30th EuroForth Conference

September 26-28, 2014

Hotel Amic Horizonte
Palma de Mallorca
Spain

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 30th EuroForth finds us in Palma de Mallorca for the first time. The two previous EuroForths were held in in Oxford, England (2012) and in Hamburg, Germany (2013). 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 was one submission to the refereed track, and one was accepted (100% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 17 submissions, 10 accepts, 59% acceptance rate. The paper was sent to three program committee members for review, and they all produced reviews. The reviews of all papers are anonymous to the authors. I thank the authors for their papers and the reviewers and program committee for their service.

Several papers were submitted to the non-refereed track in time to be included in the printed proceedings.

These online proceedings (`http://www.euroforth.org/ef14/papers/`) also contain papers and presentations that were too late to be included in the printed proceedings. Also, some of the papers included in the printed proceedings were updated for these online proceedings.

Workshops and social events complement the program.

This year's EuroForth is organized by Janet and Nick Nelson.

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
Peter Knaggs
Phil Koopman, Carnegie Mellon University
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas, Adsensus Ltd.

Contents

Refereed papers

Andrew Read: Concept and implementation of an extended return stack to enhance subroutine and exception handling in FORTH 5

Non-refereed papers

Stephen Pelc: Compiling to Flash 23
Stephen Pelc: VFX Forth for ARM Linux 26
Paul E. Bennet: High Integrity Systems — CODE 30
Bill Stoddart: HiTex — LaTeX gets a helping hand from Forth 35
M. Anton Ertl: Region-Based Memory Allocation in Forth 45

Late non-refereed papers

Klaus Schleisiek: Doing C-style structs on cell addressed uCore 50
Gerald Wodni: Forth — The Next Generation 52

Presentations

Ulrich Hoffmann: Saturation Arithmetic 54
Bernd Paysan: net2o: Command Language 57
Paul Bennet: Forth in Education — Spreading the word 61
Anton Ertl: How to get rid of C 63

Concept and implementation of an extended return stack to enhance subroutine and exception handling in FORTH

Andrew Read

June 2014

andrew81244@outlook.com

Abstract

A conceptual generalization of the FORTH return stack is obtained by complementing it with one or more additional stacks that are tightly synchronized in operation according to some precise logical rules. With additional stacks specifically deployed to support subroutine and exception handling, a model is obtained whereby CATCH and THROW can be implemented as single machine language instructions and a number of other features emerge that enhance the flexibility, speed, and robustness of subroutine and exception handling in FORTH. The extended return stack model has been successfully implemented on the N.I.G.E. Machine.

1 Introduction

This paper discusses the concept and implementation of an extended return stack in FORTH. The traditional stack data structure comprises an array of cells in memory with a movable stack pointer. Push and pop operations are defined for placing data onto the stack or removing data from the stack. Typical FORTH implementations utilize two stacks: the parameter stack (generally used for program data) and the return stack (generally used for flow control and holding a subroutine return address). This paper describes a conceptual generalization to the return stack and its implementation on the N.I.G.E. Machine.

The key conceptual idea is that additional stacks can be linked to return stack in a structured arrangement. These additional stacks can be leveraged to support enhanced flexibility, speed, and robustness of subroutine and exception handling in FORTH.

The N.I.G.E. Machine is a complete computer system implemented on an FPGA development board [1]. It comprises a 32 bit softcore processor optimized for the FORTH language, a set of peripheral hardware modules, and FORTH system software. The N.I.G.E. Machine was presented at EuroFORTH in 2012 and 2013 [2, 3] and is available open source [4]. It follows in the footsteps of a number of significant FORTH processors [5, 6, 7, 8, 9, 10].

2 Review of prior work

Flexibility, speed and robustness are critical in all aspects of computer science. Much prior work has been done to apply these topics to subroutine and exception handling in FORTH.

Klaus Schleisiek conducted some of the first research into error trapping in FORTH and noted that "everything to do with a subroutine belongs on the return stack" [27, 28]. Klaus's comment inspired the author's thinking on the topic and is the intellectual germ of this project. Brad Rodriguez was also one of the pioneers of FORTH exception handling in the era before CATCH and THROW, including applying stack frames to FORTH [29, 30]. The development of the standard words, CATCH and THROW has been explained by Michael Milendorf who also provides a reference implementation [16].

Many authors have discussed ways to enhance the reliability of FORTH software. The ideas presented in this paper build off several:

Jaanus Pöial studied stack effects at a conceptual level and developed an algebraic formalism for validating FORTH code [21, 22].

Paul Bennett and Malcolm Bulger discussed the certification of high integrity software and explained that in order to be able to fully certify an application, then it is first required to certify the programming surface. They made the point that this would only need certifying once, and then could be used as the platform for many products [12]. This is especially relevant if the programming surface can be built into the base hardware.

Anton Ertl introduced a construct that guarantees that the cleanup code associated with resource usage is always completed, and demonstrated a more efficient implementation approach for cleanup code than using a full exception frame [13].

Nick Nelson contrasted approaches in the search for reliability of a large and complex FORTH system, including the idea of developing system which tries to struggle on despite programming errors [17].

Bill Stoddart and Peter Knaggs have contributed significantly to making FORTH robust [23, 24, 25, 26].

Considering the topic of subroutine local variables, Bailey, Sotudeh and Ould-Khaoua identified that local variable management and its efficient support in hardware is a prime concern in developing efficient stack based computation [11]. Anton Ertl stressed that the appropriate use of local variables has the potential to significantly unburden the data stack [14].

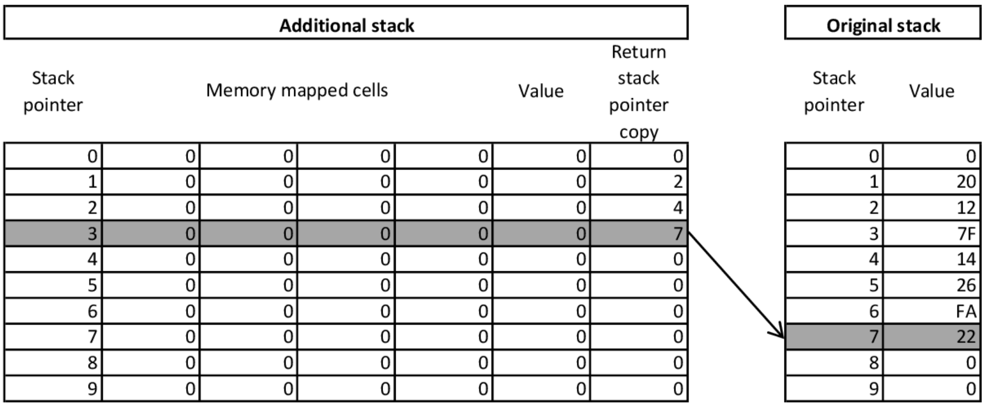
The use of a third stack for local variable storage is not new. Philip Koopman pointed out that this idea has often been proposed, but he suggests a better solution to support local variables may be a frame pointer into a software-managed program memory stack [19].

Lastly, regarding flexibility in subroutine handling, Glassanenko discussed programming techniques using return stack manipulations such as the implementation of new control structures and backtracking [15].

3 Two linked stacks as a conceptual model of subroutine and exception handling behavior

The motivation for starting with a conceptual scheme (rather than with a design specification for some desired output) was to find a straightforward logical model of the behaviors that occur in subroutine and exception handling as an intellectual goal in itself. The approach taken to finding a suitable model was an iterative series of thought experiments and pen and paper investigations. (The author accepts that this approach has the weakness of missing a formal proof. Therefore, as with the N.I.G.E. Machine overall, structured and extensive testing is a prerequisite to use in any critical system.)

This section presents the key features of the logical model that was found to represent subroutine and exception handling (the extended return stack) at a purely conceptual level. The section



Additional stack

Stack pointer	Memory mapped cells				Value	Return stack pointer copy
0	0	0	0	0	0	0
1	0	0	0	0	0	2
2	0	0	0	0	0	4
3	0	0	0	0	0	7
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0

Original stack

Stack pointer	Value
0	0
1	20
2	12
3	7F
4	14
5	26
6	FA
7	22
8	0
9	0

Figure 1: Illustration of basic conceptual scheme of the extended return stack. The current position of the stack pointer is highlighted in gray.

following will explain exactly how this conceptual scheme has a direct relationship with subroutine and exception handling in FORTH.

Consider a simple, traditional stack that has been extended by making available an additional stack and creating certain logical connections between the two. In this paper the diagrammatic convention will be that the original stack is shown on the right and the additional stack on the left. The additional stack is structured as follows (figure 1):

The additional stack is at least two cells wide. Each cell can be read independently. The data contents of all cells change simultaneously when the stack pointer moves. The first cell (by convention shown as the rightmost cell in this paper) of the additional stack holds a copy of the stack pointer to the original stack. This stack pointer copy is read and written on certain events as described below. The second cell of the additional stack is the conventional part of the stack; it is where data that is PUSHed to the stack will be placed. The remaining cells of the additional stack are mapped to registers at fixed locations in system memory. This is configured in such a way that regardless of the position of the stack pointer, the top of stack values are always found at the same physical addresses in system memory.

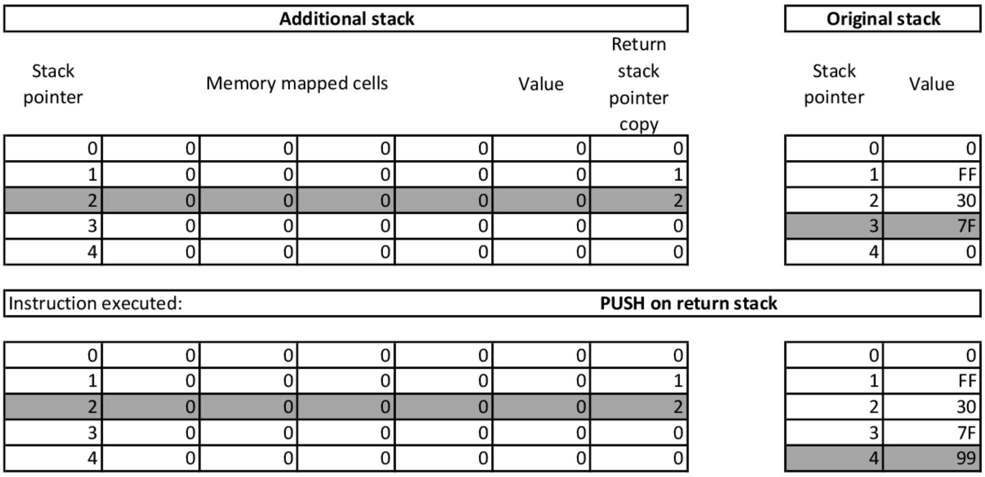
The rules for interaction between the original stack and the additional stack are as follows (illustrated in figures 2, 3, 4, 5, 6):

1. Push and pop operations are separately available for each stack. (Looking ahead, it is the mapping of these push and pop operations on separate stacks to an appropriate set of machine language instructions that makes for efficient support of subroutine and exception handling).

2. When the original stack is pushed then there is no impact on the additional stack.

3. When the original stack is popped then there is no impact on the additional stack unless the value of the original stack pointer after the pop would be less than the copy value held on the additional stack. In this case the additional stack is simultaneously also popped.

4. When the additional stack is pushed then two additional operations occur simultaneously: firstly the current value of the original stack's stack pointer is pushed to the first (rightmost)



Additional stack

Stack pointer	Memory mapped cells				Value	Return stack pointer copy
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	2
3	0	0	0	0	0	0
4	0	0	0	0	0	0

Original stack

Stack pointer	Value
0	0
1	FF
2	30
3	7F
4	0

Instruction executed: **PUSH on return stack**

0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	0	2
3	0	0	0	0	0	0
4	0	0	0	0	0	0

0	0
1	FF
2	30
3	7F
4	99

Figure 2: Illustrating the rule that when the original stack is pushed then there is no impact on the additional stack. The "before" state of the stacks is shown above and the "after" below.

cell on the additional stack and secondly the original stack is a pushed with the same value as was pushed to the additional stack.

5. When the additional stack is popped then the original stack's stack pointer is reset to the copy value held on the additional stack before the pop.

