ISSN 0265-5195

November
2001
Issue 114

Forthwrite

news events people reviews projects programming

Tiny Open Firmware

FIGUK magazine:

From the ‘Net

An Interview with Tom Zimmer

A Call to Assembly

Charles Moore Interview on Slashdot

Vierte Dimension 3/01

Did You Know? – Large Forth Projects

Deutsche Forth-Gesellschaft

Would you like to brush up on your German and at the same time get first-hand information about the activities of fellow Forth-ers in Germany?

Become a member of the German Forth Society for 80 DM (£28) per year (32 DM (£11) for students and retirees). Read about programs, projects, vendors and our annual conventions in the quarterly issues of *Vierte Dimension*.

For more information, please contact the German Forth Society at the e-mail address
SECRETARY@ADMIN.FORTH-EV.DE

or visit http://www.forth-ev.de/
or write to
Forth-Gesellschaft e.V.
Postfach 161204
18025 Rostock
Germany
Tel.: 0381-4007872

November
2001
Issue 114

events

euroFORTH 2001 21
AGM Report.............................. 22

news

Forth News 2
Did you Know?
– Large Forth Projects 34

reviews

Tiny Open Firmware................. 5
From the 'Net 11
Vierte Dimension 3/01 31

people

An Interview with Tom Zimmer ..12
Charles Moore interview on Slashdot
... 30
Letters 35

programming

A Call to Assembly 2/3 23

Editorial

Feedback is always welcome and our contributors are especially keen to get some response to their efforts, so recent comments on “NEAR Spacecraft”, “A Call to Assembly” and “Arithmetized Logic” recently were well received.

I’m pleased to report that this issue includes three more contributions from non-members – these items widen our horizons and help FIG UK contribute to the wider community of Forth users. However the pages of Forthwrite remain open to all and we are especially keen to encourage new members to venture into print.

Although Forthwrite is the most tangible service from FIG UK, it is only one of several:

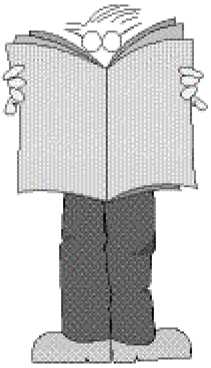
- IRC takes place every month (the only regular Forth chat session anywhere) with a healthy mix of members, non-members, UK and overseas.
- Our web site is the primary route to FIG UK membership and Jenny reports about 1,000 visitors a month.

Welcome to new member John Phythian from Kettering and welcome back to old member John Olwoch.

PS. Don’t forget the monthly IRC session. Our next one is Saturday 1st December on the server “IRCNet”, channel #FIGUK from 9:00pm.

Until next time, keep on Forthing,

Chris Jakeman



Forth News

People

Charles Moore Interview

The “Slashdot Interview” announced in the previous Forth News has taken place; see the report by George Morrison elsewhere in this issue.

Commerical Systems

Benchmark Corrections

In the previous issue, we exchanged the results for iForth and SwiftForth. The correct version is given below.

MPE's **VFX Forth** for Windows build 3.40.0685 is available for download. A summary of the optimisation results was posted by Stephen Pelc.

Primitives using no extensions, test time (ms) including overhead for VFX3.4, iForth and SF2.0

1.Eratosthenes sieve
2.Fibonacci recursion
3.Hoare's quick sort
4.Generate random numbers
5.LZ77 Comp.
6.Dhrystone

Total time in msecs:
1,893 for MPE ProForth VFX 3.40.0686
5,445 for iForth by M. Hendrix, v1.12.1121
16,103 for SwiftForth 2.00.3

Tiny Open Firmware

Brad Eckert, author of the free Tiny Open Firmware is offering more demonstration hardware for Tiny Open Firmware (see the article in this issue). Tokenized boot code on expansion modules runs on both 8051 and 68331 platforms. At startup, the board (whichever processor it's based on) evaluates tokenized driver code resident in a serial EEPROM in each module. Tokenized code gets translated to native machine code and linked into the application at startup. More info at:

http://www.tinyboot.com/eval31.html

Non-commercial Systems

Forth now available for .NET

The Forth language is now available for the .NET platform. Valer Bocan (currently completing a PhD at Timisoara, Romania) has released his Delta Forth compiler at

http://www.dataman.ro/dforth

Delta Forth .NET requires Microsoft's .NET framework to be installed and generates .NET executables.

FICL Upgrade

FICL release 3.01 is now available for download at

http://sourceforge.net/project/

showfiles.php?group_id=24441

This release includes contributions and bug fixes. Thanks to Larry Hastings for the optional FILE wordset. Larry also did the very nasty task of moving all of those static pointers into FICL_SYSTEM so that you can create and destroy FICL_SYSTEMs in any order. Ye Xiaofeng contributed a SWIG adaptation for FICL - this generates your wrapper code for you automatically, saving your wrists. Thanks also to David McNab and Leonid Rosin for bug fixes.

FIJI

FIJI, the ForthIsh Java Interpreter, is now a SourceForge project. The current release is 1.2 Beta. See http://fiji.sourceforge.net

kForth

A new release of kForth (Rls. 9-26-2001) has been announced by Krishna Myneni for Linux and Win32. It is available at

http://ccreweb.org/software/kforth/kforth.html

PetForth

This is a new system based on eForth and close to ANS developed by Petrus Prawirodidjojo. See http://www.geocities.com/petrusp_id/petforth.zip

Forth Resources

Forth Books For Courses

"Forth Programmer's Handbook" and "Forth Applications Techniques" are available from Forth Inc. (http://www.forth.com). McMaster University, Ontario, have bought 67 copies, presumably to support a new course.

Improved FTRAN

Julian Noble has posted an improved version of FRAN on his computational methods page at

http://www.phys.virginia.edu/classes/551.jvn.fall01/

under "Forth system and example programs".

You can now evaluate an expression interactively as in:

```
fvariable x  fvariable y  ok
3e0 x f!     4e0 y f!  ok
f$" x*(x^2+y^2)" f. 75.0000  ok
```

Also included is a limited ability to handle complex variables. The code conforms to ANS Forth and has been tested on Win32Forth v4.2.

Russian FIG

Michael L. Gassanenko reports that a "translate" button has been added to the RuFIG site at

http://www.forth.org.ru

This allows you to view the site in English, French or German translation. The quality of translation has “improved from syntactic ramblings to lexical misuse”.

Forth Primer

Julian Noble has converted his on-line primer, “A Beginner's Guide to Forth”, to HTML format. It now has a hyper-linked table of contents and links to manoeuvre around internally.

It now includes a section on actually writing a program, from start to finish (as opposed to defining words that perform simple tasks).

Win32Forth Fan Club

John Peters has launched an on-line fan club for Win32Forth at

http://go.to/win32forth/

In addition, John is organising a collaborative project to continue development of Win32Forth, WinView and tools based on it. John can be reached at japeters@pacbell.net

FIG UK mailing list

The FIG UK mailing list dedicated to users of the F11-UK processor kit has moved. Previously hosted at Robert Gordon University, Graeme Dunbar has now moved this to the group FIG-Forth-UK hosted at http://groups.yahoo.com.

FIG UK is grateful to the university and Graeme Dunbar for providing and maintaining this service since its inception.

Brad Eckert
brad@tinyboot.com

Tiny Open Firmware

- Extensibility for Small Embedded Systems Brad Eckert

Most of us have heard of Open Firmware – see November 2000 issue – which provides a “plug and play” facility for all Apple, Sun and Power PC computers. Brad Eckert has developed “Tiny Open Firmware” to provide similar facilities for small embedded systems.

Introduction

Today’s microprocessor-based electronic and electromechanical equipment can often be designed to accommodate add-on accessories. Add-on hardware must either live with the design constraints of the original firmware or provide a means of upgrading firmware in the field. Ideally, the add-on hardware should be able to patch the application at run-time so as to take advantage of the new hardware.

A common way to patch code at run-time is to put hooks (function pointers in C) at strategic points in the application. These hooks point to function pointers in RAM, which can be patched. For example, you might decide that `putch` may change, so you define a default ROM version `DefPutch` and then declare

```
    void (*putch)(byte) = DefPutch;
```

To define a patch, you can compile a new version of `putch` for an absolute location in memory. To patch the ROM code, you'd load the patch code into the RAM location you compiled for and then change the function pointer for `putch` to point to the new code.

See www.tinyboot.com for Firmware Studio, a public-domain development environment based on Tiny Open Firmware and commercial evaluation boards for 68331 and 8031 microcontrollers.

This method can be used with assembly too. Either way, there are serious drawbacks:

- The patch code is native code, and on many platforms, absolute code. This greatly complicates the use of multiple add-ons. Plus, you have to freeze the ROM if the add-on code is to re-use sections of ROM code.

- You need to provide enough hooks to address every anticipated need. Invariably, a need comes up that you didn't think of, so the add-on code has to replicate a big chunk of the application in order to work. This bloated code eats up system resources that may already be scarce.

- Changing the hardware design or moving to a new processor will break existing add-on code. Accessories in the field that can't practically be updated will rendered obsolete.

Besides flexibility in the interface between add-on code and application, you also need a way to handle run-time variations and provide extensibility. If you want to enable the end user to write add-on code, you can't just give them your source code and tell them to buy a compiler. An interpreter such as Lua, TCL or Java could provide a virtual machine to isolate add-on code from the implementation details of your application. But interpreters are slow and usually bulky and don't really address the interface issue. Better to compile add-on code at start-up.

One solution is to implement a computer language (preferably not a new one) in a way that solves these problems. A late-binding mechanism that renders all subroutines patchable at run-time would solve part of the extensibility problem. Any subroutine could be patched with a relatively small amount of machine code. And, although a token interpreter offers some extensibility, real extensibility requires removing the wall between application and add-on code.

