read § 6.

To build MinForth on Windows execute `make32.bat` or `make64.bat`. Here you are:

```
e:\src>make64
Making mf2c.exe with tcc 64 bit

MinForth V3.4 to C Transpiler
-----------------------------
Reading Forth source file mf3.mfc
Including core.mfc
Including except.mfc
Including facility.mfc
Including double.mfc
Including float.mfc
Including complex.mfc
Including memory.mfc
Including search.mfc
Including locals.mfc
Including string.mfc
Including file.mfc
Including block.mfc
Including tools.mfc
Total definitions: 955
 - in dictionary:  874
 - therein hidden: 242
Writing C target file mf3.c
Making mf3.exe with tcc 64 bit
Renamed mf3.exe to mf64.exe

Ready to start mf64 and enjoy your new
MinForth V3.4 64-bit :-)

Press any key . . .
```

Similarly, to build and start MinForth on Linux, enter from your shell command-line:

```
~$  cc mf2c.c -o mf2c
~$  ./mf2c
~$  cc mf3.c -o mf3 -std=gnu99 -funsigned-char -lm
~$  ./mf3
```

or simply use the ./lmake script

```
~$  ./lmake mf2c
~$  ./mf2c
~$  ./lmake mf3
~$  ./mf3
```

1.3 Memory Model

MinForth uses real host memory addresses, no sandboxed virtual memory or neutralized (obfuscated) memory addresses. This allows direct co-operation with other C programs or libraries without address translation.

Functions or Forth words processed through the transpiler become machine code (primitives in Forth speak) that can be called by function pointers. MinForth execution tokens are function pointers in C. The Forth dictionary is a simple array or list of structures (headers in Forth speak) comprising link fields, flags, name fields and those function pointers (execution tokens).

The linked dictionary is continued in MinForth's dataspace for high-level words and data. New headers are created by MinForth's _HEADER word (in core.mfc). The dataspace is defined as a character array (mfdsp[] in file mf3.h) while stacks are defined as cell arrays. Forth stacks are only used by high-level words, primitives use C stack frames of course.

Memory access violations (segfaults) are caught by a signal handler (defined in `mf3.sys`).

1.4 Safety Aspects

In 'reasonable' circumstances MinForth should crash 'nicely' on errors, with some message about error cause and error location. To this end many C primitives can be compiled with runtime check code (`mferr` macro) to throw an exception e.g. when a stack underflow is detected.

To improve speed, runtime error checking can be switched off and back on by the MinForth commands `FAST>` and `SAFE>`. But benchmark tests showed relative speed improvements of only about 5%. Other hardware platforms might show better figures, but it demonstrates also that this additional comfortable safety harness comes relatively cheap and thus should remain enabled.

To disable runtime error checking permanently, set the compiler flag `MFRTCHECK` within `mf3.h` to 0 and recompile MinForth without runtime error checking code. This might also be useful when MinForth is embedded within another host application that brings its own safety harness.

1.5 64-bit Applications

MinForth can be compiled as 32-bit or 64-bit program version from the same codebase. In the 64-bit version, 128-bit integer arithmetic for Forth mixed/double numbers is provided. Floating-point numbers occupy 64 bits in either version, following the IEEE 754 standard.

2 MinForth System

2.1 Overview

This chapter concentrates on the MinForth command-line interpreter, the "system". A Forth system is an interactive programming environment for Forth application programs. Forth systems are often rather compact and offer program execution, compilation and testing in one shell for an efficient seamless development workflow.

This document does not provide basic Forth information or explanations of standard Forth words. Those can be extracted from the enclosed Forth-2012 draft standard proposal (in `./doc` directory), or found online along with lots of additional information at https://forth-standard.org/

2.2 Running a MinForth Session

Start a MinForth session by executing `mf3.exe` (or `./mf3` on Linux). A terminal will appear looking somewhat like

```
┌──────────────────┐
│ MinForth V3.4 - 64 bit │
└──────────────────┘
Dataspace: 1000000 / 999076 unused
Stacks: d:100 / r:100 / f:20
198816 bytes free
#
```

MinForth uses ANSI terminal command sequence for text colours and cursor control. If this is not available in your system and if you see garbled letters on your screen, try with VT OFF.

Try some keyboard inputs and MinForth will give you immediate feedback if it can accept your inputs or not. Note the automatic stack diagram in the system prompt. F.ex. enter `hex` (or `HEX`,) the decimal `#` prompt will change to a `$` hex prompt.

You can enter Forth words in upper or lower case; word names are automatically converted to upper case by default. To change this automatism, enter `CAPS OFF`.

If something goes wrong, MinForth shows the actual source line, points out the exception causing word and tries to analyse stack contents and backtrace nested high-level words. With `WHERE` you can redisplay the last backtrace.

With `DIR` you can inspect the files in your working directory; use `CHDIR <NAME>` to change your working directory of your current session.

To execute OS commands use `SYSTEM ( ac uc -- )` by command string (ac uc) including parameters.

```
┌──────────────────┐
│ MinForth V3.4 - 64 bit │
└──────────────────┘
Dataspace: 1000000 / 999076 unused
Stacks: d:100 / r:100 / f:20
# s" uname -a" system
Linux localhost.localdomain 5.3.18-lp152.60-default #1 SMP Tue Jan 12 23:10:31 UTC
2021 (9898712) x86_64 x86_64 x86_64 GNU/Linux
 ok
#
```

With the Facility wordset included, typical command-line editing functions are provided, including history for the last 8 keyboard entries. The last 8 lines can be recalled with cursor up/down keys.

When the session is closed by the word `BYE` the history is copied to file `mfhist.blk`.

2.3 System Command-Line Arguments

With the Facility wordset included, the following system command-line arguments are available which are executed during MinForth start:

`/e "<words>"`	evaluate given words at start, e.g. `mf3 /e "cr s\" hello\" type"`
`/d <size>`	resize data stack, e.g. `mf3 /d 200`
`/f <size>`	resize floating-point number stack, e.g. `mf3 /f 100`
`/r <size>`	resize return stack, e.g. `mf3 /r 200`
`/i <filename>`	evaluate given file line-by-line at start, e.g. `mf3 /i mft.mf` (for more flexibility use: `/e "include <filename>"`)
`/nl`	do not show the logo at start-up

(above flags start with a / slash character to leave the – dash character to application programs)

When MinForth finds a file `autoexec.mf` in your current work directory, it will be evaluated automatically during program start (useful for project-specific settings).

2.4 Low-Level and High-Level Internals

A MinForth system is a MinForth program built from ground up with little Forth words written in mixed C and Forth definitions.

For example the standard Forth word FM/MOD is defined as

```
: FM/MOD  \ ( d n -- mod quot ) mixed floored division
C  mfCell n=mftos, s=mftos^mfsec;
   sm/rem
C  if (mfsec&&(s<0)) mftos-=1,mfsec+=n; ;
```

which is transpiled to get the equivalent plain C code

```
void mf66F1FDC7(void) { // : FM/MOD
  mfCell n=mftos, s=mftos^mfsec;
  mf1705E412(); // SM/REM
  if (mfsec&&(s<0)) mftos-=1,mfsec+=n;
}
```

to be compiled by a C99 compiler (see § 6.2) to become the executable primitive integer division word. One can have as many primitives as desired or required.

The MinForth system also provides a Forth compiler for new ‘high-level’ words defined within the system, without needing the C compiler any more. Such ‘high-level’ compiled words build upon ‘low-level’ primitive words, or on other high-level words.

The high-level compiler translates them to address pointer lists which can be executed through the ‘:’ colon runtime function `_[:]` that is the workhorse of nearly all MinForth hi-level words. It is defined a classical virtual machine for direct-threaded code:

```
: _[:]  \ ( -- .. ) nest into hilevel definition
C  mfrpush(mfip); mfip=(mfXT**)mfw+1;
C  while(mfw=*mfip++,mfw) (*mfw)(); // MinForth hilevel VM
C  mfip=(mfXT**)mfrpop(); ;
```

So in Virtual Machine terminology: the MinForth system is subroutine threaded, whereas high-level definitions are direct threaded.

That's the general concept. Finer details 😉 are described within MinForth's *.mfc source files.

Side note: when you browse through MinForth source files, you'll observe that it is written in a very tight horizontal style, and not easily legible. This originates from early times and working with tiny field monitors and has continued since then; but also because I prefer it this way personally. But be my guest and add as many line feeds and tabs as you like to get to a more modern vertical style.

2.5 Building a new MinForth System

The development system of MinForth is radically minimal: no IDE, no autotools or makefiles, no libraries. Instead: very small scripts to call a C compiler as shown in § 1.2

The first step is to call the mf2c transpiler with the following files in its path:

- `mf3.mfc` build instructions
- `mf3.h` data space, stacks and type declaration, useful macros (syntactic sugar)
- `mf3.sys` machine-specific and some general routinies
- `*.mfc` all source files mentioned in mf3.mfc

A typical build instruction in file `mf3.mfc` looks like (your actual distribution may look slightly different):

```
\ ------ MinForth Wordsets ---------------------------------------------------

#INCLUDE core.mfc          \ MINFORTH kernel
\ optional:
#INCLUDE except.mfc        \ EXCEPTION wordset
#INCLUDE facility.mfc      \ FACILITY wordset
#INCLUDE double.mfc        \ DOUBLE-NUMBER wordset
#INCLUDE float.mfc         \ FLOATING-POINT NUMBER wordset
#INCLUDE complex.mfc       \ COMPLEX-NUMBER wordset
#INCLUDE memory.mfc        \ MEMORY-ALLOCATION wordset
#INCLUDE search.mfc        \ SEARCH-ORDER wordset
#INCLUDE locals.mfc        \ LOCALS wordset
#INCLUDE string.mfc        \ STRING wordset
#INCLUDE file.mfc          \ FILE-ACCESS wordset
#INCLUDE block.mfc         \ BLOCK wordset
#INCLUDE tools.mfc         \ PROGRAMMING-TOOLS wordset

\ ------ MinForth System Start -----------------------------------------------

: MAIN     \ ( -- ) MF Hilevel Forth system, called by C main() function
   _boot   \ initialize system
   _logo   \ startup banner
   _abort  \ initialize and start Forth text interpreter
   bye ;   \ clean system before program end
```

So MinForth comprises all standard Forth wordsets except the XCHAR Extended Characters wordset. Being a C-based Forth system, some few words from the Programming-Tools Wordset like CODE..;CODE are also not implemented.

On the other hand MinForth provides many additional words and functions that are useful for programming in Forth, without becoming too fat.

With those instructions `mf2c` builds a C file `mf3.c`. The second step is to compile the final executable MinForth system `mf3.exe` (or `./mf3` in Linux).

2.6 MinForth Kernel Words

The standard Forth Core Wordset alone does not make a usable system because it does not (and of course can not) reflect target-specific hardware and software environments, like presence or absence of an underlying operation system.

Therefore the MinForth core system consists of

- the standard core and extended core wordset
- additional words to make a small runnable stand-alone program: the MinForth kernel.

A number of words are deferred to enable added functionality through including additional wordsets during the transpiler phase, e.g. more comfortable exception handling or local values.

If you want to experiment with a minimal Forth kernel (in the spirit of eforth) the `./ecore` directory provides the necessary sources.

2.6.1 Core Wordset

All standard words as of ISO/ANS standard document §6 are provided within source file `./src/core.mfc`.

From user perspective, important additional words are:

`-ROT ( a b c -- c a b)`
rotate top three stack elements anti-clockwise

`PLUCK ( a b c -- a b c a )`
copy third over top stack element

`RDROP ( r: x1 x0 – x1 ) 2RDROP ( r: x2 x1 x0 – x2 )`
drop top / top 2 return stack element(s)

`RPICK r( n |r: xm..x0 – xn |r: xm..x0 )`
copy n-th return stack element from return stack to data stack

`NOT ( f -- !f )`
logic not, identical to 0=

`ROL ROR ( n p -- nr )`
bit-wise left/right rotation by p bit-positions

`0<= 0>= <= >= U<= U>= ( n -- f )`
common logic comparison operators

`ON OFF ( a -- )`
store true/false flag at address a (used on logic variables)

`INCR DECR ( a -- )`
increment/decrement cell at address a (used on counter variables)

`CHAR/ ( a u -- a+1 u-1 )`
ignore first string character

`STR= ( a1 u1 a2 u2 -- f )`
compare two strings, true if both have identical length and content

`CELL ( -- u )`
size of one cell in bytes (4 on 32-bit, 8 on 64-bit machines)

`CELL- ( a -- a-u )`
reduce address by size of one cell

`,. U,. ( n -- ) ,.R U,.R ( n w -- )`
numeric output with ,-separator between every 3 decimal digits, e.g. `1000000 .. 1,000,000`

`DEC. ( n -- )`
display number in decimal notation, regardless of current `BASE`

`HEX. ( n -- )`
display number with `$` in unsigned hexadecimal notation, regardless of current `BASE`

`CAPS ( -- a )`
variable when set controls upper case conversion for WORD, ENVIRONMENT? (and SEARCH-WORDLIST in the Search Wordset)

`NAMED ( a u -- )`
create next word with given string as name, e.g. `S" RTCflag" NAMED VARIABLE`

`COMPILE-ONLY ( -- )`
flag latest word as 'compile-only'

`RECOGNIZER ( rec-xt exec-xt comp-xt <name> -- )`
append new recognizer to recognizer chain (see § 2.7.12.2)

`+TO ( x <name> -- )`
add x to `VALUE NAME`

`.\" ( string"-- )`
display extended string, equivalent to `S\" xxx" TYPE`

`BOUNDS ( a u -- end start )`
convert address range a u for following `DO` or `?DO`

`FAST> SAFE> ( -- )`
disable or re-enable runtime error checking

2.6.1.1 Hidden Core Words

Many MinForth primitive words are hidden from normal dictionary search to reduce clutter of wordlists.

