Emulating Forth: Interactive Cross Development.

Dr.-Ing. Egmont Woitzel
University of Rostock, Department of Electrical Engineering
Institut for Applied Microelectronics and Computer Science
Richard-Wagner-Straße 31, D-18051 Rostock, Germany
Phone +49 381 4983538
Fax +49 381 4981126

Abstract

The power of the Forth programming language is heavily based on its self-extensibility. There is no significant difference between the elsewhere strictly distinguished process of compilation and the run-time of a user program. Due to the incremental nature of compiling Forth code uncertain parts of the underlaying Forth system and of the currently compiled user program are executed either at compile-time or run-time or both.

Unfortunately this simple design doesn't work in cross development environments, because the compiler and target system are running at different machines, or the target system exists only inside the memory of the compiler system.

This paper describes a new approach to solve the problems caused by the necessity to execute parts of the user code at compile-time. Most of the known cross compiler designs were avoiding the execution of target words by the execution of compiler words of a specialised and limited word set instead of the execution of the target word itself. This technique is useful but has some drawbacks mainly caused by doubling user source code used at both compile-time and run-time.

It will be shown that it is much more efficient to emulate the underlaying virtual Forth machine instead of words. Furthermore it will be demonstrated how this approach allows to remove most of the differences between typical cross compiler environments and interactive remote target environments.

Forth Compilation: Compiling by Interpreting

If you are talking with a computer scientist, who does not know Forth at all, sooner or later arises the question if it is an interpreting or compiling language. Most often than nor it will be a serious problem to explain that the user interface consists of an interpreter but the execution speed of the Forth code is comparable to that of compiler languages. The remaining confidence will be blown away if you add, that the code density of Forth is usually higher than that of other compiled languages. It seems that all Forth programmers know all these things but it is almost impossible to believe it without the background of using Forth.

Surprised I have experienced analogous difficulties when I have tried to explain the nature of Forth cross-compilers even to advanced Forth programmers. Today I believe that both problems in explanation have the same roots. For that reason this paper starts with a short repetition of some well-known facts about Forth compilers in general trying to find a more common language to be understood by the majority of computer scientists.

First of all, the use of the terms *interpreter* and *compiler* is different between Forth programmers and the rest of computer scientists. Usually the term *compiler* designates a software program which translates source code, written in a formal computer language, into executable code for a computer. Traditionally, Forth programmers use this term only for one particular kind of source code translation instead, namely that between colon and semi-colon. Outside colon and semi-colon the system, or exactly spoken the text interpreter, is called to be in the *interpretation state*. This terminology wrongly implies for most non-Forth-programmers that in this state no executable code will be generated. Fortunately the ANS Forth standard clarifies most of these confusions by its chapter about definition of terms. Anyway, we have to consider that *compilation* does not exclusively occurs when the text interpreter is in its *compilation state*. Due to the dual nature of Forth code, which is focused later on, we will find two general methods to generate code involved in program execution. Additionally to the process of "extending the current definition by the execution semantics [… of a given word]", as the ANS Forth standard describes the default compilation behavior of the text interpreter, the term *compile* also includes the *definition* of words. Furthermore, *defining* stays for the interpretation of defining words as well as the interpretation of all words in the source code which are involved in constructing a new definition.

The second problem in understanding Forth compilers is to accept their extensibility. Only a small number of other languages are able to extend its set of predefined keywords or able to change their semantics by using keywords of exactly this set. The most simple approach to allow this kind of extensibility was and is the implementation of a Forth compiler by using an interpretative technique. This allows a growing set of keywords during compilation. in my experience the most simple way to explain to computer scientists why Forth systems have an interpreter and generates and executes compiled code is to say the Forth compiler is implemented as an interpreter.