Forth to the Rescue

Much flexibility can be attained if add-on code is compiled at boot-time. The compiler needs quickly to translate source code in the add-on module to machine code at start-up. It also needs to be able to execute commands that bind the new firmware features into the application. The combination of run-time compilation and immediate execution renders the application extensible.

installation instructions reduce to “plug it in”

Extensibility is a key feature of the Forth programming language. Forth is an industry-proven computer language originally developed for real-time control. Forth is used by the IEEE1275 "Open Firmware" standard to boot up millions of Sparc and PowerPC workstations. On a workstation motherboard, Open Firmware probes the busses for add-on cards and loads driver code from a ROM resident on the card. The driver code is in tokenized form, meaning that Forth keywords are represented by numbers instead of ASCII strings. The most often used Forth keywords are represented by one-byte tokens, leading to very compact code. Semantically, it's still processor-independent Forth source code.

A dialect of Forth called Tiny Open Firmware (TOF) implements the features of IEEE1275 useful to small embedded systems. TOF uses subroutine/native-threading in which Forth keywords (tokens) are converted to subroutine calls or in-line machine code. TOF adds an extra level of indirection to each subroutine call in the form of a RAM-based jump table. Instead of calling a subroutine directly, compiled code calls into an array of jump instructions. For each subroutine call the extra overhead is one jump instruction. On some processors there is a pipeline stall penalty, but it's still an efficient way to achieve late binding.

The RAM-based jump table, called the Binding Table, is initialized from ROM at start-up. You can think of a TOF-based system as an object with hundreds or thousands of late-bound methods. You can patch any ROM-based subroutine by putting new code in RAM and changing the appropriate binding table entry to jump to it. All code that uses it will be redirected to the new version. Besides giving the application fine-grained patchability, the binding table also provides rapid compilation.

Implementing TOF

A freeware Windows program called Firmware Studio implements TOF for several processors. It's available at http://www.tinyboot.com . Full source is included.

TOF uses two compilation modes: *static* and *dynamic*. Both compile `CALL` instructions. In static mode, the destination is the address of the word. This is typical of subroutine-threaded Forths. In dynamic mode, the destination is computed from the word's execution token (xt) so as to compile calls into the binding table. Most words are compiled in dynamic mode.

Tokenized Code

Firmware Studio uses an ordinary Forth interpreter to compile Forth code onto the host PC and to compile machine code onto the target processor. It also has a specialized interpreter called a tokenizer. Instead of compiling machine code, this converts Forth keywords into tokens for the target processor. The resulting tokenized code is semantically equivalent to its textual Forth source, but is stripped of comments and stored in a compact form.

source … stored in a compact form

The tokenizer is the part of TOF that runs on the host PC. The host knows the token assignments of Forth keywords and can compile tokenized code. Tokenized code may be used as boot code for add-on hardware.

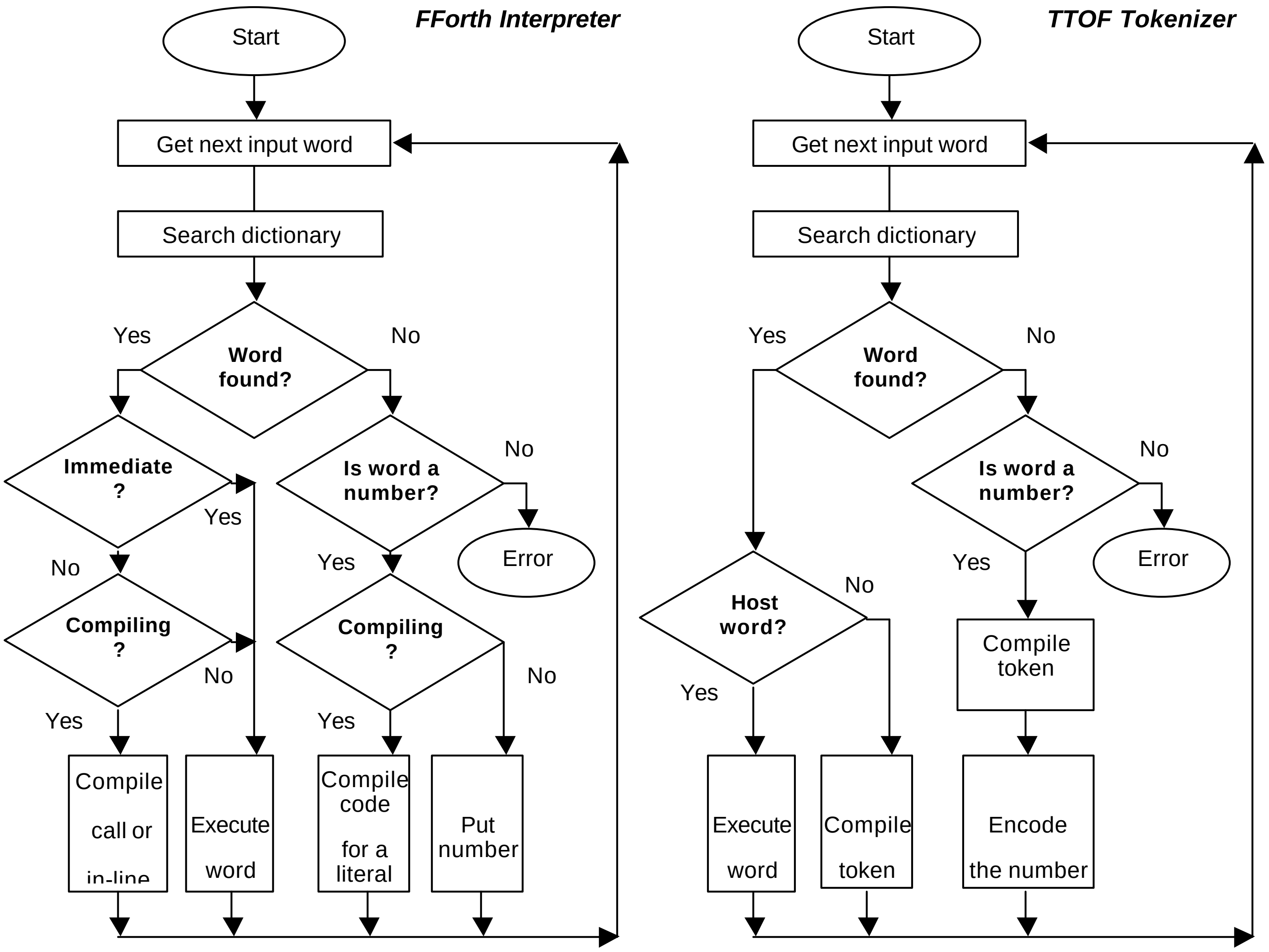
When the user interacts with the target hardware, tokenized code is sent to the target over a communication link for immediate evaluation. Typically, console input is tokenized and sent to a free part of RAM in the target. Then a program resident in the target evaluates the tokenized code. Tokenized code may also be stored in the program ROM to archive temporary features so as to pack more features into a space-limited application.

The Tokenizer is very similar to a typical Forth interpreter and the diagrams below show these side-by-side for comparison. Token values between 0x20 and 0xFF are encoded using one byte, others are encoded using two bytes. Two-byte values concatenate the lower five bits of the first byte with all of the second byte for a possible range of 0x0000 to 0x1FFF.

Host words are parts of the tokenizer resident in a special wordlist. They're mostly defining words. The tokenizer has a state flag, which defining words use to keep semantic consistency with the original Forth source.

When a ROM image is built, token numbers are assigned to word names. These token assignments can be saved to a file. A token file is really an interface specification that connects add-on code to ROM routines. This file can be used instead of the original source code to set up token assignments. It can be given away without revealing proprietary source code, enabling third parties to write add-on code.

Tokenized code serves as input for the evaluator. This is TOF's version of the classical Forth interpreter. Instead of feeding it textual source, you feed it tokenized source. This source can come from a host PC, non-volatile memory in add-on peripherals, program ROM, or any other data source available at run-time.



The Evaluator

The evaluator is a Forth program resident on the target hardware. It converts tokenized Forth source to machine code. The evaluator is like a traditional Forth interpreter except that it computes the location of Forth words instead of traversing a header list looking for them.

The evaluator uses the token value as an index into the binding table. This index is used as the call destination for compiled words. Every word is preceded by a header byte containing an immediate flag. To get this flag, the index is used to extract the address of the code from its binding table entry and the byte immediately before the code is fetched.

Numbers are represented by an immediate word like (LIT8) followed by one or more data bytes. (LIT8) and similar words fetch data from the input stream and sign or zero extend it if necessary.

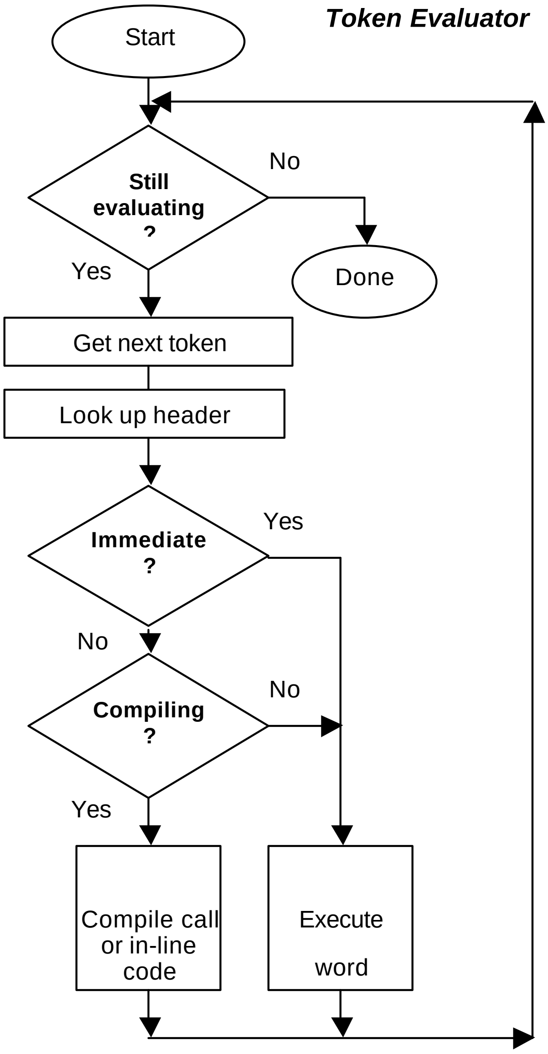
The evaluator fetches bytes sequentially from the input stream until the END token is executed. END causes evaluation to end.