Word names starting with a single _underscore character are just hidden within the dictionary. But they can still be made visible with the commands

`HIDE> UNHIDE> ( -- )`
make hidden words visible and invisible again

`HIDDEN ( -- )`
unhide only next word

```
┌──────────────────────┐
│ MinForth V3.4 - 64 bit │
└──────────────────────┘
198816 bytes free
# 1 2 3 4 5 6 7 8   ok
1 2 3 4 5 6 7 8 # unhide> 0 _sp! hide>   ok
#
```

Primitive words starting with two or more __underscore characters are not mapped by the transpiler into the dictionary at all. These words become simple C functions.

High-level words do not follow this scheme. But one can use the words _REVEAL and _HIDE to take the latest compiled word in or out of the current compilation wordlist.

2.6.2 Literal Recognizers

The text interpreter is extendable to process customized literal (or word) classes.

`RECOGNIZER ( rec-xt exec-xt comp-xt <name> -- )`
defines a new named recognizer and appends it to the recognizer list, with
`rec-xt` execution token of pertaining literal checker
`exec-xt` execution token of the recognized literal, mostly NOOP
`comp-xt` compilation token of the recognized literal

Example: `' >FLOAT ' NOOP ' FLITERAL RECOGNIZE REC-FLOAT`
(which is actually included in the boot sequence of the fp-number wordset)

The literal checker must follow the stack diagram `( a u -- n..x true | false )` with
`a u` token string provided by the text interpreter
`n..x` any number of stack elements provided as recognized

The programming tools wordset provides some additional words to support literal checker development, see § 2.7.12.2.

MinForth's recognizer scheme is very simple. A more elaborated and more comfortable recognizer proposal is pending in the Forth Standard committee.

2.7 Optional MinForth Wordsets

In any standard Forth system at least the standard Core Wordset shall be present. Other wordsets are optional. MinForth follows this rule.

2.7.1 Exception Wordset

All standard words as of ISO/ANS standard document §9 are provided within source file `./src/except.mfc`.

The Exception Wordset is the 'least optional' of all wordsets because even core words can throw exceptions. Therefore CATCH and THROW are already prepared as deferred words within the kernel.

MinForth exception frames reside within the C frame stack. Throwing a high-level exception through the Forth word THROW unwinds the Forth stack as well as the C frame stack through C's setjmp/ longjmp mechanism.

ABORT is organized accordingly through longjumping to _ABORT from within THROW (in core.mfc).

When the Exception Wordset is included, most throw codes are translated to plain-text exception messages. More detailed error information is provided through display of the source line, indicating the exception causing word, stack content(s) and a tentative backtrace to callees (the calling high-level words). On exceptions up to 6 stack values are remembered within the internal structure mfxcs[].

```
┌──────────────────┐
│ MinForth V3.4 - 32 bit │
└──────────────────┘
19999172 bytes free
# : T3 rot ; : T2 3 t3 ; : t1 2 t2 ;   ok
# 1 >r t1
? 1 >r t1
        ^ ?? stack underflow
Stacks: r: $1 $0 ^ ^ | #2 #3 --
Backtrace: ROT <- T3:1 <- T2:3
#
# where  -> stack underflow
Stacks: r: $1 $0 $30774C $307724 | #2 #3 --
Backtrace: ROT <- T3:1 <- T2:3 ok
#
```

Carets ^ in the return stack mean return addresses to the callees displayed within the backtrace. Otherwise the backtrace would show only long cryptic address numbers.

The number separated by a colon : from the word name in the backtrace indicates the calling position within the caller's definition. In the example T2:3 means that T2's 3rd cell contains T3's execution token, which lead to the stack underflow later i.e. ROT found only 2 stack cells and couldn't operate.

With WHERE you can re-display the last backtrace. The ^ carets are now resolved to their real addresses withn the return stack at the time of the last exception.

2.7.2 Double-Number Wordset

All standard words as of ISO/ANS standard document §8 are provided within source file ./src/double.mfc. In a 32-bit system double numbers represent 64-bit integers, in a 64-bit system they represent 128-bit integers.

Additionally the following words are provided:

`2NIP ( d1 d2 – d2 )  2TUCK ( d1 d2 – d2 d1 d2 )`
like standard NIP and TUCK but for cell pairs or double numbers

`D<> D> D<= D>= DU> DU<= DU>= ( d1 d2 -- flag )`
more double number comparison operators

`DROL DROR  ( d p -- dr )`
double bit-wise left/right rotation by p bit-positions

`D* ( d1 d2 -- dprod )`
double number multiplication d1 * d2 (all 128 bits in 64-bit systems)

`D/ ( d1 d2 -- dquot )`
double number division d1 / d2 (all 128 bits in 64-bit systems)

`D/REM ( d1 d2 -- drem dquot )`
double number division d1 / d2 with symmetric remainder (all 128 bits in 64-bit systems)

`D/MOD ( d1 d2 -- dmod dquot )`
double number division d1 / d2 with floored remainder (all 128 bits in 64-bit systems)

`UD. ( ud -- ) UD.R ( ud w -- )`
print unsigned double number (UD.R right-aligned in w-wide field)

`D,. UD,. ( d -- ) D,.R UD,.R ( d w -- )`
numeric output with ,-separator between every three decimal digits

`+TO ( d <name> -- )`
adds to `2VALUE NAME`

When the Double Number Wordset is included before the Floating-Point Number Wordset:

`DPL ( -- a )`
decimal point location variable to support fixed point arithmetic applications

```
┌──────────────────────┐
│ MinForth V3.4.5 - 64 bit │
└──────────────────────┘
Dataspace: 1000000 / 998736 unused
Stacks: d:100 / r:100 / f:20
# -1. hex 30 ud.r                  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF ok
$ #10.   ok
A 0 $ decimal 1.2345 dpl ?  4  ok
10 0 12345 0 # 2drop 2drop   ok
# \ input a 37 digit number   ok
# 1.23456789012345678901234567890123456 dpl ?  36  ok
3259073885544336263 66926059427634869 # 10. d/   ok
-1518767018816521536 6692605942763486 # d.  123456789012345678901234567890123456  ok
```

Traditionally Forth recognizes numbers that are directly followed by a trailing dot as double numbers, not as floating-point numbers as might be expected by new users. Recognition of double numbers is implemented by appending a recognizer during MinForth's boot process.

However, when the Double-Number Wordset is included after the Floating-Point Number Wordset, numbers with a trailing single dot are recognized not as double but as floating-point numbers, similar to other programming languages.

When the Programming Tools Wordset is included this can be changed and verified by the commands `SWAP-RECOGNIZERS` and `SHOW-RECOGNIZERS` (see § 2.7.12.2).

When the Locals Wordset is included (see § 2.7.5) many double-number operations can be simplified by using MinForth's double number locals with the `D:` prefix.

2.7.3 Floating-Point Number Wordset

All standard words as of ISO/ANS standard document §12 are provided within source file `./src/float.mfc`.

MinForth uses C double numbers (64-bit IEEE 754 fp-numbers) on a separate floating-point stack.
When the fp-number wordset is included, the `QUIT` prompt is extended to display the fp-stack content.

Additionally provided words:

`FDEPTH ( f: fn..f1 -- | d: -- n )`
actual fp-numbers in fp-number stack

`FPICK ( f: fm..f0 -- fm..f0 fi | d: i -- )`
pick i-the fp-number from within fp-number stack

`FNIP ( f: f1 f2 – f2 )  FTUCK ( f: f1 f2 – f2 f1 f2 )`
like standard `NIP` and `TUCK` but for fp-numbers

`F>R ( f: f -- | r: -- f )  FR@ ( f: -- f | r: f -- f )`
`FR> ( f: -- f | r: f -- )`
like standard `>R R@ R>` but for fp-numbers

`FINV 1/F ( f: f -- 1./f )`
inverse fp-number

`FSQR ( f: f – f^2 )`
square fp-number

`F2* ( f: f -- f*2. )  F2/ ( f: f -- f/2. )`
like standard `2* 2/` but for fp-numbers

`FMA ( f: a b c -- (a+b*c) )  F2/ ( f: f -- f/2. )`
quick multiply-add (Horner scheme)

`F0> F0<> F0>= F0<= ( f: f -- | d: -- flag )`
`F> F>= F<= F= F<>  ( f: f1 f2 -- | d: -- flag )`
more comparison operators (beware of rounding effects, for precise identity check use `F~`)

`ISNAN ( f: f – | d: -- -1|0|+1 )`
signed ‘not-a-number’ check

`ISINF ( f: f – | d: -- -1|0|+1 )`
signed infinity check

`FSIGN ( f: f – | d: -- -1|0 )`
sign function

`RAD>DEG ( f: r -- d ) DEG>RAD ( f: d -- r )`
conversion between radians and degrees, useful for trigonometric calculation

`FHYPOT ( f: a b -- sqrt(a^2+b^2) )`
hypotenuse or complex magnitude

`PI  EULER  FSQRT2 ( f: -- x )`
common fp-number constants

`G. GS. GE. ( f: f -- )`
print fp-numbers like their corresponding standard words `F. FS. FS.` but without trailing 0s.

`F, ( f: f -- )`
compile fp-number into dataspace

`+TO ( f: f <name> -- )`
adds f to `FVALUE NAME` (or to fp locals)

When the Double-Number Wordset is included before the Floating-Point Number Wordset, numbers with a trailing single dot are recognized as double integer numbers. This can be changed by changing the recognizer order (see § 2.7.12.2).

```
┌──────────────────┐
│ MinForth V3.4 - 64 bit │
└──────────────────┘
19998912 bytes free
# 1.   ok
1 0 # 2drop swap-recognizer rec-float   ok
# 1.   ok
f: 1. | #
#
```

2.7.4 Complex Number Wordset

Some popular operations and functions with complex numbers are provided within source file `./src/complex.mfc`. Complex numbers are not specified by standard Forth. MinForth complex numbers are fp-number pairs, their imaginary part occupying the higher fp-stack position.

MinForths complex trigonometric and hyperbolic functions take branch cuts into account and "behave well" in complex operations.

Recognition of complex numbers in `<re>+/-j<im>` syntax is implemented by appending a recognizer during MinForth's boot process.

```
┌──────────────────┐
│ MinForth V3.4 - 32 bit │
└──────────────────┘
199116 bytes free
# 2+j3   ok
f: 2. 3. | # zsqr   ok
f: -5. 12. | # z.  -5.+j12.  ok
#
```

