MINFORTH

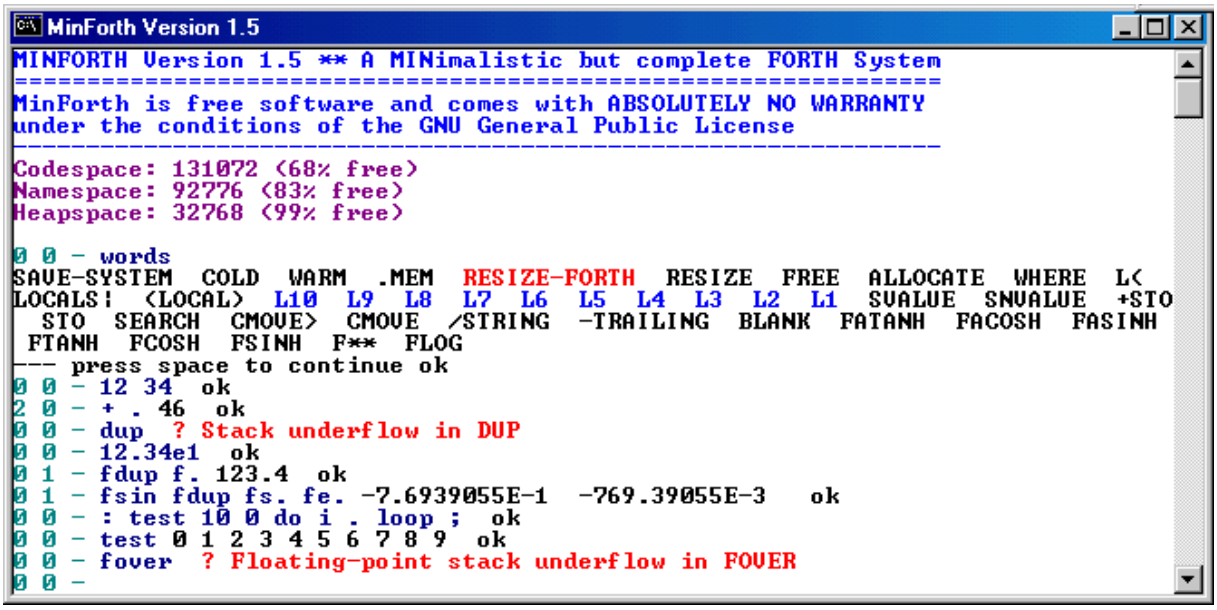
a MINimal but complete FORTH system
in C and ANS Forth
for DOS, Windows and Linux operating systems

Warning: ☺
The language used in this document and in the source code is Genglish (German English)..

1. Introduction

1.1. What is MinForth

Have a look at MinForth running in a Windows 2000 console window:



MinForth greets you shamelessly with claiming to be “a minimalistic but complete Forth system”. What does that overstatement mean:

Forth: if you don’t know about Forth please go to chapter 1.3.

Minimalistic: MinForth consists of a very simple virtual machine and only few low-level words, written in simple C. All the rest is built in high-level Forth by combining low- and high-level words.

Complete: practicallly all words defined by the Forth standard ANS X3.125 are provided. Thus for hardcore minimalists MinForth is a fat system. ☺

1.2. More Features of MinForth

Portability: MinForth runs on PCs. The sources can be compiled without modification for DOS, Windows or Linux operating systems. Images can run on any operating system.

Extendible: it is very simple to add new low-level routines in C, e.g. for accessing platform-specific hardware or for direct OS interfacing

Crash-proof: robustness goes before speed. All low-level words include maximum runtime checks before harm can be done. Dangerous conditions generate a high-level exception.

Flat memory: MinForth uses 32-bit-wide addresses to access a contiguous memory region. Word headers are separated from code space.

Floating-point math: MinForth uses 64-bit IEEE floating-point numbers

ANSI terminal: support for ANSI terminals is provided including colour and command-line history functions.

Debugger: a simple classic single-stepping debugger is integrated in the virtual machine.

Goodies: private headerless word definitions, string values, easy-to-use locals, vocabularies (like in "good old" F83), etc.

1.3. What is this all about: Forth?

There are so many good sites in the web where you can learn about Forth. The authors did it much better than I could ever do. Therefore I recommend to visit the homepage of the British Forth user group:

```
http://www.fig-uk.org/
```

And you can also participate in the always active Usenet forum:

```
news://comp.lang.forth.
```

If you are interested in trying out other Forth systems or if you want a Forth for a different hardware platform, visit

```
http://www.taygeta.com/forthcomp.html
```

1.4. An official explanation of Forth

Somehow officially, an extract from the ANS X3.125 standard document explains:

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 systems.

Forth provides an interactive programming environment. Its primary uses have been in scientific and industrial applications such as instrumentation, robotics, process control, graphics and image processing, artificial intelligence and business applications. The principal advantages of Forth include rapid, interactive software development and efficient use of computer hardware.

Forth is often spoken of as a language because that is its most visible aspect. But in fact, Forth is both more and less than a conventional programming language: more in that all the capabilities normally associated with a large portfolio of separate programs (compilers, editors, etc.) are included within its range and less in that it lacks (deliberately) the complex syntax characteristic of most high-level languages.

Forth is not derived from any other language. As a result, its appearance and internal characteristics may seem unfamiliar to new users. But Forth's simplicity, extreme modularity, and interactive nature offset the initial strangeness, making it easy to learn and use. A new Forth programmer must invest some time mastering its large command repertoire. After a month or so of full-time use of Forth, that programmer could understand more of its internal working than is possible with conventional operating systems and compilers.

