Porting Forth Applications and Libraries

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8631 441
f: +44 (0)23 8033 9691
e: sfp@mpeforth.com
w: www.mpeforth.com

Abstract

Porting Forth applications between host Forth systems is less difficult than many people assume. It just requires discipline, management, preparation and a minimal ego. This paper discusses how to perform a successful port with reference to several ports I have been involved with. Some guidelines for both producers and consumers are suggested.

Introduction

Successful code tends to have a long lifetime and to evolve over time.

The MPE Forth cross compiler has evolved from a code base originally supplied in 1982. Although hardly a line of code from the original sources has survived unchanged, the code is a direct descendant of the original.

The Candy Construction Project Modelling and Project Control software from Construction Computer Software (CCS) in Cape Town is approaching one million lines of Forth source code dating from the early 1980s. Over that period it has been ported between two CPU architectures (M68000, i386) and four operating systems (HP98xx, DOS16, DOS32, Windows) as well as several Forth hosts from several suppliers.

Hanno Schwalm recently ported his fJACK audio interface to VFX Forth. The code base now runs on iForth and VFX Forth under both Windows, Linux and OSX where the host supports it.

Bernd Paysan is currently porting his Minos/Theseus GUI design tools from the original BigForth host to VFX Forth for Linux.

Brad Eckert provided the FAT file system used with the MPE Forth cross compilers.

The Forth Scientific Library (FSL) has been ported to many different Forth systems and was probably the first example of a significant Forth source code library being widely ported. Its success is in no small measure due to the assumption that library code **should** be ported. The FSL uses the same harness approach as is discussed here.

The issues involved in porting Forth source code are very little different from those in porting source code in any other language. My observation is that standards have helped a great deal in making Forth source code more portable.

Why do the port?

The motivations for porting Forth code depend heavily on whether the code is an application or a library.

Maintainers of applications usually see themselves as tool **users** rather than tool **makers**. Although this line is blurred in large applications, many companies producing applications see the compiler as outside their core competency. Why should a manufacturer of construction planning software or mass spectrometers write Forth compilers? Using a third-party compiler enables then to take advantage of improvements introduced for all users rather than just those required by the application. In turn, if their current Forth system does not support a particular operating system, e.g. Intel OSX, the authors can turn to a Forth host that does support that operating system.

Source code libraries gain by widespread use. You don't reinvent the wheel, you use existing code. You don't waste time learning the details of a web protocol, an audio interface or a GUI, you use existing tools and get your job done faster. Successful libraries influence standards and make other peoples lives easier. Successful use of libraries enables you to do more with your time.

Why not do the port?

The main reason for not doing a port, either from one system or to another, is that you need your code to be smaller or faster. In the desktop world, size is no argument for any Forth application I know of, and both compilers and PC hardware improve over time. The speed/space reason is often advanced in the embedded systems world as it would force a hardware change. In many instances this is a fallacious argument.

The majority of embedded Forth applications are produced in volumes of less than 10,000 units per year. Changing from a 8/16 bit CPU with 60kb of Flash and 4kb of RAM to a much faster 32 bit CPU with 512kb Flash and 64kb of RAM and vastly more peripherals will cost in the range of 1 to 2 dollars/EU/pounds, and greatly extend the lifetime and potential features of the product. The additional hardware cost is easily saved in reduced software development costs.

MPE has had several clients who have stayed with what they know, only to come back five years later saying that they now need to change and that the five years have been very expensive.

When should I port?

Most people port code to another platform when they need to make a step change in the capability of the application.

Programmers of desktop applications may need to move to a new platform or use a feature or library that is unique to a particular Forth host. Embedded systems developers make the change when they run out of memory on an 8/16 bit system or need facilities such as USB, file systems or TCP/IP stacks. Don't even **think** about bank switching – it will cost you a fortune!

Whatever the reasoning, the decision to port must be considered and the porting process managed.

Process

The successful application ports that I have been involved with have all followed a similar process. Library ports to a new host follow the same basic process.

1) Preparation – eliminate host specific code and/or move it to a host-specific harness.
2) Line in the sand – draw a line in the code. What's below it can be changed between hosts, what's above it cannot be changed.
3) Dual build – compile the same code base on the old and the new hosts and retest on both hosts
4) Decision time – can you add your new features to the both hosts, or must you abandon one? If you are moving from DOS to Linux or Windows, your objective may well be to abandon DOS.
5) New features – only at this stage should you introduce new features.