Provided words:

```
ZDROP   ( f: z -- )
ZSWAP   ( f: z1 z2 -- z2 z1 )
ZNIP    ( f: z1 z2 -- z2 )
ZROT    ( f: z1 z2 z3 -- z2 z3 z1 )
ZDUP    ( f: z -- z z )
ZOVER   ( f: z1 z2 -- z1 z2 z1 )
ZTUCK   \ ( f: z1 z2 -- z2 z1 z2 )
```

stack operation words like their counterparts for other numeric types

```
Z>R   ( f: z -- | r: -- z )
ZR@   ( r: z -- z | f: -- z )
ZR>   ( r: z -- | f: -- z )
```

move to / read / fetch complex number to / from return stack

```
COMPLEX   ( -- u )   COMPLEXES   ( n -- u )
```

storage size in bytes for 1 or n complex numbers

```
Z!    ( f: re im -- | d: adr -- )
Z@    ( f: -- re im | d: adr -- )
```

store / read fp-number pair or complex number at / from address

```
Z0=  ( f: z1 z2 -- d: f )
Z=   ( f: z1 z2 -- d: f )
Z~   ( f: z1 z2 r3 -- flag )
```
comparison operators like their counterparts for fp-numbers

```
ZREAL  ( f: z -- re )
ZIMAG  ( f: z -- im )
```
real / imaginary part of complex number

```
ZCONJ  ( f: z – z' )
```
conjugate complex number

```
ZABS   ( f: z -- |z| )
ZARG   ( f: z – arg )
```
absolute value (modulus) / argument (angle) of complex number

```
Z>POLAR     ( f: re im -- |z| angle )
Z>CARTESIAN ( f: |z| angle -- re im )
```
convert between polar and cartesian representation of complex numbers

```
Z+  ( f: z1 z2 -- zsum )
Z-  ( f: z1 z2 -- zdiff )
Z*  ( f: z1 z2 -- zprod )
Z/  ( f: z1 z2 -- zquot )
Z** ( f: z1 z2 -- power )
```
add / subtract / multiply / divide / power of two complex numbers

```
ZSCALE  ( f: z f -- zscaled )
Z2*     ( f: z -- z*2 )
Z2/     ( f: z -- z*2 )
ZNEGATE ( f: z -- -z )
1/Z     ( f: z -- 1/z )
```
scale / double / halve / negate / invert a complex number

```
ZSQR    ( f: z -- z^2 )
ZSQRT   ( f: z -- sqrt(z) )
ZEXP    ( f: z -- exp(z) )
ZLN     ( f: z -- ln(z) )
```
complex square / square root / exponential / natural logarithm functions

```
ZSIN  ( f: z -- sin(z) )    ZASIN  ( f: z -- asin(z) )
ZCOS  ( f: z -- cos(z) )    ZACOS  ( f: z -- acos(z) )
ZTAN  ( f: z -- tan(z) )    ZATAN  ( f: z -- atan(z) )
```
complex trigonometric and their inverse functions

```
ZSINH  ( f: z -- sinh(z) )    ZASINH  ( f: z -- asinh(z) )
ZCOSH  ( f: z -- cosh(z) )    ZACOSH  ( f: z -- acosh(z) )
ZTANH  ( f: z -- tanh(z) )    ZATANH  ( f: z -- atanh(z) )
```
complex hyperbolic and their inverse functions

```
Z.  ( f: z -- )
```
display complex number in `<re>+/-j<im>` syntax

```
>COMPLEX  ( a u -- f, true: r: -- fr fi )
```
try to convert string in `<re>+/-j<im>` syntax to complex number (like its counterpart `>FLOAT`)

```
ZLITERAL  ( f: z -- )
```
compile complex number into dataspace

`ZCONSTANT ( ct: z <name> -- | rt: -- z )`
complex `CONSTANT`

`ZVARIABLE ( ct: z <name> -- | rt: -- adr )`
complex `VARIABLE`

`ZVALUE ( ct: z <name> -- | rt: -- z )`
complex `VALUE`

`TO +TO ( f: z <name> -- )`
store / add to `FVALUE`

When the Locals Wordset is included (see § 2.7.5) many complex number operations can be simplified by using MinForth's complex number locals with the `Z:` prefix.

Syntactic Considerations

In textbooks and some popular math software, complex numbers are presented in +/-re>+/-<im>i notation, f.ex. as `z=2+3i`. While this looks pleasing to the eye at first, it becomes quickly annoying in real-world technical computations with longish real and imaginary parts like in `-12.9602E-3-0.2301E-12i`. The eye has to parse the string. It is easier to separate in <re>+/-j<im> notation like `-12.9602E-3-j0.2301E-12`. Here the substrings `+j` or `-j` serve as unambiguous separation markers between real and imaginary parts. The letter `j` (instead if `i`) is used, because `I` is already frequently used in many technical applications e.g. to denote electrical current.

2.7.5 Locals Wordset

All standard words as of ISO/ANS standard document §13 are provided within source file `./src/locals.mfc`.

Three syntax options for locals declarations are available:
- standard `{: .. :}` syntax
- legacy Forth-94 `LOCAL| .. |` syntax
- the widespread `{ .. }` syntax

In addition floating-point, complex, and double number locals are supported by type annotation prefixes with {: .. :} and { .. } syntax:

`F: NAME`
declare a floating-point number local

`D: NAME`
declare a double-number (double integer) local

`Z: NAME`
declare a double floating-point number (complex) local

Local type annotations are persistent until they are reset by

`N: NAME`
declare a normal (integer) local

`TO +TO ( x <name> -- )`
store / add to local `NAME` of any type

Example:

```
: RLC-Impedance  {: f: R L C freq | omega -- Zre Zim :} \ serial RLC resistance
   2e pi f* freq f* to omega     \ polar frequency
   R                             \ Zre on stack
   omega L f* omega C f* 1/f f-  \ Zim on stack
;
```

2.7.5.1 Output Locals

MinForth provides a handy extension to standard locals. It enhances the standard -- syntax (that only starts a comment) by a novel == syntax that indicates output locals. Output locals are put automatically on the stack before a word ends. Be aware that this does not happen when a word is prematurely left through EXIT.

Output locals can significantly reduce stack juggling and typing (and mistakes) for words with a large number of parameters i.e. words that cannot factored much otherwise.

== (--)
   mark all following locals declarations as output locals (unless there is already an input local with the same name) thereby replacing the standard -- end-of-locals separator

:= += (x <name> --)
   assign / add to local NAME of any type (:= borrowed from Pascal, += from C)

Examples:

```
┌──────────────────┐
│ MinForth V3.4.6 - 64 bit │
└──────────────────┘
998736 bytes free
# : Z-ROT { z: a b c == c a b } ; \ complex -ROT  ok
# 1+j 2+j2 3+j3 z-rot   ok
f: 3. 3. 1. 1. 2. 2. | # ..  ok
# : MYOVER { a b == a b a } ;  ok
# 4 5 myover  ok
# 4 5 4 #
```

In this first example the notation looks already very much like a typical Forth stack diagram. The compiler does the rest without much (no) ado.

The next example demonstrates more possibilities.

```
\ Complex division, flag true if |quot|<1

: ZDIVLESSUNIT { f: xr xi yr yi | r == qr qi | n: f }
   yr yr f*  yi yi f*  f+        := r
   xr yr f*  xi yi f*  f+  r f/  := qr
   xi yr f*  xr yi f*  f-  r f/  := qi
   qr qi fhypot  1e f<           := f  ;

# 1e 2e 3e -4e zdivlessunit  ok
-0.2 0.4 | -1 #
```

Note that local declarations, incl. type annotation, are also possible with output locals.

For better readability the word := can act as placeholder for TO. It is generally recommended to use TO or +TO for global values and := or += for locals.

P.S. a shorter solution, just for fun:

```
: ZDIVLESSUNIT3 { z: x y == q | n: f }
   x y z/ := q  q zabs 1e f< := f  ;
```

2.7.6 String Wordset

All standard words as of ISO/ANS standard document §17 are provided within source file `./src/string.mfc`.

Internally `mf2c` compiles most strings with a trailing zero character to ease co-operation with other C programs.

Additionally provided:

`PARSE-STRING  ( a u -- a' u' ap up )`
parse string (a u) for blank-delimited string-token (ap up), leaving (a' u') as remaining unparsed string; ap is the address of the first character following the string token; if either u' or up are zero, no string-token was found

2.7.7 Memory-Allocation Wordset

All standard words as of ISO/ANS standard document §14 are provided within source file `./src/memory.mfc`.

MinForth's dataspace is defined in `./src/mf3.h` as a static character array of constant size `MFDSPSIZE`. This size might not be sufficient for some programs e.g. if they `ALLOT` a lot. Then one can either `ALLOCATE` external memory instead, or increase `MFDSPSIZE` and recompile MinForth.

Stack sizes can also be adapted through command-line arguments (see § 2,3)

2.7.8 Search-Order Wordset

All standard words as of ISO/ANS standard document §16 are provided within source file `./src/search.mfc`.

MinForth can hold up to 10 wordlists (in the WORDLISTS array), and search up to 8 wordlists in the search order (in the CONTEXT array.

MinForth provides also classic vocabularies like in previous Forth standards.

```
MinForth V31 - 64 bit
4193616 bytes free
# vocs
WORDLISTS:  FORTH  ROOT  FILES ok
# order
CONTEXT:  FORTH
CURRENT:  FORTH ok
# vocabulary myvoc  ok
# also myvoc definitions  ok
```

```
# order
CONTEXT:  MYVOC  FORTH
CURRENT:  MYVOC ok
# previous definitions  ok
# order
CONTEXT:  FORTH
CURRENT:  FORTH ok
```

Provided default wordlists (vocabularies):

FORTH	comprises all MinForth system words
ROOT	contains the minimum standard wordset
FILES	contains included file names in file-access wordset

2.7.9 File-AccessWordset

All standard words as of ISO/ANS standard document §11 are provided within source file `./src/file.mfc`.

Additionally provided words:

`STDIN STDOUT STDERR ( -- fid )`
standard terminal stream IDs

`EXIST-FILE ( a u – false|1|-1 )`
check if named file exists, returns 1/-1 when it is readable/writable

`FILES ( -- )`
vocabulary (see SEARCH wordset) holding all file names as MARKERs, so executing a file name eliminates all later included or defined words.

`FORGET-INCLUDED ( -- a )`
when flag variable is set, reset dataspace after file include error and thereby eliminate all compiled new words

2.7.10 Blocks Wordset

All standard words as of ISO/ANS standard document §7 are provided within source file `./src/block.mfc`.

MinForth implements block mass-storage through binary block-files. If no block-file is selected, a default block-file `mfblock.blk` is opened or created if it does not exist.

2.7.10.1 Block-File Management

MinForth provides many words for comfortable block-file handling. F.ex. to work with old fig-Forth source files:

```
MinForth V3.4 - 64 bit
198816 bytes free
# s" FORTH.SCR" open-blockfile   ok
# 10 list
--- BLOCK # 10 of 39 in forth.scr
 1: ( "Starting Forth" line editor - load screen )                  :
 2: ( FD 3:80 ) CR ." Editor loading, please wait..."               :
 3: 1 WARNING !                                                     :
 4: FORTH DEFINITIONS HEX                                           :
 5: : TEXT  HERE C/L 1+ BLANKS WORD HERE PAD C/L 1+ CMOVE ;         :
 6: : LINE  DUP 0FFF0 AND IF 17 MESSAGE DROP QUIT ENDIF             :
 7:         SCR @ (LINE) DROP ;                                     :
 8:                                                                 :
 9: VOCABULARY EDITOR IMMEDIATE HEX                                 :
10:                                                                 :
11: : WHERE  DUP B/SCR / DUP SCR ! ." Scr " DECIMAL .               :
12:          SWAP C/L /MOD C/L * ROT BLOCK + CR C/L TYPE            :
13:          CR HERE C@ - SPACES 5E EMIT [COMPILE] EDITOR QUIT ;    :
14:                                                                 :
15:                                                                 :
16: -->                                                             : ok
#
```

Provided words are:

`OPEN-BLOCKFILE  ( a u -- )`
Open file named (a u) in binary mode and use it as current block-file

`CREATE-BLOCKFILE  ( a u -- )`
Create new file named (a u) in binary mode, fill it with 40 empty blocks (16 line x 64 columns pe block) and use it as current block-file

`CLOSE-BLOCKFILE  ( a u -- )`
Write current block-file to disk and close it

`BLOCKFILE-NAME  ( -- a u )`
get name of current block-file

`BLOCKFILE-SIZE  ( -- u )`
get number of blocks in current block-file

`RESIZE-BLOCKFILE  ( u -- )`
add or cut blocks to/from current block-file to hold u blocks at the end

`BLOCK-READ  ( adr u -- )`
read block u to buffer at adr (buffer must be able to hold 1024 bytes)

`BLOCK-WRITE  ( adr u -- )`
write buffer at adr to block u (buffer must hold 1024 bytes)

`BLOCK-FILL  ( u c -- )`
fill block u with 1024 characters c

2.7.10.2 Block Screen Editor

MinForth provides a comfortable screen text editor to edit individual blocks. It is available when the optional source file `bledit.mfc` had been included in the build.

Use the keyboard and cursor keys to enter text and to move the cursor. Press `SHIFT-F1` to display a help screen,

The block-editor keeps only the actual screen in memory, leaving the screen with update flag set will flush the screen to the block-file. A separate backup block-file is not created automatically.

Example:

Edit the block-file HUFFMAN.BLK (from old F83).

```
# s" HUFFMAN.BLK" open-blockfile  ok
# 1 block-edit  ok

MinForth BLOCK-Editor          shF1:Help  ^r:reRead  ^s:Save  ^x:eXit
--- BLOCK # 1 of 42 in HUFFMAN.BLK
u1: \ Load Screen for Huffman Encoding/Decoding          29MAY84HHL :
 2:                                                                 :
 3: 3 20 THRU   FORTH                                              <:
 4: CR .( Huffman Utility Loaded )      EXIT                        :
 5:                                                                 :
 6:                        USAGE                                    :
 7: To compress a file type:                                        :
 8:    COMPRESS INFILE1.EXT OUTFILE1.EXT                           :
 9: To expand a file type:                                          :
10:    EXPAND   INFILE2.EXT OUTFILE2.EXT                           :
11:                                                                :
12: Where INFILE2.EXT had better be the OUTFILE1.EXT of a prior    :
13: compression.  After either a COMPRESS or EXPAND executing      :
14: EMPTY will reset the dictionary back to its original state.    :
15:                                                                :
16:                                                                :
      ^
```

You can move the cursor around with the arrow keys of your keyboard and type/edit words on the screen, very similar to a conventional text editor. Shift-F1 will show an overlay screen with helpful information about editor key settings.