The most unconventional feature of Forth is its extensibility. The programming process in Forth consists of defining new words - actually new commands in the language. These may be defined in terms of previously defined words, much as one teaches a child concepts by explaining them in terms of previously understood concepts. Such words are called high-level definitions. Alternatively, new words may also be defined in assembly code, since most Forth implementations include an assembler for the host processor.

This extensibility facilitates the development of special application languages for particular problem areas or disciplines.

Forth's extensibility goes beyond just adding new commands to the language. With equivalent ease, one can also add new kinds of words. That is, one may create a word which itself will define words. In creating such a defining word the programmer may specify a specialized behavior for the words it will create which will be effective at compile time, at run-time, or both. This capability allows one to define specialized data types, with complete control over both structure and behavior. Since the run-time behavior of such words may be defined either in high-level or in code, the words created by this new defining word are equivalent to all other kinds of Forth words in performance. Moreover, it is even easy to add new compiler directives to implement special kinds of loops or other control structures.

1.5. The full stuff

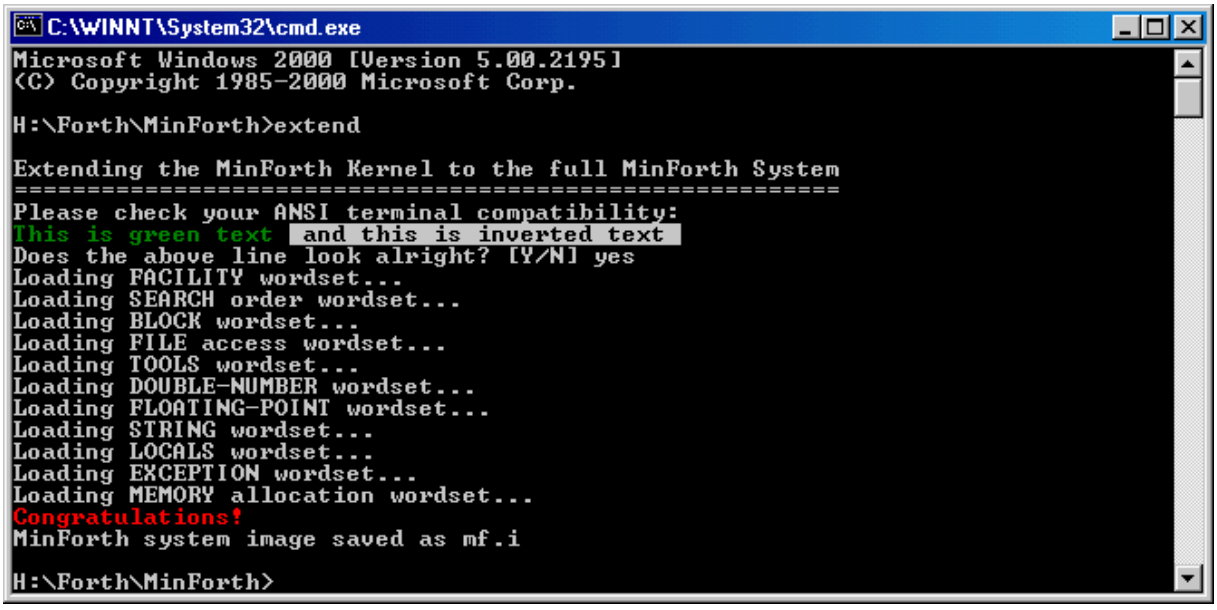
Not for the faint-hearted: get the ANS standard document, print it out (!) and spend (spoil) a rainy weekend with reading. You should start with Appendix C.

2. How to build MinForth

2.1. Easy steps

The steps are simple
1. compile the C sources if you don't want the downloaded executables
2. start metacomp to build the kernel system
3. start the extend "script" to build the MinForth system (you will be asked once if you have ANSI terminal capability)

That's all (if everything went well). Here is a screen copy while making a DOS version of MinForth:



2.2. Download MinForth

In fact there is only one common set of source files, regardless of the target operating system. The different packages offered below differ only in their precompiled executable files, the source files are 100% identical, in C and in Forth.

Please select your favourite package:

mfdoswin.zip zipped vocabulary for DOS systems
mflinux.tar.gz gzipped tar-file for Linux systems

2.3. Compiling MinForth for 16-bit DOS

You can compile MinForth with the Turbo C 2.0 compiler. Today Turbo C is freely available in the web. The following link once worked for me:

`http://community.borland.com/article/images/20841/tc201.zip`

Please contact Borland directly for any license matters.

Download the MinForth package and extract it into a separate directory, e.g. `C:\MinForth`. Make sure that you have loaded ANSI.SYS in your config.sys. The simplest way to compile the programs is by using the batch file tcmake.bat. Edit tcmake.bat and adapt the TCPATH variable to your configuration. Then use the following commands:

```
C:\MinForth> tcmake metacomp (make metacomp.exe)
C:\MinForth  metacomp        (make kernel.i)
C:\MinForth> tcmake decomp   (make decomp.exe)
C:\MinForth> decomp          (make kernel.d)
C:\MinForth> tcmake mf       (make mf.exe)
C:\MinForth> extend          (make mf.i)
C:\MinForth> mf              (start new MinForth)
```

If you prefer to use the Turbo-IDE instead of tcmake please don't forget to select the "huge" compiler model option.

2.4. Compiling MinForth for DOS with DPMI

You can compile MinForth with the djgpp compiler which is an adaptation of the gcc to work with DOS. Gcc is also the same compiler that is used for Linux system programming. You can get djgpp from:

`http://www.delorie.com/djgpp/`

Programs made with djgpp require a DPMI memory extender to run and a 386 processor at least. The advantage is that you can address nearly all of your RAM as one flat memory space. The easiest way is to use cwsdpmi that comes along with djgpp. You could either include cwsdpmi –p in your autoexec.bat file or start cwsdpmi (without paramerters) before each program that had been made with djgpp. But if you use the DOS version of Windows 9x upwards you can run the program directly, because DPMI services are already there.