Here is a simple tokenized Forth word. Words associated with the tokens E2, F0 and F2 are immediate words.

```
E1 02 06 : STARS
4B       0
F0       DO
02 05    STAR
F2       LOOP
E2       ;
```

Token values between 0x1000 and 0x1FFF are regarded as relative tokens. The evaluator subtracts 0x1000 and adds the highest unused token value of the last evaluation session. This has the effect of mapping the relative tokens of each add-on peripheral onto a different set of unused absolute token values.

Token Evaluator



Consider the case where your application's ROM has xt values up to 0x480, peripheral A has xt values ranging from 0x1000 to 0x1021, and peripheral B has xt values ranging from 0x1000 to 0x1014. Peripheral A's words will be mapped to the xt values 0x481..0x4A2 and peripheral B's words will be mapped to 0x4A3..0x4B7.

Putting it all together

Tiny Open Firmware removes the wall between application and add-on code. Add-on boot firmware is free to invade the application and do anything it wants, to any hardware or code that it wants. You control the add-on code, so you know your guests are reasonably well behaved. The software equivalent of a bouncer can keep out unknown code.

A typical system has some kind of expansion bus. At start-up, TOF probes each module on the bus looking for boot code. If it finds it, the evaluator verifies its boilerplate and checksum and then evaluates the boot code. TOF continues probing the bus until all boot code has been evaluated.

Add-on modules usually aren't just generic hardware. They are designed to supplement the application. As such, their boot firmware patches part of the application to make use of the new hardware. A typical boot program defines driver code and an application extension for the new hardware, initializes it, and links the code into the application.

The evaluator can handle boot code from multiple modules, each device containing tokenized source that's compiled to native machine code at startup. Installation instructions for the end user reduce to “Plug it in”.

For debugging, the tokenizer and evaluator together act as a normal Forth interpreter. Keyboard input is tokenized by the host PC, sent to the target and evaluated. The resulting output is read and displayed by the host PC. Since most Forth code is inherently reentrant, the debugger has its own execution thread that lets the application run while debugging is underway. TOF supports live debugging with which you can probe and patch a live, running system.

Summary

Tiny Open Firmware brings self-installing plug-and-play hardware to small embedded systems. A TOF implementation is small, typically under 32K for 68K/Coldfire and 20K for 8051 processors. Its efficient late-binding mechanism renders all ROM-based routines patchable and enables rapid compilation of processor-independent add-on code. Run-time compilation of add-on code simplifies applications whose configurations will change as customers modify their systems.

Brad Eckert holds a degree in Physics and is currently a Hardware/Firmware Engineer. He’s been designing and programming embedded systems for about 15 years.

From the 'Net

"I Hate Forth"

Did any of you see this deliberately provocative article at http://www.embedded.com by Jack Ganssle (July 31st)?

Fortunately his musings attracted some informed responses and the web site published 14, almost all positive about Forth, from:

Tamara Cravit, Taylored Software

Ed Beroset, ABB Automation

Wil Blake, Embedded & Mobile Systems Inc.

Graham Smith, Programmer and FIG UK member

Steven R. Commer, Debitek

Elizabeth D. Rather, FORTH, Inc.

Michael Losh, American Systems Technology, Inc.

David Graham, Graham Automation, Inc.

Bob Applegate, Ulticom, Inc.

Don Warbritton, Ametek/Dixson

Dennis R. Miller, Philips Semiconductors

Tom Mazowiesky, Global Payment Technologies, Inc.

Troy Flowers, Iconn Wireless

Robert (Bob) E. Cronan, RiverDelta Network

"If you give someone a length of rope and they hang themselves with it, you can hardly blame the rope. If someone's Forth code is unreadable, ineloquent or unstructured, the fault is theirs--not Forth's.

I've been programming in Forth for 17 years. For embedded systems applications, you just can't beat it."

Steven R. Commer. Senior Systems Analyst, Debitek

An Interview with Tom Zimmer
- Forth System Developer

Jim Lawless

If you've ever used a Forth compiler, chances are you've heard the name Tom Zimmer. Tom's been a staple in the Forth community for a couple of decades. Tom developed a number of Forth systems for popular 8-bit microcomputers that dominated the home-computer market in the 80's. Tom is the creator of the freeware Win32Forth system.

What's your educational background?

I received no formal programming training. I graduated from high school in 1968, long ago and far away. I was interested in electronics at the time, and I had a friend Dick Cappels who bought me the components for a computer, and told me to go down to Wiley Elmar in Sunnyvale CA. and pick up my new computer.

The CPU was an RCA CDP-1802, a static processor. It was something of an oddity at the time, most processors were dynamic, and wouldn't run below about 500 kHz. The 1802, being static, would run all the way down to 0 Hz. I had wired the 1802 with 1k of static memory into a simple computer, and programmed it in machine language. It had three clock rates, single step, 10 Hz, and about 500 kHz.

My first exposure to computers. After high school, I worked for Pacific Telephone as a COEM (central office equipment man). That was in the days when job names could specify a gender.

After a stint in the military, as a communications controller, my same friend hired me as an electronic tech for a small company that built the first video disk recorders. They were nothing like you might imagine today, being much larger, with many custom mechanical parts.

The video recorders contained a micro controller, that was programmed in Forth by Mike O'Malley at Berkeley. He did this work on a consulting basis. He would bring us an EPROM, we would plug it in, and it would work. We were always amazed when his code worked, because he didn't have any hardware to develop the code on; he claimed to have some sort of simulator that he used for testing.

Later Dick had me design a hardware controller for a video disk recorder that was not processor-based, because Mike charged us around one or two dollars a byte for code, and we thought that was expensive. So I designed the controller. My first big hardware design project.

I didn't have any formal hardware education either, unless you count a course in electronics in high school. Anyway, the controller worked, and was even shipped in a product, but it wasn't nearly as trouble-free as Mike's Forth-coded controller version, so we abandoned the idea of using hardware alone to control the recorder.

Anyway my life in electronics and computers was sealed at that point, and I have never looked back.

How did you first encounter Forth?

I already mentioned my first Forth exposure, but the first time I tried to use it was later when I worked for Calma. They built CAD workstations, and I was hired to work in the hardware diagnostics area. I obtained a barely readable photocopy listing of Forth for the 8080 processor. I typed it into an Intel MDS (Micro controller Development System), assembled it, and got it to run.

I had no idea what Forth was supposed to be, but I had heard that it was good for interactive debugging, and I was interested. It had within it the concept of virtual memory, but that was far beyond me at that point, so I just stubbed that all out.

At this time, in the later 1970's, I hadn't even heard of the Forth Interest Group (FIG), so I had no contact with that group, or anyone else in the Forth community. I was just exploring this interesting concept of an interactive computer language.

Toward the end of my time at Calma, I got a FIG listing of Forth for the VAX, and got that to run. We used Forth to write hardware diagnostics. VAX Forth was quite a challenge, because I could assemble it, but I couldn't (or didn't know how to) link it into VMS, the VAX operating system, so I had to dig into some of the system files, to extract system call locations so I could interface with VMS.

According to what I've seen on comp.lang.forth, you had developed (or co-developed) Forth software for a variety of micro-computers in the 80's. What events led to your involvement in the development of these products?

I was certainly excited about Forth after my experience at Calma. I bought a Ohio Scientific computer, which was 6502-based. I took the 8085 Forth I had evolved at Calma and hand-translated it to the 6502 assembly language, so I could run Forth on my Ohio Scientific. I was very young at the time, and I don't know why my wife even put up with all the time I spent in my work room, but I was so excited about Forth and computers, she just couldn't squash me, I guess.

Around 1979, I heard about FIG, and Robert Reiling passed along a FIG listing for the 6502. It looked interesting and seemed to be accepted by more people than my own Forth was ever likely to be, so Bob and I worked to get it working on the Ohio Scientific. I think Bob typed it in, then turned me loose to get it running on the hardware.

So, I transitioned from my own Forth to FIG-Forth around 1979 and moved forward. As various manufacturers were releasing personal computers in those days, I would buy one, and dig into it and develop a Forth for it. It was a way to have fun, and to make a little money at the same time.

The next personal computer Forth I worked on was VIC Forth, for the Commodore Vic-20. I can't remember which was next, 64Forth for the Commodore 64, or Color Forth for the Radio Shack Color Computer. Vic-Forth was an 8k cartridge, Color Forth was a 12k cartridge and 64Forth was a 16k cartridge. Each successive system had more capability.

Why did you implement each as a cartridge?

These computers didn't have disk drives, so the only real alternative was cassette. I had to use cassette to do the development, but I was interested in creating a Forth that would be easy to learn and use, so didn't want the user to have to deal with cassette, except for data storage. Later, in 64Forth, there were also concerns about security, because there were vendors selling cartridge rippers. 64Forth included copy protection that precluded running it out of RAM. It had to reside in ROM, or it would overwrite itself. Cruel, but that was in the days before I switched to making only public domain systems.

Color Forth was a 6809 processor, and was based on a Forth from the only copyrighted FIG listing. It came from a vendor in Southern california, but I can't remember his name. Anyway, I made a contract with him, to split royalties on Color Forth, and it was released. 64Forth was actually the most profitable, it was distributed by HES (Human Engineered Software) in Burlingame Ca. I personally made about $25,000 in royalties from 64Forth, before HES collapsed financially, still owing me almost $9,000 in back royalties. I didn't really care, I was very pleased that 64Forth had sold so well. I believe that they had a lot of inventory that was passed around for several years after that to various Forth vendors, 'til there wasn't any more interest.

Each of these products had a fairly reasonable manual that I wrote and HES spent a significant amount of money on the packaging for 64Forth and VicForth, so they were very attractive. I'm sure that contributed significantly to their popularity.

How did you go about publicizing and marketing each Forth product? Did you have contacts in the industry at this time?

I didn't have any contacts, but in those days, there was much less software available, so I would just contact a software publisher, and ask them if they wanted to distribute my software with their line. There was a huge hunger for software.