When you change screen content, a small `u` appears in the upper left corner of the screen. This flag indicates that the block buffer has been updated, but has not yet been flushed to disk. To undo changes press ctrl-R, to save changes to the block-file press ctrl-S. Thereafter the update flag will disappear.

Note: When MinForth is used within a Linux terminal (there are many), some (function) keys or some alt/ctrl+key combinations may be reserved for terminal control. If this collides with your block editor control keys, adapt the key settings in source file `./src/bledit.mfc` in word `BLOCK-EDIT` and rebuild MinForth. Don't forget to update the help screen in word `_BEHELP` as well.

2.7.11 Facility Wordset

All standard words as of ISO/ANS standard document §10 are provided within source file `./src/facility.mfc`.

Additionaly provided words

`TIMER-RESET  ( -- )`
(re)set internal timer variable for execution timing

`TIMER-STOP  ( -- )`
calculate internal time difference since last TIMER-RESET

`.ELAPSED ( -- )`
(stop timer if not already stopped) display elapsed time difference

`RANDOM ( -- n )`
generate cell-wide pseudo random number from time seed

`WEEKDAY ( -- n )`
identify current weekday from 0=sunday to 6=saturdaa

`DAYLIGHTSAVING ( -- flag )`
identify daylight saving calendar state

MinForth provides code for VT100 terminal escape control sequences. This can be dis-/enabled through the `VT` virtual terminal flag. When you start MinForth and see many garbled letters on your screen instead of the square logo, your terminal does not VT100 escape sequences. Then put the command `VT OFF` somewhere in your `autoexec.mf` file and restart.

VT100 sequences are also used to set text cursor attributes and for command line editing with history buffer (enhanced `ACCEPT`). Keyboard scan codes (in Windows) or VT100 key codes (in Linux) are converted to MinForth-internal key codes.

Provided additional words:

`VT ( -- a )`
flag variable to en-/disable ANSI terminal control sequences

`SAVE-XY ( -- )`
save current text cursor position internally

`RESTORE-XY ( -- )`
reset cursor to last saved position by `SAVE-XY`

`GET-XY ( -- col row ) Windows only`
get text cursor position, counterpart to standard `AT-XY`

`VTNORMAL VTBRIGHT VTREVERSE VTUNDERLINE VTHIDDEN ( -- )`
set terminal text attributes

`VTBLACK VTRED VTGREEN VTYELLOW`
`VTBLUE VTMAGENTA VTCYAN VTWHITE ( -- )`
set terminal text colors

`VTUP VTLEFT VTRIGHT VTDOWN ( n -- )`
move text cursor by n positions

`VTERASE ( -- )`
erase current line from cursor position ot end of line

`PRINTF ( a u -- )`
type string with embedded terminal control characters starting with
% for frequently used number formats:

%d	decimal number	%u	unsigned decimal number
%x	hex number	%y	unsigned hex number
%D	double number	%f	floating-point number

~ for terminal control:
~k ~r ~g ~y black, red, green, yellow text

~b ~m ~ c ~w	blue, magenta, cyan, white text
~n ~t ~u ~v	normal, bright, underlined, reversed text attribute

`P"  ( string" -- )`
print formatted string like ." but with embedded % or ~ terminal control characters

```
# 12. 2dup cr p" A green dozen is ~g~t%D~n and reversed ~g~t~v%D~n"
A green dozen is 12 and reversed 12 ok
#
```

To evaluate command-line argument or execute OS shell commands in an application program:

`ARGC  ( -- n )`
number of command-line arguments including called program name

`ARGC  ( n -- a u )`
get command-line argument n as Forth string (a u)

`SYSTEM  ( a u – errno )`
execute system sub-shell command given by string (a u) returning error number (0 on success)

`DIR  ( -- )`
display files in current working directory

`CHDIR  ( <path> -- )`
change current working directory

2.7.12 Programming Tools Wordset

Practically all standard programming tools words as of ISO/ANS standard document §15 are provided within source file `./src/tools.mfc`.

Missing words in MinForth are mostly due to its transpilation method to C (where `CODE .. ;CODE` make not much sense) or because they are somewhat "too specific" (like `EDITOR`).

Additionally provided words:

`ALIAS  ( xt <NAME>-- )`
create a synonymous word NAME that executes xt

`.RS  ( -- )`
display actual return stack content

`.FS  ( -- )`
display actual floating-point number stack content

`XT-SEE  ( xt -- )`
"disassemble" high-level word with given execution token xt (useful for analyzing `:NONAME`s that cannot be accessed by standard word `SEE`)

Special functions in this wordset:

2.7.12.1 Quotations

Forth quotations are essentially unnamed 'guest' definitions nested within other 'host' definitions. An unnamed guest definition is 'bracketed' between the following words:

`[:` (--)
start a quotation

`;]` (--)
terminate a quotation

An interior quotation like in the following (trivial) example

```
: MOTOR-WARNING  ( -- f )
  prep-checks [: check-condition ;] >r suspect?
  IF   get-motor r> execute
  ELSE acknowledge-warning r> drop false
  THEN ;
```

is functionally equivalent to the more verbose exterior formulation

```
: CHECK-MOTOR-FLAGS  ( m – f )
  check-condition ;

: MOTOR-WARNING
  prep-checks suspect?
  IF   get-motor check-motor-flags
  ELSE acknowledge-warning false
  THEN ;
```

So quotations are like ‘packed subroutines’ and can lead to more compact code.

Another example cited from the current working draft Forth standard:

```
: HEX.  ( u --)
  base @ >r [: hex u. ;] catch r> base ! throw ;
```

The advantage of using `CATCH` here is that the `BASE` is restored even if there is an exception (e.g. an output stream interrupt) during the `U.`.

2.7.12.2 Recognizers

MinForth recognizers have already been introduced with the kernel word `RECOGNIZER` in § 2.6.2.

Recognizers are part of MinForth’s text interpreter. Whenever a parsed text token is neither found nor convertible to a single integer number, the hidden core word `_LITERAL?` walks the recognizer list and tries to process the text token at each recognizer element.

`SHOW-RECOGNIZERS` (--)
display actual recognizer list

`SWAP-RECOGNIZER` (<name> --)
exchange position with preceding recognizer in list

```
┌──────────────────────┐
│ MinForth V3.4 - 32 bit │
└──────────────────────┘
199116 bytes free
# show-recognizers
Recognizers:
REC-DOUBLE : _DOUBLE?  NOOP  2LITERAL
REC-FLOAT : >FLOAT  NOOP  FLITERAL
REC-COMPLEX : >COMPLEX  NOOP  ZLITERAL  ok
# 1.  \ 1. is a double number   ok
1 0 # swap-recognizer rec-float   ok
1 0 # 1.  \ 1. is now a fp number   ok
f: 1. | 1 0 # show-recognizers
Recognizers:
REC-FLOAT : >FLOAT  NOOP  FLITERAL
REC-DOUBLE : _DOUBLE?  NOOP  2LITERAL
REC-COMPLEX : >COMPLEX  NOOP  ZLITERAL  ok
f: 1. | 1 0 # swap-recognizer rec-double  \ back to normal   ok
f: 1. | 1 0 # 10.   ok
f: 1. | 1 0 10 0 #
```

When programming new recognizers it is sometimes necessary to re-parse the text token or the input source. This is supported by

`<IN  ( -- )`
reposition `>IN` back to the beginning of the actual text token within the input source

2.7.12.3 Stepping Debugger

The mf2c transpiler and C compiler catch many syntactic errors, but they cannot detect algorithmic or program flow errors. A seasoned C programmer can use the debugger tools of his toolchain for compiled primitives. But they are often awkward to use for debugging high-level words.

As additional debugging aid MinForth offers step debugging of primitives through the `+tr` and `-tr` transpiler commands (see § 3.6)

Standard Forth just foresees `DUMP ( adr u  -- )` to display the byte content of a memory region, and `SEE ( <name> -- )` to display a 'disassembled' content.

More comfortably, MinForth provides a stepping debugger to visualize execution of highlevel words:

`DEBUG ( <name> -- )`
enable stepping debugger for colon word `NAME`; tap space bar to step through its execution; special command keys are

`<d>`	for decimal numeric display
`<x>`	for hexadecimal numeric display
`<t>`	to type a string
`<r>`	to display the return-stack
`<q>`	to quit debugging, but the rest of the word is still executed
`<e>`	to exit debugging, the rest of the word is not executed
`<u>`	to disable the debugger for the current word, the rest of the word is executed
`<h>`	to display a help line for debugger command keys

`UNBUG ( <name> -- )`
disable stepping debugger for colon word `NAME`

`UNBUG-ALL ( -- )`
disable stepping debugger for all colon word within current compilation wordlist

Example:

```
# : STARS  10 0 DO I '*' EMIT LOOP ;   ok
# debug stars   ok
# stars
$7FF72E8AC0C8: d# _[LIT] 10 <r: on/off>
$7FF72E8AC0C8: r: 0 | d# _[LIT] 10
$7FF72E8AC0D8: r: 0 | 10 d# _[LIT] 0
$7FF72E8AC0E8: r: 0 | 10 0 d# _[DO]
$7FF72E8AC0F0: r: 0 10 -10 | 0 d# _[LOOP]
$7FF72E8AC0F8: r: 0 10 -10 | 1 d# _[JMPZ] 9
$7FF72E8AC108: r: 0 10 -10 | d# I
$7FF72E8AC110: r: 0 10 -10 | 0 d# _[LIT] 42 <hex>
$7FF72E8AC110: r: 0 A -A | 0 d$ _[LIT] 42 <Help Decimal heX Type Rstack Quit Exit Unbug
$7FF72E8AC110: r: 0 A -A | 0 d$ _[LIT] 42 <unbug> ********** ok
0 1 2 3 4 5 6 7 8 9 #
#
```

2.7.13 Overlay Wordset

Overlays are binary copies of a MinForth dataspace mapped to a disk file. They are available when the optional source file `overlay.mfc` had been included in the build.

Overlays can be used to freeze and re-use the actual condition of a MinForth session, or to load a binary application program without its actual source code (as it is with standard Forth words `INCLUDE` et al).

The MinForth systems between saving and loading overlays must be identical, ie. overlays between different builds will not be accepted. Primitives in MinForth codespace are not included in overlays.

`SAVED-OVERLAY ( a u -- )`
: create overlay file from name (a u) and write current data space content to the file

`LOADED-OVERLAY ( a u -- )`
: open overlay file with name (a u) and overwrite data space with binary overlay file content

`SAVE-OVERLAY ( <name> -- )`
: create overlay file with parsed name and write current data space content to the file

`LOAD-OVERLAY ( <name> -- )`
: open overlay file with parsed name and overwrite data space with binary overlay file content

The loaded overlay deletes the current dataspace and current high-level definitions therein and replaces them with the dataspace content at the time of saving the overlay. Forth system variables like STATE or BASE are not affected when reloading an overlay.

2.7.14 Big Floating-Point Number Wordset

Standard floating-point numbers offer a usable calculation precision of about up to 15 decimal digits. So-called BigFloats are floating-point numbers of arbitrary length. They are available in MinForth when the optional source file `bigfloat.mfc` had been included in the build (see § 5.6.4).

Normal floating-point numbers (see § 2.7.3) are addressed directly and fp-stack cells are fp-numbers. Bigfloats are different in that they are addressed indirectly and some attention is required from the user to observe memory load. Please read § 5.6 for more detailed information.