Making MinForth is as easy as with the Turbo C compiler above. Make sure that you have loaded ANSI.SYS in your config.sys. Edit djmake.bat and adapt the DJPATH variable to your configuration. Then do as shown above.

Of course you can also use RHIDE instead of djmake.

2.5. Compiling MinForth for Windows

2.5.1. Getting a free C compiler

To compile MinForth for Windows there are several good C compilers freely available in the web.

Perhaps the best is the MinGW compiler. MinGW stands for “Minimalist GNU for Windows”, which explains that this is an adaptation of the gcc compiler to cooperate with Windows libraries. Gcc is also the same compiler that is used for Linux system programming. You can get MinGW from its homepage:

`http://www.mingw.org/`.

Another excellent free C compiler for Windows is lcc-win32. Its homepage is:

`http://www.cs.virginia.edu/~lcc-win32/`.

However the author Jacob Navia is so busy with continuously optimising his compiler that now and then he also happens to introduce new bugs, which he will then hunt down and kill in record time ;-). (Today ecvt() seems to be broken ☺).

2.5.2. Compiling MinForth

The compilation is as simple as explained for 16-bit DOS environments. Edit the batch file gcmake.bat / bcmake.bat / lmake.bat and adapt the variable GCPATH / LCPATH. Then run the batch file, e.g. like

```
gcmake metacomp
gcmake decomp
metacomp
gcmake mf
extend.
```

Note: pure Windows NT/2000/XP consoles do not support ANSI.SYS, even when the config.nt file is set properly by including the line device=%SystemRoot%\system32\ansi.sys. That is an old Windows bug. If you like to have colours and command line editing functions, then you should use the Windows version of MinForth, which uses API functions to control the console.

2.6. Compiling MinForth for Linux

Making MinForth for Linux is as simple as for DOS or for Windows. Untar all files into a subdirectory of your home directory, say ~/minforth. If necessary you must set the correct file modes for the little "scripts" lxmake and extend, e.g. by chmod 755. Then just do as before:

```
./lxmake metacomp
./lxmake decomp
./metacomp
./lxmake mf
./extend.
```

```
MINFORTH Version 1.4 ** A MINimalistic but complete FORTH System
=================================================================
MinForth is free software and comes with ABSOLUTELY NO WARRANTY
under the conditions of the GNU General Public License
-----------------------------------------------------------------
Codespace: 131072 (68% free)
Namespace: 92776 (83% free)
Heapspace: 32768 (99% free)

0 0 - cr sh" uname -a"
Linux vm-linux 2.2.16 #1 Wed Aug 2 20:22:26 GMT 2000 i686 unknown
 ok
0 0 - sh" pwd" /home/andreas/minforth
 ok
0 0 - files
.          double.mf     file.mf      locals.mf    mf           mfptoken.h
..         examples      float.mf     lxmake       mf.c         search.mf
block.mf   except.mf     fsl          lxmake~      mf.i         string.mf
copying    extend        kernel.d     memory.mf    mfcomp.h     tconv.mf
decomp     extend.mf     kernel.i     metacomp     mfpfunc.c    tests
decomp.c   facility.mf   kernel.mfc   metacomp.c   mfpnames.c   tools.mf
 ok
0 0 - pi fdup fcos f. -1.  ok
0 1 - ' aloha  ? Undefined word in ALOHA
0 0 - _
```

The source files are identical for whatever operating system you use! The source file facility.mf that is included during extend is just responsible for correct terminal support. Terminal support programming for Linux or Unix can be a nightmare. Therefore the control sequences used in facility.mf correspond to a standard Linux text console only. If you use a different terminal you must probably adapt facility.mf.

By the way my favourite Linux tool is Midnight Commander. Its integrated editor is even clever enough to work with DOS style or Unix style text files. You don't need duconv to convert text files from CR-LF to LF line terminations.

2.7. Adaptations for other environments

If your target machine has no PC-architecture, or if you don’t use DOS or Windows or Linux as underlying operating system, or if you stubbornly want to use your own home-made compiler, you will have to adapt the C sources. And of course you need a C compiler for your machine. But due to the minimalistic design of MinForth the portation job should not be too difficult.

3. The design

3.1. Simplicity and portability

The most obvious simplification over traditional Forth systems is that MinForth has no assembler. Its language for creating low-level words and system functions is C, for some strong reasons:
- Processors and assemblers change so often that it is completely uneconomic to maintain assembler software sources.
- ANSI-C is the "lingua franca" in the world of system programming since long. It is relatively easy to port ANSI-C programs from one machine to the next. It is difficult and often impossible to port assembler programs.
- Practically every manufacturer of programmable electronic equipment, e.g. controller boards, delivers C libraries for system programming of his products.
- Compiler makers are no idiots. They will probably have spent much more effort to understand the machine architecture than you will ever like to spend and their code will probably be very efficient. Why shouldn't we rely on their work? We'll test our compiled functions thoroughly later anyhow.

MinForth has few low-level words (primitives). Most words are defined in Forth. Once you have made your primitives work on your machine, the rest will function automatically.

The core program (mf or mf.exe) loads and interprets image files that contain 32-bit-wide execution tokens and data. One can exchange image files directly between different machines (they must use the same token numbers, of course). With 32-bit addresses one does also not have to fight with memory address calculations (C does this ugly job for us).

