27th EuroForth Conference

September 23-25, 2011

TU Wien
Vienna, Austria

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 27th EuroForth finds us in Vienna for the third time. The three previous EuroForths were held in Vienna, Austria (2008), in Exeter, England (2009), and in Hamburg, Germany (2010). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there were no submissions to the refereed track. Nevertheless, I thank the program committee for its willingness to serve.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings. In addition, the printed proceedings include slides for talks that will be presented at the conference without being accompanied by a paper and that were submitted in time.

These online proceedings also contain late presentations that were too late to be included in the printed proceedings.

Workshops and social events complement the program.

This year's EuroForth is organized by Ewa Vesely and Anton Ertl.

Anton Ertl

Program committee

Contents

Standardise Forth OOP Now

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

There are so many Forth OOP implementations that is has become impossible in practice to share Forth code that uses an OOP package. This is the opposite of standardisation and promotes neither portable programs nor portable programmers. This paper suggests a rather brutal approach to promoting one Forth OOP.

Introduction

MPE has had to deal with several OOP packages over the years. In looking at packages we have been offered recently, several have come with their own OOP packages. These are in addition to the three OOP packages supplied with MPE's hosted Forths. Accepting these packages as delivered would then require MPE to maintain a minimum of six OOP packages.

Such a solution is fraught with workload and does not encourage other programmers to write to a common standard. Rather the reverse, it encourages programmers to write yet another Forth OOP package.

Brad Rodriguez' survey [1] of 1996 included 17 packages. The situation is surely worse now. Forth OOP has been an active topic for nearly 25 years – Dick Pountain's book [2] appeared in 1987.

Solution

> *Arguing that Java is better than C++ is like arguing that grasshoppers taste better than tree bark. (Thant Tessman)*
>
> *Good judgement comes from experience, and experience comes from bad judgement. (Fred Brooks)*

Discussing existing Forth OOP is mostly an exercise in discussing deficiency, with individual authors protecting their package at the expense of the community. Since nobody is prepared to accept that another package is good enough, the only solution is to start again, abandoning all the existing packages.

All authors of existing OOP packages have to accept that, in one way or another, their packages are unacceptable to other people. We then just have to pick one package to be the basis of common practice. This is not a Forth200x exercise, it is much more a Forth library exercise.

Objectives

1) Acceptable notation
2) Documented
3) Portable
4) Debuggable
5) Champion to maintain it

Acceptable notation

People are part of the design. It's dangerous to forget that. (Anon)

The notation has to be acceptable both to users and to people who consider themselves to be experts in the Forth OOP field – this probably includes everyone who has written one. I have no strong opinions on this topic.

Documented

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

If the package is not documented it does not really exist in a usable form. Documentation is not just the manual, it includes the commenting in the source code. Undocumented code is just lazy code by someone who does not care about other people – it's the opposite of social programming.

There are two forms of documentation, user and implementer. Both must exist.

At the very least, in the source code every word shall have a stack comment and a description of what it does.

Portable

In order to gain traction, the package must be available on the popular commercial and Open Source Forth systems.

In order to achieve this, people have to be able to port the package. They will be able to do this if the package is well documented for the implementer as well as the user.

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)

Someone who is porting this code will make mistakes, so the code must be simple enough to be debugged. For this reason, it will have to be much better than most existing packages. Writing portable code is not easy.

The portability requirement has the perhaps unfortunate side-effect of ruling out packages that require particular implementation techniques. However, such packages tend to be difficult to port to systems without that particular implementation. It is rare to find a notation that cannot be implemented, but not uncommon to find a notation that is much easier with a particular implementation.

Debuggable

It has been discovered that C++ provides a remarkable facility for concealing the trivial details of a program - such as where its bugs are. (David Keppel)

The trouble with C++ is that it requires gurus to maintain it. Gurus don't do maintenance. (Anon)

One reason that people find OOP code hard to debug is that one of the design decisions is encapsulation – you don't know what is going on. Debugging requires you to be able override the encapsulation should you need to.

Some level of application-level debugging should be built in to the package. Some users like tools such as a "dot parser" to open up a class, for example

```
 MyLine.p1.x0 @:
```

applies the @: method to the x0 component of the line defined by points p1 and p2.

Championed

No package succeeds unless it has an interested, active and enthusiastic maintainer. There is a large number of Forth OOP packages which have no active maintainer.

Conclusions

Collaborative software development is no longer difficult. A large number of web sites exist to support software development and exposure. Standardising an OOP package is not about the Forth200x process, it is about common practice.

The hardest part of the process is to realise that common practice is more important than preserving the minutiae of a package that you are familiar with.

I have my own opinion about where I want to start from [3], but fully realise that it may not be the final design, but is just a suitable starting point.

Acknowledgements

A large number of people have informed my opinions on Forth OOP packages in recent years. Among them are
 Leon Wagner, Manfred Mahlow, Doug Hoffman, Peter Knaggs

References

[1] A Survey of Object-Oriented Forths, Brad Rodriguez, 1996
http://www.bradrodriguez.com/papers/oofs.htm

[2] Dick Pountain *Object-oriented Forth*
Academic Press Limited, London, 1987

[3] FMS – Forth Meets Smalltalk, Doug Hoffman, 2010
http://soton.mpeforth.com/flag/fms/

A Processor as Hardware Version of the Forth Virtual Machine

Willi Stricker

Springe, Germany,

September, 12, 2011

The structure of the Forth system

in contrast to other programming languages the Forth System consists of several components:

- Forth virtual machine
- programming language
- operating system

The Forth virtual machine
In general the Forth virtual machine is based on a microprocessor of any kind and created by software. It is, at least theoretically, identical for every microprocessor and has a unique interface between hardware

The real Forth processor
It is obvious to combine the hardware part of the Forth virtual machine with its sofware part to a real Forth machine that is made of hardware exclusively. This is now called the real Forth processor and is possibly made for instance by means of a modern programmable hardware (programmable logic like FPGAs).

To evaluate its properties, ist is necessay to find the differences between the Forth virtual machine and a „general“ microprocessor of common construction. Mainly there are two features:

1. There is a parameter stack instead of the customary registers
2. Every instruction is just an address.

The Forth virtual machine and the Forth system have two kinds of instructions:
<u>1. Primitives</u>
They are written in assembler code of the used processor. They correspond to the machine code instructions (assembler instructions) of a general microprocessor.
<u>2. High level instructions</u>
Instructions that ar written in Forth (instructions that are combined of primitives). They correspond with subroutines of general microprocessors.

As mentioned bevore, both kinds of instructions in a Forth system are represented by addresses and are accesed by them.

The Forth virtual machine contains an „inner interpreter“ mostly named „NEXT-routine“. It is a „program switch“ for both kinds of instructions. For distinguishing them the instructions need an instruction prefix that is a (sub)program address. For primitives that is the address of their own assembler program, for high level instructions it is the address of a small subroutine (mostly called „DOCOL-routine“), pushing the return address to the return stack and branching to the subroutine (replacing a call instruction). High level instructions are terminated by a return instruction (;S = SEMIS-routine), that returns to the calling instruction by popping the address from the return stack and jumping to it.

The coclusion is: **There is no explicit call instruction in a Forth system.**

The Forth virtual machine makes no statements about hardware properties. That is especially true for the interrupt system. Generally the hardware of the microprocessor is used. But by the way that is true for all common programming languages.

Demands for a real Forth processor
It should have the same (software) features like the Forth virtual machine and additional hardware features of a micro processor.

The construction of a real Forth processor

Afterwords a model is shown that meets the above conditions as much as possible.
It got the name **STRIP (STack Related Instructions Processor)**

it contains the following hardware equipment:

- a separate parameter stack and a separate return stack each
- an interrupt interface
- the means to connect external or internal memory
- the means to connect internal or external input and output elements

it uses only three pointer registers:
1. parameter stack pointer: SP
2. return stack pointer: RP
3. instruction pointer: IP

There are no additional pointers or registers (especially no flag register!)

Minimum instruction set:
In Forth systems generally a set of primitives are available programmed in assembler. For the real Forth processor they have to be constructed in hardware. To minimize the hardware a “minimum instruction set“ is provided chosen due to the following critera:
It contains only instructions that are mandatory for the design of a complete Forth system and additional instructions that are mandatory for hardware control.

System instructions (used by the compiler only)

;S	(-)	return = pop the address from the return stack and store it to IP
LIT	(- data)	load immediate data on stack
BRANCH	(-)	branch to the following address
?BRANCH	(data -)	branch to the following address if data equals zero, else continue

Indirect instruction and subprogram call

EXECUTE	(address **-**)	execute instruction (address on top of stack)

Access to the parameter and the return stack pointer

RP@	(- RP)	get RP
RP!	(RP -)	store RP
SP@	(- SP)	get SP
SP!	(SP -)	store SP

Return-Stack manipulation:

R>	(- data)	pop data from return stack
>R	(data **-**)	push data to return stack

Parameter-Stack manipulation (random access to the parameter stack):

DROP	(data -)	drop data from stack
PICK	(position - data)	load data from relative stack position
-PICK	(data position -)	store data on relative stack position

Memory access

@	(address -> data)	fetch data from memory address
!	(data address ->)	store data on memory address

Logic functions

INVERT	(data - result)	bitwise NOT
AND	(data1 data2 - result)	bitwise AND
OR	(data1 data2 - result)	bitwise OR

arithmetic functions

+C	(data1 data2 - result carry)	add data1 to data2 with sum and carry (lsb)
U2/C	(data - carry result)	shift right one bit, with result and carry (msb)

Special byte instructions

CSWAP	(byte1|byte2 - byte2|byte1) swap bytes in data
C@	(address - 0|byte) fetch byte from address (upper byte = 0)
C!	(byte address -) store byte to address (only lower byte, upper byte is discarded)

Processor control instructions (interrupt instructions)

DISINT	(-) disable Interrupts
ENINT	(-) enable Interrupts

These 26 instructions are sufficient for the construction of a complete Forth system (remark: some of these instructions are not defined or usual in Forth).

The real Forth processor has two particularities against the Forth virtual machine.

1. No instruction prefix
as discribed earlier, the Forth virtual machine needs an instruction prefix to distinguish primitives and high level instructions by the NEXT routine. This is omitted in the real Forth processor. So memory space and access time is saved. Now the type of insruction has to be determined by the program address. While the primitives don't have real addresses pseudo addresses are created instead in a reserved address space.

2. Return bit
Every code address is an even one (two bytes of code). So the right most bit (least significant bit) is always zero! This bit is free for another information and used as „return bit“. The return bit causes a return after the addressed instruction has been executed, independently weather the instruction is a primitive or a high level one. In a subroutine the last instruction gets a return bit. The explicit return instruction is omitted.

While programming there are situations, that need an explicit return instructions. So in the previously defined instruction set a new insruction is inserted (NOP = no operation) and the return instruction is replaced by a NOP with a return bit:

NOP (-) No Operation

The following special cases need a return instruction with address:
Trivial case: A subprogram must have at least one instruction, even if it does nothing. The only instruction is the return.
Structures (branch instructions): A branch always needs a valid address to branch to, that mostly is the address of the following instruction. But on the end of a subprogram there is no instruction left, so it needs a NOP (with a return bit).

The STRIP Forth processor in reality

First objective of its construction obviously is high speed. So as much as possible has to be executed concurrently. Then the time of execution is resricted only by accessing external memory (access time). This limits the minimum of time. Therefore as few as possible accesses to the memory (bus) should happen.
The stacks are separated physically from the main memory. They are accessable by stack instructions only. As a result the processor has access to both stacks and main memory concurrently.

The STRIP kernel contains the three pointers, their control and the whole instruction set. It uses a clock-signal and has interfaces for the parameter stack, the return stack, an interrupt interface, and data and address busses for memory and peripheral elements (memory mapped I/O).
The kernel is extended to a working processor by adding a clock element, parameter and return stacks and an reset/interrupt controller.
A complete working STRIP Forth system needs additional memory (RAM and/or ROM) for data and programs, as well as a program and debug interface. Furthermore it is possible to add input/output elements.