Nevertheless care has been taken to integrate Bigfloats as seamlessly as possible into the whole MinForth ecosystem. Example:

```
┌──────────────────────┐
│ MinForth V3.4.7 - 64 bit │
└──────────────────────┘
Stacks: d:128 / r:128 / f:16 / B:10
Dataspace: 1,048,576
# 15 set-precision   ok
# 1000 set-bprecision   ok
# pi f.  3.14159265358979  ok
# bpi b.
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534
21170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446 22
94895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664 82
13393607260249141273724587006606315588174881520920962829254091715364367892590360011330530548 82
04665213841469519415116094330572703657595919530921861173819326117931051185480744623799627495 67
35188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277 05
39217176293176752384674818467669405132000568127145263560827785771342757789609173637178721468 44
09012249534301465495853710507922796892589235420199561121290219608640344181598136297747713099 60
51870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526 19
31188171010003137838752886587533208381420617177669147303598253490428755468731159562863882353 78
75937519577818577805321712268066130019278766111959092164201 99  ok
#
```

BigFloats can be input through three different methods by

- a capital B as number prefix; this is the most comfortable way, implemented through a Bigfloat recognizer
- string input through the word STR>B
- type conversion through one of the words F>B S>B U>B D>B

```
┌──────────────────────┐
│ MinForth V3.4.7 - 64 bit │
└──────────────────────┘
Stacks: d:128 / r:128 / f:16 / B:10
Dataspace: 1,048,576
# B12e3   ok
B: 12000. # 13e-2   ok
B: 12000. f: 0.13 | # f>b   ok
B: 12000. 0.13 # 44   ok
B: 12000. 0.13 | 44 # s>b   ok
B: 12000. 0.13 44. # s" 89.2E3" str>b   ok
B: 12000. 0.13 44. 89200. | 0 #
#
```

2.7.14.1 BigFloat Glossary

Provided number type conversion words:

`F>B ( f: bf -- B: f )`
move fp number from fp stack to BF stack (may lose precision if `BPRECISION` is low)

`B>F ( B: bf -- f: f )`
move BF number from BF stack to fp stack (may lose precision if `BPRECISION` is high)

`S>B ( d: n -- B: bn )`
move number from data stack to BF stack (may lose precision if `BPRECISION` is low)

`B>S ( B: bn -- d: n )`
move rounded BF number from BF stack to data stack (may lose precision if `BPRECISION` is high)

`D>B ( d: l h -- B: bd )`
move double number from data stack to BF stack (may lose precision if `BPRECISION` is low)

`B>D ( B: bd -- d: l h )`
move rounded BF number from BF stack to data stack as double number (may lose precision if `BPRECISION` is high)

Provided BF number rounding words:

`BFLOOR ( B: bf -- bi )`
round towards nearest smaller integer, towards minus infinity

`BROUND ( B: bf -- bi )`
round towards nearest absolute integer, symmetric

`BTRUNC ( B: bf -- bi )`
round towards nearest integer, towards zero

Provided BF stack operation words:

`BDEPTH ( B: bn..b1 -- bn..b1 d: n )`
get BF stack depth

`BDROP ( B: b -- )`
drop BTOS (can be undone by `BUP`)

`BDUP ( B: b -- b b )`
duplicate BTOS, including its memory payload

`BOVER ( B: a b -- a b a )`
duplicate BSECOND over BTOS, including its payload

`BSWAP ( B: a b -- b a )`
exchange BSECOND with BTOS, without copying their payloads

`BROT ( B: a b c -- b c a )`
rotate top three BF stack elements, without copying their payloads

`BNIP ( B: a b -- b )`
discard BSECOND (can be retrieved by `BUP`)

Provided BF input retrieval words:

The following two words have no correspondence in standard Forth.

`BFLIP ( B: a b -- x )`
exchange BSEC with BONTOS and decrement BF stack pointer, thereby freeing the new BTOS position while keeping `b` hidden as new BONTOS and `a` as new BONSEC

`BUP ( B: -- x )`
increment BF stack pointer, thereby retrieving the hidden number above (BONTOS)

Provided BF comparison words:

`BISNAN ( B: a -- d: flag )`
check if BF number is a NaN (can be retrieved by `BUP`)

`B0< ( B: a -- d: flag )`
check if BF number is less than zero (can be retrieved by `BUP`)

`B0= ( B: a -- d: flag )`
check if BF number is equal zero (can be retrieved by `BUP`)

`B< ( B: a b -- d: flag )`
check if BSEC is less than BTOS (BSEC can be retrieved by BUP)

`BU< ( B: a b -- d: flag )`
check if BSEC is unsigned less than BTOS (BSEC can be retrieved by BUP)

`B= ( B: a b -- d: flag )`
check if BSEC is equal BTOS (BSEC can be retrieved by BUP)

`B~ ( B: a b eps -- d: flag )`
check if the absolute difference |a-b| is below |eps| (can be retrieved by `BUP`)

Provided singular BF arithmetic calculations (in place):

`BNEGATE ( B: a -- -a )`
negate BTOS (in place)

`BABS ( B: a -- abs(a) )`
absolute BTOS (in place)

`B1+ ( B: a -- a+1 )`
increment BTOS (in place)

`B1- ( B: a -- a-1 )`
decrement BTOS (in place)

`B2+ ( B: a -- a+2 )`
double increment BTOS (in place)

`B2* ( B: a -- a*2 )`
double BTOS (in place)

`B2/ ( B: a -- a/2 )`
halve BTOS (in place)

Provided dual BF arithmetic calculations (arguments retrievable):

`B+ ( B: a b -- a+b )`
BF addition (`BUP` retrieves first a then b)

`B- ( B: a b -- a-b )`
BF subtraction (`BUP` retrieves first a then b)

`B* ( B: a b -- a*b )`
BF multiplication (`BUP` retrieves first a then b)

`B/ ( B: a b -- a/b )`
BF multiplication (`BUP` retrieves first a then b)

`B** ( B: a b -- a**b )`
BF power (`BUP` retrieves first a then b)

`BMIN ( B: a b -- min(a,b) )`
BF smaller number of a or b (`BUP` retrieves the other number)

`BMAX ( B: a b -- max(a,b) )`
BF larger number of a or b (`BUP` retrieves the other number)

Provided BF analytic functions:

`BPI ( B: -- pi )`
BF pi constant 3.1415…

`BEULER ( B: -- e )`
BF Euler constant 2.7182…

`BINV ( B: f -- 1/f )`
BF inversion (`BUP` retrieves f)

`BSQR ( B: f -- f^2 )`
BF square (`BUP` retrieves f)

`BSQRT ( B: f -- sqrt(f) )`
BF square root (`BUP` retrieves f)

`BEXP ( B: f -- exp(f) )`
BF natural exponentiation (`BUP` retrieves f)

`BLN ( B: f -- ln(f) )`
BF natural logarithm (`BUP` retrieves f)

`BALOG ( B: f -- 10**f )`
BF decimal exponentiation (`BUP` retrieves f)

`BLOG ( B: f -- log10(f) )`
BF decimal logarithm (`BUP` retrieves f)

Provided BF trigonometric functions:

`BSIN ( B: f -- sin(f) )`
BF sine (`BUP` retrieves f)

`BCOS ( B: f -- cos(f) )`
BF cosine (`BUP` retrieves f)

`BTAN ( B: f -- tan(f) )`
BF tangent (`BUP` retrieves f)

`BASIN ( B: f -- asin(f) )`
BF inverse sine (`BUP` retrieves f)

`BACOS ( B: f -- acos(f) )`
BF inverse cosine (`BUP` retrieves f)

`BATAN ( B: f -- atan(f) )`
BF inverse tangent (`BUP` retrieves f)

Provided BF hyperbolic functions:

`BSINH ( B: f -- sinh(f) )`
BF hyperbolic sine (`BUP` retrieves f)

`BCOSH ( B: f -- cosh(f) )`
BF hyperbolic cosine (`BUP` retrieves f)

`BTANH ( B: f -- tanh(f) )`
BF hyperbolic tangent (`BUP` retrieves f)

`BASINH ( B: f -- asinh(f) )`
BF inverse hyperbolic sine (`BUP` retrieves f)

`BACOSH ( B: f -- acosh(f) )`
BF inverse hyperbolic cosine (`BUP` retrieves f)

`BATANH ( B: f -- atanh(f) )`
BF inverse hyperbolic tangent (`BUP` retrieves f)

Bigfloat Values:

`BVALUE ( ct: bf <name> -- | rt: -- bf )`
Bigfloat `VALUE`

`TO +TO ( f: bf <name> -- )`
store / add to `BVALUE`

Provided words for memory management:

`SET-BPRECISION ( n -- )`
set BF calculation precision to n decimal digits

`BPRECISION ( -- n )`
get actual decimal BF calculation precision

`BPAYLOAD ( B: bf -- bf d: n )`
get used memory net payload (without structure) of BF number

`.. ( -- )`
reset all stack pointers including BD stack pointer (but does not free payloads)

`FREE-BSTACK ( -- )`
free memory payloads of all unused BF stack elements (above BTOS)

`FREE-BVALUE ( <NAME> -- )`
free memory payload of named `BVALUE`

Bigfloat Input / Output:

`STR>B ( a u -- ior | 0 B: f )`
convert string to BF number, with flag on data stack (zero on success)

`B. ( B: bf -- )`
print BF number in decimal notation

`.BS ( -- )`
print BF stack content

2.8 Caveats

MinForth V3.4 is rather complete from a literary viewpoint in relation to the Forth standard document, but otherwise also rather limited for software development. This is partly due to its naked command-line interface, but mostly due to the nature of Forth that is more geared towards machine programming than to text processing or high-abstract programming concepts.

Specifically, not supported out-of-the-box are:

- Integrated Development Environment (IDE)
  but you can use your favorite code editor …

- Assembler programming and disassembling to machine code
  but it should not be difficult to use the built-in assembler and debugger of your C compiler

- Multitasking
  MinForth V3.4 is a command-line program and not thread-safe

- ABI for external libraries
  - for dynamic Windows DLL linkage see directory `./demo/windll`
  - calling C functions from MinForth is straightforward through building new primitives
  - calling MinForth functions from C means injecting/reading stack items and using MinForth mangled primitive names or search the dictionary to get the right function pointers

- Internationalization
  although the standard proposes an Extended-Character Wordset, and UTF-8 is dominant in the Unix and WWW world, M$ Windows does not support UTF-8 well; so it seems that two differing wordsets would be required…

- Other Platforms
  should not be too difficult, but.. right now MinForth V3.4 supports big-endian CPUs (Intel or ARM with set CPSR flag like some Raspberry PIs) and Windows and Linux operating systems (there is a limited FreeDOS version too)

- Heap memory management, dynamic strings or arrays, plot interface, linear algebra wordset .. just to name a few

3 MinForth-to-C Transpiler

The central component within the MinForth environment is the separate `mf2c` transpiler program. It translates MinForth text source to equivalent C text source. In other words: `mf2c` is a text converter.

There isn't much more to it and so `mf2c` can't execute any single Forth command. All commands that do specific conversions during a transpilation process (equivalent to Forth IMMEDIATE words are hard-coded within `mf2c`.

`mf2c` itself is a C program. Why C and not Forth? Because when you start working with a new hardware or CPU platform, it is very typical to have just a C or C++ compiler in the OEM's toolbox and some driver and function libraries.

Usage of `mf2c` is easy:

`mf2c`	without parameter transpiles the default MinForth source file `mf3.mfc`
`mf2c <name>`	transpiles the MinForth source file `name.mfc`.

For example, transpile and run the file `hello.mfc`

```
\ Hello

#include core.mfc

: MAIN
   ." Hello World " ;

#reduce
```

by

```
> mf2c hello
> tc32 hello   \ Linux: lmake hello
> hello
Hello World
>
```

When you inspect the new generated file `hello.c`, you'll see at once what the transpiler has done.

Now change `hello.mfc` to additionally print a number

```
\ Hello

#include core.mfc

: MAIN
   ." Hello World " 123 . ;

#reduce
```

and compare the new generated `hello.c` with the first one. It has become larger because Forth's classic dot-command `.` to print a number is a rather complex word that is built from a number of other Forth words. You can look up its definition in source file `core.mfc`. Therefore the `mf2c` transpiler had to pull in the sources of all those other building words for `.` too.

3.1 File Name Conventions

The following file name suffix convention are used:

`<name>.c`	C source files
`<name>.mfc`	MinForth/C mixed source files for the mf2c transpiler
`<name>.mf`	Forth source files for the MinForth system

3.2 Function and Word Names

General all internal C function-/variable-/macro names start with the two letters 'm' and 'f' in lower or upper case to avoid name conflicts in applications with embedded MinForth code.

`mf2c` converts every MinForth word name to a hashed C function name aka name mangling. F.ex the Forth word `ERASE` becomes declared in C mangled as

```
void mfB7F60EB5(void); // ERASE
```

where `mf` are prepended letters and `B7F60EB5` is the 32-bit FNV-1a hash value of the string `"ERASE"`.

This is necessary because Forth has a very flexible syntax (e.g. `2` can be word name and/or a number) that would conflict with C's more restricted naming.

However `mf2c` transpiles to documented C code with transcription of every mangled function. You should browse an `mf3c.c` file to get the idea.

3.3 Dictionary Structure

MinForth's dictionary is implemented as a C array of header structures (the `mfdict[]` array in `mf3.c`). Headers are linked via their link structure field, so that dictionary search can be extended to high-level words that were not created through `mf2c`.

Normal MinForth words are declared with a corresponding entry (header) into the dictionary list. But words whose names start with an underscore character '_' are treated specially:

Words starting with a single underscore like _HIDDEN get a dictionary header, but they are unlinked from the linked wordlist. SEARCH-WORDLIST "jumps over" such words and thus they cannot be found by high-level programs. But they are visible from within `mf2c` and can be ticked and deferred like any normal word.

Words starting with two consecutive underscores like __AWAY don't appear in dictionary at all. They are just named C functions. They don't even have execution tokens and cannot be ticked or deferred. They become just plain C functions that can be called from within high-level word definitions transpiled by `mf2c`.

3.4 Transpiler Commands

From transpiler perspective the 'MinForth language' is just a very restricted subset of some popular Forth commands.

Most `mf2c` commands look like their standard Forth counterpart words. They can be grouped in top-level commands and control commands for colon definitions.

A very important transpiler aspect is the close interaction between `mf2c` and MinForth's header file `mf3.h` that contains (among other things) all flow control commands and stack operation macros. I.e. when `mf2c` encounters a MinForth primitive definition containing an `IF .. ELSE ..THEN` structure, `mf2c` translates them to `mfIF .. mfELSE .. mfTHEN` macro names defined in `mf3.h`. These macros are only later resolved by C's preprocessor during compilation of `mf2c`'s output resp. target source file.