MinForth has a weak link to terminals. It uses ANSI-like character terminals, e.g. as provided by the ANSI.SYS device driver in DOS systems. That makes the porting job relatively easy. (One could think of using the wide-spread ncurses library though, but ...)

3.2. Stability

With MinForth robustness comes before speed (most other Forths favour speed). A MinForth program may crash or get stuck, but it should never plunge the machine into disaster. If you really need execution speed, create an extra low-level word in C or use a different Forth system or a faster hardware.

Therefore all MinForth primitives do extensive run-time parameter checking before they do anything. If an erroneous condition is detected a pertaining exception is raised (see Throw() function in mf.c).

If your program gets stuck or runs in an endless loop, typing CTRL-C will give control back to you.

MinForth does not even rely on the memory access violation monitoring offered by most C compilers for Pentium processors. It uses its own monitoring functions instead e.g. for stack over/underflow checks.

These stability features are realized with the signalling mechanisms provided by C. This explains also the limitations: whenever an exception is caused by hardware or OS failures, you cannot blame MinForth for not being able to catch it.

3.3. Virtual machine

The MinForth “motor” is absolutely trivial (see main() function in mf.c):

```
loop:
W = AT(IP);       // fetch next 32bit token into W
IP = IP + 4;      // advance instruction pointer
ExecuteToken(W); // process the new token
goto loop;        // loop back forever
```

A valid token is just an index into a jump-table (see mfpfunc.c) which contains addresses of low-level functions. E.g. token 16 leads to the execution of the pAT() function providing the runtime behavior for the Forth word @.

There is only one exception: if a high-level word starts with large token number > 256 (in its cfa) and if this number is a valid address, then this address is processed as runtime behavior (of a word made with DOES>, see ExecuteToken() function in mf.c).

This were not very flexible if not for the important functions pNEST() and pUNNEST(). pNEST() cares for pushing the instruction pointer IP onto the return stack when a high-level word is called. pUNNEST() pops IP back and the execution continues where the calling level had been left. Thus the virtual machine processes high-evel words by continuous nesting and unnesting.

3.4. Memory layout

MinForth's addressable memory space (Forthspace) ranges from 0 to LIMIT. It is divided into 3 sections: Codespace, Namespace, and Heapspace. In this way Codespace, Namespace, and Heapspace form MinForth's data space.

Codespace ranges from 0 to NAMES and contains all Forth code. New words are compiled at DP. Codespace contains no headers so that small turnkey programs without headers can be made.

Namespace ranges from NAMES to the beginning of the I/O buffers below HEAP. Namespace contains all headers. New headers are compiled at N-HERE to make a new dictionary entry. All namespace addresses and pointers to namespace objects are relative to NAMES so that the headers can be moved anywhere between HERE and TP.

I/O buffers are below heap and are accessed with absolute addresses.

Heapspace ranges from HEAP to LIMIT and can be used freely. New ALLOCATEd memory is appended to LIMIT and increases heapspace.

```
Address   Memory region utilization
=======   ========================
          --- Codespace ---
0         initial word sequence (_INIT _START _BYE)
16        system variables
256       Forth code
HERE      dictionary pointer (new code compiled here)
PAD       transient memory
...       (free codespace)
          --- Namespace ---
NAMES     first header
N-HERE    dictionary pointer (new headers compiled here)
...       (free namespace)
TP        transient headers
BCB       2 block buffer control blocks
FCB       8 sequential file control blocks
SIB       2 string buffers
TIB       1 terminal input buffer
```

```
          --- Heapspace ---
HEAP      first allocated user memory
HP        heap pointer
...       (transient heap memory)
LIMIT     Forthspace ends here
```

3.5. Adapting Memory Usage

There are 3 system variable defined at the beginning of kernel.mfc which can be adapted to adjust MinForth's memory footprint: NAMES, HEAP and LIMIT. Have a look at kernel.mfc and you'll see how. Then execute metacomp and extend from the OS command line, and you'll have your resized MinForth image files.

4. The C sources

4.1. Kernel image metacompiler

A metacompiler is a compiler that compiles a compiler. The program metacomp.c reads the kernel definition file kernel.mfc and compiles the kernel image kernel.i which contains the code for the MinForth kernel. The kernel is already a fully functional Forth interpreter and compiler, but with a limited set of words. So metacomp does not generate processor code but code for MinForth virtual machine. (For word purists metacomp is not a metacompiler because it is a C and not a Forth program – but who cares…)

Metacomp is a very simple and straightforward program. It stupidly walks through the kernel.mfc text file in a single pass, and repeats the DoCommand() function on each isolated word until the end of kernel.mfc is reached. When the word `_:` is encountered that marks the begin of a high-level definition, the while-loop in the CompileHilevel() function is entered. As "side-effect" the arrays codespace[] and namespace[] become filled. Metacomp knows all low-level words by including the file mfpnames.c which contains a single string array of all the names.

Metacomp understands only some few commands. All of them begin with an _underscore. Many of the commands - i.e. compiling words - have a prefix syntax (unlike their Forth counterparts) to make the program simpler. E. g.
in metacomp: `_const TRUE -1`
in Forth: `-1 CONSTANT TRUE`

In this way the kernel definition file kernel.mfc does not contain syntactically correct Forth code, but a kind of pseudo-Forth. Metacomp also knows nothing of immediate words, so be careful when you make your own high-level words in kernel.mfc.

Here follows an explanation of the compiling words provided in DoCommand():
_code mfname pname: compile a header with mfname and initialise its code field vector with the execution token number of the primitive pname.
_const name number: compile a constant.
_variable name: compile a variable and initialise it with zero.
_user name: address: compile a system variable whose address is in the user variable area of < 256 in codespace.
_store address number: store number at address.
_allot number: advance metacomp's code pointer.
_defer name: compile an execution vector.
_refers name: if the last definition was an execution vector, patch it to execute the high-level word name - if the last word was a high-level word, patch the execution vector name to execute it.
_immediate: set the immediate flag in the last word's header.
_compile-only: set the compile-only flag in the last word's header.
_alias aname word: compile a second header for another word.