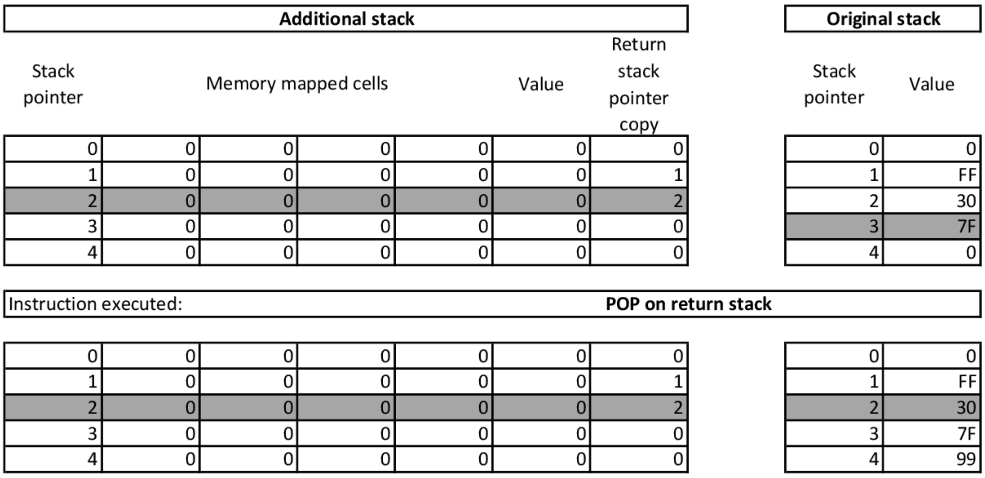
The conceptual scheme may be extended with multiple additional stacks as follows:

Each new additional stack is added on the "left", so that it has the same relationship with the current "leftmost" additional stack as the first additional stack has with the original stack according to the rules above. When an additional stack is pushed, all of the stacks on its right are simultaneously pushed with the same value. When an additional stack is popped, the reset of stack pointers flows to each stack on its right in a chain sequence until the reset of the stack pointer of the original stack. When a stack is popped to such a position that it causes a pop of the additional stack on its "left", then it is not necessary to propagate that behavior further to the left.

4 High level application to the FORTH programming language

To apply this conceptual model to the FORTH programming language the following arrangement is made. The FORTH return stack is the "original stack" in the terminology of the previous section and two additional stack are added (figure 7). The first additional stack is termed the subroutine stack and the second additional stack is termed the exception stack. PUSH and POP operations on each of the three stacks are mapped to FORTH primitives as follows: >R and R> operate on the return stack. EXECUTE and EXIT operate on the subroutine stack and CATCH and THROW operate on the exception stack (table 1).

The first cell on the subroutine stack holds a copy of the return stack's stack pointer. The second cell on the subroutine stack is reserved for storing the return address of a subroutine call. There are at least an additional 16 cells on the subroutine stack that are memory-mapped to the system

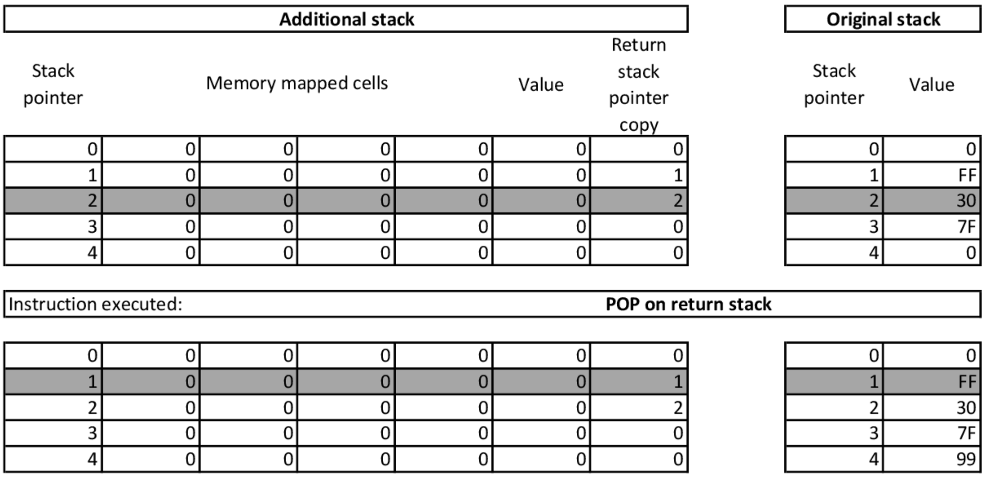


Additional stack							**Original stack**	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0

Instruction executed: **POP on return stack**

0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 3: Illustrating the rule that when the original stack is popped then there is no impact on the additional stack provided that the value of the original stack pointer after the pop is not less than the copy value held on the additional stack.

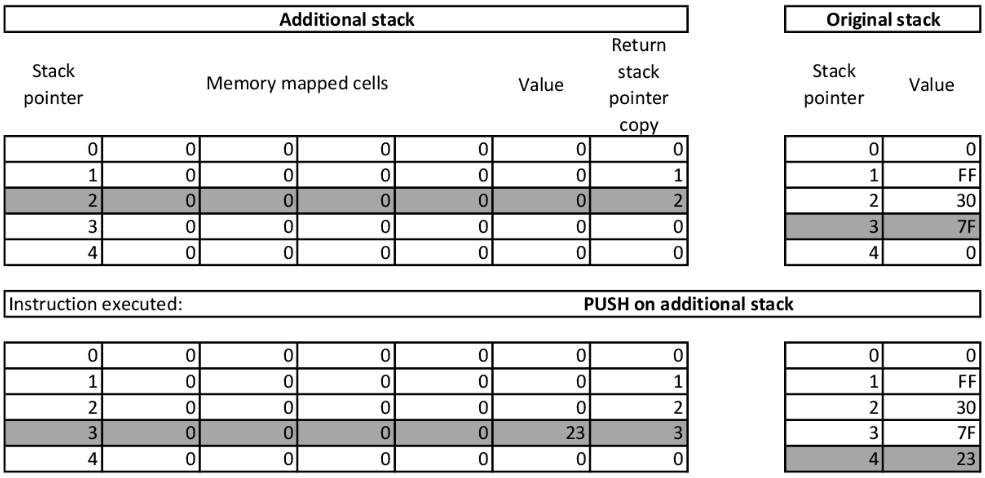


Additional stack							**Original stack**	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	0

Instruction executed: **POP on return stack**

0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 4: Illustrating the rule that when the original stack is popped then the additional stack is also popped if the value of the original stack pointer after the pop is less than the copy value held on the additional stack.

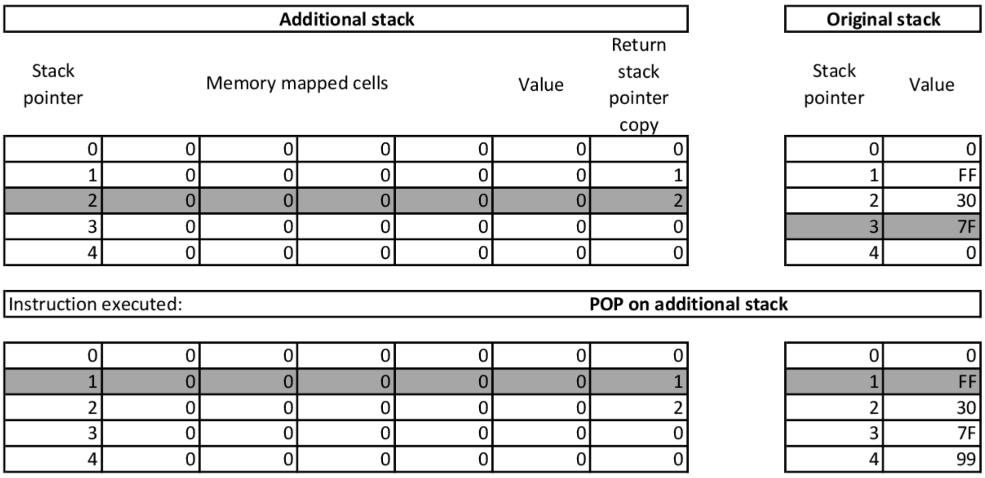


Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	**0**	**0**	**0**	**0**	**0**	**2**	2	30
3	0	0	0	0	0	0	**3**	**7F**
4	0	0	0	0	0	0	4	0

Instruction executed: **PUSH on additional stack**

0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	0	0	0	0	0	2	2	30
3	**0**	**0**	**0**	**0**	**23**	**3**	3	7F
4	0	0	0	0	0	0	**4**	**23**

Figure 5: Illustrating the rule that when the additional stack is pushed then two additional operations occur: firstly the current value of the original stack's stack pointer is copied to the first (rightmost) cell on the additional stack and secondly the original stack is also pushed with the same value as was pushed to the additional stack.

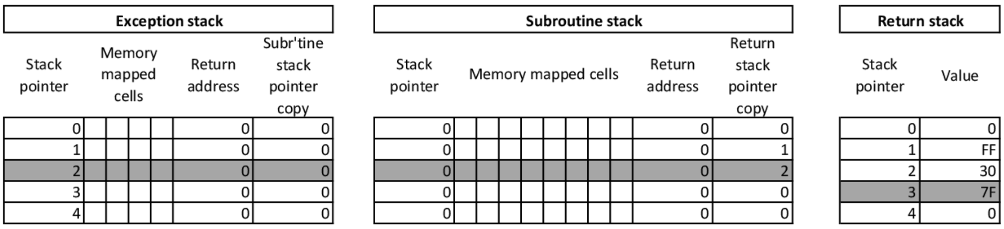


Additional stack							Original stack	
Stack pointer	Memory mapped cells				Value	Return stack pointer copy	Stack pointer	Value
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	FF
2	**0**	**0**	**0**	**0**	**0**	**2**	2	30
3	0	0	0	0	0	0	**3**	**7F**
4	0	0	0	0	0	0	4	0

Instruction executed: **POP on additional stack**

0	0	0	0	0	0	0	0	0
1	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**FF**
2	0	0	0	0	0	2	2	30
3	0	0	0	0	0	0	3	7F
4	0	0	0	0	0	0	4	99

Figure 6: Illustrating the rule that when the additional stack is popped then the original stack's stack pointer is reset to the copy value held on the additional stack.



Exception stack			
Stack pointer	Memory mapped cells	Return address	Subr'tine stack pointer copy
0		0	0
1		0	0
2		0	0
3		0	0
4		0	0

Subroutine stack			
Stack pointer	Memory mapped cells	Return address	Return stack pointer copy
0		0	0
0		0	1
0		0	2
0		0	0
0		0	0

Return stack	
Stack pointer	Value
0	0
1	FF
2	30
3	7F
4	0

Figure 7: Application of the extended return stack model to FORTH with an exception, subroutine and return stack. The additional memory-mapped cells on the subroutine and exception stacks are shown narrower for clarity of the diagram only.

address space via registers in such a way that the top-of-stack values are always visible at fixed addresses regardless of the position of the stack pointer. These cells are used to hold subroutine local variables. Because of the way the subroutine stack operates with EXECUTE and EXIT, the correct set of local variables will always be available at the relevant memory addresses. (The choice of 16 cells for local variables is made to comply with the minimum ANSI FORTH requirement, but in the case of the N.I.G.E. Machine sufficient FPGA BLOCK RAM is available to allow for more cells if desired, table 2.)

The first cell on the exception stack holds a copy of the subroutine stack's stack pointer. The second cell on the exception stack holds a copy of the exception return address. The exception stack also has a number of additional cells that are memory-mapped to fixed addresses. Just as with subroutine local variables on the subroutine stack, these memory mapped cells are used to hold local variables that have scope within a single exception.

Instruction	Exception stack	Subroutine stack	Return stack
PUSH	CATCH	EXECUTE	>R
POP	THROW	EXIT	R>

Table 1: Mapping of PUSH and POP operations on the exception stack, subroutine stack and return stack to FORTH primitives

The operation of the extended return stack as concerns execution flow control will be discussed in relation to the following code example (reordered to match the flow of the text). The stack effects are also illustrated in figure 8.

```
:  mainloop
    ’ innerloop CATCH
    if .”  Error code” .  then
;
:  innerloop
    sub1
\ code omitted
;
:  sub1
    sub2
    255 THROW
\ code omitted
;
:  sub2
    10 >R
;
```

Within the FORTH word `mainloop` the first action is to call the word `innerloop` by way of a CATCH statement. CATCH is mapped to a PUSH instruction on the exception stack. The effect of the CATCH is to increment the exception stack pointer. Because the exception stack is the “leftmost” additional stack, a PUSH to this stack also increments the subroutine stack pointer and the return stack pointer. The first cell of the exception stack is pushed with the value of the subroutine stack pointer as it is before being incremented. The second cell of the exception stack is pushed with the return address for this CATCH statement. Since a PUSH operation has also been applied to the subroutine stack, the first cell of the subroutine stack is pushed with the value of the return stack pointer as it is before being incremented. The second cell of the subroutine stack is pushed with the return address for this CATCH statement. The return address for this CATCH is also pushed onto the return stack. At this point control has been passed to the word `innerloop`.