3.4.1 Top-level Commands

`mf2c`'s outer transpiler loop is implemented by its internal function `ParseFFile()`. It parses the current source file to string tokens and translates them to equivalent C definitions.

`: NAME`
start a high-level colon definition (see next § 3.5)

`C [TEXTLINE]`
(non-standard) literal copy of text line into transpiler output, C must be first character in line

`n CONSTANT NAME`
declare a constant for given number n

`[n] VARIABLE NAME`
declare a variable and initialize to n, or 0 if n is not given (non-standard)

`n BUFFER: NAME`
declare a buffer of length n and initialize to 0

`n CELLS`
modify N to cell sizes for CONSTANT, VARIABLE or BUFFER:
Example: `4 CELLS BUFFER: MYBUF`

`n FLOATS`
modify N to floating-point number sizes for BUFFER:
Example: `4 CELLS BUFFER: MYBUF`

`DEFER NAME`
declare an execution vector initialized to crash, to be resolved later by `IF`

`DEFER: NAME`
declare an execution vector initialized by following high-level words (initial hook function), can be modified later by `IF`

`IS NAME`
(non-standard) either set already declared execution vector NAME to execute the just preceding definition, or set just preceding execution vector to execute the already declared definition NAME
Examples: `DEFER VECT … : OUT1 101 THROW ; IS VECT`
`: OUT2 102 THROW ; DEFER VECT IS OUT2`

`IMMEDIATE`
mark preceding definition as IMMEDIATE

`COMPILE-ONLY`
mark preceding definition as COMPILE-ONLY (can be combined with IMMEDIATE)

`[DEFINED] <NAME>`
`[UNDEFINED] <NAME>`
set flag for follow `[IF]` depending on existence of word `NAME`

`N [IF] .. [THEN]`
conditional transpilation of definitions between `[IF]` and `[THEN]` when `N` <> zero.
(note that there is no `[ELSE]`, repeat the instruction with inverted N)

`#INCLUDE <FILE.MFC>`
include another source definition file `file.mfc` into the current transpilation process

`#REDUCE`
exclude all words, by backtracking from MAIN, that are not required to execute MAIN
(only useful for small programs that don't require the Forth outer interpreter, see § 5.1)

3.5 Colon Definition Commands

MF2C's inner transpiler loop for high-level colon definitions is implemented by its internal function `DoColon()`.It parses the current source definition to string tokens and macros and translates them to equivalent C code.

3.5.1 Program Flow Control

MF2C translates flow control commands to equivalent C macros declared in file `mf3.h`.

`IF … ELSE … THEN  CASE … n OF … ENDOF … ENDCASE`
standard conditional branching instructions, branch when number/flag on stack is not FALSE

`BEGIN … WHILE … REPEAT … UNTIL … AGAIN`
standard conditional looping instructions

`~IF … ~WHILE …~UNTIL …~OF`
non-standard branch instruction equivalents when number/flag on stack is FALSE

`^IF … ^WHILE … ^UNTIL …^OF`
non-standard branch instruction equivalents, but not consuming the number/flag on stack

`DO … ?DO … I … LOOP … +LOOP`
similar to standard counted loops, but cannot be nested (UNLOOP is not required)

It is often more practical to use the following commands for defining counted loops:

`+n FOR … N … NEXT`
(non-standard) loop n times between FOR and NEXT thereby incrementing counter N starting from 0
Example: : COUNT-TEN 10 FOR N . NEXT ;
COUNT-TEN → 0 1 2 3 4 5 6 7 8 9

`start +n el-size FOR> … N … NEXT`
(non-standard) iterate upwards over array beginning at `start`; the array containing `n` elements of size `el-size`; loop index `N` retrieves the memory address of each consecutive array element

`start +n el-size <FOR … N … NEXT`
(non-standard) iterate backwards over array beginning at `start`; the array containing `n` elements of size `el-size`; loop index `N` retrieves the memory address of each consecutive array element

Unlike standard Forth, MinForth's `FOR`-loops do not clutter the Forth return stack. `FOR`-parameters reside within a C stack frame.

`CONTINUE`
(non-standard) immediate jump back to head of current loop; don't forget to put a flag on the stack in a `BEGIN..UNTIL` loop because although it appears last, `UNTIL` is the loop head here

`BREAK`
(non-standard) immediate jump out of current loop or `CASE` construct

`EXIT`
exit current word immediately

3.5.2 Inlining C Instructions

Although standard Forth offers a huge number of words for handling different programming tasks, it cannot cover everything. Some applications may require using special POSIX or C library functions, or call Linux syscalls (or Windows WINAPI functions) or even assembler code.

For such tasks the transpiler provides

`C [TEXTLINE] [ ;]`
(non-standard) literal copy of input text line to transpiler output text line, upper-case character `C` must be first character in line; if line ends with ' ;' (space and semicolon) the current colon definition is finished.

`I" <C commands>"`
(non-standard) inline literal copy of input string to transpiler output
Example: `i" goto label"` can be used to emulate goto's in MinForth primitives

Many words of the MinForth V3.4 high-level interpreter/compiler are coded in C in this way.

Example for using a Linux syscall:

Transpile the following `HELLO` word definition within a MinForth build

```
C #include <sys/syscall.h>
: HELLO
C  syscall(SYS_write, 1, "hello, world!\n", 14); ;
```

Start a new session and call HELLO

```
┌──────────────────┐
│ MinForth V3.4 - 64 bit │
└──────────────────┘
3998816 bytes free
# HELLO  hello, world!
 ok
#
```

3.5.3 Special Compilations

To simulate immediate Forth compiler words MF2C provides

`N`
push single numeric literal on stack; if `N` contains a .dot-sign take it as floating-point number

`['] NAME`
push NAME's execution token on stack

`[,] NAME`
(non-standard, but similar to POSTPONE) compile NAME's compilation action, necessary for compilation of immediate Forth words

`S" <STRING>"`
push literal string start and length on stack

`." <STRING>"`
type literal string

`DEFERRED <NAME>`
(non-standard) transpile the definition that is actually linked with deferred word `NAME`

`;`
finish current colon definition sequence

`#IFDEF <NAME>.. #ENDIF`
`#IFUNDEF <NAME>.. #ENDIF`
(non-standard) condition transpilation of words between `#IF(UN)DEF` and `#ENDIF` depending on existence of word `NAME`

3.6 Tracing Debugger

In addition to detailed debugger or disassembler analysis, suspect functions can be traced with stack diagrams while they are executed. For this two simple transpiler commands are available

`+TR`
compile a halting step before each called MinForth word

`-TR`
continue with normal compilation (executed automatically when current primitive definition ends

This feature is only available when the `MFRTCHECK` flag in `mf3.h` is set.

Functionally `mf2c` simply inserts a call to the mfTR() function (in mf3.sys) before each called word.

3.7 MAIN Function

If the mfc input source file defines a Forth word MAIN, mf2c will write a standard C main() function to the C output source file. Such output files can be compiled directly to executable binaries.

If no MAIN word is defined, the C output file must be included in other C files before compilation. This is useful for embedding MinForth code into C programs.

4 Testing and Benchmarking

4.1 Standard Test Suite

The `./src/2012-test` directory contains the Forth 2012 test suite. Tests can be started by including the file `runtests.fth`. It can be very helpful when you adapt MinForth to your own liking.

Run the standard tests from within MinForth with
```
# include runtests.fth
```

Run the fp-number tests from the `./src/2012-tests/fp` directory with
```
# include runfptests.fth
# include paranoia.4th
```

```
Diagnosis resumes after milestone Number 190
         Page: 9


What message and/or values does Division by Zero produce?

   Trying to compute 1 / 0 produces ...inf

   Trying to compute 0 / 0 produces ...-nan
Diagnosis resumes after milestone Number 220
         Page: 10


FAILUREs  encountered = 0
SERIOUS DEFECTs  discovered = 0
DEFECTs  discovered = 0
FLAWs  discovered = 0

No failures, defects nor flaws have been discovered.
Rounding appears to conform to the proposed IEEE standard P754
The arithmetic diagnosed appears to be Excellent!
END OF TEST.

End of paranoia.fth
 ok
#
```

In fact MinForth passes all Forth-2012 standard tests. Some remarks to reported ‘errors’:

a) Block wordset test shows 2 false errors because the test assumes BUFFER and BLOCK to be identical. This is obviously questionable. (There is also a pertaining comment in gforth source file blocks.fs)

b) String wordset test shows 4 pseudo-errors in the SUBSTITUTE tests that can be attributed to over-zealous testing of undefined behaviour.

c) Some C compilers might cause floating-point number accuracy warnings. Clang, gcc and MSVCC cause no warnings.

d) The paranoia test should be run separately from the other fp tests because they redefine i.e. 'overwrite' some words internally that can cause false error report.

4.2 MinForth Special Words Tests

The `./src/mf-tests` directory contains the Forth 2012 test suite. The tests can be run by including files `mftest.mf` and `mfp.mf`.

4.3 Benchmarking

Basically MinForth can be as fast as your C compiler makes it, which should be pretty fast. As explained elsewhere, high-level words defined in a MinForth system are compiled to address lists and executed directly as C function pointers without intermediate conversion, which makes it more than fast enough for most cases. Some speed improvement can be achieved by disabling runtime error checking (see § 1.4).

There are other free and commercial Forths that are faster (or slower). Many use clever optimization techniques, like peephole optimization or CPU register allocation, and compile directly to machine code.

Technically spoken it is rather meaningless (unless for fun) to compare execution timings of micro-benchmarks between different Forth systems. Too many unknown factors can have significant impact on the results, like CPU cache effects or unaccounted energy saving modes.

On the other hand it can be very revealing to see how good or bad some C compiler code generation actually is. Fans of that sport just love to compare each other's (dis)assembly listings. Before you join the equipe make yourself familiar with the basics here:

> Optimizing subroutines in assembly language: An optimization guide for x86 platforms
> https://www.agner.org/optimize/optimizing_assembly.pdf

That said, we arrive at

4.3.1 The three different speeds of MinForth

With MinForth you can formulate words in 3 different 'languages'.

For fun, let us compare a dummy loop of 500 million iterations:

1. in plain C

```
: 500MLOOPC
C  for (i=500000000; i>0; i--);
```

2. in transpiled (pseudo) Forth

```
: 500MLOOPT
   500000000 BEGIN 1- dup 0= UNTIL drop ;
```

3. identical in high-level interpreted (standard) Forth

```
: 500MLOOPH
   500000000 BEGIN 1- dup 0= UNTIL drop ;
```

`500MLOOPC` and `500MLOOPT` have to be transpiled so that they become part of a MinForth system (or application). 500MLOOPH is defined within a MinForth interpreter session:

The following test was run on a little 13" laptop (1.7 GHz core-i5 gen8 CPU).

```
MinForth V3.4e
# 500mloopt  ok
# 500mloopc  ok
# : 500MLOOPH 500000000 begin 1- dup 0= until drop ;  ok
# timer-reset 500mlooph .elapsed 4411 ms ok
# timer-reset 500mloopt .elapsed 1277 ms ok
# timer-reset 500mloopc .elapsed 161 ms ok
```

Unsurprisingly the high-level threaded code is slowest with about 4.4 seconds, but it fares still okay.

The transpiled (subroutine-threaded) word arrives second, with about 1.3 seconds, which is within the speed range of some other high-performance Forth compilers.

The third (direct compiled C code) word is fastest by far. Here we could profit from advanced optimization algorithms built into the C compilers (the empty loop without side effects had even been eliminated away). A comparison with other Forth systems would not be correct.

On the other hand, this artificial example demonstrates an important MinForth design decision: "Usability goes before speed." If we need a very fast routine within an application, code it in C and let the C compiler do what it can make of it. C compiler makers are no idiots; the compiled machine code may not be top notch at micro-inspection, but it will run fast enough in most cases.

4.3.2 Optimizing Compilation

Nonetheless, it doesn't hurt to fine-tune the compilation result of the C compiler. "As provided" MinForth does not apply code optimization through compiler flags (anyhow, for gcc a simple -O seems to work well).

For starters, take one typical time-critical function from your application domain, write a short wrapper routine including a loop that executes your function x-thousand (or million) times, and measure the spent runtime with `TIMER-RESET` and `.ELAPSED`. Then draw your conclusions whether

- the algorithm could be improved
- MinForth or your function should be rebuilt and tested with different compiler flags
- the function should be reformulated in C instead of Forth
- (parts of) the function should be reformulated in CPU assembly language.

If you intend to go for C or even assembly just for the sake of optimization, there are two highly recommended books by Agner Fog that you should really study before you begin:

> Optimizing software in C/C++: An optimization guide for Windows, Linux and Mac platforms
> https://www.agner.org/optimize/optimizing_cpp.pdf

4.4 Forth Scientific Library

Somewhat more serious, and providing another good test platform, is the code collection of the Forth Scientific Library, which is provided in the `./fsl` directory.