_**: name**: compile a high-level definition until _; is encountered.
_**:noname**: compile a headerless high-level definition until _; is encountered.

Another set of compiling words for control-flow management is provided in the function DoControlWord(). They work exactly like their standard Forth equivalents:
_**if** _**else** _**then**: for alternative program execution depending on a flag value.
_**begin** _**while** _**repeat**: for recurring executions.
_**begin** _**until**: also for recurring executions.
_**begin** _**again**: for endless recurring executions.
_**exit**: to leave a high-level word incoditionally.

Metacomp does not offer counted loops like the DO-LOOP constructs in standard Forth. This is no disadvantage because one can realize counted loops easily with the available control-flow words. In addition they execute generally faster.

Two more compiling words are needed which are also provided in the function DoControlWord():
_**" string_literal**: compile the next word as string literal and replace all '_' characters by blanks.
_**' word**: compile the execution token of the next word a literal (the standard Forth equivalent is [']).

4.2. Kernel image decompiler

The program decomp.c reads the output file kernel.i of metacomp, translates its data and token sequence into readable format, and writes the translation into the text file kernel.d. Look at a kernel.d file with a text editor and you will understand what decomp does. Lines beginning with N: mean namespace addresses that are relative or offset to the `NAMES` system variable. Lines beginning with a C: mean codespace addresses.

Decomp is a valuable utility when you make modifications in metacomp, because it helps to check whether metacomp produces the expected results.

4.3. MinForth main program

4.3.1. Starting a MinForth program

Let us explore the source code of the main program mf.c. Its first comment lines explain how to use the mf program. For a better understanding you should look at two other programs: the extend-script (extend.bat in DOS or Windows) and the upper.mf program in the ./examples subdirectory.

Every “normal” C program starts with executing main(). Main() in mf.c is very short. It loads the image file, sets some signal handling routines, enters the virtual machine (see above), and does some house-cleaning before the program finishes.

An image file is a simple binary file which contains a magic number, the codespace and the namespace of a MinForth session. Look at the small `SAVE-IMAGE` definition in kernel.mfc. Thus when MinForth starts it just allocates the stacks, copies the image parts into codespace and namespace, and the virtual machine starts with the first instruction at codespace address 0, i.e. with the sequence `(BOOT) (MAIN) (BYE)`. These are deferred words which you can also set to your own definitions, e.g. with the (nonstandard) Forth word `IS`.

4.3.2. Included source files

When compiling mf.c the C preprocessor includes three special files:
mfcomp.h checks the compiler and the operating system and defines some values that are used to control a few #if-switches in mf.c
mfptoken.h contains a list of all execution tokens for the low-level words in MinForth
mfpfunc.c contains one single array of function pointers to the C definitions of the low-level words.

The primitives i.e. low-level definitions are in mf.c and form its biggest part. They are called from the low-level dispatcher ExecuteToken() by the command primfunc[token]. In other words, the primfunc[] array is MinForth's internal jump-table to its primitives.

4.3.3. MinForth primitives

"Primitive" is an old Forth slang for words that are not coded in high-level words. In many Forth systems these are coded in assembler language of the target or host machine. MinForth's low-level language is C to become hardware-neutral. Let us have a look at some simple MinForth primitives.

The Forth word SWAP is realized by the pSWAP() function:

```
void pSWAP() /* ( a b -- b a ) swap the top 2 stack elements */
{  Cell X; /* cells are 32-bit integers /
   Indepth(2); /* at least 2 elements on the stack? /
   X = TOS, TOS = SECOND, SECOND = X; /* exchange them */
}
```

The called function Indepth() is realized by

```
void Indepth(Int d) /* check number of stack elements /
{  if (stk+d > stk_min) /* stack pointer out of allowed memory? /
      Throw(-4); /* ANS exception for stack underflow */
}
```

Of course pSWAP() is one of the simplest functions, but nevertheless it shows the principle: do the maximum amount of parameter checks before a primitive function does something and throw an appropriate exception if anything is wrong. This is the way for all MinForth primitives. Because the run-time behavior of all high-level words must be a sequence of the run-time behavior of primitives in the end, MinForth provides the maximum amount of run-time error checking for all words.

4.3.4. Exception handling

In order to prevent any primitive from continuing its task with illegal parameters, Throw() must not return. Let us look at how Throw() works.

When the system variable (THROW) is set (normally to the cfa of the high-level word THROW) then Throw() executes THROW and then long-jumps directly to the virtual machine loop in main(), thereby clearing the processor's returnstack (not the Forth returnstack!).

If (THROW) is not set then we must have the rare but severe case of a double error. Double errors can not be caught because the exception of first error is just being handled. Therefore MinForth decides to terminate the session inconditionally but orderly via Abort(). To help with troubleshooting the cause of such severe errors, Abort() analyses the MinForth returnstack and tries a short backtrace and dumps the top values of the data stacks.

4.3.5. 64-bit arithmetics

MinForth cells are 32 bit wide. Therefore standard Forth double numbers are 64 bit wide in MinForth. Not all C compilers provide library routines for 64-bit integer math. Therefore the double number math primitives in mf.c contain two versions: one for compilers with 64-bit math and one "long-hand version" for compilers without. To select the right version, mfcomp.h provides the switch variable MATH_64.