Let's have a look at some basic stuff that is used in almost every Forth program, which is in the same manner nearly impossible to translate in mainstream languages. We will start with a simple compiler extension, the `DEFER-IS` construct well-known since Laxen/Perry's F83. The source code shown in Figure 1 isn't an ANS standard program, but should work under a large numer of ANS standard systems. It assumes that the nest-sys parameter of the colon, introduced by ANS, is a pointer to the next instruction, and that the code compiled by the text interpreter in its compilation state will be stored in a common memory space with data items. The source code is neither very tricky nor complete, but it should only demonstrate four different types of colon definitions, all of them involved in extending the compiler.

```
\ --- Compiler Extension Sample: Laxen/Perry like DEFERs

: DUMMY ( ds: -- )( does nothing ) ;

: (IS) ( ds: xt -- )( ip: ’defer )( assigns xt to defer )
        R> DUP CELL+ >R @ ! ;

: [IS] ( C.ds: -- )( C.ib: <spaces>name )( R.ds: xt -- )
        ( compiles run-time assignment to name )
        POSTPONE (IS) ’ >BODY , ; IMMEDIATE

: DEFER ( ds: -- )( ib: <spaces>name )( defines name )
        CREATE [’] DUMMY , DOES>
        ( ds: xxx -- yyy )( executes current assignment )
        @ EXECUTE ;
```

Figure 1: Compiler extensions by words with execution-only semantics, compilation-only semantics and defining words

The word `DUMMY` is part of the compiler extension only in that way, that its existence is necessary for definiting and using of the defining word `DEFER`. The word `(IS)` has only at run-time a determined behavior, because it expects an inline parameter within the compiled code which isn't provided during interpretation. The complementary word `[IS]` has only compilation semantics instead, it appends the execution semantics of `(IS)` to the current definition. The word `DEFER` defines a whole new class of words which are changeable in their execution behavior. All four words build together a powerful extension of the Forth compiler.

Usage is demonstrated in the next source code sample, referring to Figure 2, where we additionally find a number of other typical actions, which Forth programmers let be happen during the compilation of source code, namely the use of words defined in the same source code to calculate something during compilation and to run some words to initialise data structures.

```
\ --- Compiler Extension Usage Sample: Rounding

: 10** ( ds: +n1 -- +n2 )( calculates decimal exponential )
        1 SWAP ?DUP
        IF  0 DO 10 * LOOP  THEN ;

2 ( digits ) 10** CONSTANT ROUND-RANGE ( ds: -- u )

: FLOORED ( ds: n1 -- n2 )( round towards greatest value less or equal )
        ROUND-RANGE / ROUND-RANGE * ;

: NEAREST ( ds: n1 -- n2 )( round towards nearest value )
        ROUND-RANGE 2/ + FLOORED ;

DEFER ROUND ( ds: n1 -- n2 )( rounds with variable strategy )

: SET-FLOORED ( ds: -- )( activates ’floored’ strategy )
        [’] FLOORED [IS] ROUND ;

: SET-NEAREST ( ds: -- )( activates ’nearest’ strategy )
        [’] NEAREST [IS] ROUND ;

SET-FLOORED ( initialise rounding strategy )
```

Figure 2: Defining words by direct and/or indirect execution of compiler extensions

The question of sense concerning the applied numerical rounding in the sample should not be discussed here. Much more important is, that even this very straight-forward programmed sample contains some critical sequences if you are trying to port it to a cross environment.

Compiler Systems and Target Systems

The design of a Forth compiler as described above is quite simple as long as you want to compile code for the own Forth system. Unfortunately there are a number of situations where this approach does not work, because you have explicitely to compile code for another Forth system than to the running one. We have to distinguish between *compiler system* and *target system*. Such a compiler, which compiles code for a different system, usually will be called a cross compiler.

It is obvious to recognize that you have to use a separate compiler if you want to build a new Forth system up from the scratch. If there is anything you start with, you have to use words of a separate compiler to define words, perform compile-time calculations, compile code and so on. This situation is at least well-known by vendors who are building their own interactive Forth systems by a so-called metacompiler.