5 Sample Applications

The `./demo` directory contains some little Forth program examples.

Special MinForth programming aspects:

5.1 Reducing transpiled Code Size

When the transpiler encounters the command `#REDUCE` in a MinForth source file, `mf2c` tries a recursive elimination of all unneeded words, descending the dictionary from the word MAIN.

See the included example in the `./demo/reduce` directory. Descriptions are in file `mfreadme.txt`.

Use #REDUCE with care because of two limitations:

1. It processes only Forth words. If the words use or call user C functions/declarations directly, the resulting C file may be incomplete and the compiled program may not work.

2. Of course it cannot predict which words might be searched in the dictionary later by the application. Therefore `mf2c` stops with an error when outer interpreter words appear in the reduce process.

But in the end this is no tragedy because modern C compilers do dead-code elimination anyhow.

5.2 Overlays

Overlays are binary copies of an actual MinForth dataspace mapped to a disk file (see § 2.7.13). Overlay files are binary files; they present an alternative to open source text files that could be read by any unauthorized person.

See the included example in the `./demo/overlay` directory. Descriptions are in file `mfreadme.txt`.

5.3 Creating Turnkey Applications

Auto-inclusion of an overlay when booting a MinForth system can be used to create turnkey applications.

See the included example in the `./demo/turnkey` directory. Descriptions are in file `mfreadme.txt`.

5.4 Dynamic Linking with Windows DLLs

Windows' Dynamic Link Libraries (DLLs) can be linked dynamically (as the name goes) at runtime without prior static compilation into application programs. This can provide a very convenient way to access precompiled libraries instead of revealing data or algorithms.

See the included example in the `.\demo\windll` directory. Descriptions are in file `mfreadme.txt`.

The simple example demonstrates how one can evoke a standard Windows message box through dynamic linking with a function in the `user32.dll` Windows library. By the way, Windows comprises also a number of C runtime libraries `msvcrxxx.dll` (within

`C:\Windows\System32`) which contain many useful C functions that can be linked with dynamically.

Note: an annoying peculiarity of Microsoft Windows is that a 32-bit process can only link with functions in a 32-bit DLL, and a 64-bit process with an 64-bit DLL.
https://docs.microsoft.com/en-us/windows/win32/winprog64/process-interoperability
When you stumble over this "feature" it might be advisable to create a 32- or 64-bit MinForth system for your application, i.e. suitable to the bitness of your DLL.

5.5 Experimental Core

The MinForth `./ecore` directory comprises a minimal MinForth system with only the standard Core wordset plus some elementary words to define a usable kernel program.

It can be used as starting point to develop own application wordsets, or for experimenting with new transpiler words, without being dependent on a fat full-fledged Forth system.

The directory contains also some simple usage examples as described in file `mfreadme.txt`.

5.6 Working with BigFloats

The MinForth `./bigfloat` directory includes a comprehensive word set for comfortable calculations with arbitrary length numbers (see § 2.7.14).

Calculation with big integer numbers (so-called BigNums) is also possible. However the Double Number word set in a 64-bit MinForth system already works with up to 128 bit integer numbers, so there has been no real requirement for wider BigNums up to now.

As of today, bigfloats have been built and tested only with 64-bit gcc compilers in Windows 10 and Debian Linux.

5.6.1 BigFloat Internals

Standard Forth words use numbers in CPU registers or in memory locations directly. Addresses like in `@` or `!` are slim pointers to memory locations. Very easy.

Not so easy with Bigfloats. Every bigfloat number is a compound set of three data objects:

- a fat pointer (`bf_t` structure) to the number's dynamic payload (mantissa table)
- a mantissa table allocated somewhere in computer memory (it can grow or shrink or be reallocated to other memory addresses)
- a common shared context (`bf_context_t` structure) serving also as calculation cache

So Bigfloat numbers are indirectly handled through their fat pointers. Calculations with Bigfloats are handled through `libbf` library functions that get the pointers and other things as parameters.

Before usage bigfloats have to be initialized. MinForth does this automatically. After usage Bigfloat memory should be freed to avoid memory leakage (e.g. through invocation of a `MARKER` word). MinForth provides some memory management words for releasing Bigfloat payloads.

Anyhow when MinForth (or any process) ends and priority is given back to the OS, all of its allocated memory is released automatically though an internal OS function (very old or exotic OS should be checked nevertheless).

5.6.2 Performance Optimization

As explained before BigFloats carry an opaque memory payload. With small precision lengths this is not very noticeable on modern computers with fast CPUs and big memories. However above several hundred decimal digits computation can become sluggish.

Copying Bigfloats should be avoided whenever possible because it involves allocation of new memory and copy of the large payload to its new twin.

The MinForth Bigfloat stack is an array of `bf_t` pointers. Moving the stack pointer costs virtually nothing, swapping and rotating stack elements is cheap (because only the pointers are moved, not their payloads), but `BDUP` and `BOVER` are expensive and should be avoided.

This conflicts to some degree with Forth's principle that words should generally consume their inputs (there are some exceptions to that rule). Forth programs use `DUP`, `OVER` etc. a lot for preparation of input arguments on the stack. A classic example is

```
: MIN  2dup < IF drop ELSE nip THEN ;
```

that involves copying of two stack elements within `2DUP`.

MinForth Bigfloats can do this completely without copying:

```
: BMIN1  b< bup not IF bflip THEN ;
```

By design practically all Bigfloat operators place their output value under the input value(s) that remain preserved hidden above the actual top-of-stack TOS position. By `BUP` the stack pointer can be moved up to make the 'old' input values visible again. `BFLIP` is a handy word to free the output TOS position from an input parameter.

But following the "principle of least surprise" BigFloat words can also be used just like their corresponding fp-number words, of course:

```
: BMIN2  bover bover b< IF bdrop ELSE bnip THEN ;
```

works just as well, although somewhat slower.

5.6.3 Active Memory Management

BigFloat memory payload should not pose problems on modern hardware. Even with 100,000 decimal digits calculation precision, a Bigfloat number needs just about 42k memory. So under normal circumstances you have sufficient space for Bigfloat numbers until you fill up your machine.

Nevertheless for some memory housekeeping you can use `BPAYLOAD` to inspect a number's payload size. `FREE-BSTACK` frees the payloads of invisible BF stack elements. `FREE-BVALUE` can be used to free the payload of no longer used `BVALUE`s.

Some more caution is required when using `MARKER` below `BVALUE`s. Carelessly used, executing the marker-word could leave orphaned payloads behind (memory leakage). To avoid this put the marker word into another definition with the same name, along with `FREE-BVALUE`s as required (for this purpose `FREE-BVALUE` is an immediate word).

Should you ever use the hidden word `_B,` you are on your own.

5.6.4 Library Compilation

MinForth's Bigfloat wordset is based on the LibBF library by Fabrice Bellard. LibBF is a small library to handle arbitrary precision floating point numbers. It is available from his web site https://bellard.org/libbf/

The author released the software under the MIT license from https://opensource.org/licenses/MIT

Please observe the license conditions when you use the software for non-private usage.

The libbf sources can be found in subdirectory `./bigfloat/libbf`. To (re)build linkable object files under a POSIX development environment, use the Makefile as usual with

```
% make all
```

Adapt compilation flags as needed within the Makefile. For more information read the enclosed file `readme.txt`.

On a 64-bit Windows OS (tested only with Windows 10 64-bit) with installed MinGW-W64 compiler it is easier to just run the enclosed batch file `gc64libbf.bat`. It will compile the required libraries and some small test programs. For more information read the enclosed file `readme.mf`.

MinForth directory `./bigfloat` comprises the sources to build a MinForth system with the optional Bigfloat word set. To change the default BF stack size, adapt build parameter `MFBSTSIZE` within header file `mf3bf.h`. Adapt the path setting to your own MinGW-W64 location within batch file `gc64bf.bat`.

To build and run MinForth with Bigfloats under Windows enter

```
> mf2c
> gc64bf mf3
> mf3
```

Similarly on Debian Linux

```
~$ ./mf2c
~$ ./lmakebf mf3
~$ ./mf3
```

Within MinForth you should now be able to pass the Bigfloat tests by entering

```
# include bftest.mf
```

6 Work Environment

By design, a typical MinForth work environment is very minimalistic. Everything can be done from the command line. You just need a text editor and a C compiler.

A two-pane file commander is helpful for easy navigation and code viewing / editing.
`Free Commander` for Windows or `Midnight Commander` for Linux can be recommended.

6.1 Operating Systems

MinForth runs on Windows and Linux operating systems. It should not be difficult to port it to other operating systems like BSD or DOS, but so far it hasn't been tested.

OS-specific functions are concentrated within the source file `mf3.sys`. It comprises two separate sections for Windows and Linux terminal primitives, plus some internal MinForth routines e.g. for dictionary search or the tracing debugger for primitives.

The MinForth distribution can be 'installed' (copied) anywhere into the file hierarchy to which the user has full access. For instance in a new empty desktop folder `MF34`. There the file structure should look similar to:

```
~/Desktop/MF34
├──bigfloat          \ bigfloat library
├──demo              \ many examples
├──doc               \ Forth standard and user documentation
├──ecore             \ experimental core sources
├──fsl               \ Forth Scientific Library
└──src               \ MinForth sources
    ├──2012-tests    \ Forth standard test suite
    │   └──fp        \   floating-point number tests
    ├──linux         \ (not in windows)
    ├──mf-tests      \ MinForth tests not covered by standard tests
    ├──tcc           \ (not in linux)
    └──windows       \ (not in linux)
```

Building/adapting MinForth takes place within the `./src` directory, where two new executable files `mf2c.exe` and `mf3.exe` (or `mf2c` and `mf3` in Linux) are created.

MinForth does not impose any environment settings on the user. Depending on preferences a user can execute `mf3` through scripts or symbolic links, or just copy `mf3` to his/her work directory.

However many users prefer to have `mf3` available as system-wide program. Then the search path to mf3 must be known to the terminal or console program.

In Windows the easiest way to set use user paths goes through the command (hit winkey+R)

```
rundll32.exe sysdm.cpl,EditEnvironmentVariables
```

that opens a system properties panel. The upper list is for the current user only, the lower list requires administrator privilege. Edit the user PATH variable and add a new line

```
%USERPROFILE%\Desktop\MF34\src
```

(or whatever installation path you have).

In Linux the easiest way is to edit the user's `.bashrc` script file within his/her home directory (enable 'show hidden files' in your file manager). Then edit and add to the bottom of the file

```
PATH=$PATH:$HOME/Desktop/MF34/src
```

(or whatever installation path you have).

6.1.1 Windows

MinForth runs in Windows 10 consoles with direct provision of VT100 terminal escape sequences. Virtual Terminal mode is enabled automatically (VT ON) at MinForth start. See also https://docs.microsoft.com/en-us/windows/console/console-virtual-terminal-sequences

Some earlier Windows versions - namely Windows 7 – did not support ANSI terminal or VT100 escape sequences. To enable these it is possible to use an external program: `ANSICON` created by Jason Hood;
https://github.com/adoxa/ansicon/releases
A copy is included in the `.\src\windows` directory. You'll want the x86 version for 32-bit Windows 7.

If you don't use C much otherwise and don't want to fill your hard drive with zillions of seldomly used files, the Tiny C compiler is recommended (see § 6.2.5) for MinForth compilation. It is already included in MinForth's distribution for Windows (see `.\src\tcc` directory)

Note: consider to set a directory exclusion to your on-line virus scanner for your development directory; it makes a real difference in compile times (and possible false alerts).

6.1.2 Windows Subsystem for Linux

The Windows Subsystem for Linux (WSL) is an emulated GNU/Linux environment within Windows 10. WSL provides many Linux command-line tools, utilities, and applications. See https://docs.microsoft.com/en-us/windows/wsl/install-win10

Microsoft actually provides two versions of WSL. WSL1 is recommended over WSL2 when you want easy file access from WSL Linux to your Windows files.

For example to get a minimal MinForth environment on Debian Linux, continue with the usual

```
~$ sudo apt-get update  <confirm with your user password>
~$ sudo apt-get upgrade
~$ sudo apt clean
```

followed by

```
~$ sudo apt install gcc  <build-essentials are not required>
~$ sudo apt install mc  <recommended file commander>
```

Start Midnight Commander with

```
~$ mc
```
(use ctrl-o to toggle between command line and file panes)

With `mc` create a `~/Desktop/MF34` folder in your home directory.

On WSL1 you can use `mc`'s other file pane (press tab) to move to root directory `/mnt/` from where you have access to your Windows drives and folders from within Debian. If you have MinForth files already on your Windows drives, you can copy those from there to Debian through the F5 key.

Vice versa, to access a running Debian session from Windows press <Winkey+R> and enter \\wsl$ to open an Explorer window showing the Linux file system. User directories are grouped under the /home/ directory.