Timing
The Subroutine call and most of the primitives need only one bus access and with it only one clock cycle!
Two bus accesses and clock cycles are used only for instructions

- that need an additional parameter: **LIT, BRANCH, ?BRANCH,**
- that access memory: @**, !, C@, C!,**
- that need a second cycle with a return bit: **R>** with Return bit**, RP!** with Return bit.

Notes:
instructions with operands need two memory spaces, instructions with memory access only one. Instructions with a return bit don't need an additional bus access and no additional time. Exceptions only the instructions RP! and R>, they need a second cycle for the return (without bus access).

Pseudo addresses for primitives
In the Forth virtual machine the primitive instruction are assembler code with memory addresses. For the STRIP Forth processor pseudo addresses are defined. These addresses (numbers) address primitives and are not usable for high level instructions. But they can be used for memory. The pseudo addresses are placed into an area outside the program memory. It is a good idea to put them on the start of the memory map (starting with addrerss zero).

With the minimum instruction set 26 addresses are used. In a 16 bit system 52 bytes are necessary. In practice 64 (2^6) bytes are reserved for pseudo addresses.

Restart and interrupts
For the restart memory space is reserved for the restart address. For the interrupts there is also one space each for its address that contains the start of its corresponding interrupt service routine.

The restart is an primitive, activated by hardware. It needs one clock cycle (address fetch).
An interrupt is activated by hardware asynchronously. It is checked in S0 by the kernel and executed after the concurrently fetched instruction is executed. The interrupt doesn't need any overhead time beside its own program.

Final conclusion and construction

The STRIP Forth processor is firstly programmed into an FPGA (Actel eval kit with APA 075 with 3075 Tyles and 3 kb RAM) that is sufficient for a 16-bit system for experimental proving of the processor.

Forth on the ARM Cortex-M1 FPGA Development Kit

Leon H. Wagner
FORTH, Inc.
www.forth.com

Abstract

This paper describes the instantiation of an ARM Cortex-M1 CPU core on an Altera Cyclone III FPGA and the development of a simple Forth application to run on it.

The CPU core used here is the ARM Cortex-M1 FPGA Development Kit from ARM, Ltd. The Altera Quartus II environment is used to design an ARM system, including memory and embedded peripherals. SwiftX-ARM is used to develop and interactively test a simple Forth application on the newly instantiated Cortex-M1 core in the FPGA.

Section 1: Overview

1.1 About SwiftX

SwiftX is FORTH, Inc.'s interactive cross compiler for the development of applications for embedded microprocessors and microcontrollers. SwiftX is based on the Forth programming language and is itself written in Forth.

SwiftX has been ported to the following microprocessor and microcontroller families:

- Atmel, Cirrus, Nuvoton, NXP, ST Microelectronics (and other) ARM cores
- Freescale ColdFire
- Freescale 6801 / Renesas 6303
- Freescale 6809
- Freescale 68HC11
- Freescale 68HC12 (S12, S12X, etc.)
- Freescale 68HCS08
- Freescale 68K
- Aeroflex UTMC 69R000
- Intel (NXP, SiLabs, others) 8051
- Atmel AVR
- Renesas H8H (H8/300H, H8S)
- Intel (AMD, other) i386

- Texas Instruments MSP430
- Patriot PSC1000
- Harris RTX2010

SwiftX cross-compilers run in the SwiftForth programming environment. They inherit all the features of SwiftForth and extend its interactive development environment to manage multiple program and data spaces as well as to generate the code and data that fill them.

1.2 About the ARM Cortex-M1 FPGA Development Kit

The ARM Cortex-M1 FPGA Development Kit, available free of charge from ARM, Ltd., is delivered as an SOPC Builder design optimized for the Altera Cyclone III FPGA Starter Kit. The ARM system bus has been adapted to the Altera Avalon system interface for this implementation. However, there are no architectural changes from the standard ARM Cortex-M1 Core.

The Cortex-M1 processor is intended for deeply embedded applications that require a small processor (i.e., low gate count) integrated into an FPGA. The processor core implements the ARM architecture v6-M Thumb Instruction Set Architecture (ISA) with some 32-bit Thumb-2 extensions.

ARM Cortex-M1 FPGA Development Kit is fully compatible with Altera's SOPC Builder and Quartus II tools. This application note demonstrates how to build a simple system from the ARM Cortex-M1 FPGA Development Kit and a few of the standard Altera peripheral IP blocks, then generate a bitfile of the whole system and program it onto a Cyclone III device.

1.3 Requirements

The following items are required to implement the system described in this application note:

- SwiftX-ARM from FORTH, Inc. (www.forth.com)
- Quartus II (subscription or web edition), version 8.0 or later, from Altera. We used Quartus II Web Edition 11.0 for this paper. (www.altera.com)
- ARM Cortex-M1 FPGA Development Kit. (www.arm.com)
- Cyclone III Starter FPGA Kit. (www.altera.com)
- A simple USB-to-serial port converter, such as the DEV-09873 from Sparkfun Electronics. (www.sparkfun.com)

Section 2: Working with Quartus II and SOPC Builder

This section describes the development of a Quartus II project using the SOPC Builder tool to fabricate the ARM Cortex-M1 core with memory and peripherals. The system is synthesized and downloaded to a Cyclone III FPGA Starter Kit for testing.

We are using the Windows version of Quartus II and are placing our project files in the directory **c:\projects**. Make any necessary adjustments to path names if you are using the Linux version of Quartus II.

2.1 Starting a new Quartus II Project

Launch Quartus II, dismissing any startup wizards or tips. Select File > New Project Wizard and fill in the blanks as follows:

- Set the working directory for the project to **C:\projects\cm1\fpga**.
- Name the new project **cm1**.
- Set the top-level design entity name to **cm1_top**.
- Click the “Next” button (say “Yes” to create the new project directory).
- When you get to the “Add Files” step, just skip it for now. We’ll add some files after we build the ARM Cortex-M1 system in SOPC builder.
- For the “Family & Device Settings” step, select the device that matches the one on your Cyclone III FPGA Starter Kit (e.g., EP3C25F324C6).
- Click “Next” through the remaining wizard screens and “Finish” on the last one.

2.2 Working with SOPC Builder

The Quartus II SOPC Builder tool is used to build and integrate the ARM Cortex-M1 CPU, clocks, memory, PIO, and UART components. The Cortex-M1 CPU core is supplied by ARM, Ltd. The remainder of the IP components are from Altera and are distributed with the Quartus II system.

Select Tools > SOPC Builder in Quartus II and follow the directions in the following sections to build the ARM Cortex-M1 system.

For “System Name” in the “Create New System” dialog box, enter “cm1” and select Verilog as the HDL. Do not use the same name assigned to the top-level entity.

2.2.1 Adding the ARM Cortex-M1 core

Look for “ARM Cortex-M1 Processor” in the Component Library pane of the SOPC Builder window, select it as shown in Figure 1 and click the “Add” button[1].

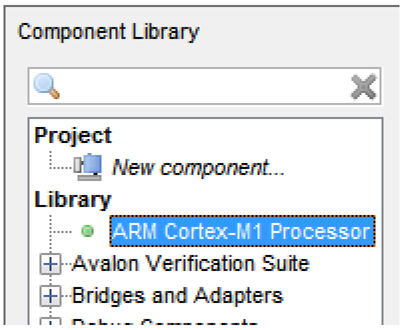


Figure 1. ARM Cortex-M1 Processor component selection

Configure the ARM Cortex-M1 processor as follows:

- Uncheck the “Debug enabled” box
- Set number of IRQs to 8
- Set ITCM size to 16 kB
- Uncheck “Read only” for the ITCM
- Check “Initialize ITCM contents”
- Set ITCM “From file” field to **itcm.hex** (this should be the default)
- Set DTCM size to 8 kB
- Uncheck “Initialize DTCM contents”
- Click the “Finish” button to add the ARM Cortex-M1 core to the SOPC design

2.2.2 Adding parallel I/O (PIO)

The SwiftX demo for the Cortex-M1 will use a 4-bit PIO to drive the four user LEDs and read the four user pushbuttons on the Cyclone III FPGA Starter Kit board.

From the Component Library, select Peripherals > Microcontroller Peripherals > PIO and click the “Add” button. Configure the PIO component as follows:

- Set the width to 4 bits
- Set direction to “InOut”
- Set the output port reset value to 0x0F
- Leave the rest of the PIO option boxes unchecked
- Click the “Finish” button to add the PIO to the SOPC design

2.2.3 Adding a UART

The SwiftX Interactive Development Environment uses a fast serial port as its

1.If the ARM Cortex-M1 Processor item is not present, use Tools > Options to add the path to ARM/CortexM1_DevKit/Component/arm_avalon_cortexm1 to the IP Search Path.

Cross-Target Link (XTL) debug interface.

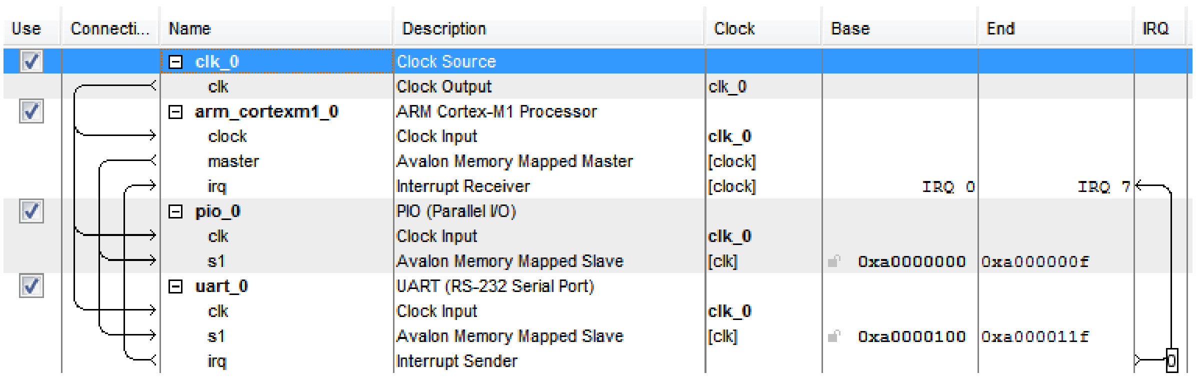
From the Component Library, select Interface Protocols > Serial > UART (RS-232 Serial Port)[1]. Configure the UART component as follows:

- Under "Basic Settings,", use the defaults (no parity, 8 bits, 1 stop bit, 2 synchronizer stages, no RTS/CTS, no EOP).
- Select a fixed baud rate of 115200 with error tolerance of 0.01
- Leave the remaining options unchecked
- Set the transmitter baud rate to "Actual" (not "Accelerated")
- Click the "Finish" button to add the UART to the SOPC design

2.2.4 Connecting the components

In SOPC builder, click on the "Filter" button and set the Filter pull-down to "All" so you can see all the connections.

Set the bus, clocks, and interrupt connections as well as the component base addresses and IRQ as shown in Figure 2.



Use	Connecti...	Name	Description	Clock	Base	End	IRQ
✓		clk_0	Clock Source				
		clk	Clock Output	clk_0			
✓		arm_cortexm1_0	ARM Cortex-M1 Processor				
		clock	Clock Input	clk_0			
		master	Avalon Memory Mapped Master	[clock]			
		irq	Interrupt Receiver	[clock]	IRQ 0	IRQ 7	
✓		pio_0	PIO (Parallel I/O)				
		clk	Clock Input	clk_0			
		s1	Avalon Memory Mapped Slave	[clk]	0xa0000000	0xa000000f	
✓		uart_0	UART (RS-232 Serial Port)				
		clk	Clock Input	clk_0			
		s1	Avalon Memory Mapped Slave	[clk]	0xa0000100	0xa000011f	
		irq	Interrupt Sender				0

Figure 2. Component connections and settings

The base address for the PIO is 0xA0000000 and the UART is 0xA0000100. Use IRQ 0 for the UART "interrupt sender" element.

2.2.5 Generate the SOPC system

Click the "Generate" button to generate the system. When prompted to save the file, name it **cm1.sopc**.

After the generation is complete, you should see the "System generation was successful" message in SOPC Builder's output pane. Click the "Exit" button to close the SOPC Builder window, saving any changes to the **cm1.sopc** file.

1.Do not use the JTAG UART component.