Human Engineered Software (HES) was a real developer, they actually invested money into packaging and advertising. They also had contact with cartridge producers that could do "Chip On Board", which eliminated the need for ROM packaging, keeping the production cost low. They produced a very nice package that was used for both VicForth and 64Forth. I am sure that the package alone was responsible for some of the sales.

Had you mastered the assembly languages for the variety of microprocessors at the time? (6502, 6809, etc.)

Assembly language is assembly language is assembly language. If you have seen one, you have seen them all, with the possible exception of the 1802, which was very different from all the others. I learned assembly language as I went along. Just buy another book, and translate its instructions mentally to the ones I already knew.

Later at Maxtor, I was employed as a diagnostics programmer for testing their disk drives. We used 8086s there; we started with Laxen and Perry Forth and developed Forths for running diagnostics on the high-capacity disk drives that Maxtor produced. Forth-based software was used in a custom networked environment, to burn-in disk drives for 48 hours, and print burn-in results. I worked there for about three years, and developed several public-domain Forths, with names like zforth, tforth, hforth, HF, ZF, and F-PC.

Have you written commercial systems other than Forth compilers?

Good question. For a while there it seemed that all I was good at was making Forth systems rather than writing applications. I guess, to me, Forth was an application. Over the years, I have worked on several applications, but they always seem to be based on having to write a Forth system first. I know that many people disagree with this philosophy, but at the time, I felt I needed to have control of the development system.

Now that Visual C++ is so prevalent, we can trust Microsoft to provide the development system (I'm kidding).

That's an interesting statement, though. Do you think that the younger programmers are missing something in their education by not being exposed to Forth?

Absolutely. Most people who are not very familiar with Forth think it is just a forgotten language of the past. The same thing could be said about our heritage, no matter which country we were born in.

History is important for several reasons, not the least of which is what it teaches us about how to deal with the future. Forth's most important feature has little to do with the fact that it is a stack language; it has instead to do with the way it interacts as a whole with the user.

Forth’s extensibility, structure, modularity and very simple syntax are key attributes that give the programmer freedom to structure solutions for problems in ways that programmers of other languages cannot understand or attempt.

Having access to the full source for your development system gives you the freedom to enhance, or correct problems that the vendor didn't consider. Freedom is very important to me, as it should be to everyone, you just have to remember, that along with freedom, comes responsibility.

Forth gives you the freedom, and the power to mould solutions that match the problem. It also gives you the power to shoot yourself in the foot, or in some other even more sensitive area, so if you can't handle the freedom and the power, then you had better stay away from Forth.

Do you presently develop software for a living? If so, what kind of software?

Yes, I work at ThermoQuest, as a programmer. I was hired by Andrew McKewan to assist in porting a very large DOS-based Forth application into the Windows NT environment. We looked at, and even bought the only commercial Forth for Windows NT available at the time.

Unfortunately it wasn't very mature at the time, and we did not have access to the kernel source code, so when we ran into bugs and philosophy differences, Andrew implemented his own 32-bit Forth kernel one weekend. We got it running using the commercial vendors assembler, which we had a license to use, but we never used any of their source code. I am sure we are guilty of using several of their ideas though.

“amazingly faster, and still as reliable”

Anyway, Andrew brought the Forth kernel into work and turned it over to me for “expansion”. The kernel started in the public domain, and I never took it out of the public domain during development. I was always careful to separate the code that was proprietary to my employer from the public domain general-purpose Forth system code.

An example of this is that, since Win32Forth was a 32-bit Forth system, we were faced with the question of whether to convert all the application source from 16-bit to 32-bit. Since the application was several megabytes, and we wanted it to be reliable, we chose to leave it as 16-bit and to write a 16-bit to 32-bit translation layer between the Forth and the application. This kept the problems we had to face down to compatibility issues and allowed easy porting.

We also added a Windows GUI to the application to make it acceptable to the Windows market. The port was completed from start to actual product release in about 9 months, with an average of four programmers working during that time. Still a large task, but the application proved to be very reliable in the field.

One interesting note, is that the translation layer had within it a lot of debugging code to do range checking on memory operations. When we shipped the product, we left the debugging code active, because we weren't confident enough that we had gotten out all the bugs. Then a year later, when we release the next version of the application, we removed the range-checking and suddenly the application was amazingly faster, and still as reliable, since we had worked out most of the problems during that year. So marketing used “much faster” as a new feature.

Andrew McKewan, Robert Smith and I were the primary contributors, followed by Y.T. Lin, and Andy Corsack. Later I talked Jim Schneider into writing a full 486 assembler, which he donated, completing the system. Andrew added object-oriented programming fairly early, modelled on the MOPS OOP Forth system for the Macintosh. OOP was very valuable in handling the complexity of the Windows API.

Over the years several people have donated bug reports, fixes and enhancements to Win32Forth. It was even sold to a commercial vendor for a year, but it proved to complex for their purposes.

Today I program mostly in Visual C++. Originally I hated C but,

after five years, it is bearable. When programming in C, I miss the power of Forth to create compile-time solutions for difficult problems.
I think I may be burned out for Forth system development, but who knows what the future will bring? If another interesting computer and OS come along, perhaps I will jump ship and dive into another Forth system development project.

What about BeOS? I saw a post recently in comp.lang.forth asking about Forth systems for BeOS.
I am a Macintosh advocate, and I was interested in BeOS when it was going to run on the Mac, but now that won't happen, so I really haven't looked at it much lately.

Have you ever entertained the idea of making a Forth compiler for a console gaming system?
No, but I might be interested in writing a Forth for a PDA-style device, though there are already Forths for the Palm. I think that market is just starting, and more interesting devices will come along. Perhaps then.

Have you thought about actually selling Win32Forth?
I have thought of it, but my experience has been that it is very hard to make money selling development systems. Win32Forth is public domain, so others can benefit from it, but also so that I can benefit from other peoples' contributions. I prefer public domain over GPL because it places less restrictions on use. True, anyone can take Win32Forth and turn it into a commercial system or write a commercial program without giving me or the other contributors credit, but I am also free to use contributors code in commercial applications I write, so while I always try to give credit where credit is due, being able to solve applications problems is what drives me, not receiving credit for some segment of code I wrote several years ago.

Interestingly Win32Forth was purchased a couple of years ago by a commercial vendor for a token fee. They were to document it and release it as a commercial product. Problem was, Win32Forth is so big, that it didn't really fit within their philosophy of development tools, so it languished and was ultimately returned to me.

How many copies had been sold of each of your commercial compilers?
I don't have good access to that information, but my recollection is that about 10,000 copies of 64Forth were produced and I got royalties on about 7000 of those before HES went out of business. There were probably 3,000 or 4,000 copies of VicForth sold, and much smaller numbers of ColorForth and OSI Forth.

What prompted you do develop a DOS Forth with an IDE resembling other compilers of the time rather than a traditional Forth IDE?
I am guessing you are talking about TCOM here, since that is the only Forth system I wrote that has a real IDE. TCOM was developed to make writing an application for DOS easier. One of the problems with all my Forth systems was their size. They were always big and fat, with lots of

tools and libraries of utilities. All that stuff results in large executables.

TCOM was designed from the start to include only the parts of the language that were needed to support the application being built. The result was very small executables.

Of course you still want to debug your programs, so I needed a debugger. Since TCOM produced .COM executables that didn't contain any debugging information, and I didn't want to burden the target application with any overhead, I chose to produce additional data files that could tell the debugger where the various source lines connected to the target application. This allowed me to create standard assembly style listing files from TCOM executable and to debug them symbolically. It worked very well.

TCOM eventually included a bunch of target processors, including at least; 8086, 8096, 8080, 68hc11, 6805 and the Samsung Super8, 56000, and 57000 processors. It included a bunch of examples applications for the 8086 target, more than 70 I think. I even wrote a simple basic compiler for the 8086 target of TCOM. TCOM included all the source for all of the compiler, the examples, the debugger and all the listing generators for each target. TCOM was built on F-PC.

Did you attend industry trade shows in the 80's?

Oh, yes. But only the Forth related ones. There was a lot of activity in Forth in the 80's. There were several hardware vendors, and a bunch of software vendors. Things are a little quieter now, but I think Forth has just

interested in it, and they never bother to spend the time to find out.

moved underground. It won't ever be a general replacement for Visual C, but it still has wonderful applicability in limited resource environments.

As we see faster and faster computers, approaching gigabytes of RAM and terabytes of hard disk storage, we might think that limited resource environments will pass away, but in the consumer product area, and pretty much any high volume product area, Forth is a viable alternative. It provides rapid development and debugging, at low cost.

I think it will always be the secret weapon of the small developer breaking into the market of the large developer with hundreds of programmers.

How does Forth fit into your future?

Well, I describe myself as a C programmer, who is really a Forth programmer. C has provided employment, and Forth provides tools for hardware and software debugging.

When I work with other C programmers on large projects, I always build in a Forth interpreter into the application, for debugging purposes. The hardware guys love it, because it gives them so much power to figure out what is going on with the hardware. For software debugging, it is great because it gives you an interactive method of figuring out how to talk to the hardware before going off and writing a driver in C.

I think most C programmers look at Forth, and don't really understand why they should be

I think of Forth like a fine set of hand tools. Microsoft on the other hand, provides the ultimate power tool, Visual C++ with MFC. It's the computer-controlled mill, that you need three PhDs to operate. Then you can get your job done really fast, but you hate doing it, because the tool is such a monster, and so unforgiving of mistakes.

MFC provides wonderful information hiding, to solve common problems, but unfortunately you have to know a lot about the information that is being hidden, or it won't work properly in many situations. It is like a house built on sand, rather than a house built on rock.

Forth on the other hand, is more like the foundation of rock that you can build your house on. It is simple to understand, and completely bug-free. Of course Win32Forth has fallen into the Microsoft trap. In attempting to deal with all the complexity of Windows, it adds huge complexity to what could otherwise be a relatively simple Forth system. The whole OOP thing was added just to help deal with the complexity and it does help, but at a price. Sometimes I think the price of increased complexity is just too high.

Well, I guess I better get off my soap box, and get back to programming in Visual C++, MFC and my latest project in Java, a whole new adventure.

