Mr-Forth — A Minimal ROM-able FORTH

Mark Hays

June 27, 2015

Introduction

Welcome to Mr-Forth, a Minimal ROM-able FORTH system.

FORTH is an unusual programming language invented by Chuck Moore in the late 1970's. Despite its oddness, FORTH has survived for 30 years for several reasons:

- it strikes a good balance between being high level and low level.
- it can be very efficient, in terms of memory and speed (though Mr-Forth is *not* fast).
- it doesn't require a lot of system resources; Mr-Forth's core dictionary uses less than 16k of ROM and well under 1k of RAM
- it is *interactive*. You can edit, compile, test, and run FORTH code on very small systems.

Today's FORTH users include NASA, microcontroller programmers, BIOS programmers, robotics hobbyists, etc.

Mr-Forth is a minimal system: it implements most of the commonly expected FORTH core with as few system dependencies as possible. The only things it expects from (or knows of) the hardware on which it runs are the ability to print characters to and receive keystrokes from the user. Mr-Forth doesn't know anything about disks, graphics, or anything like that; if you need these things, you'll have to add them to Mr-Forth yourself. As shipped, Mr-Forth gives you precisely enough so you can play with FORTH on a PC; it's easy to add support for specific hardware to Mr-Forth, though.

Getting Started

If you aren't familiar with the FORTH programming language, I'd suggest reading the online version of Leo Brodie's book *Starting Forth* (now out of print, URL given in the "Credits" section of this manual). Intermediate and advanced FORTH users should have a look at Leo Brodie's book *Thinking Forth* (also out of print, but available at the URL in the "Credits" section as well). Finally, Windows users might be interested in the Windows help file `pfe-fth.chm` from the PFE distributions (yup, URL below); this file contains a searchable version of the ANS94 FORTH standard, in particular.

To run an interactive Mr-Forth session, get to the distribution directory, open a shell or command-prompt window, and type

```
mrforth
```

I won't promise that all (or any) of the examples in *Starting Forth* will work in Mr-Forth, but it should be possible to get them working with some effort. You might have to type "./mrforth" or ".\mrforth" depending on your particular system. You can execute the following simple "smoke test" to confirm that Mr-Forth is working properly

```
        : doit ." Hello, world!" cr ;
        doit
```

If Mr-Forth is working, the standard message should be printed in your terminal window.

Sooner or later — probably sooner — you'll want to store your code in files. This is easy: create your code in a text editor such as notepad and save it (the example below assumes it's in the Mr-Forth distribution directory). To run your file in Mr-Forth, do

```
        mrforth myfile1.fth myfile2.fth
```

This will execute the code in these 2 files and then dump you into an interactive session (unless one of the files executes `bye` which causes Mr-Forth to exit immediately).

Files in the Windows `mrforth.zip` Archive

The Windows distribution ZIP file contains the following:

`Makefile`	Top level Makefile for building Mr-Forth
`README.txt`	A pointer to the Mr-Forth manual
`cygwin1.dll`	A copy of the Cygwin library, in case you don't have one or your isn't compatible with mine for some reason
`docs/*`	Mr-Forth documentation
`forth/*.fth`	Mr-Forth core FORTH source; most of Mr-Forth is written in Mr-Forth
`mrfgen.exe`	The Mr-Forth dictionary generator
`mrforth.exe`	The main Mr-Forth executable
`src/*`	C source code for `mrfgen.exe` and `mrforth.exe`

Mr-Forth Design

The idea for Mr-Forth came from reading about Bill Muench's eForth (see the "Credits" section). The eForth core consists of about 30 FORTH primitives; the rest of the FORTH words are built using this handful of words. I couldn't use eForth because it is fairly x86-specific; I liked Bill's idea of getting something working quickly, but I wanted to be *really* lazy: I didn't want to write any assembly code whatsoever.

After studying pForth for a while (see link at end), I decided that writing roughly 2 dozen primitives in ANSI C would do the trick. Mr-Forth uses a virtual machine architecture that is similar in spirit to pForth: the VM opcodes map directly to the primitives. Figuring out which primitives were needed was an interesting exercise. The list could be further reduced, but only with a large reduction in performance. Writing the core also proved interesting; I spent many hours staring at the PFE source for inspiration. The upshot of all of this is that Mr-Forth isn't very fast; its guts are, however, extremely portable.

I spent a good bit of time studying the ANS FORTH '94 standard. Some of it seemed...clumsy and... "designed by committee." After much reading, including some rants by Chuck Moore, I bailed on the idea of ANS94 compliance and simply used it as a source of ideas.

The final wrinkle is that I wanted Mr-Forth to be compatible with ROM-based and Harvard architecture systems (like flash-equipped microcontrollers) where instructions and data live in physically separate memories. Like most programming languages, FORTH is inherently a little more comfortable with the "normal" von Neumann architecture model. Getting FORTH running in a ROM/Harvard environment wasn't actually too hard: the main trick involved changing variables into constant addresses in the RAM area. Also,

From	To	Description
0000	4999	FORTH core
5000	8999	Optional additional words
9000	9999	Code to finish up core generation

Table 1: Numbering of FORTH source files in `forth/`

it was necessary to implement `mrfgen` separately from `mrforth`; since Mr-Forth is 16-bit, big-endian, and twos-complement on all platforms, image portability isn't much of an issue.

Writing Mr-Forth was a heck of a lot more work than I expected, but it was fun and much was learned. This is what hobbies are for, I guess.

Advanced Usage

The distributed `mrforth.exe` program only contains a basic FORTH core. As you develop your own FORTH words, you might want to either add them to a standalone core or create a new `mrforth.exe` that contains *your* core.

To generate a standalone core, create a file that contains the words you want to add (we'll assume the file is `mywords.fth` below) and

```
./mrfgen forth/[0-8]*.fth mywords.fth forth/9*.fth -- mycore.img
```

to generate `mycore.img`, your customized FORTH image; as an alternative, see the procedure below. To use this image with the distributed `mrforth.exe`, simply do

```
mrforth -- mycore.img
```

if you want an interactive session, or

```
mrforth myfile1.fth myfile2.fth -- mycore.img
```

if you want to load some extra source into your core.