The first action of the word `innerloop` is to call the word `sub1` with an implied EXECUTE statement. EXECUTE is mapped to a PUSH instruction on the subroutine stack. The effect of the EXECUTE is to increment the subroutine stack pointer and, since the subroutine stack is “left” of the return stack, to also increment the return stack pointer. The first cell of the subroutine stack is pushed with the value of the return stack pointer as it is before being incremented. The second cell of the subroutine stack is pushed with the subroutine return address. The subroutine return address is also placed onto the return stack. The exception stack is not affected by a subroutine call, consistent with the conceptual scheme as outlined. At this point control has been passed to the word `sub1`. The first action of the word `sub1` is to call the word `sub2`. The mechanism involved is the same again with the result that both the subroutine and return stack pointers are incremented a second time.

The word `sub2` places the value 10 on the return stack above the current return address before calling the implied EXIT. In a typical FORTH implementation exiting a subroutine after placing an arbitrary value on the top of the return stack could result in an unstable condition because the arbitrary value could be taken as the subroutine return address. However in the extended return stack arrangement as described here the EXIT statement is mapped to a POP instruction on the subroutine stack. The effect of the POP on the subroutine stack is to return execution to the return address as read from the subroutine stack (the second cell position), reset the return stack’s stack pointer to the value as read from the subroutine stack (the first cell position) and then decrement the subroutine stack pointer. As a result EXIT will return the flow of execution to `sub1` and the return stack pointer will return to the position that it was in when the subroutine call to `sub2` was originally made.

The following action is to call the word THROW with the value 255 on the top of the parameter stack. THROW with a non-zero parameter is mapped to a POP operation on the exception stack. The effect of THROW at this point is to return the flow of control to the exception return address that was copied on to the exception stack when the corresponding CATCH was called. In addition, the subroutine stack is reset to the position that it was in when CATCH was called since the copy of the subroutine stack pointer that was copied to the exception stack at that time is now written back to the subroutine stack pointer. Following this, the return stack is also reset to the position that it was in when CATCH was called as the copy of the return stack pointer on the reset subroutine stack is written back to the return stack.

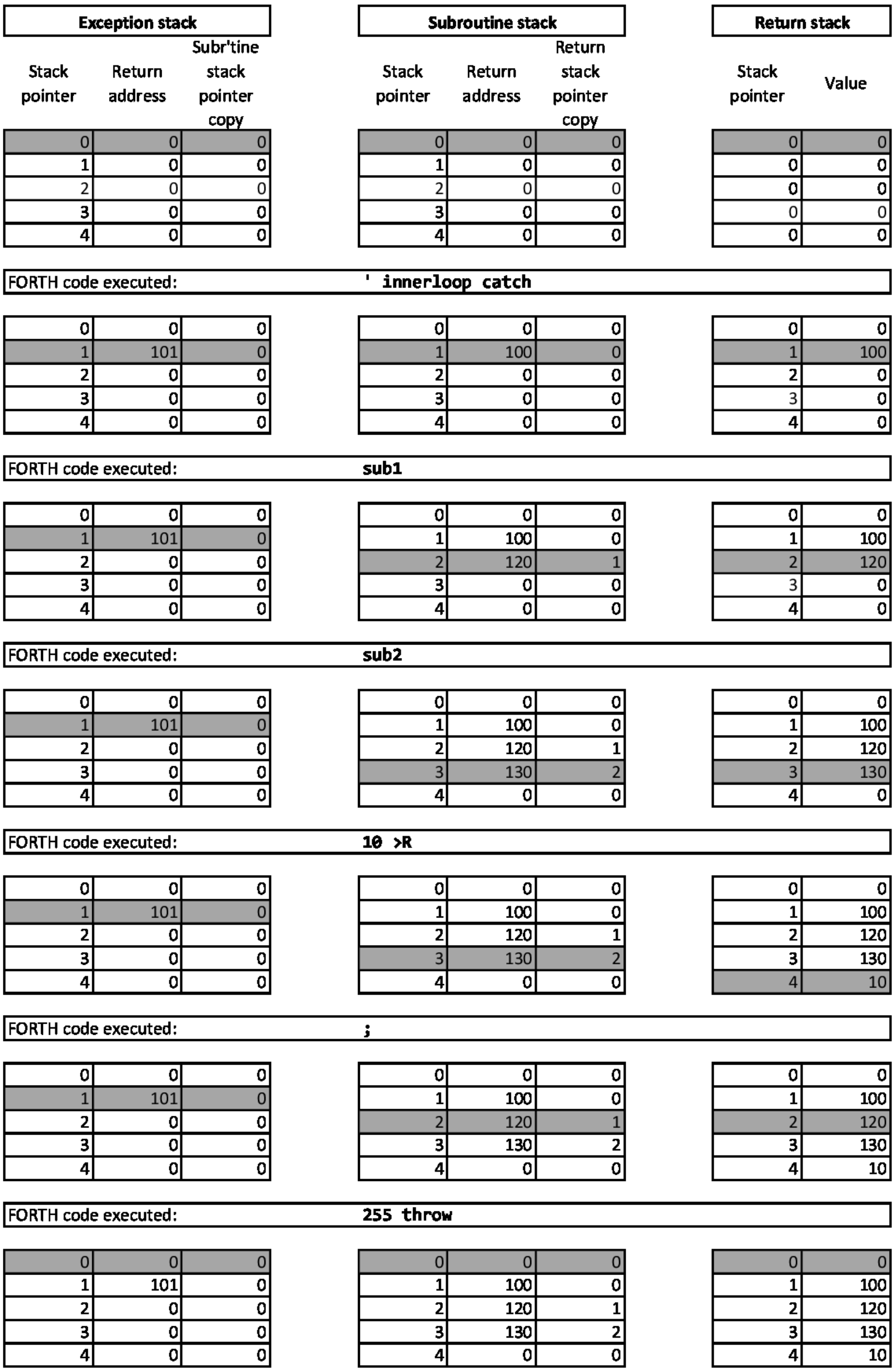
5 Additional FORTH requirements for subroutine and exception handling

The conceptual scheme for additional return stacks as described in the previous two sections provides almost all of the functionality that is needed to implement exception handling in FORTH. However to implement fully all of the required ANSI FORTH functionality in CATCH and THROW, some further mechanisms are required in addition to the additional stacks as described in the previous section.

Subroutine calls and exception handling could be operated by the additional stacks without any need for the return address of a subroutine call to be placed on the top of return stack. This is because as described, both the subroutine stack and the exception stack hold return addresses for subroutine and exception calls. However for comparability with existing FORTH code a copy of the subroutine return address should be placed on the return stack. There is a further point here. FORTH subroutines may remove the subroutine return address from the top of the return stack so that when an EXIT instruction is subsequently encountered, flow control is returned to the caller of the caller of that subroutine. This behavior, known as backtracking, may be used in implementations of CREATE DOES> and in other applications [15]. Since the conceptual scheme for the additional stacks already requires that if a stack is popped above the level of the copy of its stack pointer held by the additional stack to its right, traditional FORTH code that takes advantage of backtracking can run without alteration in the extended return stack scheme. (Note that backtracking in this form is environmentally dependent.)

Another issue that needs to be dealt is that EXIT (including an implied exit at the end of a FORTH word definition) needs to place a value of zero on the parameter stack if the subroutine was called with CATCH, but does not place any value on the parameter stack if the subroutine was called with EXECUTE (including an implied execute when a FORTH word is complied inside a definition). The conceptual scheme as discussed in the previous sections maps EXIT to a POP operation on the subroutine stack with flow control passing to the return address held on the subroutine stack. The difficulty is that subroutine stack doesn't "know" whether the current FORTH word was called with EXECUTE or CATCH.

A solution is achieved by having the return address that is stored on the exception stack to be one instruction ahead of the return address that is stored on the subroutine stack and arranging that a CATCH machine language instruction will always be followed by a ZERO machine language instruction (i.e. the instruction that places ZERO on the top of the parameter stack). As a result EXIT will direct execution flow, via the subroutine return address, to the ZERO instruction in a subroutine that was called by CATCH. On the other hand THROW, when called with a non-zero parameter, will direct execution flow via the exception return address and to the next following instruction and therefore skip the ZERO instruction. This is illustrated in the machine language excerpt below:



Exception stack			Subroutine stack			Return stack	
Stack pointer	Return address	Subr'tine stack pointer copy	Stack pointer	Return address	Return stack pointer copy	Stack pointer	Value
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	0	1	0	0	0	0
2	0	0	2	0	0	0	0
3	0	0	3	0	0	0	0
4	0	0	4	0	0	0	0

FORTH code executed: `' innerloop catch`

0	0	0	0	0	0	0	0
1	**101**	**0**	**1**	**100**	**0**	**1**	**100**
2	0	0	2	0	0	2	0
3	0	0	3	0	0	3	0
4	0	0	4	0	0	4	0

FORTH code executed: `sub1`

0	0	0	0	0	0	0	0
1	**101**	**0**	1	100	0	1	100
2	0	0	**2**	**120**	**1**	**2**	**120**
3	0	0	3	0	0	3	0
4	0	0	4	0	0	4	0

FORTH code executed: `sub2`

0	0	0	0	0	0	0	0
1	**101**	**0**	1	100	0	1	100
2	0	0	2	120	1	2	120
3	0	0	**3**	**130**	**2**	**3**	**130**
4	0	0	4	0	0	4	0

FORTH code executed: `10 >R`

0	0	0	0	0	0	0	0
1	**101**	**0**	1	100	0	1	100
2	0	0	2	120	1	2	120
3	0	0	**3**	**130**	**2**	3	130
4	0	0	4	0	0	**4**	**10**

FORTH code executed: `;`

0	0	0	0	0	0	0	0
1	**101**	**0**	1	100	0	1	100
2	0	0	**2**	**120**	**1**	**2**	**120**
3	0	0	3	130	2	3	130
4	0	0	4	0	0	4	10

FORTH code executed: `255 throw`

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	101	0	1	100	0	1	100
2	0	0	2	120	1	2	120
3	0	0	3	130	2	3	130
4	0	0	4	0	0	4	10

Figure 8: Illustration of the effect of some FORTH code on the exception, subroutine and return stacks. The current position of each stack pointer is show in grey. For clarity the memory-mapped cells for local variable storage on the subroutine and exception stacks have been omitted from the diagram.

```
PC address   Instruction               Comment
100          #.1          <sub addr>   ; load the subroutine address on stack
105          CATCH
106          ZERO                      ; 106 is the return address for EXIT
107          <next>                    ; 107 is the return address for THROW
```

Although the first two cells on the subroutine and control stack are used for control purposes, both of the stacks are multiple cells wide and the remaining cells are intended to be used for variable storage. As previously mentioned, the conceptual scheme and its implementation is such that the contents of the cells at the top of the subroutine and exception stacks are always available at fixed memory locations regardless of the value of the return stack pointer.

The remaining available cells on the subroutine stack are used to hold variables that are local to a subroutine call. As the subroutine stack is pushed down with EXECUTE or CATCH instructions, new variable storage is exposed. As the subroutine stack is popped up with EXIT or THROW instructions, the set of local variables reverts to those applicable to the appropriate subroutine level. In order to take best advantage of the cells on the subroutine stack that provide storage for locals it may be arranged that each time the subroutine stack is pushed, these cells will be written with a default value of zero.

The remaining available cells on the exception stack are used to hold variables that have scope within a particular exception. Suitable candidates for variables to hold on the exception stack are discussed in section 7.1. It should be arranged that each time the exception stack is pushed, these cells will be written with a copy of the cells immediately above them on the exception stack. In this way exception level variables will persist through subsequent CATCH statements unless they are explicitly changed.

The ANSI FORTH standard requires that THROW (with a non-zero parameter) also restore the parameter stack pointer to the value that it held at the time of the relevant CATCH statement. To accommodate this behavior one cell on the exception stack should hold a copy of the parameter stack pointer and automatically be updated at the time of a CATCH statement. Additional logic should reset the parameter stack pointer to the value as held on the exception stack at the time of a non-zero THROW. In this way CATCH and THROW can produce all of the required ANSI FORTH functionality simply by operation of the additional stacks without the need for supporting FORTH code. This is how the additional return stacks have been implemented on the N.I.G.E. Machine.