On the other hand in the field of embedded control there are often very strong restrictions in the computing power and the available memory size due to cost aspects, system size and power consumption which makes it in fact impossible to use the targetted computer system as a development platform. Hence you have also to use a different Forth compiler to create the application code as in the situation described above.

To find out the way a compiler for such a type of Forth system has to be designed let’s have a look at our previous source code example. The interpretative implementation of the compiler allows to execute uncertain portions of the user defined code at compile-time. In fact, there is no syntactical hint, whether a currently defined word will be executed at compile-time or not. If it calls `DOES>` or if it will be marked to be `IMMEDIATE` isn’t a sufficient information to correctly answer this question. It can only be answered after a complete compilation of the source text. Furthermore, the execution of a given word at compile-time doesn’t exclude that it will be required to run at run-time, too. Table 1 shows that there are some non-immediate words executed at compile-time as well as words executed at run-time only as well as words executed at compile-time only.

word name	executed at compile-time [1]	executed at run-time [2]
DUMMY	referenced [3]	
(IS)	indirect	indirect
[IS]	direct	
DEFER	direct	
10**	direct	
ROUND-RANGE	indirect	indirect
FLOORED	referenced [3]	indirect
NEAREST	referenced [3]	indirect
ROUND	referenced [3]	direct
SET-FLOORED	direct	direct
SET-NEAREST		direct

Table 1: Required execution of words defined in Figure 1 and Figure 2

At this point we have to make a critical decision. With the restrictions of embedded systems in mind we have to ensure, that only words executed at run-time will appear in the target system. All other words should exist at compile-time only. On the other hand, to ensure the correct compilation of our source code sample, some words of the target system must be executed at compile-time, too. One simple solution is to compile all required definitions into the compiler system. As we have seen above, there is no simple algorithm to decide whether a currently defined word will be executed at compile-time or not. So the programmer has to do the job of doubling and adapting all the required source code.

To avoid that kind of source code doubling the compiler should provide the ability to execute user definitions. Furthermore, the compiler has to use the memory addressing and stack model of the target system when it generates code for it.

Non-Living and Living Target Systems

On designing a cross compiler the most important fact is whether or not the target system is able to execute its words itself. In some circumstances it is possible, for instance if the target system is linked through a serial communication line and controllable by the compiler. This type of system will be referred as *living* target system and is well-known from a number of publications [1], [2].

In most other cases the target system is not able to execute its own definitions, mostly because it is existing only as an image inside the memory of the compiler system, or the target system uses a different CPU type. Such a type of system will be called *non-living* target system.

Because of the different target capabilities different compilers are necessary to co-operate with both types of target systems. Additionally, in the case of living target systems no source code doubling is necessary. Unfortunately not all problems can be solved with this kind of system. At least the start-up nucleus has to be compiled using a more metacompiler like system.

To provide a homogenous programming environment the differences between both types of target systems should be as slight as possible. The most simple approach to achieve a comparable system behavior is to enable the compiler to execute definitions of a non-living target system by emulating them. The source code doubling

[1] follows the assumption that none of the words defined here will be executed later during compilation