To generate a new `mrforth.exe`, you'll first need a Unix or Cygwin development system with `gcc`, `make`, etc. To build the `mrforth.exe` program, edit the Makefile and add the names of your FORTH files to the `APP_FSRCS=` line and then

```
make all
make install
```

This process creates both `mrforth.img` and `mrforth.exe` files. If you simply want the `mrforth.img` file, a "`make mrforth.img`" command will do the trick.

Note that the last line above will *replace* the `mrforth.exe` executable in the main distribution directory with the new one.

The FORTH sources in `forth/` are numbered as `NNNN-name.fth` which helps ensure that they get loaded in the proper order. The interpretation of `NNNN` is given in table 1.

Finally, there are a number of `#define` options in `mrfconfig.h` related to profiling and speed/safety tradeoffs; these are detailed in table 2. It is highly recommended that the `MRF_CHECK_*` defines be left turned on; the rest are optional. If profiling is enabled, `mrforth.exe` and `mrfgen.exe` will print run times and primitive counts on exit; in addition, the following primitives will be defined:

(preset) Reset all profiling counters

(pprims) Push the number of primitives executed as a dword

Symbol	Meaning
MRF_PROFILE	Enable basic primitive execution count profiling
MRF_CHECK_STACK	Check for operand stack under/overflow
MRF_CHECK_RSTACK	Check for return stack under/overflow
MRF_CHECK_XWRITE	Check for writes to ROM in runtime VM

Table 2: Mr-Forth VM compile-time options

(psecs) Push the number of secondaries executed as a dword

(ppcnt) Push the number of times the primitive whose CFA is TOS has been executed as a dword

A simple benchmarking word might look like this:

```
: (report)
  ." executed " d. ." primitives" cr
  ." executed " d. ." secondaries" cr
;

: bench
  (preset)
  execute
  (psecs) (pprims)
  (report)
;

: myword ... ;

' myword bench

\ count dup calls:
: dupcnt 'compile dup (ppcnt) d. ." calls to dup" cr ;
```

Mr-Forth cannot currently profile secondaries; this is left as an exercise.

Intermission

The remainder of this manual consist of a fairly detailed accounting of the design and implementation of Mr-Forth's innards. If you simply want to write some code in FORTH, only the "Credits" sections at the end will be of interest to you. If you'd like to understand how Mr-Forth works or are interested in porting it to another platform, the information below will hopefully give you all the needed information.

MrFLib C API

Mr-Forth exposes a simple C API via `libmrforth.dll`. This API only includes the ability to evaluate FORTH code and step the VM. The API is described in the file `src/mrflib/mrflib.h`. A simple demo of the API is in `src/mrflib/demo.c`. The distribution contains a built version of this program, `demo.exe`, which simply runs the FORTH application built by `mrfgen.exe` — an interactive FORTH session.

Virtual Machine Architecture

The Mr-Forth virtual machine (VM) provides two basic services: a virtual hardware platform and roughly two dozen FORTH primitives on which all other FORTH words are based.
The virtual hardware has a number of properties as follows:

- Characters (bytes) are exactly 8 bits wide.
- Words (cells) and addresses are exactly 16 bits wide.
- Words are stored in memory with the most significant byte (MSB) first ("Big endian" byte ordering), even when on one of the stacks.
- Memory is arranged as a list of bytes, with valid addresses ranging from `0x0000` to `0xFFFF`.
- Word loads and stores may be unaligned.
- Memory is conceptually split into two regions: ROM and RAM. The ROM is read-only and the RAM is read-write. These regions are not necessarily adjacent; their total size can be less than 64 kB.
- Memory accesses are range- and type-checked; an exception is thrown when an invalid access occurs.
- There are two stacks: the operand stack and the return stack. Both of these stacks grow down. Both stacks post-increment when popped and pre-decrement when pushed.
- Stack operations are range-checked; an exception is thrown on stack underflow or overflow.
- Numbers and arithmetic are twos complement.

The Mr-Forth VM can be implemented on a wide variety of physical hardware. In general, the physical hardware will need at least 1 kB of RAM and roughly 32 kB of ROM. Regardless of the actual properties of the underlying hardware, the Mr-Forth program always operates in the basic execution environment described above.

The Mr-ForthVM is required to implement the 23 FORTH primitives in table 3. In this table, "addr" represents a 16 bit address, "char" represents a byte, "n" and its variants represent 16 bit signed or unsigned numbers, "u" and its variants represent 16 bit unsigned numbers, "bool" is a boolean with `0` being false and `65535` a.k.a. `-1` being true, and "xt" is an execution token. The sign bit is `0x8000`. The OVER, 0<, and 0= words may be defined as secondaries, if desired; however, doing so greatly reduces Mr-Forth's performance. For this reason, they are implemented as primitives. The (EMIT) and (KEY) primitives encompass all of the physical hardware dependencies in the Mr-Forth core.

An Execution token (XT) in the Mr-Forth VM is either a primitive opcode or the CFA of an existing secondary; all XTs are 16-bit integers. If the XT is less than the number of defined primitives, the machine code implementing that primitive is called; otherwise, the usual nesting operation is performed: the IP is pushed to the return stack and the XT is loaded into the IP. Executing the instruction stream involves fetching the 16-bit XT whose address is in the IP, incrementing the IP by two, and executing the XT as above. The EXIT primitive (which terminates every secondary) pops the return stack into the IP. The EXECUTE primitive pops an XT from the operand stack and executes it as above. As you can see, Mr-Forth is a standard threaded interpreter.

The (LITERAL) primitive performs the logical equivalent of

```
    : (literal) r> dup 2 + >r @ ;
```

Unfortunately, this operation must be implemented as a primitive since the definition of (LITERAL) itself requires the use of the literal `2`.