6 Implementation on the N.I.G.E. Machine

A number of design enhancements have been made to the N.I.G.E. Machine since it was demonstrated at EuroFORTH 2013. These are briefly summarized here for context. Firstly the overall design was ported from a Xilinx Spartan 3E FPGA on the Diligent Nexys 2 development board to a Xilinx Artix 7 FPGA on a Digilent Nexys 4 development board. In the process of porting the design the direct memory access (DMA) controller that mediates access to the off-chip 16 M byte pseudo-static dynamic RAM (PSDRAM) chip was completely re-written to conform to the AXI-4 protocol, the system clock speed was increased from 50MHz to 100MHz, and the program memory space was increased from 48K bytes to 128K bytes. (16M bytes of data memory is additionally available in PSDRAM). The native FAT file system software was also upgraded to support SD cards formatted with partition tables (as is common with higher capacity micro-SD cards). Finally a new video mode was added with 1024*768 resolution. A revised user manual has been produced that includes a quick start guide and documentation of the system features.

The subroutine and exception stacks described in section 4 and the further FORTH functionality described in section 5 were successfully implemented on the Nexys 4 version of the N.I.G.E. Machine. Three new machine language instructions were created: CATCH, THROW and RESETSP.

CATCH completes execution in 2 clock cycles (table 3). THROW completes execution in 3 cycles for a non-zero parameter and in 1 cycle if the parameter is zero. RESETSP resets all of the parameter stack pointer, the return stack pointer, the subroutine stack pointer and the exception stack pointer to zero. It completes execution in 1 clock cycle. The intended use of RESETSP is to return the machine to a known configuration upon reset. Three machine language instructions were removed from the instruction set, partly so that their opcodes could be reused in the three new instructions and partly because their use would interfere with the proper operation of the extended return stack. The removed instructions were: RSP@, RSP!, PSP! which respectively read and wrote the return stack pointer and wrote the parameter stack pointer. The original instruction PSP@ remains in the instruction set and is used within the implementation of the FORTH word PICK.

The subroutine and exception stacks were implemented as FPGA BLOCK RAM. The subroutine stack is 17 cells wide (544 bits) and 512 cells deep. The first 32 bit cell is subdivided into a 9 bit cell that holds a copy of the return stack pointer and a 23 bit cell that holds the subroutine return address. There are 16 cells available for the storage of subroutine local variables. 36 K bytes of BLOCK RAM are allocated to the subroutine stack. The exception stack is 9.5 cells wide (304 bits) and 512 cells deep. The first 32 bit cell is also subdivided into a 9 bit cell that holds a copy of the subroutine stack pointer and a 23 bit cell that holds the exception return address. There is next a 16 bit wide cell that holds a copy of the parameter stack pointer. There are 8 cells available for the storage of exception variables. (The value of 8 cells was chosen somewhat arability with the expectation that it can easily be adjusted depending on future needs). 19 K bytes of BLOCK RAM are allocated to the exceptions stack, so that the total BLOCK RAM used for both additional stacks is 55 K bytes.

The N.I.G.E. Machine uses microcode to control the datapath. In this scheme the lowest 5 bits of each machine language opcode are interpreted as an address and access a BLOCK RAM element. The data value returned from the BLOCK RAM element at that address are the control lines used to configure the multiplexers in the datapath . To accommodate the extended return stack the number of control lines was extended from 14 to 21. For example, 3 bits are used to control the subroutine stack pointer. Of the allowed 8 possible configurations available in 3 bits, 5 configurations are used: no change, decrement, increment, reset to the copy value held on the exception stack, reset to zero. The exception stack pointer and return stack pointer are similarly controlled by microcode. An additional configuration was added to the control of the parameter stack pointer: reset to the copy held on the exception stack. This configuration is used by THROW.

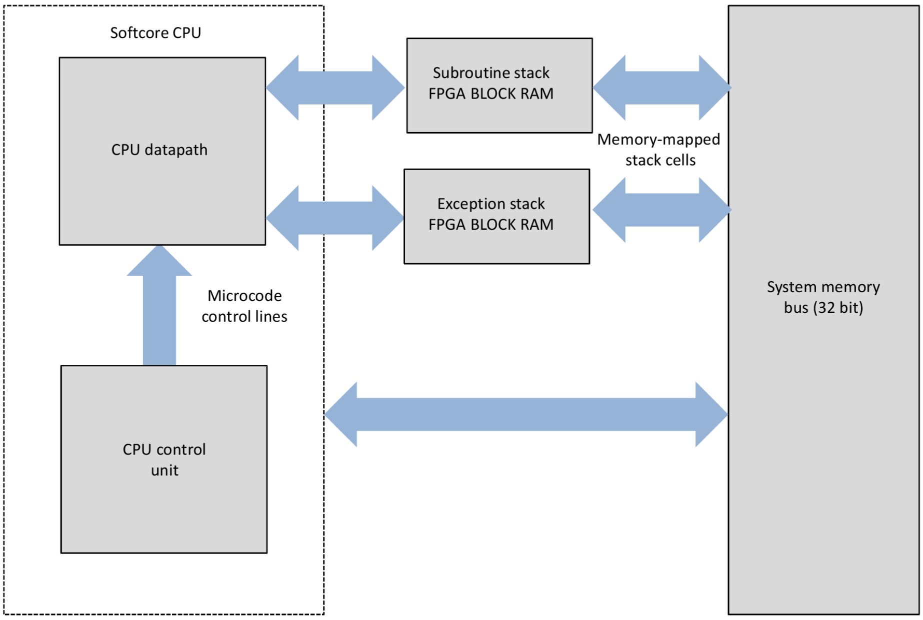

Softcore CPU
CPU datapath
Microcode control lines
CPU control unit
Subroutine stack FPGA BLOCK RAM
Exception stack FPGA BLOCK RAM
Memory-mapped stack cells
System memory bus (32 bit)

Figure 9: System diagram of the extended return stack implementation on the N.I.G.E. Machine

In order to make the subroutine and exception stacks available within the system memory space VDHL modules were created that interface the system memory space address lines with the BLOCK RAM elements holding the stacks. In each case the interface operated so that regardless of the position of the subroutine and exception stack pointers, the top of stack values are found at fixed memory locations. (The interface does not allow any other access from the system memory space into the subroutine and exception stacks, except to the at top of the stack values). These modules are also responsible for setting the newly exposed cells on the subroutine stack to zero and for "copying down" the values on the exception stack when a PUSH occurs, as described in the previous section.

Lastly, THROW was configured so that it would also signal the CPU to cancel any current interrupt condition. Thus if a non-zero THROW occurs within interrupt code the interrupt condition will be canceled when flow control is returned to the exception address. This is important since in the N.I.G.E. Machine an interrupt condition blocks further interrupts from occurring.

Making CATCH and THROW machine language instructions comes at the expense of implementing the subroutine and exception stacks in hardware. With the Artix 7 FPGA (device XC7A100T), the additional resources consumed are quite modest (table 2).

Version of N.I.G.E. Machine	**BLOCK RAM**	**FPGA fabric logic**
With extended return stack	32%	8.7%
Without extended return stack	22%	7.0%

Table 2: Artix 7 FPGA resource consumption comparing versions of the N.I.G.E. Machine with and without the extended return stack.

All of the required functionality was therefore achieved by adding BLOCK RAM elements to serve as the additional stacks, by extending the microcode used to control the datapath, by adding the required multiplexers to control the stacks within the datapath, and by making the top of stack values on the subroutine and exception stacks available in the system memory space. There was no need to resort to any unstructured "glue logic" to complete the design. Consequently the timing performance of the N.I.G.E. Machine was not affected and the design can be comfortably implemented at a clock speed of 100MHz.

The syntax for local variables implemented in the N.I.G.E. Machine system software follows VFX FORTH [20] and is documented in more detail in the N.I.G.E. Machine user reference manual [4]. The compiler utilizes a recognizer to identify local variable names ahead of searching the main dictionary, and is aware of the fixed memory addresses where local variables are stored on the subroutine stack.

7 Discussion

This section will examine the advantages and limitations of the extended return stack as compared with traditional approaches to subroutine and exception handling.

7.1 Flexibility

Using the exception stack to hold variables that have scope within a single CATCH statement ensures that if a THROW occurs all of these variables are guaranteed to be restored to their values prior to the CATCH. This restoration happens as an atomic operation inside a single machine language instruction. The feature can be leveraged for considerable utility as is illustrated by the following example. Anton Ertl has shown several models for a word `hex.` that prints a number in hexadecimal without changing BASE [13]. A new model that can be adopted with the extended return stack is as follows:

```
:  hex.-helper
    hex     \ the variable BASE is located on the exception stack
    u.
;

:  hex.
    [’] hex.-helper catch throw     \ no exception frame needed with extended
return stack.  CATCH is as fast at EXECUTE
;
```

The word `hex.` calls `hex.-helper` using CATCH, thus pushing the exception stack. Within `hex.-helper` the word `hex` stores the value of 16 to the variable BASE, which is located on the exception stack. Regardless of how `hex.-helper` exits, either at the implicit EXIT statement or due to a THROW during `u.`, the exception stack will be popped at that time and the value of BASE will be restored to the value that it held prior to the call to `hex.-helper`.

This model can be extended to a more general case with a word `debug.` that prints a number to the RS232 port in hexadecimal without changing BASE or redirecting output. The point is that any variables that have scope within a single exception will be automatically restores to their former values upon EXIT or THROW.

```
:  debug.-helper
    hex     \ the variable BASE is located on the exception stack
    >remote    \ the character output vector is updated
    u.
;

:  debug.
    ['] debug.-helper catch throw
;
```

Anton Ertl has developed various models for region based memory allocation [18]. This system could likely be used to support region based memory allocation if the critical references are held on the exception stack. A complete model for region based memory allocation using the extended return stack is a topic for further research.

An additional point of flexibility concerns local variables. FORTH implementations typically favor VALUE flavored locals because these can be implemented using index offset load/store instructions. VARIABLE flavored locals may be less suitable for typical FORTH implementations because they require a memory address to be explicitly calculated on each reference. By contrast, the extended return stack makes it straightforward and efficient to implement VARIABLE flavored locals because the memory addresses of the local variables are fixed. The relocation of local variables onto the subroutine stack also removes the possibility of interference with DO LOOP operations that may occur when the return stack is used for local storage.

However the extended return stack approach to local variables has its constraints. Firstly the number of local variables available is limited to the cells provided in hardware on the subroutine stack. In the current implementation of the N.I.G.E. Machine there are 16 cells available for local variables on the subroutine stack and 8 on the exception stack and this could be extended relatively easily. Secondly, since most FORTH subroutines do not use local variables the approach of keeping them on the subroutine stack may seem wasteful of memory resources. However this is more of a trade-off decision, since the prize obtained by adopting this approach is the ability to implement CATCH and THROW as single machine language instructions. Depending on the chosen FPGA, this trade-off may not be a significant concern (table 2).

In the N.I.G.E. Machine system software, the variable BASE, and vectors for redirecting keyboard and screen input/output to the RS232 port are held on the exception stack.

7.2 Speed of code execution

Fast subroutine execution is important in FORTH because of the highly factored nature of FORTH code. The speed of subroutine execution on the N.I.G.E. Machine is unchanged by the implementation of the extended return stack. Fast exception handling may also be important. Although CATCH and THROW were not implemented on the N.I.G.E. Machine prior to the extended return stack, considering the reference implementation of CATCH and THROW it is likely that executing CATCH in 2 clock cycles and THROW in 3 clock cycles will be an order of magnitude faster than implementing these constructs in software (table 3).

Version of N.I.G.E. Machine	EXECUTE	EXIT	CATCH	THROW
With extended return stack	2	2	2	3
Without extended return stack	2	2	n/a	n/a

Table 3: Speed of subroutine and exception execution measured in clock cycles the N.I.G.E. Machine with and without the extended return stack implemented. CATCH and THROW were not implemented on the original N.I.G.E. Machine

Speed of local variable access is also important: arguably local variables are expected by programmers to be the fastest-to-access storage available. This will be the case if local variables are held in CPU registers rather than system memory, as is likely to be the case with implementations of the C programming language or with certain native FORTH implementations. However if local variables are held in registers then there will be a time penalty on subroutine entry and exit due to the need to save and restore the register set. Alternatively, holding local variables in system memory dispenses with this penalty, but access to system memory will likely be significantly slower than access to registers. The extended return stack offers the best of both worlds. Specifically allocated local variable storage on the subroutine stack means that there is no requirement to save or restore a register set. At the same time access to local variables is directly mediated by FPGA fabric logic. On the N.I.G.E. Machine all load/store operations to local variables complete execution in 2 clock cycles (as is the case with access to the N.I.G.E. Machine's BLOCK RAM in general).