This interview was taken, by permission, from Jim Lawless' web site at http://www.radiks.net/jimbo

Les Kendall writes:
Recently we did a mod to our system and I asked a 66-year old semi-retired electronic engineer to write a full PC keyboard controller using Forth on a TDS board, controlling all the comms and handshake by bit-bashing. With no previous knowledge of Forth whatsoever, he completed the task in 2 weeks. He then described Forth as an 'interesting' language. He had done C and assembler but I thought it was pretty good to do the job alone - shows that Forth can be easy to learn even from a book.

F11-UK

provides everything needed in a professional-quality low-cost Forth controller board.

Use it in industrial or hobby projects to control a wide range of devices using the well-known multi-tasking Pygmy Forth.

Designed for hosting from a Windows or DOS PC, you can test your application as it runs on the F11-UK board itself. The board was developed by FIG UK members to provide an easy way to explore the world of controlled devices – a niche where Forth excels.

The kit includes both hardware and software and is supported and sold to members at a nominal profit through a private company.

Software

PC-based PygmyHC11 Forth compiler running under DOS produces code for Motorola HC11 micro-controller.

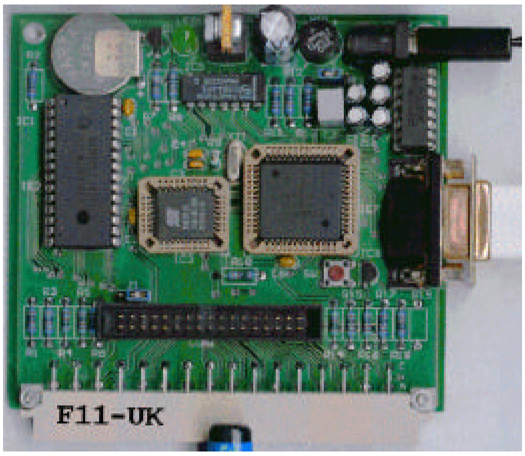
Code is downloaded via standard serial link from the PC to the FLASH memory (or RAM) on the F11-UK single board computer (SBC).

No dongle or programming adaptor of any kind is required.

Forth running on the SBC is interactive which makes debugging and testing much easier.

Multitasking and Assembly included.

The serial link can be disconnected to enable the SBC to function as a stand-alone unit.



All source code provided - 78 pages or so (unlike many commercial systems).

Around 30 pages of additional documentation is supplied including a full glossary of the 300 or so Forth words in the system.

Email mailing list for discussion and limited support.

Hardware:

Processor:
Motorola HC11 version E1 - 8 MHz (2 MHz E-Clock).
Memory:
32k x 8 FLASH
32k x 8 battery backed SRAM
512 x 8 EEPROM onboard HC11.
I/O:
20 lines plus 2 interrupts (IRQ & XIRQ).
Analogue in:
up to 8 lines using onboard 8-bit A/D.
Serial:
1. RS232, UART onboard HC11
2. Motorola SPI bus onboard HC11.
Expansion:
Via HC11 SPI serial bus using
2 or more of 20 available lines.
Timer system:
Inputs: 3 x 16-bit capture channels
Outputs: 4 x 16-bit compare channels.
PCB size: 103 x 100 mm.

Price to FIG UK members: £47.0 plus postage and packing (£2 UK, £4 overseas) plus $25.0 (US Dollars) for registration of 80x86 Pygmy Forth with the author Frank Sergeant.

Delivery: ex-stock.

More information: jeremy.fowell@btinternet.com and 0121 440 1809

euroFORTH 2001

The 17th annual euroFORTH conference on the Forth programming environment and Forth processors is being held on November 23 – 26 at Schloss Dagstuhl, near Saarbrücken, Germany.

This annual conference is held in the UK every third year and this year it returns again to Schloss Dagstuhl. (See Paul Bennett's detailed report in issue 99). For conference details, see http://dec.bournemouth.ac.uk/forth/euro/ef01.html.

Papers will include:

- The C18 ColorForth Compiler
- The 25x Emulator: for a 5x5 array of C18 processors on a 7 mm^2 die
- An HTTP Server in Forth
- A Windows Driver Program Written in Forth
- A Forth Programming System for a Coil Winding Machine
- CANed Objects, a simple object message transport mechanism for distributed transducers
- ColorForth & the Art of the Impossible.
- The mite Virtual Machine: Bridging the Complexity Gap
- A Minimal Development Environment.
- Treating Data as Source: a simple and extensible XML Parser
- A State Machine Design of the Forth Multi-Tasker
- An OO Package for Embedded Control
- Joy: The Concatenative Language of Manfred von Thun.
- Threaded Code Variations and Optimisations.
- Top Heavy Trees: a variation of Binary Trees with faster average access times
- The Common Case in Forth

Charles Moore is the Guest of Honour, so this is a rare chance to meet the inventor of Forth on this side of the Atlantic.

Chris Jakeman
01733 352373
cjakeman@bigfoot.com

AGM Report

Doug Neale offered his hospitality once again – thanks Doug and to Mrs. Neale too.

Changes to Committee

As reported previously in Forthwrite, Chris Hainsworth has retired as Chairman and **Jeremy Fowell** has succeeded him. Sylvia Hainsworth has retired as Librarian and **Graeme Dunbar** has succeeded her. The **Library** contains all the recent books and conference proceedings as well as a complete set of Forthwrite and Forth Dimensions. As books like Thinking Forth go out of print, this resource is increasing in value to our international audience.

Review of Last Year

Our web-site has become a key resource, with over 60 pages, an estimated[1] 1,000 visitors a month and 600 downloads of Forthwrite magazine. Jenny has re-vamped it again this year. We also maintain a list of web-site subscribers from all around the world who have signed up to be notified whenever an update is made.

We have several initiatives which other FIGs have yet to imitate. Our **Library** is a unique lending resource. As books like Thinking Forth go out of print, this resource is increasing in value to our multi-national audience. **IRC** is going well, with a good mix of regulars and visitors, mostly Forthers from overseas. The experimental publication of **Forthwrite on the web** will continue. We will however extend the interval between publishing each issue on paper and making it available electronically. Forthwrite is also placed into several **universities** which may bring results in the long-term.

We should try to learn from our German colleagues in Forth Gesellschaft who run a friendly and effective annual FIG conference.

The finances, as reported in the last issue, are now in balance and look healthy.

Most importantly, membership continues to be stable with around 110 members, many of whom are very active.

Plans for Next Year

We have a new project based on F11-UK – the Flickwriter – and we plan to do more for new members by supplying a free FIG UK CD which will contain the latest versions of key Forth resources.

[1] Extrapolated from a small sample.

AJulian Noble
jvn@virginia.edu

A Call to Assembly 2/3

Julian Noble

Institute of Nuclear and Particle Physics
University of Virginia
Charlottesville, VA 22901

This is the second part of a paper originally prepared for the sadly defunct Forth Dimensions magazine.

Case conversion

Many languages contain a library function for converting a string to all upper case letters or all lower-case ones, leaving digits and punctuation alone. The new Forth ANS standard[2] happens not to require such a routine, although most Forths contain a word analogous to `UCASE` as part of the compiling mechanism.

The first step is to choose our approach. In Microsoft QuickBasic® (QB), a string of N characters is stored in a contiguous sequence of N bytes of memory in the default data segment. It is referenced by a 4-byte string descriptor, with the first two bytes containing the length as a signed 16-bit integer, and the second two bytes the offset of the beginning of the string in the data segment. That is, Quick Basic strings can be up to 32 Kb long. Microsoft C stores strings in contiguous segments of N+1 bytes with the N+1'st byte containing 0 (standard C string terminator), strings being referenced by the address of their first byte.

Forth, by contrast, usually deals in counted strings up to 255 bytes long, whose count is contained in the first byte. These differences between languages present a minor problem in designing subroutines that manipulate strings, since they will not work the same in Forth as in QB or C. The easiest method is to write the code in two pieces: a language-specific header and a universal body.

We illustrate with headers for Forth, QuickBasic and C string-storage conventions.

What of the body code? If we write it first in high level Forth the design becomes clear[3].

```
: lcase?          ( char -- flag)     \ true if lower case
   DUP [CHAR] a <  ( char f1)          \ true if char < “a”
   SWAP [CHAR] z > ( f1 f2)            \ true if char > “z”
```

[2] A copy of the final draft of the ANS Forth Standard document, X3J14 dpANS-6 can be downloaded in several different machine-readable formats, including F-PC hypertext, Microsoft Word , or HTML, from the Web site http://www.taygeta.com.

[3] There are many ways to define `lcase?` including a table look-up. This way has been chosen to illustrate the use of assembler.

```
   OR              ( not[flag] )       \ combine flags
   INVERT ;                            \ logical not

: UCASE            ( beg len)
   0 DO                       \ work from left to right thru string
      DUP C@       ( -- adr char)      \ get character
      DUP lcase?   ( -- adr char flag)
      32 AND       ( -- adr char 32 if lcase | 0 else)
      -                       \ subtract 32 from lcase letters only
      OVER C!                 \ replace modified character
   CHAR+ LOOP                 \ increment address by 1 and loop
   DROP ;                     \ clean up stack
```

That is, we step through the string a byte at a time from beginning to end, testing whether the character is a lower case letter or other. If lower-case, change to upper-case; otherwise do nothing. The actual switch from lower to upper-case is accomplished by subtracting 32_d from the ASCII character code of the letter, since the upper case letters have codes 32_d smaller than their corresponding lower case values. It is worth noting that, in the words `lcase?` and `UCASE` , the programming style computes the result rather than deciding it. That is, while it is not always practical to avoid decisions[4], good style eschews branches wherever possible.