I have observed an application of 800,000 lines of Forth source code ported in six months using this process.

Preparation

If you rely on host-specific features, they will break your code later. To avoid this, move all such code to a host-specific harness file or directory of files. The harness for your existing host should be fully **documented in the source code**. The harness will become the model for the new hosts.

Code that causes problems includes standard words that have host-specific extensions, e.g. some Forths use range checks in **`/STRING`** whereas many do not. It is far better to rename this version to something else. A global search/replace on your source tree is much cheaper than days spent chasing bugs. Other nightmares come from words that are common, but have different meanings and semantics on different hosts, e.g. **`FOR`** and **`NEXT`**.

Many Forth systems use a vectored I/O model for redirecting **`KEY`**, **`EMIT`** and friends to different devices or displays. You will need to find a way to isolate the differences from your application code.

Other sources of error will come from words that effectively split execution into two words, but do not use `:` and `;` to do it. The ANS standard does not permit words to define other words inside themselves and some compilers take advantage of this. Such words will need host-specific hooks into the compiler.

Remove all coded definitions, rewrite them in high level for the harness model. You can always rewrite them later if you have to. Forth assemblers for the same host CPU are very rarely compatible, so remove problems before they occur.

This phase essentially forces you to perform a code review of your source tree. Do not be surprised if you find and correct existing bugs!

Line in the sand

The objective is find a place in your load order above which no code needs to be changed to make the application work on the new host. You will not get right immediately, but is important that the setting of the line is managed and that programmers buy into the idea. There must be a manager of the process.

There's always a big temptation to put conditional compilation into the code above the line. This only indicates that that some piece of code should be changed and the host-specific parts moved into the harness layer. The porting process is a matter of constant negotiation between the partners.

Dual build

The point of the process is to be able to test that the new version runs identically to the one on the old host. You must be able to share data between them.

Do not add new features. This is only a port, you do not want to be debugging new features yet.

At the end of this stage you will have two harnesses – one for the old system, one for the new.

Decision time

If you are porting a library, you have now completed you first port. Your harnesses need to be reviewed. It is in your own interest to reduce the size of the harnesses where possible – it will make future ports easier.

When porting an application, you are probably not going to be able to make all users convert to the new system immediately. Therefore you need to make a decision as to whether to maintain the dual build for a while so that the old host can be extended too, or whether to put the application on the old host on "care and maintenance only". This decision is essentially a commercial decision.

Where you are porting between (say) Windows and Linux, you will probably have already made, or be in a good position to make, decisions about how to manage the differences between GUIs. Embedded systems developers will be in a position to review speed, space and power budgets.

It is important to (try to) predict what evolution the code will make over the next few years. If you have moved from Windows to Linux, will you want to go to OSX as well? Now that you've moved from a 16 bit CPU to a 32 bit CPU, will you want a file system and a TCP/IP stack? What will be the consequences of these decisions on your harnesses, e.g. for vectored I/O.

New features

Now that you have made your decisions and reviewed the harness code, you can plan your new features.

Application developers who have chosen to abandon the old host will be very tempted to optimise the code base for the new host. Be **very** cautious. The process of building the harness has also contributed to layering the software, which has its own benefits. Where you can profitably take benefit is in removing code that has equivalents in the new host. In embedded systems, the appearance of vectored I/O in the new host may permit considerable simplification.

Library developers should seriously consider doing another port. The effect of another port is to consolidate the harness code. The result of this to make it much easier for third parties to do their own ports, which in turn increases the take-up of the code.

Guidelines

Harness documentation

> *Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing. Dick Brandon*

Yes, this guideline comes first. The harnesses are the specification by example.

The documentation should be in the source code. Programmers don't use Word or OpenOffice, they use UltraEdit or Vi. Stack comments must be accurate, and every word should have at least a one-line description. Literate programming tools for Forth are available. It may take 10% longer to write the code, but you'll save more time during testing and debugging. Once you start documenting your code as just a part of programming, you will see the advantages and apply it to all code.

Put the design notes at the top of each section.