7.3 Robustness / fault tolerance

Four arguments are made why the extended return stack concept, implemented in hardware, offers significant benefits for robustness and fault tolerance for FORTH programmes:

The hardware based extended return stack provides an absolute guarantee that variables held on the exception stack will be returned to their former (prior to CATCH) values upon EXIT or THROW. This occurs as an atomic operation within a single machine language instruction. This guarantee means that no further software problem solving is needed to ensure the safe handling of these variables in the event of an exception. This is valuable in high integrity software both as a feature in its own right and because, as Paul Bennett and Malcolm Bugler explain [12], once a programming surface is certified then it serves as a extensible platform for further applications.

A subroutine EXIT will execute correctly even if the subroutine has left spurious values on the top of the return stack (for example by leaving a DO LOOP without UNLOOP). This was demonstrated in section 4. Whilst it might raise a concern for the moral hazard of programmer complacency in managing the return stack, in critical situations the avoidance of the serious error that would have occurred otherwise may be a significant benefit. As Nick Nelson points out, a system that struggles on despite programming errors is a valid strategy for avoiding failures [17].

Although the literature does not suggest that exception processing is currently a bottleneck for FORTH programs[13], the extended return stack offers very fast exception processing in hardware (i.e. as fast as an ordinary subroutine call and return). This assurance on performance may encourage programmers to increase their use of CATCH and THROW, this improving software integrity.

As a final point, since the exception stack does not rely on a global variable to anchor its execution, the possibility that this variable could be corrupted, with catastrophic consequences for subsequent exception flow control, is avoided.

However on a practical level, and as also noted in section 3, before the N.I.G.E. Machine could be used in any critical systems an extensive program of structured testing (or some other approach) would be needed to certify the integrity of the N.I.G.E. Machine itself.

8 Conclusion

The extended return stack starts with a conceptual scheme for additional stacks that is not dependent on any particular hardware or FORTH implementation. The straightforward way in which this structure is able to handle exception and subroutine processing, and the one-to-one correspondence of the CATCH, THROW, EXECUTE, EXIT, >R, and R> FORTH primitives with PUSH and POP operations on the exception, subroutine and return stacks suggests that the conceptual stack scheme has a natural correspondence to the underlying logic structure of exceptions and subroutines in FORTH. The additional tweaks to this conceptual scheme that are needed to fully implement the requirements of ANSI FORTH are not extensive.

Implementation on the N.I.G.E. Machine was straightforward because the N.I.G.E. Machine's softcore is microcode based. The additional functionality is obtained by extending the number of control lines set by microcode and adding appropriate multiplexers to the datapath. The FPGA resource requirements for the extended return stack are minimal on an Artix 7.

The availability of variables on the exception stack that are guaranteed to be restored to their pre-CATCH value upon EXIT or THROW may be a genuine innovation. In addition, the implementation of CATCH and THROW as single machine language instructions makes exception processing very fast. Overall this paper has argued that the extended return stack offers significantly enhanced flexibility, speed, and robustness of subroutine and exception handling in FORTH.

Further work is intended in three areas:

- preparing additional verifications that the additional return stack design works correctly for subroutine and exception handling in all corner cases
- developing applications for variable storage on the exception stack, for example complementing with Anton Ertl's models for region based memory allocation [18]
- seeking further ways in which the extended return stack could drive further improvements in robustness and fault tolerance of FORTH software

The author sincerely wishes to thank the anonymous academic reviewers for their time and effort in providing feedback. Their comments on content and calibration have been very helpful in clarifying the author's thinking and improving the presentation of the paper.

References

[1] The author, video demonstrations https://www.youtube.com/channel/UCz_LqPfKT0r2rEID7Av-Chw

[2] The author, "The N.I.G.E. Machine: an FPGA based micro-computer system for prototyping experimental scientific hardware", in *EuroFORTH*, 2012

[3] The author, "Optimizing memory access design for a 32 bit FORTH processor", in *EuroFORTH*, 2013

[4] The author, Github open source repository https://github.com/Anding/N.I.G.E.-Machine

[5] James Bowman , "J1: a small Forth CPU Core for FPGAs" in *EuroFORTH*, 2010

[6] K. Schleisiek, "MicroCore," in *EuroFORTH*, 2001.

[7] B. Paysan, "b16-small – Less is More," in *EuroFORTH*, 2004.

[8] E. Hjrtland and L. Chen, “EP32 - a 32-bit Forth Microprocessor,” in Canadian Conference on Electrical and Computer Engineering, pp. 518–521, 2007.

[9] E. Jennings, “The Novix NC4000 Project,” Computer Language, vol. 2, no. 10, pp. 37–46, 1985.

[10] Rible, John, "QS2: RISCing it all," Proceedings of the 1991 FORML Conference, Forth Interest Group, Oakland, CA (1991), pp. 156-159.

[11] C. Bailey, R. Sotudeh, and M. Ould-Khaoua, “The Effects Of Local Variable Optimisation In A C-Based Stack Processor Environment.”, in *EuroFORTH*, 1994

[12] Paul E. Bennett, Malcolm Bugler, “Certification of High Integrity Software”, in *EuroFORTH*, 1998

[13] M. Anton Ertl, “Cleaning up after yourself”, in *EuroFORTH*, 2008

[14] M. Anton Ertl, “Ways to Reduce the Stack Depth”, in *EuroFORTH*, 2011

[15] M.L.Gassanenko, “Open Interpreter: Portability of Return Stack Manipulations”, in *EuroFORTH*, 1998

[16] Michael Milendorf, “CATCH and THROW”, in *EuroFORTH*, 1998

[17] N.J. Nelson, “Crash Never”, in *EuroFORTH*, 2011

[18] M. Anton. Ertl, “Region-based Memory Allocation”, in *EuroFORTH*, 2013

[19] P. J. Koopman, Jr., “Stack computers: the new wave”, Halsted Press, 1989

[20] Stephen Pelc, “VFX FORTH for Windows”, MPE, 2011

[21] Jaanus Pöial, “The algebraic specifications of stack effects for Forth programs”, FORML, 1990

[22] Jaanus Pöial, “Multiple stack effects of Forth programs”, EuroFORML, 1991

[23] Bill Stoddart and Peter Knaggs, “The Cell Type”, Proc. 1991 Rochester Forth Conf.

[24] Bill Stoddart and Peter Knaggs, “Formal Forth”, Proc. 1991 Rochester Forth Conf.

[25] Bill Stoddart and Peter Knaggs, “The Event Calculus: Formal Specification of Real Time Systems by means of Diagrams and Z Schemas”, 5th International Conference on putting into practice method and tools for information system design, 1992, Institute Universitaire de Technologies, Nantes, France

[26] Bill Stoddart and Peter Knaggs, “Type inference in Stack Based Languages”, Formal Aspects of Computing 5(4):289-98, Springer International

[27] Klaus Schleisiek, "ERROR TRAPPING: a Mechanism for Resuming Execution at a Higher Level.", 1983 FORML Conference Proceedings, pp. 151-154, San Jose, CA: FORTH Interest Group, 1984

[28] Klaus Schleisiek, "Error Trapping and Local Variables", 1984 FORML Conference Proceedings, CA: FORTH Interest Group, 1985

[29] Brad Rodriguez, "A Forth Exception Handler", SIGForth Newsletter Vol. 1 No. 2 (Summer 1989)

[30] Brad Rodriguez, "Stack Frames in Forth", SIGForth Newsletter Vol. 1 No. 4 (Winter 1989)

Compiling to Flash

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

In the last two or three years, a number of embedded Forth systems have emerged that are self-hosted and compile directly to Flash. This paper explores some of the issues found at MPE when we implemented such a kernel for the MPE Lite edition cross-compilers. Issues for the TI MSP430 family and several ARM implementations are discussed.

Introduction

With the vast quantity of extremely low-cost hardware provided by semiconductor manufacturers comes an attitude that there should be software at an equivalent price. For a third-party toolmaker such as MPE, there's no money in this. The race to the bottom is asymptotic to zero.

Why on earth should any compiler vendor give tools away? The only answer is to expose students, hobbyists and evaluators to good-quality tools. There's a huge number of free (of cost) Forth systems available, but to professional eyes the vast majority of them are poorly implemented and woefully documented. Overall such Forth systems damage rather than enhance the reputation of Forth.

Schools in the UK have ridiculously small budgets for electronics and technology projects, so the Lite compilers are also part of our contribution to school science and technology education.

In designing the free-of-charge Lite Forth for our cross compilers, we had to be careful not to damage our own market. This achieved in several ways:

1. The Lite Forth kernel is not compatible with the standard MPE PowerForth kernel. The Lite Forth kernel compiles directly to Flash and is subject to change.
2. The Lite Forth kernel is not ANS or Forth200x compliant. It has changes to meet the requirements of compilation to Flash.
3. The Lite Forth target code supports a restricted range of CPUs and target hardware.
4. The Lite Forth cross compiler is limited to producing no more than a certain amount of code, 16 kb for an MSP430 and 64 Kb for a Cortex M0.
5. The Lite Forth compiler may not be used for commercial purposes.

The Lite compilers are available for the MSP430 and the ARM Cortex-Mx CPUs. The MSP430 is ideal for school and low-power use, but the low-power advantage over a Cortex-Mx is much less than may be anticipated. The Cortex-Mx CPUs are the current CPU of choice – they cost no more than 8 bit CPUs, and the ease of programming 32 bit CPUs reduces time to market.

The nature of Flash

There is wide variation among Flash devices in terms of how they are programmed. What is common is that they erase to all bits set to '1', and you can only program a '1' bit to a '0'. In order to set a '0' bit back to a '1', you have to erase a range of memory known as a page or sector, which range in size from 128 bytes to 64 kbytes. In some chips, there are sectors of varying size.

In simple chips such as the MSP430 CPUs, you perform a simple operation to permit writing and then write byte by byte to an erased sector. At a later time, you can rewrite a byte if you just change '1' bits to '0' bits. In others, such as many ST devices, Flash programming is performed through a Flash controller peripheral which has additional requirements, e.g. that bytes to be programmed are set to 0xFF, all bits set.

The important thing about all of this is that you can only program a Flash location once without encountering restrictions. These restrictions have an impact on the Forth dictionary structure and Forth notation.

Direct compilation to Flash

When we did Forth Lite for the MSP430, we were using a simple Flash system that could rewrite '1' bits to '0' bits. The major issues in writing a Forth that compiles directly to Flash are the dictionary header layout and compilation of forward branches.

Dictionary header layout

In a traditional Forth, there are two flag bits that are changed during compilation or even after compilation. These are the dictionary visibility bit (“smudge” bit) and the **IMMEDIATE** bit. The dictionary visibility bit turns out to be problematic, as some code, e.g. the common words **HIDE** and **REVEAL**, imply that this bit can be changed twice, which contradicts our rule that we can only change a '1' bit to a '0' bit. The solution to this was to remove the visibility bit completely, and only to link the word into the dictionary thread when we were satisfied that the word was correct. Hence the link field is initialised to all ones.

We can deal with the **IMMEDIATE** bit by inverting it so that a '1' bit indicates non-immediate and a '0' bit indicates that the word is immediate. Another change makes the implementation even simpler. The traditional phrase

```
: foo  …  ; immediate
```

is replaced by

```
imm : foo  … ;
```

which means that the **IMMEDIATE** bit is known when the header is constructed.

Forward branches

We can deal with forward branches by setting the branch target offsets to all '1' bits so that a typical branch looks like $opc111, $11111111.

Ah, but ...

This all worked fine on an MSP430 which has what might be called a classical Flash controller. However, the whole plan fell apart on a couple of ARM Cortex parts. For one family, you can only program the flash in minimum units of 16 bytes on a 16 byte boundary. For another, you can only modify bytes that have all bits set.

It was time for a rethink and to understand what we were really trying to achieve with this Forth kernel. The main requirement is to be able to cross-compile the kernel itself, and then to be able to compile as much code as is wanted using the target hardware. In our world, the cross compiler generates highly optimised code, but we are not really concerned with the quality of the code created by the target. After all, you can always upgrade the Lite compiler. A secondary initial aim was to use a common code base for the MSP430 and Cortex devices. We also asked ourselves if we actually wanted to increase the range of Lite compilers beyond MSP430 and ARM Cortex, and we decided that we do not.