The assembly language version is easy to construct. Begin with `lcase?` and recode directly in assembly language :

```
CODE lcase?       ( char --- flag)
   POP BX         \ char - BL
   MOV AX, BX     \ copy to AL
   SUB AL, # 96   \ AL = char - 96
   CBW            \ sign AL - AH = flag1
   XCHG AX, BX    \ interchange registers BH = flag1
   SUB AL, # 123  \ AL = char - 123
   CBW            \ sign AL - AH = flag2
   OR AH, BH      \ AH = flag1 or flag2
   XCHG AL, AH    \ AL = ~flag
   NOT AL         \ AL = flag
   CBW            \ convert 8- to 16-bit flag
   PUSH AX        \ flag - TOS
NEXT END-CODE     \ terminate definition
```

(Note that the phrase `[CHAR] a <` is not expressed directly in assembler, but becomes `[ CHAR a  1- ] LITERAL ( ie 96 ) - 0>` instead - Ed.)
Test it with:

```
CHAR A DUP . lcase? . 65 0 ok
```

[4] J.V. Noble, Computers in Physics, Jul/Aug 1991, p. 386.

```
    CHAR a DUP . lcase? . 97 -1 ok
    CHAR z DUP . lcase? . 122 -1 ok
    CHAR & DUP . lcase? . 38 0 ok
```

(Note: TRUE - all bits set to 1 - is interpreted as an integer value –1 by “.” in these Forths. ANS Forth is unique among language standards in providing portability between the common 2's-complement integer arithmetic and 2 alternative schemes.)

The preceding test went well - we can test efficiently whether a character is lower case. To proceed, we will merge `lcase?` and `UCASE` into a single routine which will require a looping construct. The one we used in STIB will do fine, because once again the loop will execute a predetermined number of times. Again we must provide header code that places the count (string length, in bytes) in the CX register, and the address of its first byte in
BX. This time, however, we identify the header code as a separate section of the assembler subroutine, in order to be able to replace it later on with an appropriate equivalent that respects the conventions of a language other than Forth.

In F-PC the header will consist of the instructions:

```
    POP CX       \ count in CX
    POP BX       \ beginning of data in BX
    PUSH DI      \ save DI (index) register
    MOV DI, BX   \ start-1 in DI
```

Upon exiting we restore DI with mov BX DI, as the last instruction preceding

```
    NEXT END-CODE.
```

For comparison, a header suitable for QuickBasic would look like[5] [16]

```
    PUSH BP          ; save BP
    MOV BP, SP       ; use BP as a stack pointer
    PUSH DI          ; save DI register
    MOV BX, 6 [BP]   ; address of string descriptor to BX reg
                     ; Note: don't need to initialize CX
    MOV CX, 0 [BX]   ; count in CX reg
    ADD BX, 2        ; offset to string origin in BX
```

and the corresponding QB footer (to exit gracefully) would be

```
    POP DI
    POP BP           ; restore registers
```

[5] Note: if we were trying to generate the same function for linking to C, we would have to take into account the 0-terminated structure of strings in C, probably using a different looping method, since the count would not be readily available.

The complete program in F-PC assembler then becomes

```
CODE UCASE          \ start header
   POP CX              \ get count
   POP BX              \ get origin
   PUSH DI             \ save DI
   MOV DI, BX       \ end header, start body
HERE:                  \ begin loop
   INC DI                 \ point to next byte
   MOV BL, 0 [DI]         \ get byte
   MOV AX, # 96           \ test case
   SUB AL, BL
   CBW
   XCHG AX, BX
   SUB AL, # 123
   CBW
   AND AH, BH             \ AH = FF|0
   AND AH, # 32           \ AH = 32 if lcase, 0 else
   SUB 0 [DI], AH         \ convert letter in $
   LOOP HERE              \ loop if CX >= 0
                    \ end body, begin footer
   POP DI              \ restore DI
   NEXT             \ end footer
END-CODE
```

The subroutine is hard to read even with indented comments (which is why we prefer high-level language to assembler), but it consists of the same parts as the high level definition: a SETUP section that gets the count and origin of data; a body that LOOPs through the string; a test that determines whether a character is a lower-case letter, and if so, modifies it to upper case; and a footer that restores whatever registers have been saved on the stack and exits gracefully. Note we were able to eliminate three redundant instructions:

```
XCHG AL, AH
CBW
PUSH AX
```

whose only purpose in the `CODE` version of `lcase?` was to convert an 8-bit flag to a 16 bit integer that could be left on the stack. The code for `UCASE` is about as terse as such a routine can be made. Since assembler is used to provide raw speed, it is interesting to examine timings[6]. Looking up the number of clock cycles per instruction for the Intel 80286, we find:

[6] Abrash, already mentioned, discusses in detail the pitfalls of assuming the instruction timings given by Intel.

```
CODE UCASE            \ 0 (assembler directive)
POP CX                \ 5
POP BX                \ 5
PUSH DI               \ 3
MOV DI, BX            \ 2
            \ total = 15 for header
HERE:                 \ 0 (assembler directive)
INC DI                \ 2
MOV BL, 0 [DI]        \ 5
MOV AX, # 96          \ 2
SUB AL, BL            \ 2
CBW                   \ 2
XCHG AX, BX           \ 3
SUB AL, # 123         \ 3
CBW                   \ 2
AND AH, BH            \ 2
AND AH, # 32          \ 3
SUB 0 [DI], AH        \ 7
LOOP HERE             \ 9
            \ total = 42 for body
POP DI                \ 5
NEXT                  \ 5 (depends on the Forth)
            \ total = 10 for footer
END-CODE              \ 0 (assembler directive)
```

The instructions labeled “assembler directive” execute during compilation and carry no run-time overhead. Since the header and footer are executed once, their 25 clock cycles are immaterial for reasonably long input strings. Converting a lower-case to an upper-case letter evidently requires 42 clock cycles, i.e. about 1.3 µsec on a 33 MHz machine. The test loop

```
: TEST0 0 DO PAD COUNT UCASE LOOP ;
: TEST1 0 DO 10000 TEST0 LOOP ;
```

allows us to iterate enough times to get meaningful data: saying `10 TEST1` iterates 10^5 times. The time to convert 4,500,000 characters is 7 seconds, giving a per-character time of 1.6 µsec, in reasonable agreement with the estimate from machine cycles. This is 24 times faster than the F-PC Forth version[7]; so optimization is definitely worthwhile when we have many strings to convert.

For variety, here is a version that works with C-style 0-terminated strings. There are two obvious ways to approach the problem: first, modify the loop in UCASE so it terminates when the byte fetched is 0 (not to be confused with ASCII “0”). Alternatively, if we had a fast way to determine the string's length, we could

[7] F-PC is a direct-threaded Forth. Forth systems that optimise and generate native code are much faster (as fast as optimising C compilers), but not as fast as hand-coded assembler.

use the preceding code unmodified. Now, we know only the beginning address of a C string, so to determine its length we must search it until we find the terminating character, incrementing a counter as we go. In high level Forth the subroutine is

```
: GET_LEN   ( beg --- len)
   DUP           ( beg beg)
   BEGIN                        \ start indefinite loop
      DUP C@                    \ get char
      0 <>       ( beg adr flag)
   WHILE CHAR+   ( beg adr+1)
   REPEAT        ( beg end+1)   \ loop until character is 0
   SWAP -        ( -- len)      \ compute length
;
```

and is very slow. Unless there is a specific need for a function that determines the lengths of 0-terminated strings there does not seem to be any reason to factor out this functionality, merely to re-use the code designed for counted strings. Here is a situation where recoding UCASE from scratch is the
more efficient approach. We will call the new version UCASE.C.

Once again we begin by prototyping in high-level Forth, then translating to CODE. We want to hybridize GET_LEN and UCASE.C from before, i.e. replace the definite loop with an indefinite one.

```
: UCASE.C    ( beg --- )
   BEGIN                     \ start indefinite loop
      DUP C@   ( -- adr char)
      DUP      ( -- adr char flag)
   0<> WHILE                 \ haven't reached end
      lcase?   ( -- adr flag)
      32 AND   ( -- adr char 32 if lcase | 0 else)
      -                      \ subtract 32 from lcase letters only
      OVER C!                \ replace modified character
      CHAR+    ( -- adr+1)
   REPEAT                    \ loop until char = 0
   DROP                      \ clean up stack
;
```

The assembler version is easily coded. The use of CBW (convert byte to word) avoids decisions by computing a flag (in the upper half of the AX register) based on the sign of the subtraction operation.

```
CODE UCASE.C
   MOV DX, DI       \ save DI (in DX)
   POP DI           \ DI = beg
1 $:                \ label to return to
   MOV BL, 0 [DI]   \ get byte
   CMP BL, # 0      \ is it 0 ?
   JZ 2 $           \ jump to end if 0
   MOV AX, # 96     \ 97d is ASCII 'a '
   SUB AL, BL       \ is the byte 'a' ?
```

```
       CBW              \ if BL = 97 then AH = FFh, else AH = 0
       XCHG AX, BX
       SUB AL, # 123    \ is the byte 'z' ?
       CBW              \ if AL 122 AH = FFh; else AH = 0
       AND AH, BH       \ AH = FFh if 'a' ≤ byte ≤ 'z', else AH = 0
       AND AH, # 32     \ AH = 32 or 0
       SUB 0 [DI], AH   \ convert byte in string
       JMP 1 $          \ loop
    2 $:                \ end
       MOV DI, DX       \ restore DI
       NEXT
    END-CODE
```

Micro-mini assembler

Although I have discussed the use of the Forth assembler in the context of rapid machine code development and/or as a propaganda device to interest outsiders in Forth, of course one should not forget that it is a useful tool in the Forth programmer's arsenal. In my own work I have not worried too much about the fact that most Forths[8] tend to run somewhat slower than optimized C programs because I know that if I really need to step on the gas by hand coding an inner loop, it will not take much extra effort. (There was a time, not so many years ago, when I got so carried away with that approach that I would define words in CODE at the drop of a hat, just because it was so easy. Needless to say my work was cut out for me later on when I had to port the programs to ANS-compatible Forths. One mustn't lose one's head by over-CODEing.)

When memory is limited and only a few CODE words need to be defined, rather than load the entire assembler, it pays to insert the op-codes directly into the body of the code word. These are usually byte-sized numbers in hexadecimal format, and can be inserted with C, as in (suitable for F-PC)

```
CODE MY@ HEX 5B C, FF C, 77 C, NEXT END-CODE
```

If there are more than a few such words, but one would prefer not to load the assembler, the following word may be of use.

```
    \ Micro-mini assembler suitable for F-PC
    HEX
    : <% BASE @ HEX                  \ base 16
       BEGIN BL WORD %NUMBER
       WHILE DROP C,
       REPEAT 2DROP BASE !           \ restore base
       HERE 1+ @ 3E25 <> ABORT" Missing %> !" ; IMMEDIATE
    DECIMAL
    \ Usage: CODE MY@ <% 5B FF 37 %> NEXT END-CODE
    \ Note: to make the above work in ANS Forth we need to define
    \ %NUMBER in terms of NUMBER.
    \ : %NUMBER 0.0 ROT COUNT NUMBER NIP ;
```

[8] MPE Ltd.'s VFX Forth is a modern exception.