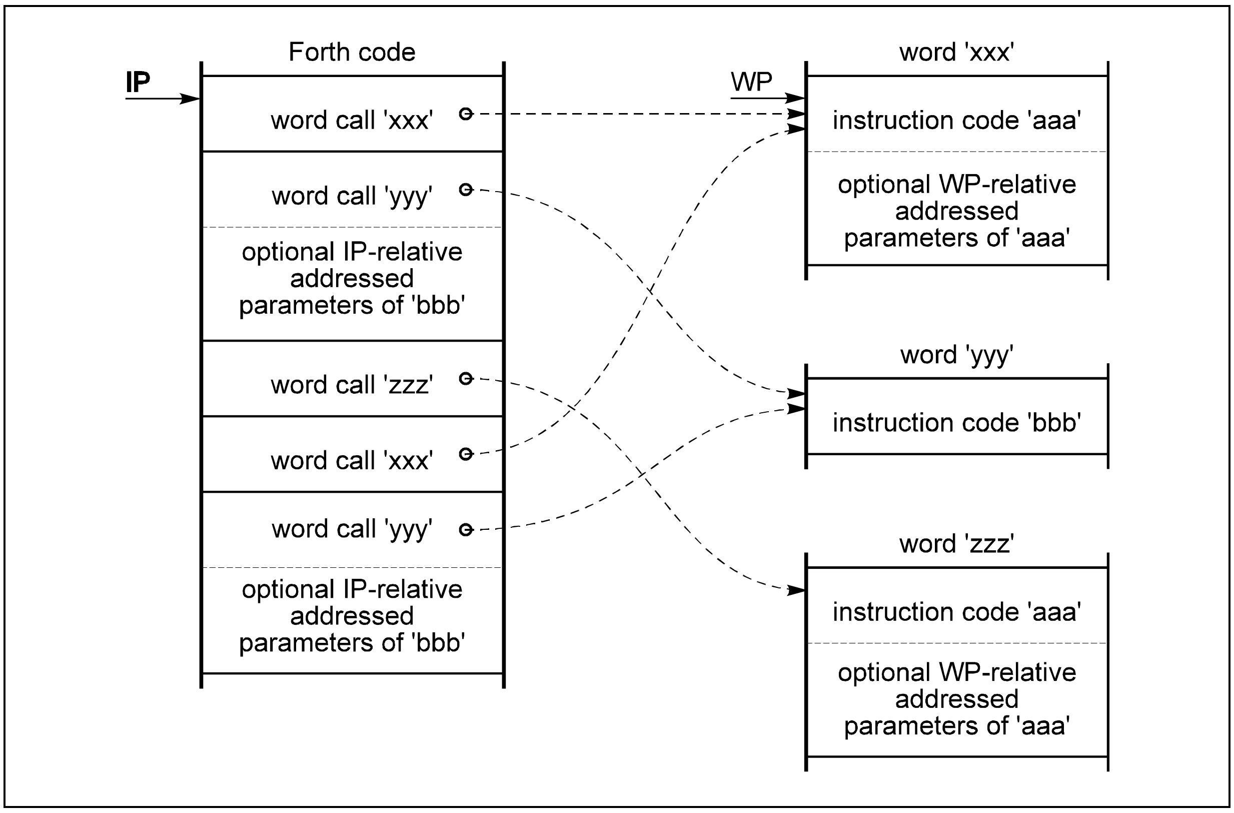
[2] follows the assumption that the 'user interface words' ROUND, SET-NEAREST, and SET-FLOORED were directly called by later defined user words

[3] the term referenced means that the existence of the word is required, but the word itself will not be executed

described above is one simple kind of emulation. But the word level isn't the most powerful level of emulation. It's much better to emulate the complete Forth engine instead of words.

Modelling the Forth Engine

Before the discussion of how to emulate a Forth engine let's have a closer look at the structure of Forth code. Figure 3 is a snapshot of one part of a dictionary of a hypothetically Forth system. Due to the dual nature of Forth-compilation we will find defined words and compiled code, spoken in Forth terms, or compiled two-staged threaded code, spoken in more computer scientists like terms.



Forth code
IP
word call 'xxx'
word call 'yyy'
optional IP-relative addressed parameters of 'bbb'
word call 'zzz'
word call 'xxx'
word call 'yyy'
optional IP-relative addressed parameters of 'bbb'
WP
word 'xxx'
instruction code 'aaa'
optional WP-relative addressed parameters of 'aaa'
word 'yyy'
instruction code 'bbb'
word 'zzz'
instruction code 'aaa'
optional WP-relative addressed parameters of 'aaa'

Figure 3: Structure of Forth code using a code-word-instruction model

The main difference between typical CPU architectures and a Forth engine is the direct maintainance of the word encapsulation mechanism. The code of non-Forth engines typically consists of instruction codes and optionally instruction pointer (IP) relative addressed parameters, access provided by the immediate addressing mode. In difference to that, the code of a Forth engine consists of word calls. The instruction which has to be executed on a word call is coded inside the word. This enables the Forth engine to use an additionally addressing mode, provided through an additional word pointer (WP) register which will be automatically adjusted during a word call. The WP relative addressing mode reveals one aspect of the mystery of Forth code density. This kind of addressing allows to reuse constant IP relative addressed parameters.