By not using a common code base, we could simplify the 16 bit target for the MSP430. We also freed up what we could do with the 32 bit targets. A typical low cost 10 Euro board for an ARM Cortex-M0 has 128 kb Flash and 16 kb RAM. Thus we continued with the "classical" Flash assumption for the MSP430, and permitted ourselves to use a more complex approach for the 32 bit targets.

Compilation to a RAM buffer

After examining a variety of approaches such as generating a linked list of partially compiled bytes and rewriting them, we concluded that this would lead to having to special-case each ARM part with a different peculiarity in its Flash controller. Instead, we would take the simple approach of compiling the code to a temporary RAM buffer and then copying it to Flash when complete. We use special versions (`C!C` `W!C` and `!C`) of the store operations which take the Flash address but actually store the data in a buffer in RAM. Words that finish compilation such as `;` flush the buffer to Flash. This approach has the advantage of requiring almost no change to on-chip compilation and requires the least sacrifice of Flash.

Conclusions

Direct compilation to Flash is tedious because it can be affected by the minutiae of specific chips. However, there is sufficient commonality that it can be done using a small number of techniques – a single technique is not enough.

We are often reminded in Forth that we should only solve the problem at hand; we should not over-generalise. Compilation to Flash is one such class of problem. In particular, our desire to use a common code base for 16 and 32 bit systems conflicted with reality. Now we use separate code bases.

Acknowledgements

Dirk Bruehl persuaded me that an educational Forth is important. Michael Kalus provided editorial comments for the Lite compilers. Juergen Pintaske stubbornly prodded me to do more and reminded me what can be achieved with a solderless breadboard.

VFX Forth for ARM Linux

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

All VFX Forth versions have been built from the same source tree. However, VFX Forth for ARM Linux is the first ARM port since 1999. This paper looks at how well the original VFX Forth source tree has stood up to the changes of the last 15 years.

Introduction

There is now a large number of ARM-based systems running Linux. These range from expensive to very low cost, e.g. Raspberry Pi (around EU 40) and Beaglebone Black. These devices are so cheap that many traditional embedded systems can be replaced at lower cost by these off-the-shelf systems.

These low-end ARM systems use CPUs that range from 450 MHz ARM11s to 1GHz Cortex-A8s. The CPUs all support the original ARM 32 bit instruction set. The systems provide several different flavours of Linux.

The project was thus to port VFX Forth for Linux from the x86 implementation to an ARM with minimal changes to the overall VFX Forth source tree. The source tree is implemented for a multiple stage build. Here we are concerned with the first two stages:

1. Production of the Forth kernel with a primitive interface to the operating system, but with a full assembler, disassembler and code generator,
2. Self-compilation by the kernel of the Linux O/S interface and development tools.

The first stage build is performed by the existing Forth cross compiler for the ARM. This only required a few minor changes to match creeping changes to internal data structures in the target Forth.

Operating System startup and coded definitions

The startup code is contained in a single file which covers the ELF headers, Forth startup from the operating system, primitive access to shared library access routines, and the callback interface. Most of this code is written in assembler, and is the largest assembler component of the whole VFX Forth system. Once the cross assembler, disassembler and code generator are working, this is one of the most critical files in the system.

Barring one or two, all the other code definitions are in a small file that contains words that are best coded. For example, the base ARM32 instruction set does not include a divide instruction. A version of **CMOVE** is available that provides four times the performance of a byte-by-byte **CMOVE** but is over 900 bytes in size. These routines were taken from the existing embedded ARM target for the cross compiler.

Other files contains the operating system specific routines required for the first-stage build, the default console (unchanged from x86 Linux) and the binary save utility (virtually unchanged from x86 Linux).s

Assembler, disassembler, and VFX code generator

The assembler is cross-compiled because it used by the code generator. The disassembler is cross-compiled because you need it to debug the code generator. The code generator is cross-compiled so that all the code in the system is optimised.

The code is taken from the cross-compiler's code tree. Changes are required for defining words. The notation used is from the ANS draft cross-compiler proposal. It's not pretty but it works. Changes are also needed because the MPE cross compilers and VFX Forth use different notations for connecting compilation semantics to word names. This could be improved. Additional minor changes were required because the cross compilers are focused on embedded systems with separate Flash and RAM, whereas hosted systems mainly have a single address space in RAM.

Library linkage

Linking the Forth to shared libraries is a fundamental part of making a Forth for a hosted system. The MPE **`Extern:`** notation emphasises being able to copy and paste a C prototype from the Linux documentation. The following example is taken from the GTK interface.

```
Extern: gboolean "C" g_file_set_contents(
 const gchar * filename,
   const gchar * contents, gsize length,
   GError ** error
);
```

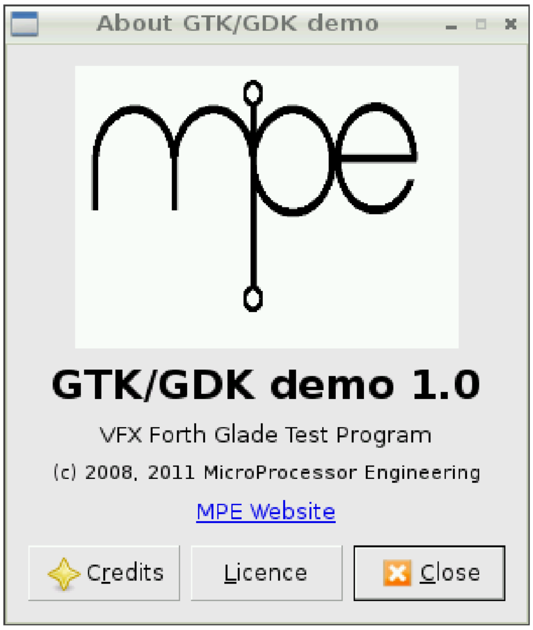
This is, in many ways, the most critical file in the port. It is affected by the startup code and the interface into **dlopen()** and friends. Although MPE has VFX Forth for 32-bit x86 Linux, ARM Linux uses a rather different calling convention with the first four integer parameters passed in registers. Several other O/S interfaces use a similar convention. The choice of how to pass floating point values to Linux affects the floating point package and the parameter passing mechanisms may affect Forth stack layout.

The floating point options are such that we do not yet know how many ABIs must be supported! There are two main ones, for the VFP hardware and for floating point emulation. In many ARM9 implementations, there is no FP hardware and software FP is used. Software FP may well use a library API that passes FP numbers in the integer registers and/or the C stack. Hardware FP may either use the same API as the software FP or may use a faster API that uses the VFP registers. The choice of API is probably defined by the choice of Linux. There is no guarantee that two Linux implementations for the same hardware will use the same FP API.

Once all the choices have been made for the shared library interface, the same choices have to be implemented for the callback interface.

GTK

MPE uses GTK for cross-platform GUIs across Windows, Linux and even Mac OS X. For Linux, GTK is also our primary GUI environment. It has been extended with a simple graphics extension that works in a similar manner to the old Borland BGI interface from long ago.



GTK/GDK demo 1.0

VFX Forth Glade Test Program

(c) 2008, 2011 MicroProcessor Engineering

MPE Website

Apart from the different shared library names on different operating systems, the GTK interface and the demo shown above uses the same source code unchanged.

Similarly the majority of the Forth examples and library interfaces compile unchanged.

GPIO

Using a Raspberry Pi as a base system, the speed of GPIO access varies hugely according to how it is done. The following link has the gory details:
http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/

Depending on language and implementation technique, you can expect to see GPIO access in the range 40 kHz (Python) to 20 MHz (optimised C). In VFX Forth we expect a generalised routine to achieve about 7 MHz, while specific access should exceed 15 MHz. To achieve such speeds, the Forth application must be run with root permissions.

Conclusions

Once code generation is good enough, the vast majority of a Forth system can be written in high level Forth. The hard parts of the remainder are involved in the operating system interface. A very few routines are still best written in assembler, for example a high performance version of **CMOVE**.

Given that the last 15 years of VFX Forth development have all been for the Intel IA32 instruction set, the addition of ARM and allowance for multiple instruction sets has caused very few changes to existing files. At least for a Linux operating system, there have been no changes to the second stage build except to automate (by conditional compilation) the selection of shared library file names.

As the cost of hardware designed to run ARM Linux has plummeted, e.g. Raspberry Pi, Beaglebone Black and Olimex OlinuXino all fall in the EU 30 to EU60 range, Linux boards are becoming cheaper than conventional embedded hardware. We can expect to see many traditional embedded applications migrate to Linux boards. In particular, we already see the Raspberry Pi (2 million sold), being modified in the B+ form to be significantly more suitable for industrial use – more I/O, better mounting holes, more USB.

Where hard real-time is still important there's always a trade-off, but we are already seeing some migration to FPGA+ARM solutions, e.g. Xilinx Zynq, where the FPGA portion handles the heavy-lifting of the hard real-time requirements. The Zynq incorporates a dual-core Cortex-A9, all the standard peripherals including Gigabit Ethernet, plus an FPGA. Such devices will, in the long term, make the traditional embedded system an extremely niche product.

Acknowledgements

Our thanks to Vic Watson, Juergen Pintaske and several others for encouraging us to generate VFX Forth for ARM Linux.

High Integrity Systems

CODE

by Paul E. Bennett IEng MIET, HIDECS Consultancy

Abstract:-

In Phil Koopman`s paper “The Grand Challenge of Embedded System Dependability” he sets out four challenges.

“Four significant challenges in embedded system dependability are:

- embedded-specific security approaches,
- unifying security with safety,
- dealing with composable emergent properties,
- and enabling domain experts to use advanced dependability techniques.”

[Koopman1]

This paper will describe the benefits of developing software as components of a system, following on from their interface specification, review/examination, functional testing and limitations testing. It will continue to explain why Forth is one such suitable Component Oriented Development Environment. We will cover the development process (including organisational requirements) and the production of well specified and correctly behaving (according to its specification) software.

Mechanics

Early Engineering was entirely mechanical in nature. The invention of the wheel and axle, the ramp, the Archimedes Screw, Nut and bolt etc. The early days saw some failures but the engineers of the time learnt how to make the components better and stronger within the constraints they were forced to deal with. They also gained the wisdom to know when to state that the constraints were too restrictive and not workable. For installations where safety was a major concern, the notion of using certified components became widespread. Such components had to be certified as meeting specifications and being compliant with standards in materials inspection and testing. Our mechanical engineering counterarts have had a very extensive head-start. An example of an early standard was the one for screw threads.

Electrical & Electronics

Slightly newer than the mechanical engineering industry sector, the electrical and electronics industry has managed to assist in the simplification of many machines and systems. Integrating with the mechanics, the electrical and electronic systems also became the subject of closer examination and thus eventual Certification of Conformity (CofC) also became compulsory for Safety Related and High Reliability Systems. Then machines and electronics began to become programmable, leading to the creation of seemingly quite clever machines. Thus was born software. However, some early programmable machines led to the occurrence of disasters.

Software

When software came along, the early implementers were highly skilled mathematicians. However, the number of people creating software grew rapidly and some of the new software writers were neither mathematicians nor engineers. This led to some sloppy code and systems that would fail regularly. Only recently have some software engineers become interested in creating software that truly is delivered by robust engineering methods. Component Oriented Software is one such engineering methodology.

However, systems are not just software. Everything that makes up that system, including the statement of requirements, the design documents, the operating manuals as well as all the nuts, bolts and logic components are a part of the overall system. Rather than different methods for developing the hardware and software a unified development approach that all involved in the process understand fully becomes the basis for managing a “Component Oriented System Devlopment”.

Development Process

If you want your systems to have real integrity it is necessary to be very adept in the early development of accurate requirements, that are testable for compliance with the clients real needs, before you launch into writing the design specification. Also, remember that prototyping is only meant to be a tool for exploring the detailed requirements space in order to hone the thinking at that level. The two best models of development still tend to be either the "V" model or Barry Boehm's Spiral Model (applied properly and with the early spins being directed solely at the elicitation of the full and correct requirements). The V model has to be applied as a two-prong approach to initially developing requirements and the System Acceptance Test Cases that prove the requirements have been fully met. Thus you will be developing the final acceptance test clauses in parallel with determining what your system needs to do. It is therefore imperative to finding the right descriptive language in which to express the requirements and allow the test cases to be written (even to the extent of handling the un-expected conditions). Tests have to not only cover the acceptance of the component unit but build to the acceptance of the overall system.