4.3.6. Low-level debugger

Debuggers are very sophisticated programs that monitor and display the processor states during a running program. Compared with a real debugger the small Debug() routine in mf.c is a plain idiot. It does neither monitor the instruction pointer nor any machine state. It just halts before an execution as long as the return stack maintains a certain level. That's all.

Nevertheless it is a very useful tool. There are two ways to start the debugger from a high-level program: by inserting the command TRACE anywhere within a high-level word, or by DEBUG 'WORD' to trigger the debugger whenever WORD is used. There is an examplary debugging session further below in this document (chapter 6.4).

Within mf.c the debugger is controlled by two flags. Only when the flag debugging is on, the Debug() routine is started. And only when the flag tracing is on, the debugger line can be displayed. Obviously tracing must not be on when debugging is off. Debugging can only be reset by the debugger commands Stop or Abort.

5. The Forth sources

5.1. MinForth kernel

5.1.1. Kernel definitions

As explained above, the kernel definitions are in the file kernel.mfc. The definitions are formulated in "pseudo-Forth" for the metacomp program. Browse through kernel.mfc with a text editor. When you are familiar with Forth you will have no problems to understand the definitions, only that some words strangely begin with an underscore character '_'. These words have special "compilation semantics" in metacomp (there is no runtime behavior anyhow).

A characteristic of Forth is that it is strongly factorized, i.e. most Forth words consist of a combination of other Forth words. A Forth program or system is like a complex building where the low-level words (primitives in machine language) are the foundation, and where all structures on top of that rely on the foundation and/or on previously built other higher-level structures.

Therefore in the first part of kernel.mfc we lay the foundation: system variables and basic primitives for stack manipulation, then arithmetics and memory operations, then file and terminal access, and finally the Forth compiler. Each layer of word definitions relies on the previous layers. Very few words, particularly THROW, must be deferred words so that they can be compiled before they are defined.

When MinForth starts, the instruction pointer is set to IP = 0 and the virtual machine "lifts off". Therefore the first definitions in kernel.mfc are three deferred execution vectors: (BOOT) for whichever initialising tasks, (MAIN) for the main application, and (BYE) for whichever cleaning tasks after the main application. The fourth execution token is always -1 to close MinForth. These three deferred words are set to appropriate definitions which you will find near the end of kernel.mfc.

From codespace address 16 and 56 you will find some system variables that MinForth uses to set up the memory regions and stack sizes. Some of these variables were set by the metacompiler (functions FinishImage() and SaveImage() in metacomp.c). But you can also overwrite the system variables in MinForth just before you use the word SAVE-IMAGE. The so created image will then start with the new sizes.

5.1.2. Working with files

In C one usually uses the internal 'file' structure for buffered input and output. There is no standard way to access the buffering mechanism. But today's machines have already different layers of file buffering or caching in the hardware and in the operating system. Adding another buffering layer is nonsense. On the other hand there are small controller boards with PC-like architecture but with restricted or emulated disk functions. In such cases direct access without hidden structures is also better.

Therefore MinForth's low-level file-access words use unbuffered file operations by file handles. The ANS Forth file-access words are created with these words. For convenience

MinForth provides also words which evaluate the ior return codes of the ANS file-access words and which generate an appropriate exception if an error occurred.

5.1.3. Private words

A special temporary wordlist is provided for private words. The syntax is:

```
BEGIN-PRIVATE
   <private words>
END-PRIVATE
   <public words>
MAKE-PRIVATE
```

After `BEGIN-PRIVATE` new headers are placed at the end of namespace instead of being appended to the last header, and they are linked into the temporary wordlist. `FIND` now always searches the temporary wordlist before the context wordlists.

After `END-PRIVATE` the temporary words are still visible to the compiler, but new words are now compiled again as usual into the current wordlist and headers are appended.

`MAKE-PRIVATE` discards the private headers and empties the temporary wordlist. Of course the code of the private words remains in codespace, they just become nonames.

5.1.4. Extending the kernel

When we have built the kernel with metacomp, we can start it from the operating system's command line with `mf /i kernel`. The kernel welcomes us with a blank screen. Hitting return results in an ok-prompt, but the famous Forth command `WORDS` leads to an error message. That is not much and we should leave the kernel with `BYE`.

To get a fully equipped Forth system we need more. This is the task of the MinForth program file extend.mf. You can load it from the kernel by `FLOAD EXTEND`. You will be asked once whether your terminal understands ANSI escape sequences for cursor and colour control. Then it creates the `ROOT`, `HIDDEN` and `FORTH` vocabularies. Then it loads all the other system files that make MinForth a standard ANS Forth system. Then it sets some deferred words to display a startup message and the typical system prompt which shows the number of elements on the data stack and the floating-point numbers stack. Finally it saves the so created system as new image file mf.i.

From now on, whenever you start mf without parameters, it will load the image mf.i by default, and the full MinForth system will be at your command.

5.2. Facility wordset and ANSI terminal interface

The facility wordset contains words which make “facility” i.e. computer functions available to the Forth system. These are particularly timing functions and handling of keyboard events and monitor output. To facilitate porting, MinForth assumes the presence of dumb character terminals.

When ANSI terminal functions are present MinForth provides the "luxury" of colours and simple command line editing functions with command history storage. Pressing 'cursor up' on the keyboard brings the latest command back to the system prompt. The system variable `ANSI` can be used to switch ANSI terminal support on and off.

A keyboard event is triggered whenever you press a key on the keyboard. It consists of a series of characters of different length. The length depends on the key, the used terminal, and the used operating system. Porting terminal support to other machines is generally a nightmare….

5.3. Other ANS wordsets

5.3.1. Block wordset

The file block.mf contains the remaining words of the ANS block wordset. MinForth uses the file structure of the operating system it runs on; it does not map blocks directly to disk segments. You can simply use `BLOAD <blockfile>` for reading in a blockfile and starting it at block 1.