More important to the topic focused here is the fact, that emulating instructions instead of words will be an alternative approach to emulate a Forth engine. Therefore we have to answer the question, when instructions will be defined. Usually instructions of the Forth engine have to be implemented in the machine code of the target systems. Most instructions occuring in a standard Forth system are using stack relative, stack indirect and IP relative addressing modes only. Without any WP relative addressed parameters they were usually defined as primary words using the `CODE name … END-CODE` construction. Instructions which are using WP relative addressed parameters are more difficult to identify. They will be implicitely implemented inside defining words using `;CODE` or `DOES>`.

Even if the model shown at Figure 3 looks like an indirect threaded code system it is applicable to all implementation variants of Forth code. It doesn’t care if word calls and/or instruction codes are implemented as tokens, pointers or machine code.

The most important advantage of emulating instructions instead of words is based on the fact, that there are significantly less instructions than words in a Forth system. The source code sample described earlier defines only one additionally instruction using the `DOES>` construct inside the definition of `DEFER`. That’s why less emulation code is needed to emulate instructions than to emulate words.

Emulating Non-Living Targets

Using the Forth engine model described above its emulation will be quite simple. The programmer has to provide an emulation for every instruction. Therefore the compiler has to provide an emulation language which gives access to IP relative, WP relative, stack relative and stack indirect addressed parameters. During emulation the compiler has to use a control procedure which is able to decode word calls and instruction codes.

The complexity of the decode procedure is extremely dependent from the chosen Forth code scheme. Especially if a native code approach is chosen, you will often find additionally code optimization which replaces word calls by inline code together with optimised parameter passing using CPU registers. In such cases it will be necessary to keep track of the original code by generating shadow code which is used during emulation.

Another difficulty occurs if the target code may contain forward references as necessary during metacompilation. The compiler has to enable the emulation of instructions which aren’t defined yet and it has to decode all references to words which aren’t defined yet. At this point it may be useful to provide both, emulation of instructions and emulation of words. This enables the complete emulation of Forth code which contains forward references to words not defined yet.

Practising Theory: The fieldFORTH Cross Environment

The cross compiler design described above has been proven to be functional yet. The cross development environment fieldFORTH, offered by the FORTecH Software GmbH, represents a Windows 3.1™ based cross compiler system which is intended to develop embedded software. It is able to co-operate with living target systems as well as non-living target systems. Figure 4 illustrates the structure of the system. The fieldFORTH shell contains the true compiler and serves as the user front-end. The so-called emulator can be used to build tailored Forth nucleus systems from scratch on. It is able to execute target definitions by using a instruction oriented emulation engine as outlined above. The component called adapter connects a living target system to the shell, allowing on-line debugging. The differences between non-living and living target systems remain encapsulated inside the emulator and the adapter respectively. All symbolic information not stored inside the target system will be transferred from emulator to adapter during deveolpment. A more detailed description of the fieldFORTH environment may be found in [3].

The compiler extensions presented at Figure 1 needs only tiny modifications to become acceptable by the fieldFORTH Shell, the usage sample shown at Figure 2 will be accepted without any changes.