George Morrison
gdm@gedamo.demon.co.uk

Charles Moore interview on Slashdot

George Morrison

Slashdot[9] is a technology news web-site largely concerned with computers and the internet which is much beloved of geeks and nerds. Its popularity has given rise to the term “The Slashdot Effect”; a web-site mentioned on Slashdot can receive so many hits that it becomes overloaded. Occasionally readers are asked to submit questions to an industry luminary which form the basis of an interview.

Charles Moore was recently the subject of a Slashdot interview[10], the main topics of which were his 25x processor chip and Forth. (Forthwrite reported over 300 questions and comments had been tabled – a measure of the intense interest in what Moore is doing – Ed.) The 25x is a new design which contains 25 independent stack machines and has a claimed speed of 60,000 MIPS. CM: “At this stage the 25x is a solution looking for a problem. It's an infinite supply of free MIPS.” He suggested possible uses might be embedded audio/video applications or voice/image recognition.

Chuck's enthusiasm for Forth shines through. CM: “I'm locked in the Forth paradigm. I see it as the ideal programming language. If it had a flaw, I'd correct it.” He was asked about the use of colorForth by colour blind people and suggested that colours could be replaced by different fonts or sizes, and also mentioned the possibility of spoken colorForth. He is unimpressed by other programming languages and sees no signs of progress in their development.

The interview was quite brief, but more information about Chuck's work can be found at Chuck's colorForth site http://www.colorforth.com and UltraTechnology http://www.ultratechnology.com.

[9] http://slashdot.org

[10] http://slashdot.org/interviews/01/09/11/139249.shtml

Alan J M Wenham
01932 786440
101745.3615@compuserve.com

Vierte Dimension 3/01

Alan Wenham

Alan provides a look at the latest issue of the German FIG magazine. To borrow a copy or to arrange for a translation of an individual article, please call Alan.

Vierte Dimension contains some very valuable material and several members have suggested that a full translation of a single article would be appreciated. We have volunteers with the skills to do this, so now we need your nominations. Contact Alan with your choice and we will publish a translation in the next Forthwrite - Ed

General

The acting editor, Martin Bitter, expresses his pleasure at receiving the Swap Dragon award and also greets three new members. These include Chris Jakeman, to whom he extends a specially warm welcome.

Lego robots and arithmetic logic in Forth

Fred Behringer

behringe@mathematik.tu-muenchen.de

This article about the representation of propositional logic as arithmetical expressions is an extended version of Fred's presentation to the German Forth convention. It also appears in English in the July issue of Forthwrite.

REORDER and continuation

Martin Bitter

martin.bitter@forth-ev.de

Martin has been spurred to reconsider Ulrich Paul's `REORDER`, a stack manipulator. `REORDER` was conceived ten years ago for F-PC and Martin has reworked it for Win32Forth. It seems to him that `REORDER` is slower because of the increased number of input values. (Naturally, since factorial(n), the number of permutations of n, increases very quickly with increase in n). Martin thinks that more systematic studies are needed.

WebForth

Chris Jakeman

This is Chris Jakeman's presentation, in English, at the German 2001 Forth convention.

MINOS examples: OpenSched GUI

Bernd Paysan

bernd.paysan@gmx.de

Bernd's presentation at the 2001 Forth convention. OpenSched is a free Linux program for project planning and progressing. Its implementation is not particularly user-friendly as its input is in the form of text data. Bernd describes his GUI system MINOS, and shows that Forth can help to provide a polished front-end for OpenSched.

Book reviews

Friederich Prinz

Friederich.Prinz@t-online.de

Friederich discusses two books, one on Windows 2000 and the other on technology of IP-networks.

Forth opens doors

Fred Behringer

behringe@mathematik.tu-muenchen.de

Fred's starts a new feature in Vierte Dimension presenting short articles, with examples, which appeal to the beginner or, as here, those who are changing to Forth. This article concerns assembler programming in the Forth environment and is the production of a DOS .COM file which opens or closes the drawer of a CD-ROM drive. It is only 92 bytes long!

Calculation with guaranteed accuracy

Christopher Poeppe

This is a repeat of the article previously presented but with correction of a number of serious mistakes.

Other journals

Fred Behringer

behringe@mathematik.tu-muenchen.de

Fred summarises the content of Forthwrite 111 and Figleaf 25 and 26.

Riddle solution

Fred Behringer

behringe@mathematik.tu-muenchen.de

Fred gives the solution to the riddle posed in VD1/2001. It relates to the representation of the decimal number 1066 in various bases. He highlights Martin Bitter in particular as solver.

solver.

Forth convention 2001 in Hamminkeln-Dingden by Wesel

Friederich Prinz, Bernd Paysan

Two reports; a general one by Friederich and a subject-oriented one by Bernd.

Outwitting the Lego-Transmitter --- with and without Forth

Martin Bitter and Fred Behringer
martin.bitter@forth-ev.de

The IR transmitter, which connects the Lego robot to the PC, disconnects automatically after 5 seconds of inactivity on the part of the transmitter. This was found to be very inconvenient when developing applications where a long sequence of characters being transmitted from the robot to the PC via the transmitter was suddenly interrupted because there was no intermediate response from the transmitter. A simple solution was found in which a high resistance was soldered between the +9 volt line and an appropriate point, which allows the voltage there to held high, thus preventing the transmission circuit from being closed down after a certain voltage drop and in turn preventing the robot-IR-transmitter character flow from ever being interrupted.

From the big Teich...

Henry Vinerts
VOLVOVID@AOL.com

Henry reports on the Silicon Valley FIG Meeting of May 2001.

Chris Jakeman
cjakeman@bigfoot.com

Did you Know?

– large Forth projects (1)

While other parts of Forthwrite bring you all the news and the latest ideas and developments, the **Did You Know?** section highlights achievements in Forth, both recent and historical (taking care always to distinguish hearsay from attested fact).

Earlier this year, the comp.lang.forth newsgroup was asked how the very large Forth installation at Riyadh Airport, Saudi Arabia, could be considered a success when it runs on such old hardware.

“Building automation and auxiliary services by AVCO/Textron for King Khaled International Airport (Saudi Arabia). System contains nine PDP 11/44s, 378 8086-based computers, 320 8085-based security processors, and 36,000 sensors.

Initially the project was a disaster, with over 100 programmer-years of code and failing to meet performance standards by a factor of 10. The AVCO/Textron group started over using Forth on all the computers, and wrote a successful version in less than 30 programmer-years and 18 calendar months.”

“The installation was designed in the early 80's and installed in the mid-80's, by which time much of the hardware was at least obsolescent. Now it's hopelessly obsolete and failing. About once a year someone from the airport gets in touch with us (always a different person/company, since contractors there rarely last more than a year, and there's virtually no information transfer from one generation to the next). They have been told that, since the project is written in Forth, there's no way the old stuff can be replaced.

On the contrary, since Forth is readily available on most modern platforms, a port would be straightforward (not a trivial project, but vastly simpler than a total rewrite in a new language). In contrast, the original programming (which we replaced) was done in PLM and FORTRAN. How would you port that to modern platforms?

When the opportunity arises, we always point this out, and have regularly offered to send someone over to make concrete recommendations and proposals regarding upgrades. So far, no one has accepted this offer.”

Source – Elizabeth Rather, Forth Inc

Letters

The Magazine Team are always pleased to get feedback and encouragement. Here we have news of Federico de Ceballos, who uses Forth in his academic work and updates us on his current activities. We also have some feedback from Fred Behringer, inspired by “Call to Assembly” in the last issue.

Federico de Ceballos

From: federico.ceballos@unican.es
Sent: 02 November 2001

Hi Chris,

Nice to hear from you.

> Hope all is well with you and looking forward to your next Forthwrite
> article sometime. Are you doing any research at present that is
> Forth-related?

In the last months I've been doing some study and research into other programming languages (apart from C++: Ada, Java and Oberon). I am also involved in a course of 'Compiler Generators'. After going through the 'well trodden path' of Aho[1] et al., I am planning to give the students some insight by the end of the course about the simplest way of getting a compiler for a high-level language running. (I don't need to tell you which language is this ;-) !)

Apart from this, I've taken some time off from my other obligations in order to (finally!) finish my Ph.D. thesis, titled "A Development Environment for High Integrity Applications". I plan to complete it by the end of this year.

As part of the research, I have developed a simple compiler in a Forth dialect that is used to produce four different cross-compilers for Windows, the PSC1000, the 68HC11 and the AVR RISC processor.

As a small offspring from this work, I am presenting a paper at EuroForth about "A Minimal Development Environment for the AVR Processor". As you probably know, Howerd Oakford, Jenny Brien and Bill Stoddart are also presenting papers.

I'm somehow fascinated by the subject of bootstrapping, meta-programming and cross-compilation. I'd like to prepare, some time in the near future, an article or a series of articles about it for Forthwrite.

[1] Aho is a standard text on Comiler Design – Ed.

Fred Behringer

From: behringe@sunstatistik1.mathematik.tu-muenchen.de
Sent: 25 October 2001

Hi Chris,

I like Julian Noble's tutorial article in issue # 113 of Forthwrite and I like his philosophy of exercising his "constitutional right to assemble". I too have said in several places that I like "misusing" Forth as a quick and most flexible means of assembling.