After you have copied all Minforth files to your home directory move to `~/MF34/src` and open a command line to enter

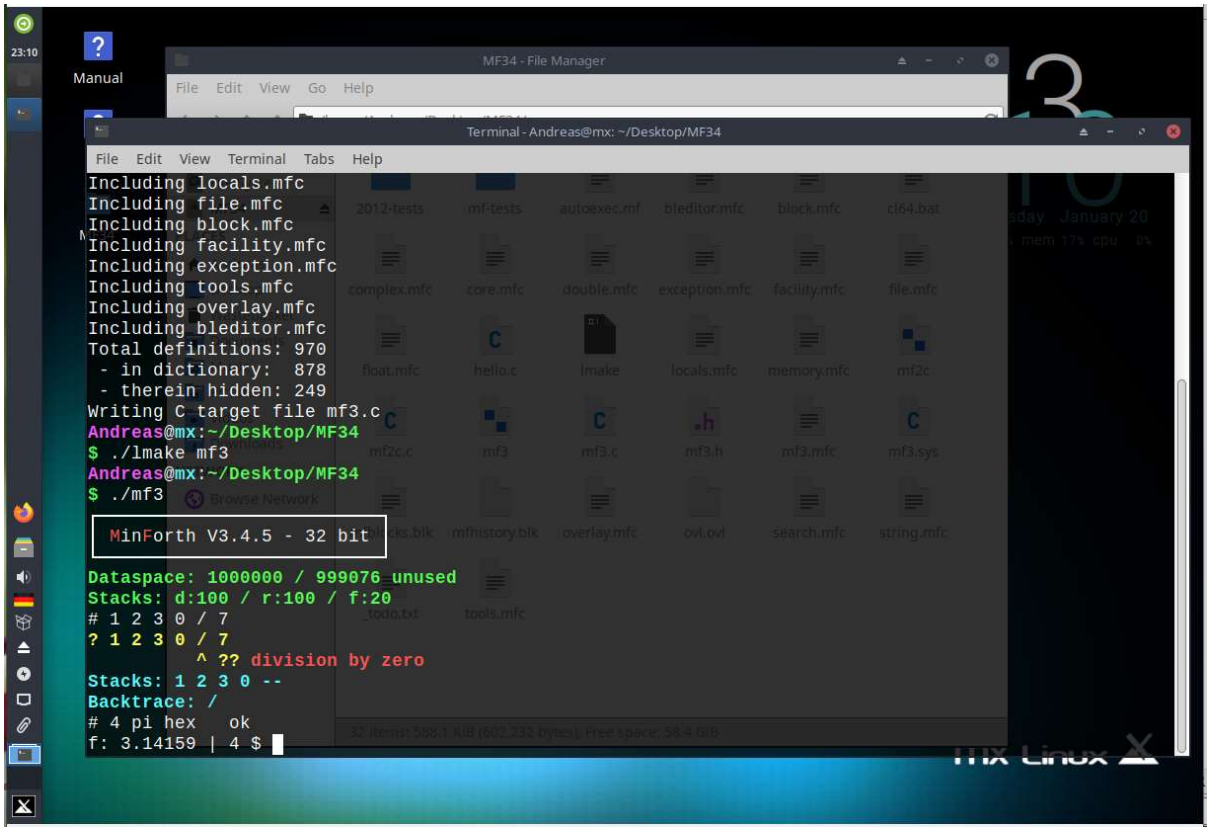
```
~$ cc mf2c.c -o mf2c
~$ ./mf2c
~$ cc mf3.c -o mf3 -Wall -std=gnu99 -funsigned-char -lm
~$ ./mf3
```

Similarly, MinForth transpiles, compiles with gcc and runs with OpenSUSE Linux on WSL.

Note: consider to set a directory exclusion to your on-line virus scanner for the WSL directory in `c:\Users\<account>\AppData\Local\Packages\<linuxpackage>` to improve operation times.

6.1.3 Linux

Screenshot showing a MinForth build session on MX Linux 32-bit:



Linux distributions are similar but not alike, particularly there is a plethora of desktop environments. Additionally, Linux terminals can have different default settings varying between distributions. Getting keyboard scan or key codes right can be a challenge because keyboard signals are processed through various layers deep within the OS, and desktop manager, and terminal program. F.ex. some function keys may be reserved for terminal commands, and thus are not available to MinForth.

Otherwise MinForth transpiles, compiles and runs fine with Debian Linux and OpenSUSE. Other Linux distributions should be okay too, but have not been tested yet.

6.1.4 DOS

The special MinForth DOS distribution comprises all required source files, a batch file `dj32.bat` to call the djgpp compiler, and test suites with adapted short file names. Extended memory should be enabled in the `autoexec.bat` file by `cwsdpmi -p`.

To build and statz MinForth enter:

```
> dj32 mf2c
> mf2c
> dj32 mf3
> mf3
```

While this process takes just few seconds in Windows and Linux, unfortunately compilation in DOS can take minutes. MinForth has been tested on FreeDOS running in a VirtualBox virtual machine, where it compiles and runs with acceptable performance (hint: use a fixed size drive, not a dynamic drive). However a test compilation in a DOSBox was too slow to be acceptable. Other options could be cross-compilation with OpenWatcom, but this has not been tested.

6.2 Supported C Compilers

Practically all Linux distributions already provide a C/C++ compiler, mostly gcc or clang. If it is not already installed on your machine (check with `./cc`) consult your documentation how to install a compiler (package build-essential or gcc on Debian derivates). Thereafter you should be able to build and run MinForth as shown in § 1.2.

Windows does not provide a C compiler out-of-the-box. But there are many options:

6.2.1 GCC

The Gnu C Compiler is undoubtedly the most versatile C/C++ compilers available for a wide range of target architectures. It is also the standard C compile for most Linux distributions.

For Windows 64-bit operating systems its mingw-w64 derivate is recommended.
https://mingw-w64.org/doku.php

MingW does not require any cygwin library to build and run MinForth.

You must use the `-D__USE_MINGW_ANSI_STDIO` argument with MingW because the underlying Windows C runtime library does not support extended fp-number output well.

The MinForth distribution comprises two batch files `gc32.bat` and `gc64.bat` to compile 32-bit or 64-bit MinForth versions from the command line. Move them up from `./scr/windows` to `./src` directory. Within the batch files you will have to edit the compiler path to your own installation location and build tools version thereafter.

6.2.2 Clang for Windows

Clang is the most prominent compiler in the LLVM toolchain. It is of more modern design than gcc and seems meanwhile on par with gcc, in some aspects even superior. Pre-built binaries can be downloaded from
https://releases.llvm.org/download.html

Note that clang-cl for Windows 'more or less officially' requires the Microsoft Visual Studio Build Tools (see § 6.2.3), but with some manual tweaking some people managed to make it work with MingW-W64.

The MinForth distribution comprises a batch file `cl64.bat` to compile MinForth from the command line. Move it up from `./scr/windows` to `./src` directory. Within the batch file you will have to edit the compiler path to your own installation location and build tools version thereafter.

6.2.3 MSVC

Microsoft's Visual C++ compiler (VC) for Windows is widely used together with the huge Visual Studio development toolchain.

Luckily to build MinForth you don't need to install the huge overhead of the Visual Studio IDE. The Visual C++ Build Tools suffice, f..ex.
https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16

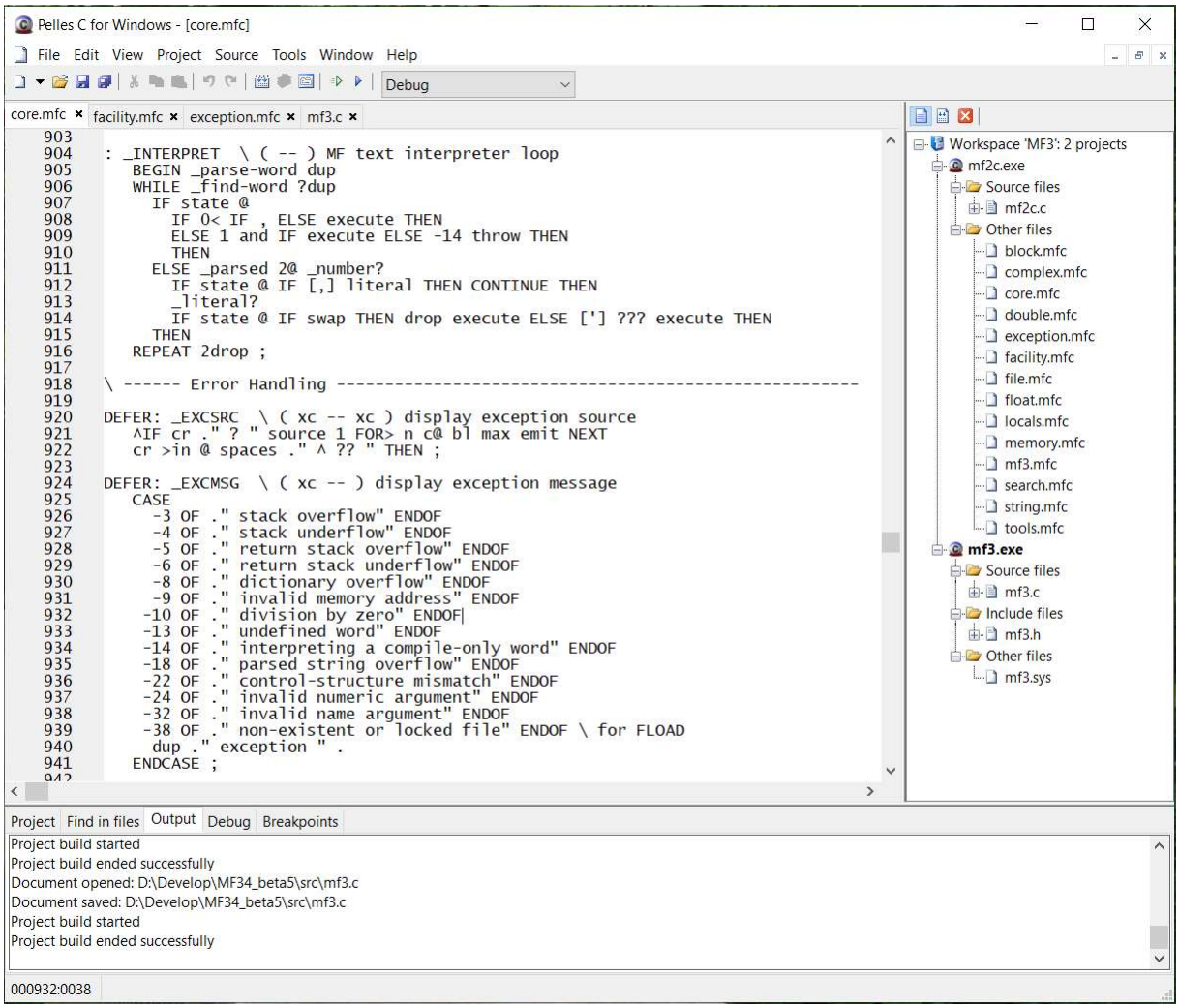
This will install the Visual Studio installer from where you can choose to install the C++ build tools. You'll need about 1.4 GB free disk space though….

The MinForth distribution comprises two batch files `vc32.bat` and `vc64.bat` to compile 32-bit or 64-bit MinForth versions from the command line. Move them up from `./scr/windows` to `./src` directory. Within the batch files you will have to edit the compiler path to your own installation location and build tools version thereafter.

6.2.4 Pelles C

Pelles C provides a nice light-weight IDE and C compiler for Windows programs. It can be found at:
http://www.smorgasbordet.com/pellesc/

The MinForth distribution comprises project workspace files `MF3.ppw, mf2c.ppj, mf3.ppj.` that can be loaded into the IDE. Move them up from `./scr/windows` to `./src` directory. Double-click on `MF3.ppw` to start the IDE:



6.2.5 Tiny C

Tiny C is a lightning-fast although non-optimizing C compiler. It can be found at:
http://download.savannah.gnu.org/releases/tinycc/
You should use version 0.9.27 (or later) which is nearly C99-compatible.

The repository comprises 32-bit and 64-bit variants. The MinForth distribution already includes a 64-bit tcc compiler for Windows in the `.\src\tcc` directory. It won't work on a 32-bit Windows system.

If you are using a 32-bit Windows version, you'll need the 32-bit tcc variant. After unzipping it into the `.\src` directory, you should have a `.\src\tcc` subdirectory with the `tcc.exe` file.

The MinForth distribution for Windows comprises two batch files `tc32.bat` and `tc64.bat` to compile 32-bit or 64-bit MinForth versions from the command line. Move them up from `.\scr\windows` to `.\src` directory. Within the batch files you will have to edit the compiler path to your own installation location and build tools version thereafter.

To build and start MinForth in one go, you can use the `make32.bat` or `make64.bat` scripts.

Note: if you get the error message 'unknown constraint t in math.h', replace all occurrences of "=t" by "=q" within TCC's <math.h> header file.

7 MinForth User License

MinForth is free software subject to a MIT license:

```
================================================================================
|                      MinForth V3 Software License                            |
================================================================================

   Copyright (c) 2020, Andreas K. Kochenburger

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```