Finally, (ABORT) is used to both terminate the VM and raise exceptions within the VM, including memory access violations, stack overflow, etc. (ABORT) expects a code on top of the stack. If the code is zero, the VM immediately terminates; this is how BYE is implemented. Otherwise, if the '(ABORT) system variable is nonzero — as it is once THROW gets defined — the XT in '(ABORT) is executed when (ABORT) is called. If '(ABORT) is zero, a message is printed (on the PC platform, at least) and the program terminates. Note that this case only occurs while the core dictionary is being created; at runtime, THROW is used instead.

Certain exceptions, return stack underflow in particular, cannot be reliably handled by Mr-Forth code; nevertheless, all Mr-Forth exceptions are handled via this mechanism. The VM resets the stacks if needed to ensure that CATCH itself won't immediately raise an exception. Unfortunately, this means that CATCH's stack frames might be trashed. It is recommended that your main application use CATCH at its outermost level and execute something like the code in (PATCH) in `2800-outer.fth`.

The Mr-Forth VM implementation provided here is written in ANSI C; some care has been taken to ensure that the C runtime uses as little machine stack space (for activation records) as possible. It should be easy to implement the Mr-Forth VM in assembler (when needed or appropriate) because there are only 23 simple primitives to write. CPU- or system-specific primitives may also be added to the Mr-Forth VM to exploit the availability of special hardware, additional memory, available instructions, etc.

The `mrfgen.exe` program used to generate the core Mr-Forth dictionary implements 5 additional primitives to facilitate core generation; however, these primitives are *not* part of the runtime VM. The primitives are needed to bootstrap the environment and are presumed available (say, on a PC running `mrfgen.exe`) while the core dictionary is being built. A built dictionary takes the form of a C include file that looks something like this:

```
static MRFuint8_t MRF_EXE[32768] = {
    60, 199,  34,  99, ...
};
```

Once this dictionary file is generated, it can be included during the runtime VM build — independent of the hardware on which the runtime VM is to execute. In other words, a built core dictionary may be embedded into any target VM implementation in a "write once, run anywhere" fashion.

The next section describes the specifics of the Mr-Forth VM needed to implement the Mr-Forth core.

Mr-Forth core architecture

The Mr-Forth VM has a 64 kB address space as described earlier. In order to implement an actual FORTH runtime inside this address space, it is necessary to settle on a memory map. Table 4 depicts the map used by the supplied Mr-Forth interpreter. The locations and sizes of each of these elements may be configured at compile-time in the `src/mrfconfig.h` file; for concreteness' sake, we'll assume the default values shown in the table. While `mrfgen.exe` is building the core dictionary, the ROM is actually writable; the runtime VM itself imposes the read-only-ROM constraint.

As mentioned earlier, five extra primitives are defined by `mrfgen.exe` so that the Mr-Forth compiler can bootstrap itself. One of these, (PARAM), is used to expose various compile-time constants to the Mr-Forth core. The core creates a bunch of CONSTANTs corresponding to the available parameters; therefore, this primitive is not needed by the runtime VM. The next bootstrap primitive, EVALUATE, has no existence purpose the FORTH code. Both `mrfgen.exe` and `mrforth.exe` require an XT for EVALUATE to load source files specified on the command line. The `mrfgen.exe` version is implemented as a C primitive. When EVALUATE gets defined in FORTH, this vector, MRF_VIREXEBASE + 2, gets set to the FORTH EVALUATE's XT. The MrFLib C API uses this vector to provide its evaluation services. The other three primitives, all dictionary-related, are (WORD), (CREATE), and ('). (WORD) is the basis for WORD and TOKEN. It takes a delimiter, an address, and reads the next delimited word from the input stream, storing it at that address. Eventually, (WORD) gets defined in pure FORTH, eliminating the need for this primitive at runtime. (CREATE) expects a counted string located at HERE, encloses both it and a link field in the dictionary, stores the CURRENT vocabulary address in the link field, and updates CURRENT to point at the length byte. Again, (CREATE) is eventually implemented in FORTH, eliminating the need for this

#	Primitive	Operand Stack	Description
1	C@	(addr – char)	Fetch byte from address
2	C!	(char addr –)	Store byte to address
3	@	(addr – n)	Fetch word from address
4	!	(n addr –)	Store word to address
5	DROP	(n –)	Discard top-of-stack (TOS)
6	DUP	(n – n n)	Duplicate TOS
7	SWAP	(n1 n2 – n2 n1)	Swap top two stack items
8	OVER	(n1 n2 – n1 n2 n1)	Push item under TOS
9	>R	(n –) (R: – n)	Move TOS to return stack
10	R>	(– n) (R: n –)	Move top of return stack to TOS
11	+	(n1 n2 – n)	Add top two stack items
12	AND	(n1 n2 – n)	Bitwise AND
13	OR	(n1 n2 – n)	Bitwise OR
14	XOR	(n1 n2 – n)	Bitwise XOR
15	(URSHIFT)	(u1 – u2)	Unsigned right shift by one bit
16	0<	(n – bool)	Test sign bit
17	0=	(n – bool)	Test for zero
18	(LITERAL)	(– n)	Push literal word
19	EXECUTE	(xt –)	Execute XT
20	EXIT	(R: addr –)	Exit current secondary
21	(ABORT)	(code –)	Raise exception if code nonzero
22	(EMIT)	(char –)	Print character to user
23	(KEY)	(– char)	Receive keystroke from user (blocking)

Table 3: The primitives implemented in the Mr-Forth VM.