2.3 Adding the Top-Level Design

We need to add a top-level design file to make some connections to the ARM Cortex-M1 core and peripherals generated by SOPC Builder. We'll do this with a simple block diagram/schematic file.

Select File > New and choose "Block Diagram/Schematic File" in the New dialog box, then click "Ok." A blank block1.bdf workspace should appear in the right window pane. Follow these steps to create and save the top-level design file:

- Double-click in the center of the schematic grid workspace. This should open the Symbol dialog.
- In the "Libraries" pane, expand the "Project" item and select cm1. This should show the cm1 block entity in the right pane.
- Click "Ok".
- Position the block roughly in the center of the schematic and click to position it.

Now we need to tie a few of the unused inputs to Vcc and Gnd levels:

- Double-click on the schematic to open the Symbol dialog.
- Expand the Quartus Libraries item and select Primitives > Other > Vcc.
- Click on the schematic to place Vcc next to the cm1 node's DBGRESETn input.
- Select a Node tool and connect Vcc to DBGRESETn.
- Double-click on the schematic to open the Symbol dialog.
- Expand the Quartus Libraries item and select Primitives > Other > Gnd.
- Click on the schematic to place Gnd next to the cm1 node's EDBGRQ input.
- Select a Node tool and connect Gnd to both EDBGRQ and NMI inputs.
- Use the Selection tool and select the entire cm1 object.
- Right-click on cm1 and select "Generate Pins for Symbol Ports."
- Save the schematic file as **cm1_top.bdf.**
- In the Project Navigator pane, select the Files tab, right-click on **cm1_top.bdf** and select "Set as Top-Level Entity."

Figure 3 shows a representation of the top-level design schematic.

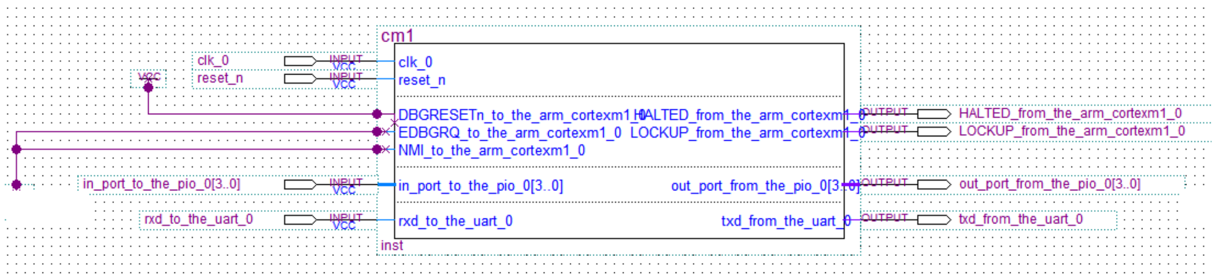

cm1
clk_0
reset_n
VCC
INPUT
VCC
clk_0
reset_n
DBGRESETn_to_the_arm_cortexm1_0
EDBGRQ_to_the_arm_cortexm1_0
NMI_to_the_arm_cortexm1_0
HALTED_from_the_arm_cortexm1_0
LOCKUP_from_the_arm_cortexm1_0
OUTPUT
HALTED_from_the_arm_cortexm1_0
LOCKUP_from_the_arm_cortexm1_0
in_port_to_the_pio_0[3..0]
in_port_to_the_pio_0[3..0]
out_port_from_the_pio_0[3..0]
out_port_from_the_pio_0[3..0]
rxd_to_the_uart_0
rxd_to_the_uart_0
txd_from_the_uart_0
txd_from_the_uart_0
inst

Figure 3. Top-Level design schematic

2.4 Synthesizing the System

The files generated by SOPC Builder along with our top-level design need to be analyzed so we can assign pin connections to the outside world.

Select Processing > Start > Start Analysis and Elaboration so Quartus II can analyze the design and figure out its pins and connections. The process should complete with the message "Analysis & Elaboration was successful."

2.4.1 Assigning nodes and pins

- Select Assignments > Assignment Editor to make the pin assignments.
- In the Assignment Editor window, click on a cell in the "To" column and select Node Finder. Select "Pins: all" ("Look in:" should be **cm1_top**) and click "List" to see all the internal pins.
- Select the pins listed in the "Signal" column of Figure 4 and add them to the pane on the right.
- In the assigment editor, set the Assignment Name field for all pins to "Location" and enter the physical pins listed in Figure 4 into the Value column and set the Enabled column to "Yes" for each pin.
- Close the Assignment Editor, saving the new assignments when prompted.

NET NAME	SIGNAL	PIN
50MHZ	clk_0	V9
CPU_RST_N	reset_n	N2
HSMC_SCL	rxd_to_the_uart_0	F3
HSMC_SDA	txd_from_the_uart_0	E1
KEY0	in_port_to_the_pio_0[0]	F1
KEY1	in_port_to_the_pio_0[1]	F2
KEY2	in_port_to_the_pio_0[2]	A10
KEY3	in_port_to_the_pio_0[3]	B10
LED0	out_port_from_the_pio_0[0]	P13
LED1	out_port_from_the_pio_0[1]	P12
LED2	out_port_from_the_pio_0[2]	N12
LED3	out_port_from_the_pio_0[3]	N9

Figure 4. Pin assignments

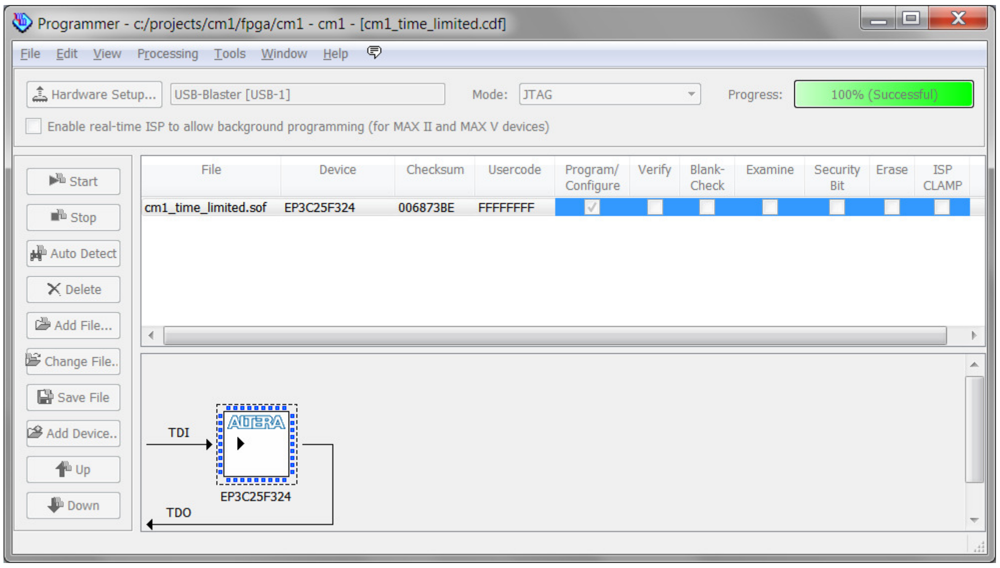
2.4.2 Compiling the system

Copy the **itcm.hex** object file from the **projects\cm1\firmware** project directory to the **projects\cm1\fpga** directory. If there is no **itcm.hex file** in the SwiftX project directory, launch the SwiftX project in that directory and do a "Build" to compile the ARM Cortex-M1 SwiftX kernel.

Then go back to the Quartus II window and select Processing > Start Compilation. The compilation step takes a few minutes to complete. When it is done, there will be some warnings in the output window, but there should be no error messages.

2.4.3 Programming the FPGA

Connect a USB port on the computer running Quartus II to the USB Blaster embedded on the Cyclone III FPGA Starter Kit board. Select Tools > Programmer and under Hardware Setup, select the USB Blaster, setting its mode to JTAG. Select the **cm1.sof** file[1] and click the “Start” button. Figure 5 shows the programmer dialog box after loading the FPGA.


Programmer - c:/projects/cm1/fpga/cm1 - cm1 - [cm1_time_limited.cdf]

Hardware Setup... USB-Blaster [USB-1]   Mode: JTAG   Progress: 100% (Successful)

☐ Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank- Check	Examine	Security Bit	Erase	ISP CLAMP
cm1_time_limited.sof	EP3C25F324	006873BE	FFFFFFFF	✓	☐	☐	☐	☐	☐	☐

Figure 5. Programmer dialog box

The target board should start running the ARM Cortex-M1 SwiftX program from **itcm.hex.**

1.The exact name of the SOF file may vary.

Section 3: Working with SwiftX

3.1 Connecting the debug interface

A serial port is used for the debug interface from the SwiftX Interactive Development Environment to the target ARM Cortex-M1 CPU inside the Cyclone III FPGA. The four pins on the Cyclone III FPGA Start Kit board originally intended for I2C connections have been reassigned in our design to be used as the UART pins. The pin assignments are as follows:

SDA	TXD
SCL	RXD
3V3	N/C
GND	GND

Connect a USB cable from the computer running SwiftX to a USB-to-serial converter[1] connected to the target board as noted above.

3.2 Establishing an interactive session

Click on the “Debug” button in the SwiftX tool bar (or select Tools > Debug or press the F9 shortcut key) to establish a serial connection with the target CPU and launch an interactive debug session. The output in the SwiftX debug window should look something like this:[2]

```
INCLUDE DEBUG

   Start      End     Size    Used   Unused   Type   Name
    0000     3FFF    16384    7640     8744   CDATA  PROG
20000000 200010FF     4352      12     4340   IDATA  IRAM
20000100 20001FFF     7936    2316     5620   UDATA  URAM

TARGET READY
SwiftX/ARM Cortex-M1 Altera  ok
```

Interactive development and testing can now proceed as described in the *SwiftX Reference Manual* and the *SwiftX ARM Target Reference Manual.*

3.3 Project source files

The project directory **projects\cm1\firmware** contains the usual set of files as documented in the the *SwiftX Reference Manual.* In addition to these, the following project-specific files are supplied:

1.TTL levels required.
2.Exact memory usage may vary.

- **reg_fpga.f** — Defines the memory-mapped register interface to the Altera PIO and UART components.
- **xtl_uart0.f** — Implements the target side of the serial cross-target link (XTL) debug interface using the Altera UART added to the design in 2.2.3.
- **leds.f** — Uses the Altera PIO core component to interface to the four user LEDs on the Cyclone III Starter FPGA Kit board.
- **buttons.f** — Uses the Altera PIO core component to interface to the four user pushbuttons on the Cyclone III Starter FPGA Kit board.
- **demo.f** — Implements a high-level `WINK` function and assigns a "walking" LED wink pattern to a SwiftX `BACKGROUND` task.

The output file generated by the SwiftX `BUILD` process is **itcm.hex.** It must be copied or moved into the **projects\cm1\fpga** directory prior to compiling the FPGA project in Quartus, as described in 2.4.2.

3.4 Demo Application

The demo application consists of a background task that drives the LEDs in a "chase" pattern. The user pushbuttons are tested in the main application loop to change the speed of the chasing pattern (by setting the LED on/off time to a longer or shorter period) and to perform an "all on" LED test.

Excerpts from the Forth source files are included in the next section.

Conclusion

The instantiation of the ARM Cortex-M1 processor on an Altera FPGA using SOPC Builder is a simple task that can be accomplished even by someone with limited FPGA programming experience.

Additional memory and peripherals can be added in SOPC Builder for a much more elaborate implemenetation. Custom logic can be included in the design from the top level using conventional FPGA tools.

Porting a Forth cross compiler and interactive debug environment to this CPU is no more daunting than a port to a conventional CPU.