Naturally, to have a robust development process, properly followed such that it meets at least the SEI CMM level 3 **[SEI]**, is paramount to providing systems with certification that will be believed. The organisation needs to properly support the development process through assignment of appropriate roles and responsibilities to suitably qualified and experienced personnel. This means that all the management chain has bought in to such development processes and its proper operation. There needs to be enough discipline in the process to succeed and the application of the process needs to provide sufficient data of its correct application. Thus robust configuration management will be required to form the basis of this development process **[Kelly]**.

Finally, for this section, the documentation you produce has to be fit for the purpose of describing the components and the system to which they will belong. There is a world of difference between good and bad documentation **[Montforton].** You don't want too much or too little documentation, but sufficient **[Koopman2]**.

Wernher von Braun once said "Research is what I am doing when I do not know what I am doing". So, to say you are involved in R&D must mean that you are on a voyage to discover what your client truly needs in a way that he can sign his full agreement with your proposals. You might spend more than 60% of project time getting to the stage that the requirements become complete and testable, but, in the long run it is often much faster and yields better quality of product than to leave testing as an afterthought. However, once you have discovered all the aspects of the system and can complete the full requirements specification, including the safety, security, environmental and aesthetic aspects, then the easier it becomes to establish the best components to utilise within the system.

C.O.D.E

As stated above, the hardware world has managed to accomplish designs with some of the highest integrity, engendering a high level of trust in such systems. We need such in the software world and some attempts were made at the network level with schemes like .NET and CORBA. These are quite large and complex components in general that have to be built to be host system agnostic. However, in control systems, such a networking level is too high a consideration in resource constrained small controllers prevalent in industrial controls (eg. field sensors and devices).

A Component Oriented Development Environment allows developers to independently create and test individual components. In software the author proposes Forth is such an ideal software Component Oriented Development Environment (C.O.D.E) that it becomes easy to develop a library of software components that can be created, tested and fully certified to similar standards as individual components in the hardware world. A Certificate of Conformity for each and every Forth word that has almost mathematical certainty about its behaviour.

Why Forth?

This question has been asked very often and a number of very varied responses have been offered in answer. Why certification works with Forth, though, is down to the underlying Virtual Machine Model (VMM) that is the central embodiment of Forth. Despite, over the years, many Forth language standards being created, the basic underlying VMM is the same as that created by Charles Moore. With this VMM, the foundation on which we build Forth software components has been stable since its creation and provides a very good platform on which to build, no matter the underlying processor architecture, register provision or memory space available. The Forth Virtual Machine is compact and easily implemented in incredibly small spaces (~512 bytes to 8kbytes), either fully on the target processor or as an interactive umbilical connection (not an option available with many other environments). With this VM as the basis it becomes almost trivial, on a word by word basis, to conduct compliance testing on each and every Forth word you use and create to support your application.

Most Functional Safety Standards require Certified Compilers and, in other languages, these become very expensive, hard to wield and the outcome is not fully certain despite the claims. However, the key is that it is a results based confirmation with a tough inspection and testing regime applied is just as valid and allowed by IEC61508 for Assembler Code. With Forth being, probably, the best macro assembler in the world, it becomes easy to see the way in which to progress. Using Forth alone does not guarantee that a product will be safe. Its use has to be coupled with a relevent and highly capable Design and Development Process that features full version control and configuration management capabilities.

The Basis of Certification

Once you have determined your approach to constructing your system to meet the requirements, you need to look at the practice of construction for a robust and certified system. You need a few standards in place to accomplish the task. I have mentioned IEC61508 but that is the over-arching Functional Safety Standard for Electrical, Electronic and Programmable Electronic Systems. It is fairly agnostic about software implementation language. In addition to this you should have a couple of other documents within your development organisation (probably better styled as guides). Some years ago I published the Forth Coding Standard as a public domain document. From this start I know that a few companies have adopted and adapted it to great benefit in making the Forth Source Code more readable. It promotes a more literate style of presenting the source code.

Adopting and following such style guidelines should also help in automating some of the more tedious tasks associated with providing a good quality documentation of the installed system. Tools such as DocGen can help here. Additionally, we have the language standards, such as Forth200X, that we can conform to in order to aid portability of code and/or programmers between Forth using organisations. Standards also aid in wider collaborations.

In Forth Certification the elements of interest are:-

- **The words name** which is a reasonably good identifier
- **The Glossary Text** which becomes the words functional specification of performance.
- **The Stack Comments** which details the input and output parameters
- **The Word Make-up** which is the words that this word uses in order to perform its intended function.

To certify that the word performs its allotted functionality and has no side effects we take the above four items and perform the following:

- **Static Inspection** is a Fagan Style examination to ensure that the apparent intent of the requirements as stated in the glossary text are implemented correctly. Such inspection will also require that the words used in the make-up of this word have been applied correctly. Earlier words should also have been certified in a similar manner.
- **Functional Test** to ensure the word actually compiles and performs as specified. Such testing should explore all pathways of logic within the word.
- **Limits Test** to explore the unwanted side effects a word may have and to ensure that such limitations have been properly documented (in the glossary text). This will ensure that only the right amount of data is pulled from the stack, the stack does not underflow and all logic paths have been exhaustively tested to ensure no adverse behaviours exist.

As most Forth systems are built bottom-up, the above becomes just an extension of the normal test as created philosophy. However, for certification the coder should pass code, he is satisfied with, to others to perform the above three processes. That, though, is just to maintain some independence between the coding stage and the certification steps. Many of the safety standards demand such independence.

Benefits of a Component Oriented Approach in Forth

Once the component has been coded and certified it may be submitted to a library repository so that it is available for re-use in other projects. If the code is stored as a package along with its certification documentation then the whole library could be considered as tried and tested code that could be used anywhere else so long as its provisions matched the requirements of the new use. Why should Certified Forth code not be as moveable and re-usable as say, an M25 nut fitting onto any other M25 bolt. Such mobility and re-use of precoded components will ease the creation of much larger certifiable systems in the future. The level of documentation for a certified component is higher than for a non-certified component due to fully accounting for each components limitations. This is, though, much like documenting the Maximum Permitted Voltages on an electronic component.

"We have witnessed hosts of microprocessors and microcomputers marching from cradle to grave, right before our eyes. Languages and operating systems come and go. Even in Forth, which I use to code for a living and write about to entertain, we've seen good work done and disappear, come and go. Have we seen the best yet?" **[C.H.Ting]**

1. I think we might be about to, and you could and should make that happen.

References & Notes:-

[Koopman1] "The Grand Challenge of Embedded System Dependability" by Koopman P. given in a panel session at Dependable Systems and Networks 29th June 2011. <http://users.ece.cmu.edu/~koopman/pubs/koopman11_embedded_dependability_challenges.pdf>

[Koopman2] "Better Embedded Software Systems" by Philip Koopman, ISBN 978-0-9844490-2

[Monforton] "Good vs. poor documentation: Don't be 'that guy'" by Jeff Monforton 17th December 2013.

[SEI] "CMMI Distilled" by Dennis M. Ahern, Aaron Clouse & Richard Turner ISBN 0-321-18613-3

[Kelly] **"**Configuration Management: The changing Image**"** by Marion Kelly ISBN 0-07-70977-9

[C.H.Ting] "Footsteps in an Empty Valley" by C. H. Ting publishd by Offette Enterprises 1985.

HiTex

LaTeX gets a helping hand from Forth

Bill Stoddart

September 17, 2014

Abstract

HiTeX is a simple LaTeX pre-processor that works through token replacement. It provides improved readability of mathematical text in a source document by allowing free use of Unicode characters and eliminating any need for specific spacing and new line commands. HiTeX gains considerable power from the ability to incorporate sections of Forth text within a document. Output generated by Forth can be directed to the output file, or can be used to define place-holders which, when used within maths mode in a HiTeX document, will be replaced by the result of the corresponding computation.

Keywords: LaTeX, Unicode, Computable Document, RVM-Forth

1 Introduction

LaTeX is a versatile type setting system that gives excellent results on both mathematical and normal text. However, the mathematical markup is not always easy to read *as mathematics.* The advent of Unicode should have improved this, allowing us to write, for example, $\sqrt{}\alpha$ instead of the standard latex markup `\sqrt\alpha`. However, Unicode and its utf8 encoding have only partially been adopted by the LaTeX community, with the promising ucs package left unmaintained and unfinished. The projects XeLaTex and LauTeX are complete reworkings of TeX and Latex which are based from the outset on Unicode utf8 input. Our research group produced some papers in XeLaTex, but it was not a happy experience. One problem is that journal editors and submission portals may not accept documents written with these tools. We also had a problem with Greek characters, due to the fact that a font suitable for publishing an article written in Demotic Greek will not be suitable for providing the Greek letters used in mathematics. Also, we felt that the availability of Unicode should make the markup language sufficiently compact that it would be possible to revise the LaTeX practice of ignoring white space and requiring specific markups for additional space and new lines. We wanted a markup language where spaces and newlines would, by default, be taken into account in the final markup.

It also seemed to us to us that, rather than completely rewrite TeX and LaTeX, which are absolutely brilliant as they are, it would be better to write a simple pre-processor to translate a Unicode mathematical language into classical LaTeX. The result is HiTex. The last page of this document gives an example of HiteX markup and the resulting output.

HiTex is a hybrid of Forth and Latex which has its own variant of the LaTeX mathematical markup language. A HiTeX document contains 3 types of text. Initially, it is in pass-through mode, in which text is just streamed from the input file to the output file. All the HiTeX interpreter is doing at this time is checking for tokens that will take it either into a mathematical mode or into Forth.

Within a mathematical mode, HiTeX performs token replacements, recognising tokens in the HiTeX source, and replacing them by a corresponding token in the LaTeX output file. A token can be any sequence of characters. Some of the tokens are Unicode characters, such as ∀, ∃, *dot* etc, which are replaced by their corresponding LaTeX markups, `\forall, \exists, \bullet`. However, a token can also be something like a new line character, a space, or a sequence of spaces. Where one token is the prefix of another (for example a token consisting of two spaces would be a prefix of a token consisting of three spaces) the longer token is matched first. This ensures a correct match for all tokens.

HiTeX is implemented in RVM-Forth and uses Frank Zeyda'a set package, (see EuroForth 2002 proceedings), which supports arbitrary finite homogeneous sets. We use ascii zero format strings.

The corresponding pairs of tokens used by HiTeX are held in the set `LaTeX-MARKUP`. Here is the beginning of its definition:

```
STRING  STRING  PAIR  { " ∀"   \forall"  ↦ ,

                        " ∃"   " \exist " ↦ ,  ...
```

Text before the opening brace gives the type information required to construct the set. The set consists of pairs of strings. The maplet operator $\mapsto$ combines two strings on the stack into an ordered pair of strings. The following comma compiles this element into the set. The set construction is terminated by a closing brace, at which point the set (i.e. a pointer to the data structure which represents the set) is left on the stack

Within a Forth section a user can add new markups using set union $\cup$ or remove markups using domain subtraction `<<|`.

2 Including computation results in a document, an integer maths example

A interesting case is where the token to be inserted in a document is produced by a Forth computation. To define a token that captures an integer result, we can use the defining word `n†`. Here is an example of its use.

```
1234 n† s
```

This defines a new dictionary entry `s` which, when executed, gives the address of an asciiz string containing the text “1234”. We adopt a naming convention that strings generated in this way that will subsequently be used as tokens will be given a name that *begins* with †. Words that create such tokens have names that end in †.

We look at a simple example where we add the values of two constants and display the original values and their sum in a document.

2.1 Source code of the supporting Forth section

In the following Forth section the definitions $\dagger\alpha$, $\dagger\beta$ and $\dagger\alpha{+}\beta$ will return asciiz strings containing the text “10”, “20” and “30” respectively. Let us suppose that these are the numeric strings that are to be placed in the LaTeX output in response to seeing $\dagger\alpha$, $\dagger\beta$ or $\dagger\alpha{+}\beta$ respectively in the HiTeX source document. The tokens are paired up in a set, which is combined with `LaTeX-MARKUP` using set union. The updates are disseminated to the requisite HiTeX data structures with the `CONFIG` command.

```
%FORTH
10 CONSTANT α  20 CONSTANT β

α n† †α  β n† †β  α β + n† †α+β

STRING STRING PROD { " †α" †α ↦ ,
" †β" †β  ↦ , " †α+β" †α+β ↦ , }

LaTeX-MARKUP ∪ to LaTeX-MARKUP CONFIG END
```

Here is how these tokens can be used in a HiTeX math environment, along with the result.

HiTex markup	**Final output**
`$α=†α,  β=†β,  α+β=†α+β$`	$\alpha = 10,\ \ \beta = 20,\ \ \alpha + \beta = 30$

3 A floating point example

Floating point results are captured in a similar way, but using the defining word `f†` to define the output tokens. After the first line of Forth code the definition †$\sqrt{2}$ returns the address of an asciiz string representing $\sqrt{2}$ to 6 decimal places (our default output precision).