Keep it simple

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. (Brian Kernighan)*
>
> *The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. (Edsger Dijkstra)*

Simple, well factored code is the key to maintainable code. Guru code and clever tricks will bite you later. Identify these, re-factor them, and move the nasty bits to the harness layer. An example comes from the MPE assemblers, which can be switched between opcode-first (prefix) and opcode-last (postfix) modes. This is done using **`?PREFIX`** to separate the opcode construction code according to when it has to be executed. The original code is a Forth classic by Bob Smith from the 1987 FORML conference.

```
: prefix?       \ -- t/f ; true if in prefix mode
  <prefix> @  ;

2variable aprior

: ?prefix       \ struct -- struct' ; executes previous opcode
  prefix? if
    r>                          \ struct ret-addr --
    aprior 2@  2swap  aprior 2!   \ exchange with contents of APRIOR
    >r                          \ will use previous return address
  endif
;
```

Opcode descriptions often take the form:

```
: opType        \ opcode -- ; --
  create  , does> ?prefix  …  ;
```

This code makes several assumptions that are dangerous:

1) The return address is a single cell on the top of the Forth return stack.
2) It performs a flow control which makes assumptions about the return stack depth.
3) What follows is effectively a nameless word and the compiler is not told about it.

A better solution that only makes the return stack assumption follows.

```
: (?prefix) \ struct -- struct' ; executes previous opcode
\ This word assumes that the return address is on the top
\ of the return stack and that an xt is inline. It exits
\ the caller.
  r> @                          \ -- struct xt
  prefix? if
    aprior 2@ 2swap aprior 2!   \ exchange with contents of APRIOR
  endif
  execute                       \ use previous xt
;

: ?prefix   \ -- ; finish and start nameless word
\ This word assumes that the return address is on the top of the
\ return stack and that it can compile an xt inline.
  postpone (?prefix)  here 0 ,            \ (?PREFIX) gets xt inline
  state off  smudge
  :noname swap !  !csp
; immediate
```

This code still makes the return stack assumption. This is safe on most hosted systems, which is acceptable but still host-dependent. However, it still makes assumptions about the compiler and data alignment. We can fairly easily reduce it to the return stack assumption only.

```
: (?prefix) \ struct -- struct' ; executes previous opcode
\ This word assumes that the return address is on the top
\ of the return stack and that an xt is inline. It exits
\ the caller.
  r> aligned @                  \ -- struct xt
  prefix? if
    aprior 2@ 2swap aprior 2!   \ exchange with contents of APRIOR
  endif
  execute                       \ use previous xt
;

: ?prefix   \ -- ; finish and start nameless word
\ This word assumes that the return address is on the top of the
\ return stack and that it can compile an xt inline.
  postpone (?prefix) align here 0 , \ (?PREFIX) gets xt inline
  >r  postpone ;  :noname r> !  !csp
; immediate
```

What is also interesting about the changes is that the resulting code is more robust and makes assembler macros easier to handle.

Fix bugs first

Fixing a piece of code with two bugs in in is much more difficult than fixing one bug. So fix any bugs as soon as you detect them. Never, ever, leave a bug alone.

Crash early and crash often

When MPE wrote its first Windows Forth back in the days of Windows 3.1, we made many mistakes. A natural consequence was programmers wrote extremely defensive code. When we wrote VFX Forth for Windows, we didn't do that. VFX Forth is brutally intolerant of programming errors, but has good integration with the Windows exception handler. When CCS moved their application to VFX Forth, there were initially complaints that previously working code was crashing. After a month or two it emerged that VFX Forth was revealing bugs that had lurked in the production code for years. When these bugs were fixed, both systems had been improved.

The good thing about a crash is that it's a show-stopper – you have to fix it.

People

> *The trouble with C++ is that it requires gurus to maintain it. Gurus don't do maintenance. (Anon)*
>
> *People are part of the design. It's dangerous to forget that. (Anon)*
>
> *Never attribute to malice that which can be explained by stupidity. (Hanlon's Razor)*
>
> *Stupidity maintained long enough is a form of malice. (Richard Bos's corollary)*
>
> *A man who is right every time is not likely to do very much. (Francis Crick).*

Porting an application or library is not a competition, it's a collaborative exercise. People skills are an important part of the process. Once your code is shared, you will have to deal with a wide range of people with a wide range of expectations.

Keeping your ego out of the way is just part of the process. None of us is capable of being correct all the time.

Conclusions

Porting a library or application is mainly a matter of discipline and management.

The harness approach is practical and proven over a number of ports.

People and their management are part of the solution.

Acknowledgements

Willem Botha at CCS taught me a great deal about porting code in a disciplined fashion.

Damian Brasher provided a perspective outside the Forth world.