First	Last	Size	Description
`0xFFB0`	`0xFFFF`	80	Terminal input buffer (TIB)
`0xFEB0`	`0xFFAF`	256	Return stack
`0xFDB0`	`0xFEAF`	256	Operand stack
`0xE04A`	`0xFDAF`	7526	HERE; PAD is HERE+128
`0xE024`	`0xE049`	38	RAM used by core routines
`0xE000`	`0xE023`	36	RAM used by system variables
`0x8000`	`0xDFFF`	24kB	Empty
`0x0000`	`0x7FFF`	32kB	System ROM

Table 4: The Mr-Forth runtime memory map. ROM is 32kB starting at `0x0000`; RAM is 8kB starting at `0xE000`. The basic Mr-Forth core requires less than 1 kB of RAM.

primitive. Finally, (') is a "hard-wired" version of TICK and ultimately gets replaced with a pure-FORTH version. Although `mrfgen.exe` must also implement its own number parser, dictionary manager, and outer interpreter, direct access to these routines is not needed during the bootstrapping process: the bootstrap is the *result* of the execution of these routines. The Mr-Forth core implements its own version of them, but those definitions (mostly) aren't used until the runtime VM executes.

Except for (PARAM), these primitives are implemented without dictionary entries; instead, their XTs are poked into three system variables named '(WORD), '(CREATE), and '(') and the moral equivalent to the following definitions are created:

```
: (word)   '(word)   @ execute ;
: (create) '(create) @ execute ;
: '        '(')      @ execute ;

( usage examples for these words )

: token here (word) ;
: word  pad (word) ;

( colon isn't actually defined this way, but... )
: :     bl token (create) 1 state ! ;
```

Doing things this way makes it possible to reimplement these complex operations in FORTH with the goal of keeping the Mr-Forth runtime primitives simple. . . and fast.

It is anticipated that Mr-Forth will need to handle interrupts on certain embedded platforms. Though the details of interrupt handling are extremely CPU-specific, the Mr-Forth VM was designed with interrupts in mind: interrupts will only be enabled between primitives; i.e., the execution of Mr-Forth primitives will be atomic. This design greatly simplifies the VM implementation at the cost of requiring all primitives to be fast — in order to minimize interrupt latency. Of the VM primitives, 21 of 23 are so simple as to be downright silly; the hardware-dependent primitives, (EMIT) and (KEY), are only primitives because the PC platform running one of the usual modern operating systems requires that they be implemented in C. On an embedded system, (EMIT) and (KEY) could interface with ISRs, themselves possibly written in FORTH. In other words, implementing (EMIT) and (KEY) as primitives is a platform-specific requirement (or option).

The complete list of Mr-Forth system variables is given in table 5. With three exceptions, these variables have one thing in common: `mrfgen.exe` uses them to emulate the runtime VM. For example, (WORD) uses the input-buffer-related ones, the number parser uses BASE, etc. The variable (OUTER) is not used by `mrfgen.exe`. Instead, the runtime VM `mrforth.exe` loads the initial secondary XT to run from this location. The other two exceptions are (SP) and (RSP), the stack pointers.

FORTH systems typically use CPU registers for the stack pointers. Mr-Forth could follow this course this as well; however, it would then be necessary to code SP@, SP!, RSP@, and RSP! as primitives. These words are seldom-used and it seemed reasonable to expose them as system variables to avoid "primitive pollution." This choice has some impact on Mr-Forth's performance, of course, but laziness outweighed efficiency here. Since Mr-Forth is written in C, the compiler will likely use any hardware stack pointer for the C runtime anyway.

There is a final, hidden system variable called (RAMP) lurking around inside Mr-Forth. On "normal" von Neumann computers, FORTH code and variables live side by side in the same address space. In a ROM-based (or Harvard architecture) system, application code will live in ROM and variables in the physically separate RAM. Mr-Forth's answer to this is to have two dictionary pointers while `mrfgen.exe` is running: DP is the code dictionary pointer which points to the next available ROM slot, and (RAMP) is the RAM pointer and points to the next available RAM location. During the build, (RALLOC) adjusts (RAMP) much like ALLOT adjusts DP. At the end of the build, the usage-count and contents of RAM are copied to HERE, DP is set to (RAMP), and (RAMP) is set to the previous value of HERE. When `mrforth.exe` starts up, it reads the address stored in ROM at (RAMP), reads the count stored at that address, and copies

Offset	Name	Description	Smudged
0x00	DP	Dictionary pointer	N
0x02	STATE	Compiler state	N
0x04	CONTEXT	Search context	N
0x06	(IBLEN)	Input buffer length	Y
0x08	(IBLIM)	Input buffer maximum length	Y
0x0A	(IBUF)	Input buffer start	Y
0x0C	>IN	Input buffer offset	N
0x0E	(SOURCE)	Input source user/buffer flag	Y
0x10	BASE	Current number base	N
0x12	(OUTER)	Application start address	Y
0x14	(SP)	Mr-Forth VM operand stack pointer	Y
0x16	CURRENT	Compilation vocabulary	N
0x18	(FORTH)	FORTH vocabulary head address	Y
0x1A	'(ABORT)	ABORT execution address	Y
0x1C	'(CREATE)	(CREATE) execution address	Y
0x1E	'(')	Tick execution address	Y
0x20	'(WORD)	(WORD) execution address	Y
0x22	(RSP)	Mr-Forth VM return stack pointer	Y

Table 5: The Mr-Forth system variables.

that number bytes following that address into RAM. This allows all variables to be initialized to the values they had when `mrfgen.exe` terminated. This also allows new FORTH definitions to be stored in RAM. The define-time behavior of variables is controlled via FORTH code as follows:

```
: (svariable)                 \ build time variable definer
  cell (ralloc) constant
;

here 0 ,
constant '(variable)          \ create the vector '(variable)
' (svariable) '(variable) !   \ store (svariable) into vector
: variable                    \ now it looks normal
  '(variable) @ execute
;

: (variable)                  \ runtime variable definer
  create 0 , does>
;

( in 9999-finit.fth: )
' (variable) '(variable) !    \ change vector at end of the build
```

The Mr-Forth dictionary header format is pretty standard and is depicted in table 6. Mr-Forth names can be up to 31 characters in length; up to 8 characters are stored in the dictionary header. The length field precedes the stored name and serves double-duty by storing some per-word flags. The address of the name field is called the NFA. The link field follows the stored name and points to the previous definition's NFA; the link field's address is called the LFA. Vocabularies store the NFA of the most recent definition in that vocabulary; vocabulary searching is implemented by following NFAs and LFAs.