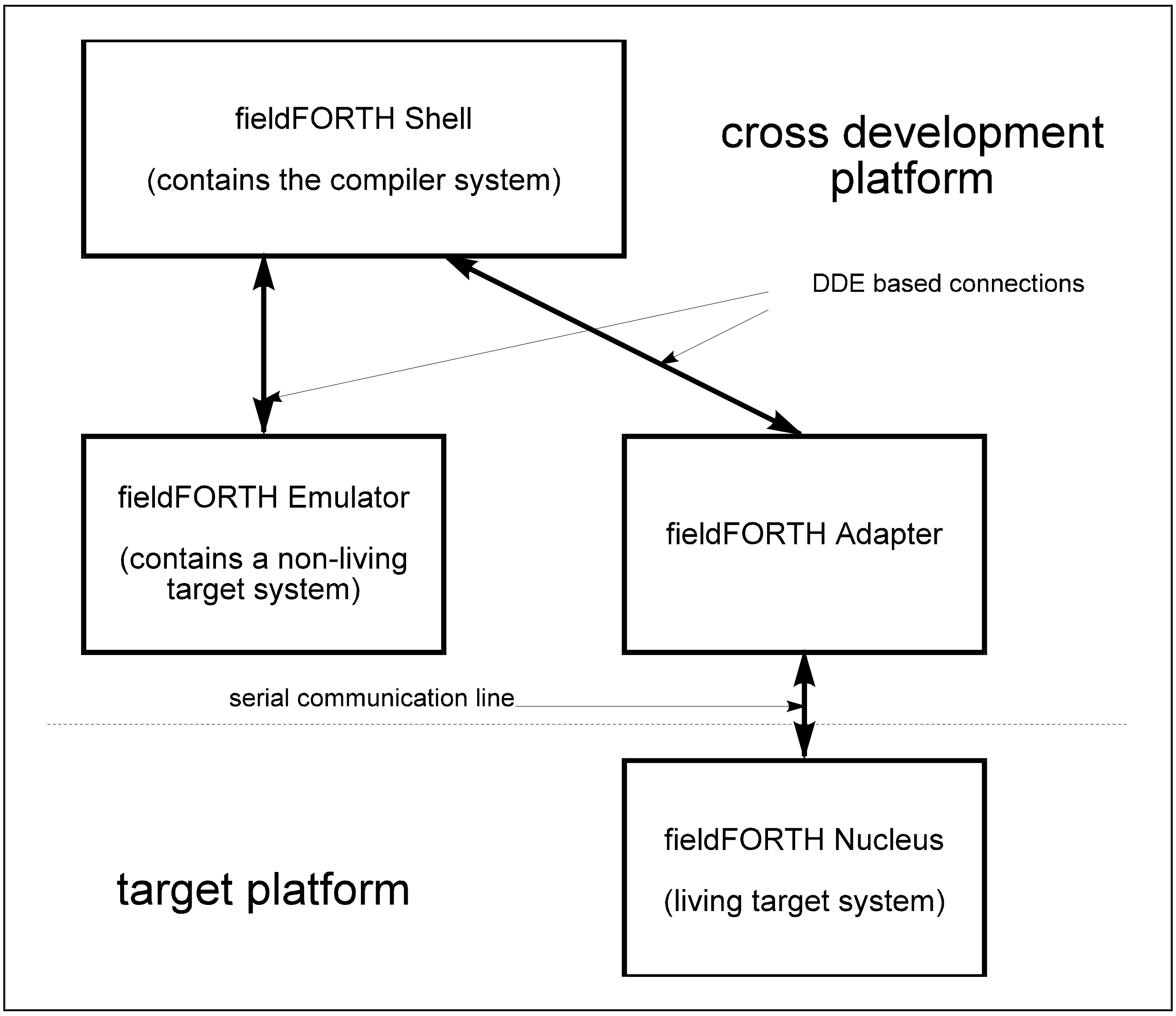


Figure 4: The components of the fieldFORTH cross environment

Remaining Problems

The introduction of an emulation component solves most of the problems connected with the execution of target code during compilation. However, many other problems in the field of cross compilation technology remain still unresolved. First of all, no common syntax exists to extend the compiler system by the means of defining words, necessary for compile-time only extensions like record or arry contructs, which causes syntactical differences between source code destinated to a cross system or to a standard Forth system respectively.

References:

[1] Robertson

[2] Woitzel, Egmont:
EuroFORML '90

[3] Woitzel, Egmont:
Forth '95