Section 4: Program Listings

4.1 LED Outputs

```
{ -----------------------------------------------------------------
LED control

!LED writes a pattern to the 4 LEDS in bits [3:0] of the PIO output
register.
---------------------------------------------------------------- }

: !LEDS ( x -- )   INVERT PIO_BASE ! ;
: /LEDS ( -- )   0 !LEDS ;
```

4.2 Button inputs

```
{ -----------------------------------------------------------------
Read switches

?PRESSED returns true if button defined by mask is pressed.
BUTTON1..BUTTON4 define masks for passing to ?PRESSED.
---------------------------------------------------------------- }

: ?PRESSED ( mask -- flag )   PIO_BASE @ AND 0= ;

1 CONSTANT BUTTON1
2 CONSTANT BUTTON2
4 CONSTANT BUTTON3
8 CONSTANT BUTTON4
```

4.3 Demo program

```
{ -----------------------------------------------------------------
LED demo program

LEDTIME holds the number of milliseconds for LED on and off period.

FAST, MEDIUM, and SLOW are the number of milliseconds for three speeds.
SPEED sets LEDTIME to u milliseconds.

CHECK-BUTTONS does the following based on button(s) pressed:
  Button 1: Set slow speed
  Button 2: Set medium speed
  Button 3: Set fast speed
  Button 4: Turn all LEDs on

LED-CHASE performs one loop of the LED "chase" sequence.
CHECK-BUTTONS is called just before the on or off delay time.

CHASER is the task assiged to perform the chasing LEDs behaviour.
/CHASER starts the task.
DEMO initializes the LEDS and time period, then starts the task.
---------------------------------------------------------------- }

VARIABLE LEDTIME        \ Millisconds on/off time

50 CONSTANT FAST
100 CONSTANT MEDIUM
250 CONSTANT SLOW

: SPEED ( u -- )   LEDTIME ! ;
: CHECK-BUTTONS ( -- )
   BUTTON1 ?PRESSED IF  SLOW SPEED  THEN
   BUTTON2 ?PRESSED IF  MEDIUM SPEED  THEN
   BUTTON3 ?PRESSED IF  FAST SPEED  THEN
   BUTTON4 ?PRESSED IF  $0F !LEDS  THEN ;

: LED-CHASE ( -- )   1
   4 0 DO  DUP !LEDS  CHECK-BUTTONS  LEDTIME @ MS  2*  LOOP
   4 0 DO  2/ DUP !LEDS  CHECK-BUTTONS  LEDTIME @ MS  LOOP DROP ;

|U| |S| |R| BACKGROUND CHASER

: /CHASER ( -- )   CHASER ACTIVATE
   BEGIN  LED-CHASE  AGAIN ;

: DEMO ( -- )
   0 !LEDS  MEDIUM SPEED  CHASER BUILD  /CHASER ;
```

Crash Never

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
4-5 Great Western Court
Ross-on-Wye
Herefordshire
HR9 7XP
UK
Tel. +44 1989 768080
Email njn@micross.co.uk

Abstract
Two approaches are contrasted in the search for reliability of a large and complex Forth system.

1. Introduction

In our terms, the reliability of a program means that it continues to work without stopping, for a very long period - at least months and maybe years.
Two possible approaches can be imagined for obtaining software reliability:
a) "Crash early, crash often"
You make a system that is extremely intolerant of programming errors, and thus try to force them to reveal themselves at an early stage.
b) "Crash never"
You accept that you can never be a perfect programmer. You make a system which tries to struggle on despite programming errors.
It is a combination of these approaches which we have been using (mostly by accident) for many years.

2. Crash early, crash often

This approach is best suited to applications which can be thoroughly and systematically tested. Every possible program path can be simulated, and hopefully a wide variety of potential data.
This does not always work, no matter how many resources are put into the testing process. It has been claimed that Bill Gates managed to "blue screen" every new version of Windows in front of a live audience.

3. Crash never
This is perhaps better labelled "Don't crash in front of the customer". This approach is better for systems for which thorough testing is not an economic possibility. In our case, it's not economic because each copy of our software is different, and is sold only once. And the reason it is different is that it is controlling a set of mechanical equipment which is unique to each customer.

4. Why do programs crash?
First, we need to set aside logical errors. For example, the customer may inform you "every time I press this button this happens, instead of that". These are usually not programming errors, but errors of specification (the mechanical engineer explained it wrongly to the programmer). Logical errors are usually easy to fix. By crash I mean that all or part of the program stops working. Typical causes for this are:
a) Invalid address errors
b) Huge, or infinite loops
c) Division by zero, or overflow
d) Race, or gridlock conditions

5. Invalid address errors, and what to do about them
Most invalid addresses are caused by stack errors. In a Forth environment, the compiler does not check the number and type of parameters which are passed to and from a word (function). This leads to the possibility of accumulating stack errors, even if there is no logical error in the function. Depending on how frequently a function is called, this may result in an invalid address either immediately, or at some time in the future.

```
: DIE-IMMEDIATELY
  1000000 0 DO DROP ( or DUP, if you fancy ) LOOP
;

: DIE-IN-A-MONTH
  BEGIN DROP 1000000 WAIT AGAIN
;
```

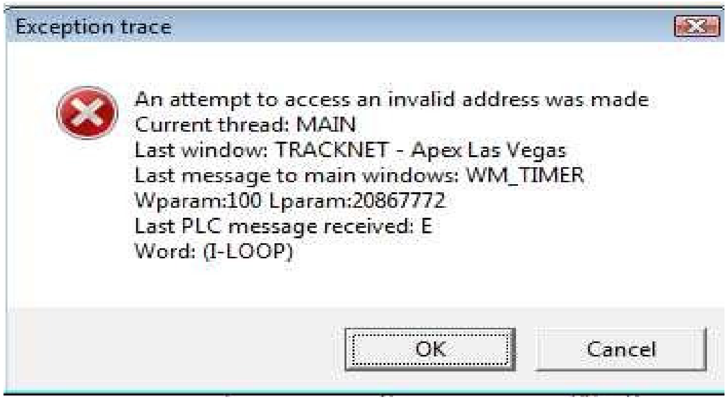
In a Windows program, the majority of the code handles responses to messages that Windows sends you - for example when a key is pressed or the mouse is moved. It so happens that the compiler that we have been using for many years (MPE ProForth V2.1) creates a new stack, every time a new message is processed. Quite by accident, an extremely fault-tolerant system is created - the system is largely immune to the most common programming error! It is due to this one factor (and certainly not to our own programming expertise) that we have a reputation for producing highly reliable software.

Of course, more sophisticated Windows programs consist of more than reponses to Windows messages. Additional threads of execution (TASKs in Forth) are created, to carry out background processing. These tasks essentially consist of infinite loops, called at predetermined intervals, and are therefor prone to die-in-a-month syndrome. Our solution is to create a stack guardband.

```
: SETDEPTH ( n--- ? ) \ Set stack depth as protection against under/overrun
  DEBUGGING 0= TURNKEY? OR    \ Always protect if in turnkey mode
  STACKPROTECT @ OR IF        \ And in debug mode, if stack protection on
    BEGIN
      DUP 1+ DEPTH <>         \ Depth not as required
    WHILE
      DUP DEPTH < IF          \ Too much
        NIP                   \ Down a bit
      ELSE                    \ Too little
        0 SWAP                \ Up a bit
      THEN
    REPEAT
  ELSE
    DROP                      \ The requested depth
  THEN
;

TASK: MYTASK( --- ) \ My processing task
  BEGIN
    10 SETDEPTH         \ Guard against stack errors
    50 WAIT             \ Do it every 50ms
    DO-SOME-WORK        \ The background work
  AGAIN
;
```

Although by far the majority of invalid address errors are caused by stack mistakes, they can also be caused by other factors, such as the miscalculation of an index into an array. There is nothing we can do to immunise the system against these, so the response is to indicate in as much detail as possible the location of such an error, whenever it occurs.



```
An attempt to access an invalid address was made
Current thread: MAIN
Last window: TRACKNET - Apex Las Vegas
Last message to main windows: WM_TIMER
Wparam:100 Lparam:20867772
Last PLC message received: E
Word: (I-LOOP)
```

The message box describes

- the type of exception
- the thread in which it occurred
- details of the last window message which was started
- special information about a particularly error-prone communications function
- the name of the Forth word which caused the exception.

This is a very valuable tool. The code required to produce this information is complex, and in this paper, there is only time to examine the outmost word. Any delegate is welcome to see the full source if they are interested. Note that the detail is highly compiler dependent, and adjustments would be needed for other compilers.

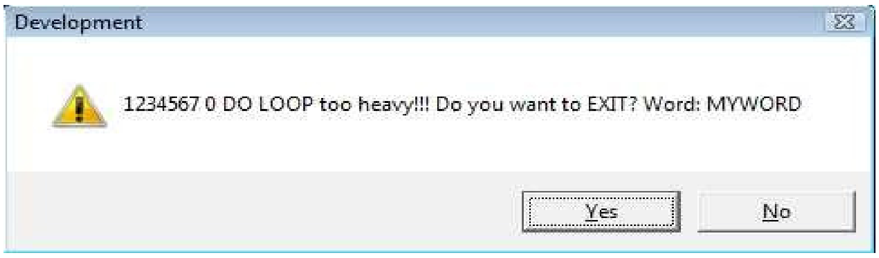
```
: (EXCEPTION-HANDLER) { | esi -- } \ Display exception message box
  WINAPPHANDLE@                                \ Owner
  LOAD-ADDR @ @                                \ Get exception code
  GET-EXCEPTION-STRING $>ASCIIZ                \ Convert to string
  ZCRLF Z+ Z"" Current thread: " Z+            \ Show name of thread
  SELF ZTHREAD Z+
  ZCRLF Z+ Z"" Last window: " Z+               \ Show name of last window
  CURR-WINDOW HANDLE ZWINNAME Z+
  ZCRLF Z+ Z"" Last message to main windows: " \ Show last main message
  Z+ LASTWM @ WMNAME Z+
  LASTWM @ WM_COMMAND = IF                     \ It was a command message
    ZCRLF Z+ Z"" Command: " Z+                 \ Show command
    LASTCM @ ZTEXT Z+
  THEN
  ZCRLF Z+    Z"" Wparam:" Z+ LASTWPARAM @     \ Last w l params to main
  ZFORMAT Z+ Z""  Lparam:" Z+ LASTLPARAM @
  ZFORMAT Z+
  ZCRLF Z+ Z"" Last PLC message received: " Z+ \ Show last PLC message
  LASTPLCMESSAGE Z+
  ZCRLF Z+ Z"" Word: " Z+                      \ Attempt to identify Forth
word
  LOAD-ADDR @ 4 + @ ABS>REL                    \ Get base of context
structure
  112 +                                        \ Offset for
floating_save_area
  12 cells+ @ -> esi                           \ Get Esi
  esi FORTH-BASE - IDENTIFY-IP Z+
  ZCRLF Z+ TRACESTRING Z+                      \ Concatenate trace
information
  Z"" Exception trace"
  MB_APPLMODAL MB_ICONHAND or MB_OKCANCEL OR
  WINMESSAGEBOX IDCANCEL = IF                  \ Display box, did user
cancel?
    BYE
  THEN
  TRACESTRING OFF                              \ Clear trace
  LOAD-ADDR @ OFF                              \ Reset so cold will work OK
  ABORT                                        \ Warm restart
; ASSIGN (EXCEPTION-HANDLER) TO-DO EXCEPTION-HANDLER
```

6. Huge, or infinite loops, and what to do about them

There are two types of loop errors:
a) DO..LOOPs with miscalculated input parameters
b) BEGIN.. with miscalculated WHILE or UNTIL parameters

6a. Miscalculated DO.. LOOPs

Modern PCs are rather speedy, and can process very large loops without noticeable delay. Nevertheless, if a DO.. LOOP is asked to execute, say, a million times, then it is reasonable to assume there may be something wrong.



```
1234567 0 DO LOOP too heavy!!! Do you want to EXIT? Word: MYWORD
```

The word which contains the offending loop is shown. A definition of the words DO, ?DO and LOOP is required to achieve this, and again code is available to anyone interested.

6b. BEGIN.. with miscalculated WHILE or UNTIL parameters

The effect of these, when included in a Windows message, is that the system becomes unresponsive to the mouse and keyboard. If such a problem happens very infrequently, it can be extremely hard to find. Fortunately, although our programs run continuously for very long periods, in practice there is only an operator interacting with the system for a small percentage of that time. If we were able to detect a non-responsive program, and close then restart it automatically, then if we're lucky the customer might not notice! At the very least, we may buy some time to locate the error.

This is one of those occasions when age is an advantage. We can remember writing code for early primitive microprocessors, which were very prone to disruption caused by electromagnetic incompatibility. We therefore "invented" (contemporaneously no doubt with many others) a simple and foolproof external circuit which, if not reset regularly by the software, would in turn reset the microprocessor. We called it the "prodder", and it later became ubiquitous in microcontrollers, as the watchdog.