The per-word flags are displayed in table 7. If the smudge bit is set, the search mechanism will ignore

Field	**Bytes**
Length/flags	1
Name	1 to 8
Link	2

Table 6: The Mr-Forth dictionary header format.

Bits	**Description**
7	Smudge bit
6	Immediate bit
5	Primitive bit
0...4	Name length

Table 7: The Mr-Forth dictionary header length-byte format.

this word. The immediate bit works as expected. The primitive bit is a bit unusual (so to speak) and deserves a bit of explanation (again, so to speak). When a word like TICK executes, it is expected to return an XT. If a secondary is found, its CFA (immediately following the word's LFA) is returned. In "classical" FORTH systems, primitives also have a CFA which begins that primitive's machine code. Since Mr-Forth is VM-based, there *is* no machine code; instead, there's simply an opcode that the VM can interpret.

One of the jobs of `mrfgen.exe` is to create dictionary entries for every valid opcode "by hand" so that the VM primitives can be "ticked" and used in new definitions. The primitive bit, when set, tells TICK and friends that a primitive has been found. Instead of returning the CFA associated with the found header, the *contents* of the CFA are returned. `mrfgen.exe` stores each primitive's opcode in the CFA following the header. The opcode itself is stored in compiled definitions; the primitives only need headers so that the search machinery can determine their opcodes.

At the time of this writing, the Mr-Forth core dictionary is about 16 kB in size. Roughly one quarter of this space is used by the dictionary headers! I had considered making Mr-Forth be a target compiler in which the object code contained no headers at all. This would allow Mr-Forth code to run on smaller systems — no headers and no interaction-related core words would be needed. There is one major downside to implementing this: every word would require an extra two header bytes to store the word's CFA. This would add about 1 kB to the current core dictionary for interactive applications; I consider interaction essential during development so I favor keeping headers around. Mr-Forth is pretty small, but it's not *that* small. On a PC, the `mrforth.exe` text segment is roughly 11 kB; assuming a similar heft on embedded CPUs, 32 kB of ROM/flash is required. Shaving 4 kB off the dictionary won't get you down to a 16 kB ROM. To get *really* small, you need a true target compiler that not only strips out the headers, but also includes just the subset of the 500+ Mr-Forth words actually used by your application. This is certainly do-able, but whatever such a product is called, it isn't called Mr-Forth.

If we ignore the hardware-dependent (EMIT) and (KEY) primitives, only 21 primitives need to be implemented to get Mr-Forth running on a new platform; these primitives are very simple. It is possible to reduce this number to 18 by eliminating OVER, 0<, and 0= in favor of the definitions in table 8. In fact, the earliest versions of Mr-Forth used these exact definitions! The problem is that use of these high-level definitions slowed Mr-Forth to a *crawl*: the 0= primitive requires 168 primitive executions alone! Who cares? You do!

You may have noticed the absence of any branching primitives in table 3 — the fact is that all branches in Mr-Forth are implemented in Mr-Forth as secondaries based on (0BRANCH):

```
: (0branch) ( flag -- ) ( R: ra -- ra' )
  0= dup invert  ( z !z ) ( 0 => -1 0 // !0 => 0 -1 )
  r> tuck        ( z ra !z ra )
```

```
: over          \ 5 prims vs 1
  >r dup r> swap
;

: 0<            \ 76 prims vs 1
  2/ 2/ 2/ 2/
  2/ 2/ 2/ 2/
  2/ 2/ 2/ 2/
  2/ 2/ 2/
;

: tuck          \ 7 prims vs 3
  swap over
;

: abs           \ 87 prims vs 8
  dup 0< tuck + xor
;

: 0=            \ 168 prims vs 1
  abs 1- 0<
;
```

Table 8: Pure FORTH definitions of OVER, 0<, and 0=, showing the number of primitives executed as secondaries and primitives.

```
  cell+ and      ( z ra taddr|0 )
  >r @ and r>    ( faddr|0 taddr|0 )
  or >r
;
```

With OVER and 0= defined as primitives, (0BRANCH) requires 22 primitive executions. With OVER, 0<, and 0= defined in FORTH, this skyrockets to 193, almost a factor of nine! Adding these three primitives yielded a handsome speedup.

One could eliminate C@ and C! in favor of @ and !, but a lot of bit-banging would be required and the resulting words would be a lot slower. Since microcontrollers tend to have lots of 8-bit control registers, having fast byte-transfer primitives seems… wise. Eliminating @ and ! in terms of C@ and C! is pure madness.

Either of AND/OR could be eliminated in terms of the other given the existence or XOR (from which one constructs NOT); however, this is a marginal reduction and would adversely effect conditional speed. XOR is not for sale: it is used for various nefarious purposes throughout the core. You cannot do without +.

The primitives (URSHIFT), (LITERAL), (ABORT), and EXIT cannot be disposed of. It might seem possible to replace EXECUTE with something like ">r exit" but this won't work: EXECUTE runs the XT in TOS while EXIT loads RTOS into the IP. For secondaries either operation achieves the desired result; for primitives, the EXIT strategy will not work.

This leaves DROP, DUP, SWAP, >R, and R>. These could all be expressed in terms of (SP) and (RSP) with appropriate fiddling, but the performance cost is utterly devastating; i.e., I tried it.

The minimum conceivable FORTH VM size seems to require 10 primitives, not including (EMIT) and (KEY). They primitives are @ ! + AND XOR (URSHIFT) (LITERAL) (ABORT) EXIT EXECUTE. I cannot imagine a machine fast enough to run it! I think the Mr-Forth 21-primitive VM is a decent tradeoff between difficulty of implementation and runtime performance.

You can actually do it with only 9 primitives; see the file `forth/ucode.fth` for details. Note that this file is not used my the Mr-Forth core; instead, it is an illustration of what a nine-primitive bootstrap might look like.