One observation that I should like to make goes as follows: ZF and Turbo Forth, my own favourites, are 16-bit systems. With almost no extra effort and in almost the same computer time (I'm referring to the Pentium processor), the bit order reversion can be immediately done with up to 32 bits. The cpu cycles I'm referring to are taken from the book by T.E. Podschun: Das Assembler-Buch (in German), Addison-Wesley, 1996.

```
HEX

: OP: 66 C, ;           \ Prefix for switching to 32-bit registers

\ Show the bottom n bits of double precision number d in reverse order
CODE DSTIB ( n d -- )
  OP: BX POP            \   1 cpu cycle
  OP: C1 C, CB C, 10 C, \   1 cpu cycle   10 # EBX ROR
      CX POP            \   1 cpu cycle
  OP: DX DX XOR         \   1 cpu cycle
  HERE
    OP: BX SHR          \   1 cpu cycle
    OP: DX RCL          \   1 cpu cycle
        CX DEC          \   1 cpu cycle
  JNE                   \   1 cpu cycle
  OP: C1 C, CA C, 10 C, \   1 cpu cycle   10 # EDX ROR
  OP: DX PUSH           \   1 cpu cycle
  NEXT END-CODE         \  70 cpu cycles for a reversal of 16-bits
                        \ 134 cpu cycles for a reversal of 32 bits
                        \ (4*n)+6 cycles for a reversal of  n bits
                        \ length = 40 bytes
\ The following is Julian Noble's 16-bit version (Forthwrite # 113)
\ Show the bottom n1 bits of single precision number n2 in reverse order
CODE STIB ( n1 n2 -- )
  BX POP                \   1 cpu cycle
  CX POP                \   1 cpu cycle
  DX DX XOR             \   1 cpu cycle
  HERE
    BX SHR              \   1 cpu cycle
    DX RCL              \   1 cpu cycle
  LOOP                  \   6 cpu cycles
  DX PUSH               \   1 cpu cycle
  NEXT END-CODE         \ 132 cpu cycles for a reversal of 16-bits
                        \ (8*n)+4 cycles for a reversal of  n bits
                        \ Reversal limited to 16-bit numbers
                        \ length = 25 bytes
```

Julian's STIB consumes almost twice as much cpu time as DSTIB, and covers only 16-bit reversal. If memory is of minor relevance, the time needed for DSTIB can be reduced even further (see DSTIB-2 to follow).

```
\ Show the bottom n bits of double precision number d in reverse order
CODE DSTIB-2 ( n d -- )
  OP: BX POP            \   1 cpu cycle   Get d
  OP: C1 C, CB C, 10 C, \   1 cpu cycle   10 # EBX ROR
      CX POP            \   1 cpu cycle   Get n
      20 # AX MOV       \   1 cpu cycle   AX = nmax
      CX AX SUB         \   1 cpu cycle   AX = nmax - n
      AX DI MOV         \   1 cpu cycle   Equivalent
      DI SHL            \   1 cpu cycle   to AX*6
      DI AX ADD         \   1 cpu cycle   but with
      AX SHL            \   1 cpu cycle   less cycles.
  OP: DX DX XOR         \   1 cpu cycle   DX = 0
  HERE 7 + # DI MOV     \   1 cpu cycle   Add 7 bytes for MOV, ADD,
                        \  and JMP.
      AX DI ADD         \   1 cpu cycle   n copies of EBX SHR EDX RCL
      DI JMP            \   2 cpu cycles  left to execute after
                        \  jumping.
  HERE                  \                 For XXX to operate on
  0C0 ALLOT             \ 2*n cpu cycles  32 times EBX SHR EDX RCL
  OP: C1 C, CA C, 10 C, \   1 cpu cycle   10 # EDX ROR
  OP: DX PUSH           \   1 cpu cycle
  NEXT END-CODE         \  48 cpu cycles for a reversal of 16-bits
                        \  80 cpu cycles for a reversal of 32 bits
                        \ (2*n)+16 cycles for a reversal of n bits
                        \ length = 245 bytes
: XXX ( -- )
  DUP 0C0 + SWAP
  DO  66 I     C! 0D1 I  1+ C! 0EB I  2+ C!  \ EBX SHR
      66 I 3 + C! 0D1 I 4 + C! 0D2 I 5 + C!  \ EDX RCL
  6 +LOOP ;

XXX                     \ Fill DSTIB-2 with 32 copies of EBX SHR EDX
                        \ RCL
FORGET XXX              \ Remove auxiliary word XXX, not needed any
                        \ more
```

Note that it will be disastrous to choose n excessively large ! Try `DECIMAL 33 4 DSTIB-2` . However, spend a few more cpu cycles and the security problems will have gone. There is no problem with that with DSTIB or STIB, though one needs to reflect a moment in order to interpret the results correctly in case of an excessive n .

It's amazing how far I can go to reach my goal although there is no 32-bit assembler in ZF or Turbo Forth. Only imagine trying one of the number of quick and dirty tricks I've applied above in any of the numerous "mainstream" languages other than Forth !

Julian`s STIB needs 25 bytes. DSTIB needs 40 bytes. DSTIB-2 needs 245 bytes. This is an interesting amount of trade-off between time and memory. However, what I am owning is no less than 768 megabyte of RAM!

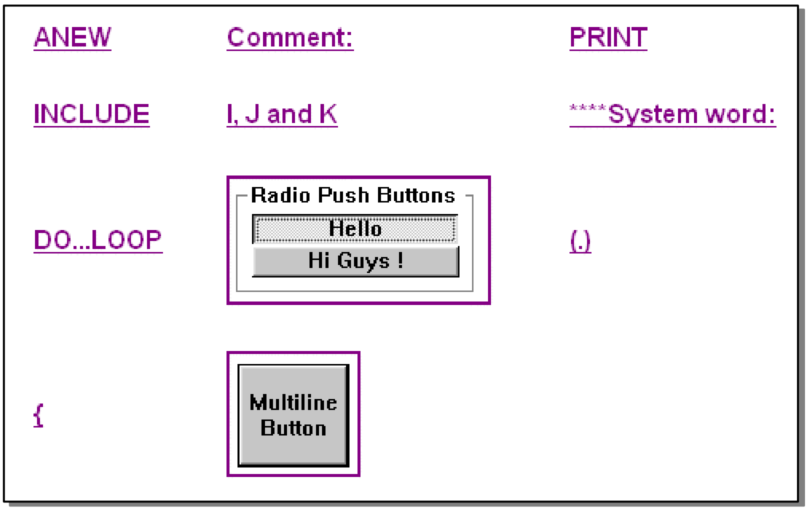
What then do I care about memory - as long as my straightforward Forth system (ZF or Turbo Forth) can manage that amount. Can it? It can – as the reader is invited to find out for himself from my article in the 2-1998 issue of Vierte Dimension.

Many of you will be familiar with Dave Pochin's popular web-site providing help with Win32Forth, especially his advice on Getting Started.

Dave has now added a section providing tips on using Win32Forth at

http://www.sunterr.demon.co.uk/AddValue/AddedVal_1.htm

If anyone would like to send him anything similar, he'll add them (quoting the source).



Chairman **Jeremy Fowell**, 11 Hitches Lane, EDGEBASTON B15 2LS
0121 440 1809 jeremy.fowell@btinternet.com

Secretary **Doug Neale,** 58 Woodland Way, MORDEN SM4 4DS
020 8542 2747 dneale@w58wmorden.demon.co.uk

Editor **Chris Jakeman,** 50 Grimshaw Road, PETERBOROUGH PE1 4ET
01733 352373 cjakeman@bigfoot.com

Treasurer **Neville Joseph,** Marlowe House, Hale Road, WENDOVER HP22 6NE
01296 62 3167 naj@najoseph.demon.co.uk

Webmaster **Jenny Brien,** Windy Hill, Drumkeen, BALLINAMALLARD,
Co. Fermanagh BT94 2HJ
02866 388 253 jennybrien@bmallard.swinternet.co.uk

Librarian **Graeme Dunbar** Electrical Engineering, The Robert Gordon University,
Schoolhill, ABERDEEN AB10 1FR
01651 882207 g.r.a.dunbar@rgu.ac.uk

Membership enquiries, renewals and changes of address to Doug.
Technical enquiries and anything for publication to Chris.
Borrowing requests for books, magazines and proceedings to Graeme.

FIG UK Web Site

For indexes to Forthwrite, the FIG UK Library and much more, see **`http://www.fig-uk.org`**

FIG UK Membership

Payment entitles you to 6 issues of Forthwrite magazine and our membership services for that period (about a year). Fees are:

National and international	£12
International served by airmail	£22
Corporate	£36 (3 copies of each issue)

Forthwrite Deliveries

Your membership number appears on your envelope label. Please quote it in correspondence to us. Look out for the message "SUBS NOW DUE" on your sixth and last issue and please complete the renewal form enclosed.
Overseas members can opt to pay the higher price for airmail delivery.

Copyright

Copyright of each individual article rests with its author. Publication implies permission for FIG UK to reproduce the material in a variety of forms and media including through the Internet.

FIG UK Services to Members

Magazine

Forthwrite is our regular magazine, which has been in publication for over 100 issues. Most of the contributions come from our own members and Chris Jakeman, the Editor, is always ready to assist new authors wishing to share their experiences of the Forth world.

Library

Our library provides a service unmatched by any other FIG chapter. Not only are all the major books available, but also conference proceedings, back-issues of Forthwrite and also of the magazine of International FIG, Forth Dimensions. The price of a loan is simply the cost of postage out and back.

Web Site

Jenny Brien maintains our web site at http://www.fig-uk.org. She publishes details of FIG UK projects, a regularly-updated Forth News report, indexes to the Forthwrite magazine and the library as well as specialist contributions such as “Build Your Own Forth” and links to other sites. Don’t forget to check out the “FIG UK Hall of Fame”.

IRC

Software for accessing Internet Relay Chat is free and easy to use. FIG UK members (and a few others too) get together on the #FIG UK channel every month. Check Forthwrite for details.

Members

The members are our greatest asset. If you have a problem, don’t struggle in silence - someone will always be able to help. Do consider joining one of our joint projects. Undertaken by informal groups of members, these are very successful and an excellent way to gain both experience and good friends.

Beyond the UK

FIG UK has links with International FIG, the German Forth-Gesellschaft and the Dutch Forth Users Group. Some of our members have multiple memberships and we report progress and special events. FIG UK has attracted a core of overseas members; please ask if you want an accelerated postal delivery for your Forthwrite.