We therefore came up with a software analog, called "Prod". This is a simple independent buddy program, which is started automatically by the main program. It has two functions:

a) It sends regular messages to the main window of the main program. If this does not respond promptly, it forces the main program to close.
b) It tests regularly for the presence of the main program, and if it is not present, it restarts it, and closes itself.

The latter also deals with another issue. Sometimes, Windows simply closes a misbehaving application without calling its exception handler. This may be because the Forth program has corrupted its own code.

The Prod program satisfies the "simple and foolproof" test - it is only 150 lines long.
The important work is done by a 1s Windows timer.

```
: PRODWIN-TIMER ( hwnd,mess,wparam,lparam---res ) \ 1 second timer
  1 RESTART-TIMER +!                             \ Increment restart timer
  NULL ABS>REL PARAMETER-BUFFER $>ASCIIZ         \ See if main window still
there
  WINFINDWINDOW IF
    0 RESTART-TIMER !                            \ Clear restart timer
  THEN
  1 HUNG-TIMER +!                                \ Increment hung timer
  HTRACKNET @ ISHUNGAPPWINDOW 0= IF              \ Program is responding
    0 HUNG-TIMER !                               \ Clear hung timer
  THEN
  HUNG-TIMER @ HUNG-TIME U> IF                   \ Exceeded hung time
    HTRACKNET @ FALSE TRUE ENDTASK DROP          \ Kill it
  THEN
  RESTART-TIMER @ RESTART-TIME U> IF             \ Exceeded restart time
    START-TRACKNET                               \ Start the main program
    PRODWIN-CLOSE                                \ Close self
  ELSE                                           \ No message yet
    4DROP 0                                      \ Continue
  THEN
;
```

7. Division by zero, or overflow, and what to do about it

It always amazes me that Microsoft and Intel between them invented a problem that was never there before. Just because you divide by zero, they decide to throw a tantrum and close you down! When one had to write the code for a division by hand, this was never an issue. The correct answer to divide by zero is infinity - or at least the closest to infinity that the computer can approximate. So the fix is simply to rewrite the various Forth words to trap for zero, and give the correct answer immediately. I'm not sure, philosophically, whether there is a difference between plus and minus infinity, but I always carry the sign of the dividend into my maximum value, on the basis that I didn't really mean zero, I just meant a rather small number.

8. Race, or gridlock conditions and what to do about them

These problems are usually caused by incorrect interaction between threads. For example, thread A is waiting for X to be set before it sets Y. But thread B is waiting for Y to be set before it sets X.

Again, experience with microcontrollers is useful. These devices often have a large number of potential hardware interrupts triggered by peripherals that require attention - for example, when a serial port has received a character. It is very tempting to write an interrupt service routine for each peripheral in use. But it soon becomes clear that the interaction between the various interrupt service routines (ISRs) and the main program is a major source of programming errors. It is much safer to use a single, timer-driven ISR, and poll all the peripherals. This eliminates all inter-ISR problems. The minimum possible work should be done by the ISR itself, and its interaction with the main program should use only one integer for each distinct operation, so that the interrupt never needs to be disabled and re-enabled to prevent partial data update.

The same principles can be extended to Windows threads.

a) Keep the number of threads to the absolute minimum. For example, if several processes all need attention every 100ms, they should all be called from the same thread. Then, the grouped processes can communicate with each other without special consideration.
b) Do only the absolutely necessary work within the thread - usually either time-consuming, or time-critical operations.
c) Use the Windows messaging system for thread to window communication - it is debugged more thoroughly than anything you could write yourself, and handles all the hard bits.
d) Avoid all the locking mechanisms (such as critical sections) like the plague. They are a prime cause of software errors.
e) Instead, define extremely simple and exactly specified inter-thread communication using single integer reads and writes that cannot be interrupted by the task scheduler.

9. Conclusion

A reliable Windows program can be written in Forth by using a combination of techniques including fault tolerance, highly targeted detection, and adherance to strict programming principles.

NJN
September 2011

SWIG & The Forth Net: Hands-On

Gerald Wodni[*] M. Anton Ertl [†]

Euroforth 2011

Abstract

We have shown the basic functionality of SWIG and The Forth Net in the past. Now we want to provide two basic examples which explain how to use them and to show how easy it is to create C-interfaces or Forth-libraries and share them.

1 MySQL in Forth

As all mayor Forth-systems provide an interface to call libraries written in the C-programming-language, it should be quite easy to load and use a library like MySQL. But as all of them have their very own interface with their own set of words, this task can be quite cumbersome. The Forth extension for SWIG [1] attempts to provide an easier way, by creating a C-source-file which is platform independent and once it gets compiled, outputs the interface information for the target Forth-system.

First we need to create an interface file for SWIG[2]:

```
1 %module mysql
2 %insert("fsiinclude")
3 %{
4 #include <mysql/mysql.h>
5 %}
6
7 %include <mysql.h>
```

Line 1 tells SWIG the name of our module.

Lines 2-5 put Line 4 into the output-file in the section "fsiinclude", which happens to be right after the default #include-directives.

Line 7 orders SWIG to parse mysql.h .

Having completed the interface file we invoke SWIG:

```
$ swig -forth -stackcomments -includeall -I/usr/include/mysql -o mysql-fsi.c mysql.i
```

Depending on our system we may add some more include directories (`-I...`). The resulting file (mysql-fsi.c) is platform independent and is exactly what we can upload to The Forth Net (see Section 2), or share in general.

On the target machine we compile the mysql-fsi.c file using the machine's C-compiler, which includes the C-headers and thereby assigns the correct values for constants, the correct typedefs and cares about all other C-related problems.

We get a binary which we assume to be named mysql.fsx . Executing it will print the selected system's C-interface definition. Redirecting it into mysql.fs gives us the desired Forth-source-file which we can simply include.

```
$ mysql.fsx  -gforth > mysql.fs
```

[*]TU Wien; gerald.wodni@gee.at

[†]TU Wien; anton@mips.complang.tuwien.ac.at

We now have enabled Forth to access MySQL. See Appendix A for a simple example of the library in action. All of the above steps after creating the interface file can be done at once by using the Makefile of Appendix A. If we want to share our newly created library, Section 2 gives an example of how to do just that.

2 Library for The Forth Net

Let us consider we wrote a library for 3D-graphics and want to share it. However, in order to make this library truly system independent and easily accessible, we have to introduce some new words explained below. All words start with “f” which relates to The Forth Net[3].

frequire *(... "file" – ...)* Some Forth-systems use the working directory as the base directory for including files by relatives paths. Others use the file which is currently parsed as the base directory. To circumvent ambiguities for our library we could use absolute paths. Unfortunately this decision would force an absolute path on all users, which is not an option. *frequire* prepends the libraries base path on the current system to *"file"* and loads it.

finclude *(... "name" – ...)* Loads the library by including its main file.

fget *(... "name" – ...)* Downloads the library from The Forth Net[4] to the libraries-root-path.

Using these words we can now extend respectively write our library. Our directory structure is as follows:

```
3d
├──3d.fs
├──models
│  ├──model.fs
│  ├──sphere.fs
│  └──cube.fs
├──data
│  ├──tree.tga
│  └──water.tga
├──doc
│  └──README.txt
└──VERSIONINFO
```

“3d.fs” is the libraries main file which is included by the user who uses “`finclude` 3d”. The library allows the user to draw cubes and spheres, so “3d.fs” will provide them by using “`frequire` models/cube.fs” and “`frequire` models/sphere.fs”. Cubes and spheres use words which are common amongst all models, and are defined in models/model.fs, so both files use “`frequire` models/models.fs” (we always address files from the libraries base-path).

All other files are optional: the directory “data” provides the library with some textures, “doc” holds the documentation, which explain the user how to use the 3d-library. The content of the file “VERSIONINFO” is displayed on The Forth Net when browsing different versions, and hold some information about recent improvements or bug fixes.

The final step of sharing our library is to upload it to The Forth Net. This is accomplished by creating a new project, clicking manage, zipping the contents of the directory 3d (not including the directory itself), and submitting them as a new version.
From this moment on others can fetch the library by using “`fget` 3d”

A MySQL Demo[5]

```
\ (c) 2011 by Gerald Wodni
\ very small example for interfaceing with the c-api of mysql

\ load binary shared library
s"␣mysqlclient" add-lib

\ include functions and constants
finclude fsi-mysql-client

\ display 0-terminated string
: .cstr ( addr -- )
        begin
                dup c@
        while
                dup c@ emit
                char+
        repeat drop ;

\ -- real program --
\ create connection element
0 mysql_init constant connection

\ connect
connection s\"␣localhost\0" drop s\"␣forth\0" drop s\"␣h4x0r\0"
drop s\"␣forth\0" drop 0 0 0 mysql_real_connect [if]
        ."␣connection␣established" cr
[else]
        ."␣connection␣error" cr bye
[then]

\ launch query
connection s\"␣SELECT␣*␣FROM␣`systems`␣\0" drop mysql_query [if]
        ."␣Query-Error:␣" connection mysql_error .cstr cr bye
[then]

\ print result
: tab 9 emit ;
: show-result ( -- )
        \ get result
        connection mysql_use_result
        begin
                dup mysql_fetch_row ?dup
        while
                connection mysql_field_count 0 u+do
                        dup @ tab .cstr
                        cell+
                loop drop
                cr
        repeat
        mysql_free_result ;

."␣Result:" cr
show-result

connection mysql_close
."␣connection␣closed" cr

bye
```

A FSI-Makefile[5]

```
SWIG        = swig
OUTPUT      = mysql
INTERFACE   = mysql.i
OPTIONS     = -forth -no-sectioncomments -stackcomments -includeall\
              -I/usr/include/mysql -I/usr/include\
              -I/usr/lib/gcc/i486-linux-gnu/4.1/include/

$(OUTPUT).fs: $(OUTPUT).fsx
	./$(OUTPUT).fsx -gforth > $(OUTPUT).fs

$(OUTPUT).fsx: $(OUTPUT).fsi
	$(CC) -o $(OUTPUT).fsx $(OUTPUT)-fsi.c

$(OUTPUT).fsi:
	$(SWIG) $(OPTIONS) -o $(OUTPUT)-fsi.c $(INTERFACE)

.PHONY: clean

clean:
	rm -f $(OUTPUT)-fsi.c
	rm -f $(OUTPUT).fs
	rm -f $(OUTPUT).fsx
```

References

[1] Gerald Wodni and M. Anton Ertl. SWIG-Gforth-Extension. In *Euroforth*, 2009.

[2] David M. Beazley et al. Simplified Wrapper and Interface Generator (SWIG). URL `http://www.swig.org`.

[3] Gerald Wodni and M. Anton Ertl. The Forth Net, 2010.

[4] Gerald Wodni. The Forth Net. URL `http://theforth.net`.

[5] Gerald Wodni and M. Anton Ertl. SWIG Erweiterung für Forth Neuigkeiten, 2011.

Ways to Reduce the Stack Depth

M. Anton Ertl[*]
TU Wien

Abstract

Having to deal with many different data can lead to problems in Forth: The data stack is the preferred place to store data; on the other hand, dealing with too many data stack items is cumbersome and usually bad style. This paper presents and discusses ways to unburden the data stack; some of them are used widely, others are almost unknown or new.

1 Introduction

The data stack is the primary mechanism for passing data around in Forth. Its advantages include: words that deal only with the stacks are reentrant, i.e., they can be used recursively and in several tasks running at the same time; and straight-line code using the stack can be factored easily (just split any subsequence off into a separate colon definition).

The limitations of the data stack are: It can contain only cell-sized items. And while it may contain many items, accessing more than a few alternatingly requires quite a bit of stack shuffling and is hard to read; idiomatic in Forth usage tries to avoid stack shuffling.

However, some problems inherently have to deal with more than the about three data items that can be managed without too much shuffling.

A commonly-used example problem is drawing a rectangle specified, e.g., by the lower left and upper right point, using line-drawing primitives that take the start point and the end point: If each point is specified by two numbers, the rectangle is represented by four numbers. Moreover, each number is needed after the first line is drawn, so just before the first line primitive we would have the four numbers for the rectangle on the stack, plus the four numbers needed for the line primitive. Many different ways have been suggested for dealing with this example problem and others.