5.3.2. Double number wordset

The file double.mf contains the ANS double number wordset. In addition the MinForth double values are defined here.

5.3.3. Exception wordset

The complete ANS Forth exception wordset is in the kernel. The file except.mf contains the word `WHERE` which helps to find the cause of the latest exception by analyzing the return stack.

5.3.4. File-access wordset

The file file.mf is just a place-holder because the complete ANS Forth memory-allocation wordset is in the kernel.

5.3.5. Floating-point wordset

The file float.mf contains the ANS floating-point wordset. In addition the MinForth floating-point values are defined here.

By default MinForth's floating-point numbers are 64-bit IEEE double precision floating-point numbers. According to the ANSI/POSIX standard for C, math functions are executed for 64-bit doubles anyhow. When you use floats the C compiler inserts additional conversions. Therefore you should use doubles in C instead of floats whenever possible, unless you have space restrictions.

5.3.6. Locals wordset

The file locals.mf contains the ANS Forth locals wordset. In addition an improved locals syntax is provided whose local values do not appear in reversed order and that offers additional free local values for storing intermediate calculation results.

As a silly example one could define:

```
: ROT L( a b c ) b c a ;
```

Another example is the star-delta-transformation for 3-phase electric circuits:

```
: DELTA2STAR L( r1 r2 r3 | rsum )
r1 r2 r3 + + to rsum
r2 r3 rsum */ ( -- ra )
r1 r3 rsum */ ( -- ra rb )
r1 r2 rsum */ ( -- ra rb rc ) ;
```

5.3.7. Memory-allocation wordset

The file memory.mf contains the ANS Forth memory-allocation wordset. As already said, MinForth allocates contiguous data space within its heapspace. It does not "borrow" external memory from the outside world i.e. from the operating system.

But if heapspace becomes too small you can apply for an increase by the low-level word `RESIZE-FORTH`.

5.3.8. Search-order wordset

The file search.mf contains the remaining words of the ANS Forth search-order wordset. In addition a classic vocabularies scheme is defined here.

After the kernel image is created with metacomp, there is only one wordlist in a very long single thread. When search.mf is loaded, all words are reorganized in the multi-threaded ROOT and FORTH vocabularies.

5.3.9. String wordset

The file string.mf contains the remaining words of the ANS Forth string wordset. In addition the MinForth string values are defined here:

```
14 SNVALUE V1           \ empty string value for length 14
S"  Sawyer" SVALUE V2   \ initialized string value
S" Tom" STO V1          \ store a string
V2 +STO V1              \ concatenate strings
CR V1 TYPE
```

5.3.10. Programming-tools wordset

The file tools.mf contains some of the words of the ANS Forth programming-tools wordset. As already said MinForth's low-level words are written in C. There are no provisions for processor-specific assemblers.

The standard word SEE displays a word's definition in a "disassembled" raw form. The MinForth word VIEW displays the original source code if it can be located in a present source file. Note that only sources that had been included by FLOAD or REQUIRES can be viewed in this way.

The required editor wordlist contains just one word: EDIT, which uses an OS-command to start your favourite editor (vi of course ;-). A task for the adventurous: you could create a word LOCATE which starts your editor and puts it directly to the source of a desired word.

6. Examples

6.1. Test programs

The subfolder ./tests of the MinForth directory contains some small test programs for MinForth:

core.fr: John Hayes' classic test program for the core words of an ANS Forth system. Run it with FLOAD CORE.FR.

double.fr: some tests for double number arithmetics.

fltest.fr: some tests for floating-point number arithmetics.

postpone.fs: Anton Ertl's test for the correctness of POSTPONE.

tlocals.mf: for testing ANS-style and MinForth-style locals.

blktest.blk: a very incomplete check for classic block files. Run it with BLOAD BLKTEST.BLK.

values.mf: for testing MinForth's different VALUE's.

upper.mf: a DOS filter program made with MinForth.

struct.mf: a small but (not so) simple static structures package for MinForth.

reln.mf: online relocation of MinForth headers; try .MEM BAL-N .MEM.

winapi.mf: simple programming examples with the Win32 API function interface.

6.2. Forth examples

The subfolder ./examples of the MinForth directory contains a small collection of Forth proggies by various authors:

black.blk: a very old blackjack game in Forth.

bubble.tst: measures and displays execution times.

factorl.mf: calculates factorial numbers in different ways.

hanoi.mf: recursive solution of the classic Towers of Hanoi problem.

horst.mf: defactorization of single integer numbers.

magic.mf: creates magic squares.

matmult.mf: matrix multiplication benchmark.

md5.f: a MD5 hashing algorithm in ANS Forth.

perm.mf: permutation benchmark.

savage.f: an idiotic floating-point number benchmark.

sha1.f: a SHA-1 hashing algorithm in ANS Forth.

sieve.mf: the Byte sieve benchmark.

theory.4th: enjoy the nostalgia of early computer dialog experiments.

treesort.mf: a sorting benchmark.

6.3. Forth Scientific Library

The Forth Scientific Library (FSL) is a large and free collection of mathematical algorithms, of course written in Forth. The FSL web homepage is:

```
http://www.taygeta.com/fsl/scilib.html
```

You can apply this collection to your own needs. The subfolder ./fsl of the MinForth directory contains the FSL sources, which compile and run perfectly in MinForth. When you examine the sources you will find that the routines are really not trivial. That you can use them here is a good sign for the quality of MinForth, "in my humble opinion".

6.4. A debugging session

In addition to its intensive run-time checks MinForth offers simple but effective tools to find programming errors. It is best explained with an example. For this we use the little program tconv.mf which contains conversion routines between different temperature scales in degrees Celsius, Fahrenheit, Kelvin and Reaumur (an old French unit).