Adding Primitives to the Mr-Forth VM

Adding primitives to Mr-Forth is fairly straightforward. Before getting into the details, let's have a look at the Mr-Forth C runtime API in table 9. The available system variables appear as the MRF_SV_<*name*> constants in the file `mrfvars.c`. The steps for adding a primitive to Mr-Forth are as follows:

- Create and populate a file with the primitive's C implementation
- Add this file to `APP_CSRCS` in the Makefile
- Include this file after `mrfos.c` in `mrforth.c` and `mrfgen.c`
- In `mrfoptab.h`, add the next available opcode for your primitive before MRF_NCORE_OPS is defined. Bump MRF_NCORE_OPS as well. It is important to use the next available opcode and use MRF_NAMEOP so that the profiler will behave correctly
- In `mrfgen.c`, add a call to mrf_makeprim() in mrf_makeprims() if your opcode is to have a dictionary entry. The `flags` parameter should be zero or MRF_FLAG_IMMED
- In `mrfrun.c`, add code to call your primitive in the mrf_runxt() switch statement
- Rebuild and test everything

To add a new system variable, create an MRF_SV_ constant for it in `mrfvars.c`, bump MRF_SVSIZE by 2, and add initialization code for it in mrf_sv_initialize() in `mrfgen/mrfsvtab.c`. All system variables are currently turned into FORTH constants in the `0300-defining.fth` file.

Notes on Porting the Mr-Forth VM

The contents of this section have **never** been tested – be advised! Below are my thoughts on porting Mr-Forth to microcontroller platforms, including working theories of how interrupt handling might work in Mr-Forth on these platforms. Again, this information is only a thought experiment!

In the simplest case, porting Mr-Forth basically involves a rewrite of `mrforth.c` and its dependencies. You should strongly consider keeping your port 100% compatible with the C implementation so that you can continue to use `mrfgen.exe` to create your FORTH core dictionaries. This requires preservation of the overall VM architecture and therefore a thorough understanding of the provided C implementation. Though Mr-Forth doesn't currently provide one, use of a VM test suite is strongly recommended: If the VM passes all tests, the FORTH core should run fine.

If the VM target language isn't C, you'll need to modify the end of `mrfgen.c` to emit the built dictionary in the target language. The output filename for the dictionary include file is MRF_EXE_FILE in `mrfconfig.h`. If your core includes target-dependent primitives, they should be stubbed appropriately for `mrfgen.exe` by wrapping the stubs in an `#ifdef MRF_SYSGEN...#endif` block. For extra performance, you might consider converting some of the Mr-Forth secondaries into primitives; if you do this, it is suggested that you make these changes on the "straight" PC version and test them before beginning your port as problems are much easier to resolve on systems with niceties like a monitor. Aside from these details, `mrfgen.exe` should be usable as-is.

Below we'll assume the worst case: you intend to port the VM to some sort of assembly language and therefore need a complete rewrite. The first decision you need to make is how to handle exceptions. The

Signature	**Description**
Memory Access	
MRFuint16_t mrf_cat(MRFaddr_t addr)	Fetch byte with sign extension
MRFuint8_t mrf_cbang(MRFaddr_t addr, MRFuint8_t val)	Store byte; returns val
MRFuint16_t mrf_at(MRFaddr_T(addr)	Fetch word
MRFuint16_T mrf_bang(MRFaddr_t addr, MRFuint16_t val)	Store word; returns val
System Variables	
MRFaddr_t mrf_sv_addr(svnum)	Return virtual address for MRF_SV_<*name*>
MRFuint16_t mrf_sv_get(svnum)	Fetch value of MRF_SV_<*name*>
MRFuint16_t mrf_sv_set(svnum, MRFuint16_t val)	Store value to MRF_SV_<*name*>; returns val
Operand Stack	
MRFuint16_t mrf_getsp()	Get SP
MRFuint16_t mrf_setsp(MRFaddr_t val)	Set SP; returns val
MRFuint16_t mrf_depth()	Get stack depth
MRFuint16_t mrf_tos()	Returns top stack item
MRFuint16_t mrf_stos(MRFuint16_t val)	Set TOS to val; returns val
MRFuint16_t mrf_push(MRFuint16_t val)	Push val onto stack; returns val
MRFuint16_t mrf_dupe()	Push TOS; returns TOS
MRFuint16_t mrf_pop()	Pop stack returning old TOS
MRFuint16_t mrf_drop()	Increment SP
MRFuint16_t mrf_over()	Push second stack item, returning new TOS
void mrf_swap()	Swap TOS and second stack item
Return Stack	
MRFuint16_t mrf_getrsp()	Get RSP
MRFuint16_t mrf_setrsp(MRFaddr_t val)	Set RSP; returns val
MRFuint16_t mrf_rdepth()	Get return stack depth
MRFuint16_t mrf_rpush(MRFuint16_t val)	Push val onto return stack; returns val
MRFuint16_t mrf_rpop()	Pop return stack returning old RTOS
Execution	
MRFaddr_t mrf_getip()	Get IP
MRFaddr_t mrf_setip(MRFaddr_t val)	Set IP, returning val
void mrf_runxt(MRFaddr_t xt)	Run an XT
void mrf_runsecs(MRFuint16_t count)	Run XTs until done
Exceptions	
void mrf_abrt(MRFuint16_t code)	Raise exception; code == 0 means “exit app”

Table 9: The Mr-Forth runtime API.

code in `mrfabrt.c` prints an appropriate message and terminates the program. Depending on the target platform, neither of these actions may be acceptable or even possible. You should think carefully about this issue.