This paper looks at various ways to deal with such problems, and discusses the advantages and disadvantages.

[*]Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

2 Grouping data in memory

One approach is to store much of the data in one or few structures in memory, and putting only the address of the structure(s) on the stack. The disadvantage here is that it requires managing the memory for the structures; this includes specifying who is responsible for deallocating the memory.

In our rectangle example, we might represent each point as a structure, both in the input of our rectangle word and in the calls to the line-drawing words, but this would mean that we would have to allocate, fill, and later deallocate at least two such point structures (for the lower right and upper left point of the rectangle), and let the caller deallocate the structures (somewhat contrary to the Forth idiom of consuming stack items):

```
: line-line ( p1 p2 p3 -- )
  \ draw a line between p1 and p2
  \  and one between p2 and p3
  over line line ;

: rect ( ll ur -- )
  over point-x @ over point-y @ make-point
  ( ll ur ul )
  >r 2dup r@ swap line-line r> free-point
  over point-y @ over point-x @ swap make-point
  ( ll ur lr )
  >r 2dup r@ swap line-line r> free-point ;
```

Of course, we could pass in the data to `rect` in a structure and pass it to `line` on the stack, so the memory management would not show up in `rect`, but that would be less instructive, failing to show that memory management overhead occurs.

Given that Forth does not normally have automatic memory management (aka garbage collection), I tend to avoid such solutions where possible.

3 Multiple Stacks

A common way to deal with many stack items is to put some of them on the return stack. The return stack allows no shuffling and only direct access to the top item, and data has to be moved or copied back to the data stack for computations, so this is usually limited to just one or two items. The disadvantage of this strategy is that we lose the nice factoring property of data-stack-only code: there are some straight-line code sequences in code using the return stack that cannot simply be split of into a separate colon definition.

Floating-point code keeps floating-point numbers on the separate FP stack, so the number of items on the data stack (and the number of items on the FP stack) is usually smaller than in equivalent integer code, and there is usually much less shuffling necessary.

As an example, consider the following word from the integer matrix multiplication benchmark:

```
: innerproduct ( a b -- n )
  \ a points to a column in a matrix
  \ b points to a row    in a matrix
  0 row-size 0 do
    >r over @ over @ * r> + >r
    swap cell+ swap row-byte-size +
    r>
  loop
  >r 2drop r> ;
```

Note that this code already reduces the stack load by passing `row-size` and `row-byte-size` in constants. A floating-point variant of the word looks like:

```
: finnerproduct ( a b -- r )
  0e row-size 0 do
    over f@ over f@ f* f+
    swap float+ swap row-byte-size +
  loop
  2drop ;
```

All the return stack usage went away.

The main disadvantage of the FP stack is the additional implementation cost: managing another memory area for this stack, per task; and having another stack pointer that has to be saved and restored by context switches and in exception handling.

Some people have suggested additional stacks, e.g., an address stack or a string stack. This has not really caught on yet. In addition to the costs mentioned above one often wants to use operations like `-` on addresses and, e.g., string lengths. Keeping these types on separate stacks would require moving these data between the stacks for such operations, which will increase the stack noise in some cases.

4 Locals

Locals offer a way to deal with lots of data. E.g., for our rectangle example a solution using locals would be:

```
: rect {: x1 y1 x2 y2 -- :}
  x1 y1 x1 y2 line
  x1 y2 x2 y2 line
  x2 y2 x2 y1 line
  x2 y1 x1 y1 line ;
```

The result is readable and this approach scales to dealing with many data.

Despite these advantages, using locals has been vilified often by a considerable portion of the Forth community. One disadvantage of this approach is that we lose the nice factoring properties of data-stack-only code, but using the return stack has the same disadvantage without having the same acceptance problems as locals.

A common argument against using locals is that locals discourage proper factoring; they only discourage it in the same way that the return stack does, but maybe the complaint really is that due to the scalability they fail the encourage factoring in the same way that stack-based approaches do; i.e., that locals take away the pressure to factor for less active data at a time, because words that deal with lots of data still remain manageable.

The question then is why we want a more highly factored program. If locals achieve that goal (say, readability) with less factoring, do we need more factoring? If not, maybe we should be aware of and strive for the desired property instead of just avoiding some programming language features.

5 Global/`user` variables

5.1 ... within single definitions

Another related approach is using global variables. As long as you use them inside a single colon definition, the readability and scalability is similar to using locals. And in contrast to locals, in some respects they keep the nice factoring properties of data-stack-only code. The disadvantages are that the result is not reentrant or usable recursively unless special measures are taken.

The most serious problem, though, is: if you make use of the factoring properties, now the data does not just flow through the stack from caller to callee and back, but through an arbitrary set of global variables. This makes the data flow hard to track, and makes programs hard to maintain. If you want to avoid that, you lose the factoring property with globals just as you lose it with locals.

Also, one nice property of normal factors is that they are often useful for other purposes; however, factors involving globals do not have a nice stack-based calling interface, but something more complicated, so they are usually not nice factors.

In conclusion, while globals are in theory less of an obstacle to factoring than locals, it's usually better to avoid this kind of factoring.

If we use user variables to make the word reentrant in the presence of multiple tasks or threads, the variables consume space in each task, all the time. With cooperative multi-tasking, we can avoid that as long as the variables live only between task switches (which creates another maintenance problem), but with true concurrency this trick no longer works; and we want to use true concurrency on the increasingly pervasive multi-core CPUs.

5.2 ... across definitions

Global/user variables are sometimes used as additional input or output parameters for words. An example in standard words is #, which takes base as additional input and produces additional output in the *pictured numeric output buffer*.

This global state complicates the interface, which reduces reusability and causes maintenance problems. An example is trying to debug code between <# and #> using .s.

One programming practice for reducing these kinds of problems is to save the global variable before changing it and restoring it before returning to the caller. An example of this practice is:

```
: hex.-helper ( u -- )
  hex u. ;

: hex. ( u -- )
  base @ >r
  ['] hex.-helper catch
  r> base ! throw ;
```

However, this practice is somewhat cumbersome to program and Charles Moore prefers to just set global state whenever that is needed [Bro04, Page 212]; this is a bad idea for reusability, but Moore does not value reusability of code.

6 Context wrappers

Saving, changing, and restoring a global variable can be factored out into a context wrapper. E.g., Gforth has a word base-execute that saves base, changes it, executes an xt, then restores base. A usage example is:

```
: hex. ( u -- )
  ['] u. $10 base-execute ;
\ base-execute ( xt u -- )
```

Another example is execute-parsing, which saves the input stream, sets it to the passed-in string, executes an xt, and restores the input stream. A usage example is $create, which takes the name of the created word from a string instead of the input stream:

```
: $create ( c-addr u -- )
  ['] create execute-parsing ;
\ execute-parsing ( c-addr u xt -- )
```

A third example of this pattern is >string-execute which redirects the console output (e.g., type) into a string. This allows the programmer to construct strings from many or complicated words without having to deal with intermediate strings on the stack.

```
: fe.>string ( r -- c-addr u )
  ['] fe. >string-execute ;
\ >string-execute ( xt -- c-addr u )
```

The general convention used in Gforth (with the exception of base-execute, which came before the convention) is to pass the execution token into the context wrapper on the top of stack, because it is usually a literal.

An advantage of context wrappers is that they make it possible to use words that would otherwise be specific to some global resource (e.g., the console output in case of fe.) in a more general way; without a word like >string-execute, if you want to transform an FP number into engineering notation, you have to reimplement most of fe. yourself.

So context wrappers do not only make it possible to reduce stack shuffling in new code, but also to reuse some code in ways for which it has not originally been written.

Gforth also has the words infile-execute and outfile-execute that allow to use console input (key) or output (type) words for input from or output to a file. A special advantage of using a context wrapper here is that it restores the old, working setting if an error is thrown; in contrast, if an error occurs during a global redirection of console I/O, the user has problems recovering from the error (especially if input is redirected).

The usual implementation of context uses global/user variables and saves the contents of these variables on the return stack when performing a context wrapper. The disadvantage of this approach is that each context requires space for another user variable in each task.

Hanson and Proebsting [HP01] discuss a related concept: dynamically scoped variables; they also present several implementation techniques that may require less memory (but more run-time) than approach of using global/user variables with saving.

7 Implicit Parameters and Results

A common pattern in reducing the number of items on the data stack is implicit parameters and return values. The context in context wrappers and global/user state like `base` are two examples of this pattern.

Another one is the loop control parameters in `do` loops. The equivalent `begin` loops would often have too many items on the stack to manage easily.

Finally, a number of object-oriented Forth extensions have an implicit *current object*, e.g., `this` in `objects.fs` [Ert97]; in this objects extension, the current object is set automatically from the top-of-stack when entering a method and the old current object is restored when leaving the method. By contrast, in Bernd Paysan's oof.fs model, the current object is set explicitly, but is then used implicitly whenever calling a method. In both objects extensions, the current object is used implicitly when accessing fields of the current object.

8 Registers

ColorForth has the programmer-visible register A, which is used for memory accesses (e.g., Forth's `!` becomes ColorForth's `A! !`); moreover, the top of the return stack R serves a similar function. Virtual machine models with even more registers have been proposed [Pel08].

A value in A does not have to be kept on the stack, reducing stack load. This is supported by `@+` `!+`, a fetch and store that autoincrement the address in A.

These registers are global resources and share many of the disadvantages of globals. However, their usage model is somewhat different from ordinary globals:

- Most globals are specific for one particular purpose, whereas any word that accesses memory will set A in ColorForth. So, the usage of registers is much more temporary, and programmers typically don't expect the contents to survive across calls (unlike, usually, globals). So, they don't reduce the stack load across calls.

- Interrupt handlers and task switchers will preserve the contents of the registers across the interrupt or for the next execution of the task, so the registers can be used in reentrant code.

9 Example: Postscript Graphics Model

The Postscript graphics model demonstrates some of the ideas presented up to now in action. Here is the rectangle example, written in Forth, but with Postscript graphics operators as words:

```
: rectangle ( x y w h -- )
  2swap moveto
  over 0 rlineto
  0 swap rlineto
  negate 0 rlineto
  closepath stroke ;
```

First, we have adapted the parameters of `rectangle` (width and height instead of the coordinates of the other corner), because that requires less stack shuffling in combination with the Postscript graphics operators. Now, to the essential parts:

Postscript has the *current point* as implicit parameter. We start out by setting the current point with `moveto`.

Then we draw the first line with `rlineto`; the current point determines the start of the line, and the basis for our relative operation[1], so we change x by w and y by 0; `rlineto` also sets the current point to the end point of the line.

The next `rlineto` draws the second, vertical line of our rectangle, the third `rlineto` draws the third line.

Actually, these words did not draw lines, they created a path in the (implicit) graphics state (which contains the current point and other information). Now we add a final line to the path with `closepath` that goes back to the start of the path.

Finally, `stroke` draws the lines described by the path onto the canvas (the in-memory representation of the page). It takes a number of additional implicit parameters into account: line width, colour, dash patterns, corner shape, scale and rotation (more generally, a transformation matrix).

As we can see, Postscript makes effective use of implicit parameters through the global graphics state. To avoid some of the problems of global state, Postscript provides `gsave` to save the current graphics state on a dedicated graphics state stack, and `grestore` to change it back to the old value.

10 Staged Execution

Another way to reduce the number of items on the stack at any one time is to divide the computation into different stages. The first stage deals with some

[1]In addition to the relative `rlineto`, which takes a coordinate relative to the current point, Postscript also has `lineto`, which draws a line to an absolute coordinate.

of the data and generates code for the second stage, the second stage deals with more data, and either completes the computation or generates code for a further stage, etc.

As an example, consider `innerproduct` from the matrix multiply benchmark. The single-stage version (already shown in Section 3) looks like this:

```
: innerproduct ( a b -- n )
  0 row-size 0 do
    >r over @ over @ * r> + >r
    swap cell+ swap row-byte-size +
    r>
  loop
  >r 2drop r> ;

a b innerproduct .
```

This version already uses a number of techniques to reduce the stack depth: a `do` loop to get rid of the stack items for loop control; the number of elements in the vectors (`row-size`), and the strides (cell and `row-byte-size`) are not passed in through the stack; and the return stack is used for the intermediate result.