After starting MinForth from the OS we load the program with `fload tconv`. To test it we type `32 f>k` which should give us the expected result of 273 (Kelvin) on the stack. But instead we get a red stack underflow exception message.

MinForth remembers the latest exception number and its top 4 returnstack cells (see except.mf). We can try to find the fault with `WHERE` which tries to analyze the remembered cells. But that does not give us any real clue. The fourth return stack element is the number 59 which cannot be an address. It had probably been put there by r> for interim storage. We type `view */mod` to investigate the source code of `*/MOD` and find that that was the case.

We have not found the bug yet and therefore release the debugger with `debug f>k`. Again we try `32 f>k` and the debugger halts before the first word within f>k., which is f>c. We can single-step through f>k by pressing the space bar on the keyboard. But instead of halting before the second word we get the same old exception. The bug must be within f>c.

The debugger is still sharp, thus we retry with the cursor-up and return key. Again we are halted before f>c, but now we press the H key to get some help. The command for entering other high-level words was Nest, thus we press the N key to dig down into f>c.

```
0 0 - fload tconv  ok
0 0 - 32 f>k  ? Stack underflow in F>K
0 0 - where
Backtrace for last exception # -4 : Stack underflow
1168  ->  2DUP + 1 cells ->  OVER
2064  ->  M* + 1 cells ->  2DUP
2300  ->  */MOD + 2 cells ->  M*
59 -> data? ok
0 0 - view */mod in file kernel.mfc line # 457:
_: */MOD   \ ( n1 n2 n3 -- r q ) n1*n2/n3 with double intermediate result
   >r M* r> SM/REM _; ok
0 0 - debug f>k  ok
0 0 - 32 f>k
40484: d 32 > F>C  ? Stack underflow in F>K
0 0 - 32 f>k
40484: d 32 > F>C  Help
Help RSt FSt heX Nest Unnest Cont Stop Abort Bye
40484: d 32 > F>C  Nest
40352: d 32 > [LIT]
40360: d 32 32 > -
40364: d 0 > [LIT]
40372: d 0 59 > */  ? Stack underflow in F>K
0 0 - see f>c
: F>C   _LIT 32 _MINUS _LIT 59 */ ; ok
0 0 - _
```

The debugger now halts before every single word within f>c. But after some steps we get the exception again. Now it is obvious: */ requires 3 stack arguments, but the data stack contains just 2, namely 0 and 59 on top. With `see f>c` we can look at the decompiled internal representation of f>c. The debugger was correct, but we had a typo mistake in the source: it should have been 5 9 with a blank in-between and not 59.

6.5. Adding new primitives: DOS system programming

System programming belongs to the most difficult and to the worst paid jobs in the software industry. Therefore it is a source of happiness for any REAL programmer.

Adding new primitives to MinForth is easy. You just have to extend the tables in the three files mfptoken.h, mfpnames.c and mfpfunc.c. And don't forget to increase the PRIMNUMBER value in mfptoken.h accordingly! Then you must insert the new functions into mf.c. It is probably wise to create an extra include file for that and to #include it in mf.c, just after the MinForth primitives.

If you have downloaded the DOS version of MinForth then you have a subdirectory ./bios to the MinForth root directory. It contains the sources and executables for a MinForth system which can be used for system programming in DOS. Just study the above mentioned files with a text editor (or file commander) to see the small changes. The new primitive functions reside in the file mf_bios.c. Without modification they can be compiled only with the Turbo C compiler.

The new basic system programming primitives become available in MinForth when you load the source file bios.mf. If you need it very often you should include bios.mf in extend.mf. The exemplary BIOS video interrupt words in bios.mf are good enough only for 'historic' PC architectures. In order to make them fit your needs you will have to work through your own BIOS documentation. But the principle is clear.

6.6. Using Windows API functions in MinForth

The same principle that we used for accessing BIOS interrupts can be used for accessing Windows API functions. Application programming interface (API) functions are hundreds of

precompiled procedures in Windows library files. C compilers for Windows have large header files that are used for linking these external procedures into application programs.

If you want to experiment with simple API functions just have a look at how the console interface is realized in winterm.mf. And try the winapi.mf program in the ./tests subdirectory.

By the way, C programmers might find the source of the p_RUNPROC() function in mf.c interesting. IMHO it is a fine small example of what can be done with function pointers.

7. MinForth Software License

MinForth is free “copylefted” software. MinForth comes without absolutely no direct or indirect guarantee or warranty and you can use the software on your own risk. You can use MinForth according to the widely used GNU General Public License. You can obtain latest copies of the GPL license from the web-site:

`http://www.gnu.org`

The original MinForth download packages also contain a text file `COPYING` of the license.

8. Some personal remarks

Please note that the public release of MinForth is a snapshot of a standard Forth system (without multitasking). I have taken up the effort to write this documentation and to brush and polish the software a bit before releasing it into the public domain. But it remains an old snapshot anyhow. In other words: it may contain bugs and it certainly can be improved in many aspects.

By the way for those who are interested: the original development target for MinForth were controller cards for process controllers. The cards had Intel processors and were in fact small PCs with live access to input/output signals. The VM on the controllers was of course programmed in Assembler and reasonably fast. Unfortunately the project never made it into products. Nevertheless I am convinced that MinForth contained some good ideas. It would be a shame if it just went down the sink of software history.

I am publishing the software with the idea in mind, that others might find MinForth interesting as well. Go ahead and have fun!

Should you stumble over a real bug please drop me a line at `minforth@gmx.de` or post a message in the Usenet forum `comp.lang.forth`.