The next thing to consider is the target implementation of VM memory. The core dictionary should live in an array in program memory, typically ROM or Flash, and the VM's RAM should live in a fixed array in the target's available RAM. You'll need to decide how to size VM RAM based on the available RAM, as well as the amount of RAM needed for VM globals, activation records, and the like. It will also be necessary to decide on virtual address mappings for code and RAM. Don't forget to modify `mrfconfig.h` to include the map so that the generated core will match the VM's expectations! Having made these decisions, porting the memory access operators C@ C! @ and ! should be fairly straightforward. With the memory accessors in place, you'll also need access to a few system variables to make the runtime work, specifically MRF_SV_ABORT, MRF_SV_OUTER, MRF_SV_SP, and MRF_SV_RSP, as well as an implementation of mrf_sv_copytab(). A minimal C main() for your VM (without exception handling) would look like

```
void main(void) {
  mrf_sv_copytab();           /*** initializes SP and RSP */
  mrf_setip(mrf_sv_get(MRF_SV_OUTER));
  while (1)
    mrf_runsecs(0);
}
```

If you choose to implement exception handling, it will appear in the target equivalent of main() and various other places throughout your VM.

With the MRF_SV_SP and MRF_SV_RPS accessors implemented, it should be fairly simple to write the stack accessors and 6 associated primitives DROP DUP SWAP OVER >R and R>. Have a beer: your port is almost halfway complete and the hard part is over!

The arithmetic and logical operators are instruction-set specific, but shouldn't pose any particular problems. In the arithmetic camp appear + AND OR XOR and (URSHIFT). The last of these is a unary operator, while the other 4 are binary. The unary logical operators are 0< and 0=. At this point, 17 out of 23 primitives are complete!

Aside from the platform dependencies discussed eariler, (ABORT) should look much like the C version. If your application uses CATCH and THROW to handle exceptions, and if "0 (ABORT)" shouldn't exit the application, the platform dependencies actually disappear.

EXECUTE, EXIT, and (LITERAL) will also probably look much like their C counterparts.

The two remaining primitives, (EMIT) and (KEY) may or may not be present, depending on the target's capabilities and the needs of the application. If they are in fact present, they may or may not be coded as primitives; this decision is left to the implementor.

One important ingredient remains: `mrfrun.c` must be ported to the target. The C function mrf_runxt() takes care of all the details of executing a primitive directly and nesting a secondary call. It can implemented in a variety of ways, but a table lookup is probably fastest. The mrf_runsecs() loop can be unrolled under main() if desired; it is a simple IP-banging, mrf_runxt()-calling loop.

One last issue remains: on non-PC systems, the VM will need to handle interrupts. If your application doesn't need to deal with interrupts at the FORTH level, interrupt handlers will be handled in the target language. Any FORTH-bridge can be handled by having your FORTH code poll variables and the like. In this case, creating new FORTH system variables is probably the cleanest way to handle the bridging. The handling of critical sections is application dependent. Assuming that you need ISRs written in FORTH, there are three problems to solve: getting FORTH code to be executed safely, getting FORTH code to be executed at all, and getting the FORTH application and ISR to play well with one another. The safety issue can be addressed by modifying main() and mrf_runsecs() as follows

```
void main(void) {
  cli();                     /*** disable interrupts */
  mrf_sv_copytab();
  mrf_setip(mrf_sv_get(SV_OUTER));
  while (1)
    mrf_runsecs();
}

static void runsecs(void) { /*** simplified code */
  MRFaddr_t i;

  /*** NB interrupts are disabled here! */
  i = 1;
  while (i) {
    sti();                    /*** enable interrupts */
    /*** interrupts can only occur right here! */
    cli();                    /*** disable interrupts */
    i = mrf_getip();           /*** load IP here in case it changed */
    mrf_setip(i + 2);
    mrf_runxt(mrf_at(i));
  }
  /*** NB interrupts are disabled here! */
}
```

With this setup, interrupts can only occur *between* VM instructions; this means that all VM opcodes execute atomically. Again, it is important to make sure that all primitives execute quickly in order to minimize interrupt latency. Since interrupts occur at a well-defined point, the VM implementation is guaranteed to be in a consistent state when the ISR fires and prods the VM to execute FORTH code.

Getting a FORTH ISR to run involves making an interrupt handler tell the VM to begin executing a FORTH handler. Given the Mr-Forth VM design, the following assumptions are made:

- The FORTH ISR shares the operand and return stacks with the application
- When the ISR exits, the stacks are restored to their pre-ISR state; in particular, CATCH and THROW must be used to handle any ISR-related exceptions. Alternatively, it must be guaranteed that ISRs will never generate VM exceptions
- The FORTH ISR shares RAM with the application; therefore, the FORTH programmer must be aware of the reentrancy properties of both core and application words used by the ISR. Note that certain core words are not reentrant!
- The application can handle the fact that ISRs may execute asynchronously at any point in the FORTH application

To summarize, when a FORTH ISR fires, it is as if the ISR were *called* by the application at the moment the ISR executes. The mechanics of having an ISR execute inside the VM can be described in C pseudocode as follows:

```
/*** one way to do it */
MRFaddr_t *vectors = &MRF_EXE[4];

/*** each possible interrupt handler calls this: */
void do_isr(int interrupt_num) {
  mrf_runxt(vectors[interrupt_num]);
  /*** possible special return-from-interrupt */
  reti();
}
```

```
/*** another way to do it */
MRFaddr_t *vector = &MRF_EXE[4];

/*** each possible interrupt handler calls this: */
void do_isr(int interrupt_num) {
  mrf_push(interrupt_num);
  mrf_runxt(*vector);
  /*** possible special return-from-interrupt */
  reti();
}
```

The actual setup of the CPU interrupt handlers is completely platform-specific. Note that you'll certainly need to provide either primitives for banging on physical hardware registers or add mappings for these registers to your virtual memory map.

What's with the "&MRF_EXE[4]" bit? When `mrfgen.exe` runs, the (RAMP) system variable is stored in MRF_VIREXEBASE a.k.a. &MRF_EXE[0]. At the end of the build, (RAMP) is repurposed as described earlier so the the runtime VM's mrf_sv_copytab() can initialize the system's RAM. There's a pointer to EVALUATE stored at MRF_EXE[2] as well; this allows `mrforth.exe` to execute user code before running the main application.