Here is a version that divides the execution into two stages:

```
: gen-innerproduct ( a[row][*] -- xt )
\ xt is of type ( b[*][column] -- n )
 >r :noname r>
 0 ]] literal SWAP
 [[ row-size 0 do ~~ ]]
   dup @
   [[ dup @ ]] literal * under+ cell+
   [[ row-byte-size + loop
 drop ]] drop ;
 [[ ;

a gen-innerproduct b swap execute .
```

This code uses the syntax `]] x y [[`, which is equivalent to `postpone x postpone y`, but more readable. The staged code uses the same stack depth reduction techniques (except the return stack, which becomes unnecessary) as the original code, but it adds staged execution; this results in less stack shuffling, and no need to use the return stack (except to get the parameter a past the `:noname`. The second stage then contains an unrolled loop that contains the values from the vector a; in source code it would look like this (for a three-element vector a containing 5, -3, 2):

```
:noname
  0 swap
  dup @  5 * under+ cell+
  dup @ -3 * under+ cell+
  dup @  2 * under+ cell+
  drop ;
```

Of course, this technique incurs the CPU and memory costs of generating the code for the second stage, and is normally only efficient if the second stage is used several times (if it is used often enough, it can be significantly more efficient than the single-stage code [LL96], if the Forth compiler supports fast code generation); and the programmer may have to deal with recovering the memory for the generated code.

So, this technique is not very general-purpose, but it still is an interesting addition to the arsenal of stack depth reduction techniques.

11 Pipelines

An program can be organized as multiple tasks that are connected in a pipeline. One reason for this organization is that it allows the flexible composition of useful reusable parts; that is the main reason for using pipelines in Unix. Another benefit of pipelines is that the tasks of a pipeline (pipeline stages) can be executed in parallel.

In the context of our topic the benefit is that each task has its own stack; if we have multiple parameters to pass in and multiple data to handle, hopefully each task needs only a part of these parameters and only needs to deal with a part of the data, reducing the stack depth pressure in that task, compared to a program that tries to do it all in a single task.

I am not aware of an implementation of this idea, but it should not be hard to implement on a multitasking Forth system. In any case, the following is just a somewhat elaborate idea, not something you can use as programmer at the moment.

The following code fetches a vector x from memory, multiplies it with an FP number a, and adds the product vector to another vector y in memory. This is a common linear algebra function (called SAXPY, DAXPY, etc. in BLAS, depending on the type). In our pipelined implementation each of these steps (fetching, multiplying, adding) has its own pipeline stage:

```
: v@ ( f-addr nstride ucount -- )
    0 ?do
        over f@ fput
        tuck + loop
    endput 2drop ;

: vf* ( ra -- )
    begin fget? while
            fover f* fput repeat
    fdrop ;
```

```
: v+! ( f-addr nstride -- )
    begin fget? while
            over f@ f+ over f!
            tuck + repeat
    2drop ;

: faxpy ( ra f-addr-x nstride-x
             f-addr-y nstride-y ucount -- )
    rot rot 2>r ['] v@ xxx|
    ['] vf* f|
    2r> v+! ;
```

Here the input parameters are the start address address, stride[2], and size of x, the value of a, and the address and stride of y (the size is the same as for x). So we have five cell-sized parameters and one FP parameter, too many for handling in one function with stack manipulation words only (therefore I present no version without pipelining).

In our pipelined version each pipeline stage only has to handle a few of the parameters, and consequently there is little stack manipulation code. The interface word `faxpy` sees them all, but only has to pass them to the stages, which is relatively simple.

Each pipeline stage passes its FP result with `fput` to the next stage, which receives it with `fget?`.

The connections between the stages are implicit, so we can only have a linear pipeline. Linear pipelines have been good enough for a lot of work in Unix, but one still might want to consider less restricted options (ideally, any data-flow DAG); the main disadvantage would be that we then have to identify which connection an `fput` or `fget?` refers to, and since this identifier would be passed on the data stack, this would increase the stack load. Another disadvantage of data-flow graphs beyond trees is that simple pipeline implementations can lead to deadlocks.

12 Conclusion

In this paper we look at various ways to reduce the stack load. There is no silver bullet, except locals. Yet, using a combination of the other techniques, most of the time it is possible to keep the stack load manageable even if we do not use locals: using the return stack, the counted loop parameters and various implicit parameters present in the Forth system.

The Postscript graphics model shows how a problem that appears hard for stack-based languages can be solved using such techniques.

We also present the more exotic (in Forth) techniques of staged execution and pipelines, which provide additional weapons against high stack item pressure.

[2]The stride parameter allows using the function on vectors that are not consecutive in memory, e.g., a column or diagonal of a matrix.

References

[Bro04] Leo Brodie. *Thinking Forth.* Punchy Publishing, 2004. reprint of the 1984 edition.

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997.

[HP01] David. R. Hanson and Todd A. Proebsting. Dynamic variables. In *SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 264–273, 2001.

[LL96] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, 1996.

[Pel08] Stephen Pelc. Updating the Forth virtual machine. In *24th EuroForth Conference*, pages 24–30, 2008.

Forth concurrency for the 21st century, Part 2:

What are we going to do about *volatile*?

Andrew Haley

Motivation

- VFX forth compiled

```
    counter @ dup 2 +
```

into machine code equivalent to

```
    counter @ 2 +  counter @ swap
```

- This breaks a common idiom for a shared counter:

```
begin
   counter @ dup N +
   counter compare&swap
until
```

Where are we, *recap*

- Moore's law has not been cancelled: every 18 months, the number of transistors per unit area doubles
- However, clock speeds have not been increasing for several years, and if anything have got slightly slower
- Performance can still be increased with new pipeline and cache designs, but not by much.
- There seems to be a 4 GHz barrier
- Future machines will have more and more cores

Motivation

- I sent an email to Steve, who replied:

  “It's not a bug ... it's the volatile problem!”
  “Note that this is only a problem for x86 and hosted systems.”

- So, there is a volatile problem. And that's official!

Profane languages and *volatile*

- C has *volatile*. Is it the first language to have it? I think so
- Hans Boehm points out that there are only three portable uses for it. Summarizing,
  - * marking a local variable in the scope of a *setjmp* so that the variable does not rollback after a *longjmp*.
  - * variables may be "externally modified", but the modification in fact is triggered *synchronously* by the thread itself, e.g. because the underlying memory is mapped at multiple locations
  - *A volatile *sigatomic_t* may be used to communicate with a signal handler in the same thread

Memory barriers and data races

- Current processors feature *out-of-order execution*. From the point of view of a thread, memory accesses appear to occur in program order. However, from the point of view of another thread, there is no guarantee that memory writes from another thread will occur in the same order, or that its stores into memory will *ever* be visible!
- The new C++ standard takes the view that all access to data shared between threads that are not explicitly protected by locks or atomic operations are undefined
- Because some Forth systems are implemented in C, they will necessarily have the same problem

Profane languages and *volatile*

- Java has *volatile*. But its meaning is totally different from that of C's *volatile*
- Java's *volatile* is a memory barrier

data races

- Thread 1:

```
99 result !  1 ready !
```

- Thread 2:

```
begin  pause  ready @ until
result @  ...
```

- The value in `result` is undefined

redhat

load acquire and *store release*

- What would it take to make this work?
- Thread 1:

```
99 result !  1 ready volatile!
```

- Thread 2:

```
begin  pause  ready volatile@ until
result @  ...
```

redhat

load acquire and *store release*: x86

- On x86, *load acquire* is:

  mov memory -> register

- *store release* is:

  mov register -> memory

- Compilers don't have to do anything special because on the x86 there is a guarantee that all threads see stores in the same order. Other processors don't have such guarantees, so naive programmers may accidentally write non-portable code

redhat

load acquire and *store release*

- `volatile!` and `volatile@` must be *memory barriers*
- Every store to memory that *happens before* a `volatile!` is visible to another thread after
- that other thread does a `volatile@`
- These operations are known as ***load acquire*** and ***store release***

redhat

load acquire and *store release*: ARM

- On ARM, *load acquire* is:

  ld memory -> register
  dmb (flush all locally cached loads)

- *store release* is:

  dmb (force all pending stores)
  mov register -> memory

- There are other equivalent sequences

load acquire and *store release*: PowerPC

- On ppc, *load acquire* is:

```
ld memory -> register;
lwsync ( flush all locally cached loads)
```

- *store release* is:

```
lwsync  ( force all pending stores)
ld register -> memory
```

- There are other equivalent sequences

Forth and the future

- The C an C++ standard committees used to think that they didn't have to worry about these things. They have now changed their minds, because threading issues can't be left to libraries
- If Forth has a future on multi-core systems these issues must be addressed. We should start thinking about them now

net2o: Application Layer

Factor the Content

Bernd Paysan

EuroForth 2011, Vienna

Outline

Motivation

Requirements

Solutions
- Some Basic Insights
- Factor Data
- Distribute the Code

net2o Recap: Lower–Level Parts

- Shared memory datagrams with short headers, packet size a power of two, see [1].
- Abstraction: files and attributes (name $\longrightarrow$ value pairs)
- Network model: peer to peer distributed file system with version control, “single data cloud,” see also [2]
- Real–time ability allows data streaming
- Legacy implementation based on UDP
- P2P principle: access data by cryptographic hash (Merkle trees for larger data)

Status: Legacy Flow Control

- TCP flow control too aggressive, fills buffers until packet drops occur — huge delays (in the order of 4s for a typical DSL account)
- Somebody did my homework: LEDBAT flow control adds constant delay — not so aggressive
- Can’t use LEDBAT in naive form, needs adaption — I like to have a bandwidth control, not a window or even packet size control
- BitTorrent already implements LEDBAT, seems to be a good idea [3]
- Assumption is single bottleneck with few shared connections

What for?

People like to share information (*share* means making *copies*)

- Messages, photos, videos, music
- Longer, structured documents
- Two–way real–time (chat, telephony, video conferencing)
- Collaborative gaming

How to find?

Information needs to be organized, or we are lost

- By person/group
- By topic
- By content
- By relevance (“page rank”)
- By preference (“like it”–button, visited regularly. “hate it”–button to rate annoying pages down)
- By date or geographic location

Web Pages as Application

How to Present

Elements of a web page, classified by user experience

1. User interface — navigation, modes, and actions
2. Textual content
3. Graphics and photos
4. Videos
5. Games. . .

Factoring 1.0

- HTML glues together all textual and user interface elements into one file
- Separated CSS for layout
- Separated JavaScript for application logic
- Separated graphics and videos
- Plugins for games and videos

Browser as Application Environment

Remember: The browser is meant as application environment for net–centric applications!

- Requires a general purpose language to write the applications in
- Requires a good development and debugging environment
- Must be fast/low power budget (performance is key on mobile devices, PCs are fast enough)
- HTML5+CSS+JavaScript doesn't cut the mustard

Downsides

- Client has to pull data together, more network traffic
- Data assembly may be pretty static
- Collated queries work in a client–server environment, but not in a P2P environment

Factor the Text!

- Dynamic web pages use AJAX to partially replace elements of the page
- Why put everything into a single file first, and then start tearing it apart?
- Put each element (article, comment, message, navigation bar, images, videos, sound) into a file of its own, and *refer* to embedded objects, regardless of their type
- Use scripts to generate dynamic references ("the last three comments to that article")
- Provide low–level services, and let the application logic and libraries do the rest

ACID on Data Clouds

- You want your data to be in shape, thus you need the ACID[1] properties of the data repository
- You can't do read–modify–write on data clouds. You can push new data, that's it. Forget about "delete," there's no easy way to call your data back. You can't implement locks. All you get is durability: "the net will not forget."
- However, you can say "this is the next revision of document *x*."
- Two concurrent edits will produce a fork. It's up to the authors to decide how to merge forks back.
- When you publish something, you can't guarantee that it's available in order. If the reader gets an incomplete transaction, he must either retry or fall back to the previous version.

[1] Atomicity, Consistency, Isolation, and Durability.