This reservation of MRF_VIREXEBASE is achieved in the function mrf_sv_initialize() in `mrfgen/mrfsvtab.c` where DP is set to MRF_VIREXEBASE + 4. If you want to put your FORTH interrupt vector or vector table in ROM, you can bump this DP initialization up to make room for the vector or table — whose start address is &MRF_EXE[4]. If you place the table in ROM like this, you must of course initialize those entries from FORTH code via `mrfgen.exe`. The proper way to perform this initialization is to

```
(exe_base) cell+ constant isr_vectors

: myhandler
  ( whatever )
;

' myhandler isr_vectors 0 cells + !
' myhandler isr_vectors 1 cells + !
( etc )

( might want to: )
smudge isr_vectors
smudge myhandler
```

If you might need to change ISR handlers at runtime, you can store them in RAM, too. One approach is to create a system variable vector or table; another is to allocate a table in RAM and store its address in a system variable or in ROM.

Based on what we've done so far, we're able to have FORTH code service interrupts. Invariably, the FORTH ISR will communicate with the application through a set of variables. This means that we've got to avoid race conditions. Addressing this issue is the last piece of the ISR puzzle. There are at least a couple of ways to proceed.

If a multitasked FORTH is on your agenda, you might focus on implementing semaphores. The minimum required facility is something akin to a POSIX threads mutex. To make this work you need a primitive that does something like this (there are many ways to do it):

```
( atomic test-and-set )
: tas  ( addr -- addr old-val, set to -1 and return old value )
```

```
    dup @ over -1 swap !
  ;
```

This must be implemented as primitive so that it executes atomically; a sample implementation looks like this:

```
  void op_tas() {
    MRFaddr_t a = mrf_tos();

    mrf_push(mrf_at(a));
    mrf_bang(a, -1);
  }
```

With this simple primitive, we can now implement our mutex in FORTH as follows:

```
  : unlock ( addr -- , unlock mutex we ALREADY own )
    0 swap !
  ;

  : locked ( addr -- bool, is mutex currently locked? )
    @
  ;

  : lock ( addr -- , lock mutex -- BLOCKING )
    begin
      tas while
    repeat drop
  ;

  : trylock ( addr -- bool, true if we own it )
    tas nip not
  ;

  ( EXAMPLE CODE )
  variable mymutex

  : do_critical
    ( modify data shared with ISR )
  ;

  ( everyone must use this word: )
  : safe_do_critical
    ( acquire mutex for critical section )
    mymutex lock

    ( be sure and catch all exceptions! )
    'compile do_critical catch

    ( release the mutex )
    mymutex unlock

    ( propagate any exceptions )
    throw
```

```
;

: myword
  ( ... )
  safe_do_critical
  ( ... )
;
```

There is one last important rule that FORTH ISRs must follow: they must **never** use the word `lock` defined above. In general, ISRs should never block — because the only thing capable of unblocking them is another ISR! If the application has a mutex locked and an ISR tries to acquire the mutex with a blocking lock call, the entire application will be deadlocked! ISRs should instead use `trylock` and take appropriate action. If the mutex is busy, the ISR will probably ignore the interrupt because there is no way (in this interrupt handling model) for the ISR to temporarily return control to the application (for the purpose of unlocking the mutex) and regain control once the mutex is available. One consequence of this is that an ISR can assume that it will not be interrupted by the application. Despite these limitations, use of suitable data structures like double-ended queues permits the implementation of interrupt-driven serial I/O from which one can build (EMIT) and (KEY) routines at the application level.

The possibility of nested interrupts and interrupt prioritization is, again, completely platform dependent. Though it is always possible to prevent interrupt nesting in software, doing so prohibits exploitation of the benefits nested interrupts can provide. For maximum application portability, one should always assume that the program's ISRs can — and will — be interrupted by other ISRs.

A completely different approach to avoiding race conditions is to have the application itself control the CPU's interrupt enable bits. To do this, mrf_runsecs() would be modified as follows:

```
/*** interrupt disable count */
static unsigned long int_dis = 0;

static void mrf_runsecs(void) {
  MRFaddr_t i;

  /*** NB interrupts are disabled here! */
  i = 1;
  while (i) {
    i = mrf_getip();
    mrf_setip(i + 2);
    mrf_runxt(mrf_at(i));
    if (!int_dis) {
      sti();
      /*** interrupts occur here */
      cli();
    }
  }
  /*** NB interrupts are disabled here! */
}
```

and implement primitves CLI and STI. The first would increment int_dis and the second would decrement it. At the FORTH level, your code would look like the mutex example, except that the LOCK and UNLOCK calls would be replaced with calls to CLI and STI. Again, use of CATCH/THROW is mandatory, lest interrupts be disabled forever! On a side note, it is possible to implement "locked" words with something like

```
: :locked
```

```
  ( start definition )
  :
  ( disable interrupts )
  compile cli
  ( push xt as literal )
  compile (literal)

  ( save room for xt )
  here 0 ,

  ( execute, enable interrupts, reraise )
  compile catch
  compile sti
  compile throw

  ( done with wrapper )
  [compile] ;

  ( set xt slot above )
  :noname
  swap !
;

variable shared
:locked myword
  42 shared +!   ( +! is a complex operation )
;
```

The main disadvantages of controlling the interrupt enable bits from FORTH are that interrupt latency is obviously greatly increased and programming mistakes are potentially harder to track down. For example, if a locked word indirectly calls KEY, the application will hang: KEY is a blocking call and the ISR that services KEY will never get to execute!

Credits

- The book "Starting Forth" is available online at `http://home.iae.nl/users/mhx/sf.html`
- The book "Thinking Forth" is available in the file `thinking-forth-color.pdf` from `http://thinking-forth.sourcef`
- The file `pfe-fth.chm` can be found in the Portable Forth Environment distribution at `http://pfe.sourceforge.net`
- eForth is available at `http://www.baymoon.com/∼bimu/forth/`
- Much of the Mr-Forth core us inspired by pForth available at `http://www.softsynth.com/pforth/`