Hash–Indexed Content

Hashes as “handle” to actual content are the key to data management

- $h(data) \longrightarrow hash$ produces a unique hash for each data file
- $d(hash) \longrightarrow data$ allows to retrieve the data when the hash is known
- Hash trees provide a mean to distribute large files
- Relationships between data revisions are stored as graph, using the hashes as symbol for the actual data

Secure Execution Environment

- Any sufficiently powerful language let you write malware
- The libraries of any sufficiently powerful environment (even with a very restricted language) contain enough exploits to write malware

Sandbox

- Sandbox the process, restrict network access (read is ok, write needs user permit)
- Using your keys (decyption, encryption, signing) must be outside the sandbox
- “Same origin”–policy doesn’t work for a data cloud — the destination is again “the cloud”

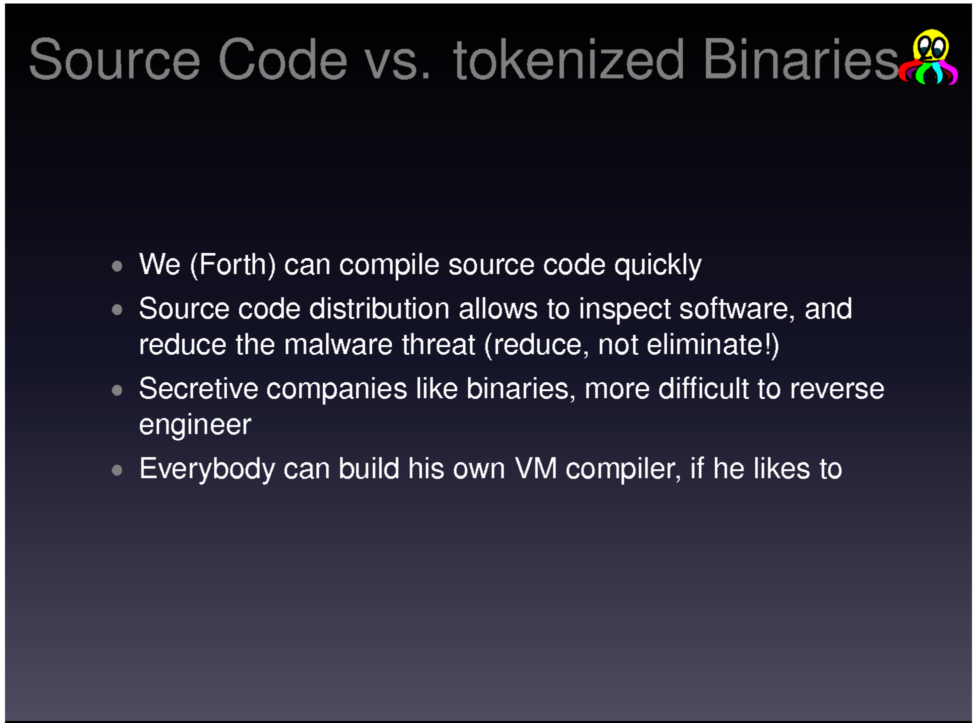
- Signed scripts and social control can help to some extend
- The boundary to malware is non–trivial to define. Is Farmville malware?

Privacy

- You can control with whom you share (cryptography)
- Recalling information requires cooperation (“the net will not forget”)
- You can’t control with whom your receivers re–share (impossibly of DRM)
- There is no real anonymity, but your traces can be lost in the clouds

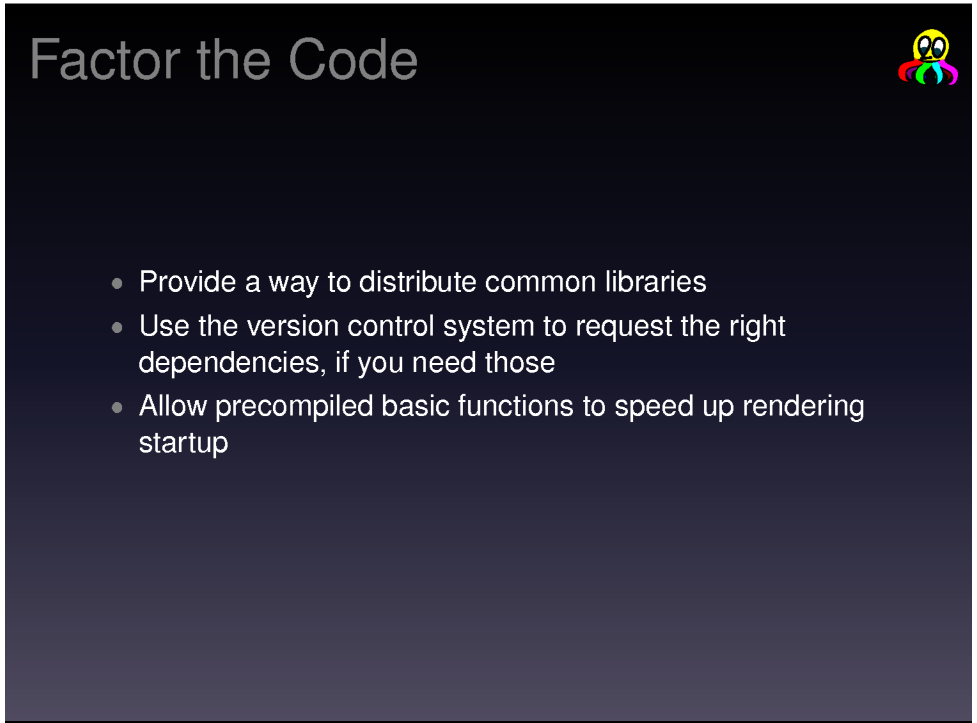
Push vs. Polling

- Polling is stupid
- Push–style solutions require open connections or ports
- Stored procedures in the cloud — or better call them “callbacks,” because they can only call the originator



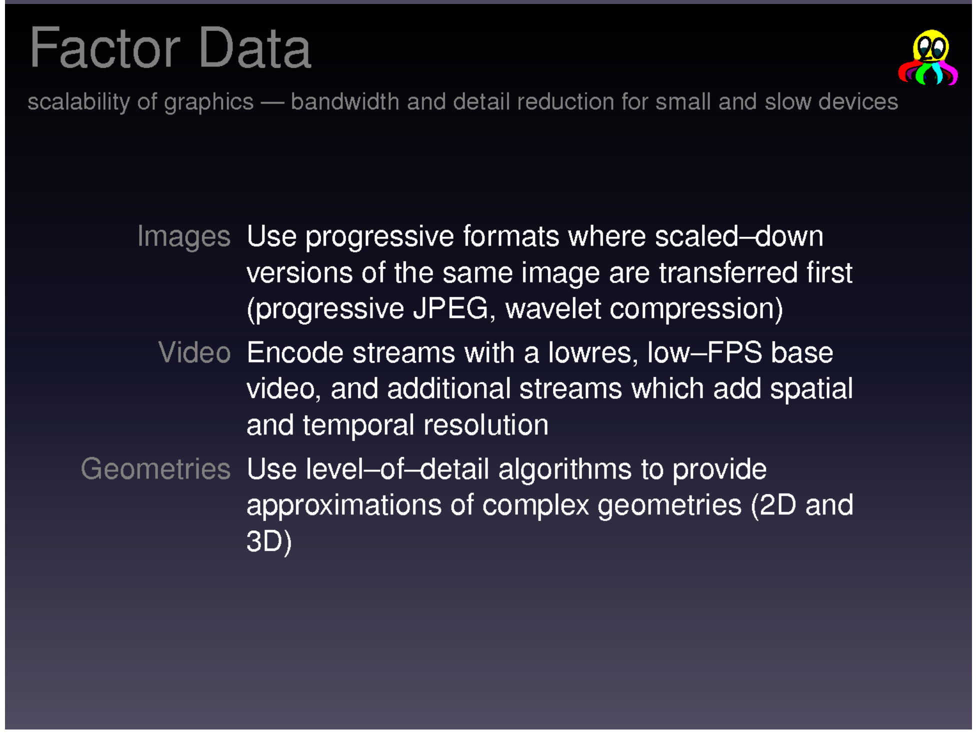
Source Code vs. tokenized Binaries

- We (Forth) can compile source code quickly
- Source code distribution allows to inspect software, and reduce the malware threat (reduce, not eliminate!)
- Secretive companies like binaries, more difficult to reverse engineer
- Everybody can build his own VM compiler, if he likes to



Factor the Code

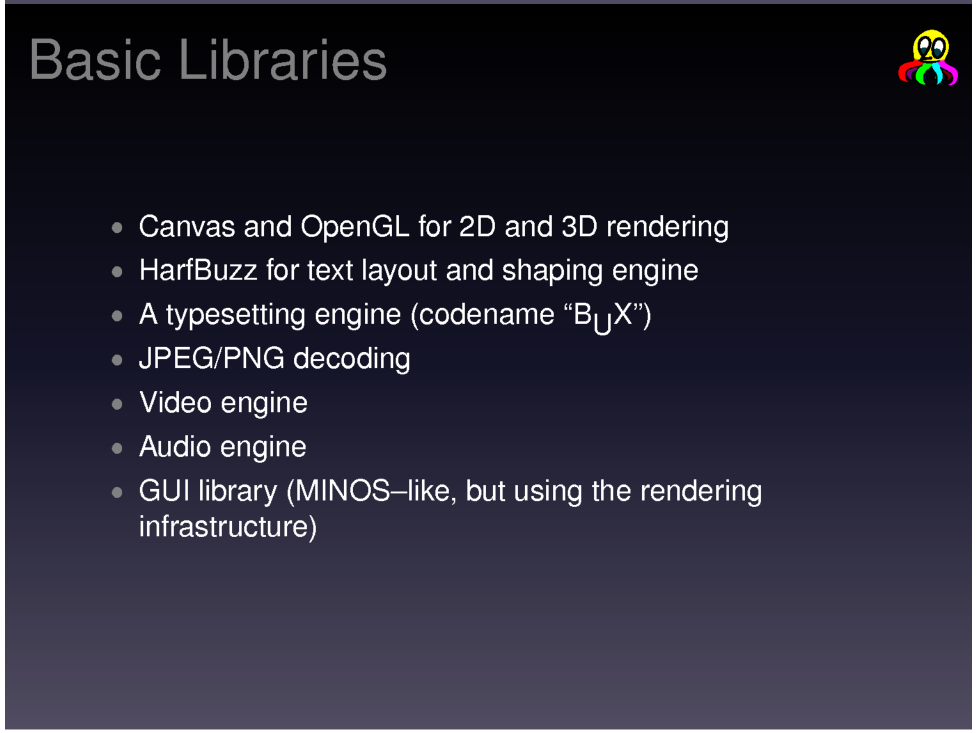
- Provide a way to distribute common libraries
- Use the version control system to request the right dependencies, if you need those
- Allow precompiled basic functions to speed up rendering startup



Factor Data

scalability of graphics — bandwidth and detail reduction for small and slow devices

Images	Use progressive formats where scaled–down versions of the same image are transferred first (progressive JPEG, wavelet compression)
Video	Encode streams with a lowres, low–FPS base video, and additional streams which add spatial and temporal resolution
Geometries	Use level–of–detail algorithms to provide approximations of complex geometries (2D and 3D)



Basic Libraries

- Canvas and OpenGL for 2D and 3D rendering
- HarfBuzz for text layout and shaping engine
- A typesetting engine (codename “B$_U$X”)
- JPEG/PNG decoding
- Video engine
- Audio engine
- GUI library (MINOS–like, but using the rendering infrastructure)

Most Stupid Mistakes of the Net 1.0

- There really must be an easy to use (i.e. WYSIWYG) *in–browser editor* for the content!
- *Client–server* instead of peer to peer — the original idea behind the Internet was peer to peer, but it was soon forgotten
- *Postel principle* — do not be liberal in what you receive, do explicit consistency checks even if that is costly (Rose principle)
- *Unencrypted* by default (was too costly; but then, cryptographic protocols such as SSL are really ugly and full of mistakes)

For Further Reading

BERND PAYSAN
Internet 2.0
http://net2o.de/

POUWELSE, GRISHCHENKO, BAKKER
swift, the multiparty transport protocol
http://libswift.org/

ROSSI, TESTA, VALENTI
Yes, we LEDBAT
http://www.pam2010.ethz.ch/papers/full-length/4.pdf