3.1 The supporting Forth section

```
%FORTH 2. FSQRT f† †√2   3. 2. F/ FSQRT f† †√(3/2)

STRING STRING PROD { " †√2" †√2 ↦ ,
" †√(3/2)" †√(3/2) ↦ ,  " √" " \sqrt " ↦ , }

LaTeX-MARKUP ∪ to LaTeX-MARKUP CONFIG END
```

And here is an example of HiTeX markup and the resulting final output.

HiTex markup **Final output**

```
\[
√2=†√2
√«3/2»=†√(3/2)
\]
```

$$\sqrt{2} = 1.41421$$
$$\sqrt{3/2} = 1.22474x$$

4 Configuraton tasks

A Forth section can be used for general configuration tasks, both of the HiTeX application and of the underlying Forth system.

In the example above, French «guillemets »were used as HiTex scope delimiters. These are preferred to the standard tex/latex delimiters `{` and `}`, as we use the latter as set brackets, and consider them to be essential mathematical symbols.

HiTeX holds its scope delimiters in the `VALUE`s `{SCOPE` and `SCOPE}` .

The following Forth section shows how we change these delimiters to Unicode bold brackets.

We also change the precision of the floating point output, recalculate the string produced by printing $\sqrt{3/2}$, update our markups, and reconfigure.

4.1 The supporting Forth section

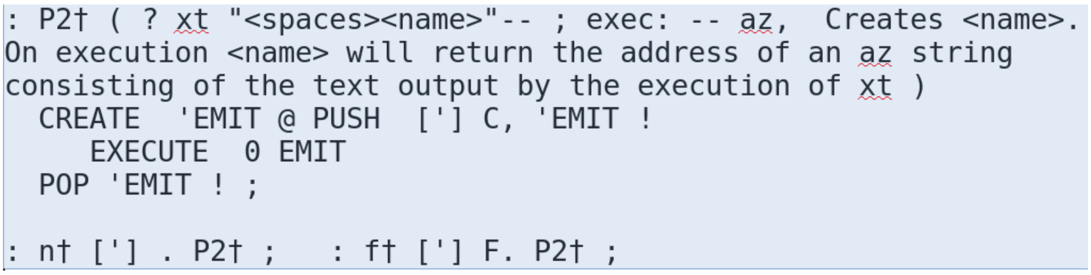
```
%FORTH  " ❨"  to  {SCOPE     " ❩"  to  SCOPE}

( Regenerate the result for √(3/2) at higher precision)
8  SET-PRECISION   3.  2.  F/  FSQRT  f†  †√(3/2)

( remove the previous entry for the placeholder " †√(3/2)")
STRING {  " †√(3/2)"  , }  LaTeX-MARKUP  <<|

( Add the new entry )
STRING STRING PROD { " †√(3/2)"   †√(3/2)  ↦  ,  }   ∪
to  LaTeX-MARKUP  CONFIG  END
```

Now our markup for $\sqrt{3/2}$ and the corresponding output are as follows

HiTex markup	**Final output**
`\[ √❨3/2❩ = †√(3/2) \]`	$\sqrt{3/2} \; = \; 1.2247449$

5 Implementation note 1

The defining words `n†` and `f†` have a lot in common, and both are defined in terms of a more primitive word `P2†` as follows:



```
: P2† ( ? xt "<spaces><name>"-- ; exec: -- az,  Creates <name>.
On execution <name> will return the address of an az string
consisting of the text output by the execution of xt )
  CREATE  'EMIT @ PUSH  ['] C, 'EMIT !
     EXECUTE  0 EMIT
  POP 'EMIT ! ;

: n† ['] . P2† ;   : f† ['] F. P2† ;
```

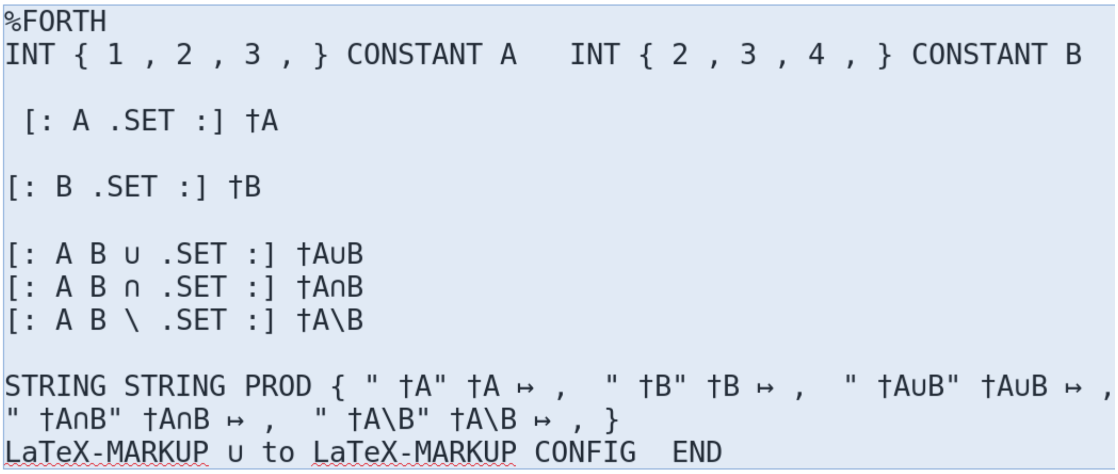
`P2†` takes an execution token fromthe stack, plus whatever extra parameters are required for the token's execution. It `CREATE`s a new dictionary entry and vectors `EMIT` to compile its output into the dictionary. It executes xt, and restores `EMIT`

6 A more general example

.

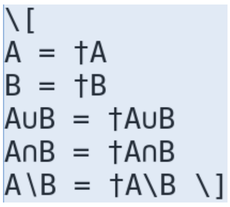
We provide for an arbitrary section of Forth source code to produce output, which we assume will be in the HiTeX markup format, rather than in Latex. This output must therefore be processed by the HiTeX maths pre-processor before being inserted in the LaTeX document. This is done with the pair of words `[: ... :]`. For example, suppose `A .SET` gives the output `{1,2,3}` This is not suitable to be immediately passed into the output document, since LaTeX will not see the braces as set delimiters, but as scope delimiters, and they will not appear on the final output. The phrase `[: A .SET :] †A` creates the Forth word `†A` which returns the address of the string obtained by passing the text output by the Forth between `[:` and `:]` through the HiTeX math pre-processor. Thus this defines `†A` as the string `" \{1,2,3\}"`, which is the correct LaTeX markup for the value of set `A`.

6.1 The supporting Forth section



```
%FORTH
INT { 1 , 2 , 3 , } CONSTANT A   INT { 2 , 3 , 4 , } CONSTANT B

 [: A .SET :] †A

[: B .SET :] †B

[: A B ∪ .SET :] †A∪B
[: A B ∩ .SET :] †A∩B
[: A B \ .SET :] †A\B

STRING STRING PROD { " †A" †A ↦ ,  " †B" †B ↦ ,  " †A∪B" †A∪B ↦ ,
" †A∩B" †A∩B ↦ ,  " †A\B" †A\B ↦ , }
LaTeX-MARKUP ∪ to LaTeX-MARKUP CONFIG  END
```

HiTeX markup



```
\[
A = †A
B = †B
A∪B = †A∪B
A∩B = †A∩B
A\B = †A\B \]
```

Final output

$$\begin{aligned} A &= \{1,2,3\} \\ B &= \{2,3,4\} \\ A\cup B &= \{1,2,3,4\} \\ A\cap B &= \{2,3\} \\ A\setminus B &= \{1\} \end{aligned}$$

7 Implementation note 2

HiTeX reads a source file into an input buffer, and places its LaTeX output in an output buffer. An asciiz string computed within a Forth section, and whose address is on the top of the stack, can be sent directly to the output buffer with the phrase:

```
DUP AZLENGTH TO-OUTBUFF
```

The outermost HiTeX interpreter passes text from the input buffer to the output buffer until it encounters a token that causes it to enter either Math mode, or Forth. The mathmode interpreter checks at each point in the input buffer whether the following characters match one its tokens. These tokens are those from the domain of `LaTeX-MARKUP` plus other tokens that require special action. If the token is from the domain of `LaTeX-MARKUP` the corresponding token from the range of LaTeX-MARKUP is added to the output buffer. Other tokens are special cases which require additional action. For example, a new line character in the input buffere requires a line count to be incremented, and the new line itself plus the LaTeX markup for a newline must be passed to the output buffer.

The input and output buffers are managed by a collection of VALUEs holding buffer start addresses, pointers to the current position in each buffer, etc. When text generated within a Forth section is to be processed by the HiTeX maths pre-processor, e.g. when using a [: . :]. structure, these buffer management values are saved, and the pointers etc are set to work from temporary buffers. After the text is processed, the resulting LaTeX markup is compiled into the dictionary and the temporary buffers are free for future use.

We return to the point of distinguishing between tokens such as †A and †A+B. The first of these tokens matches the start of the second, i.e. the first token is a prefix of the second. How do we ensure that †A+B won't be mistaken for †A?

We do this by searching for tokens in the same order as they occur in a sequence. We place our tokens in a sequence in such a way that any token that has prefixes that are also tokens will occur before its prefixes in the sequence. Reverse lexical order will achieve this.

The properties of our set implementation and the reversible features of RVM-Forth make this simple to implement. Every set is held as an ordered set, and the CHOICE operator selects the maximal element of each set, or if invoked after backtracking will select the maximal element not yet chosen.

For strings the ordering is lexical. Thus “†A” comes before “†A+B’’

We can create a sequence in which tokens in the domain of LaTeX-MARKUP occur in reverse lexical order using the following code:

LaTeX-MARKUP DOM SET2SEQ

Where the definition of SET2SEQ is:

```
: SET2SEQ ( x:P(X) -- y:seq(X), ran(y)=x )  (: set :)
   set [ <RUN set CHOICE RUN> ] ;
```

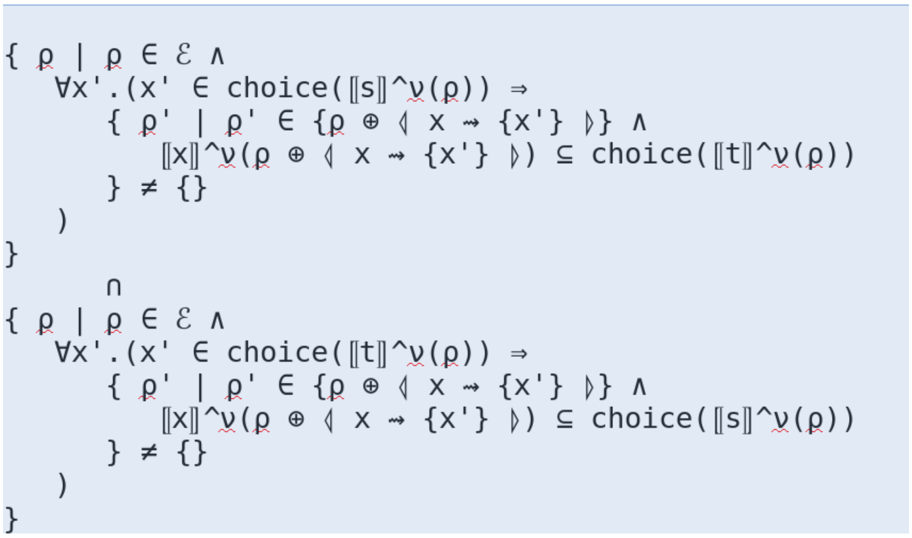
In this code the square brackets enclose a sequence construction. (They are not the Forth Standard square brackets). The set before the open square bracket provides type information. The code bracketed by `<RUN ... RUN>` chooses an element of `set` and compiles it as the next element sequence. Execution then reverses back to `CHOICE`, which makes a different choice if one is available, and this is then added to the sequence. This is repeated until no further choices are available, at which point execution continues beyond `]` The result is a sequence of strings held in reverse lexical order. This code is based on the premise that our sets are ordered; we know how but we can't control how. But the order of elements in a sequence is entirely under programmer control.

8 Conclusions and Future Work

HiTeX has been very valuable to us for writing dense mathematical documents. Its main limitation is that it does not support a verbatim mode which accepts Unicode - that's why we have used screen shots for the most of the Forth source code and HiTeX markup examples in this document.

Appendices

A HiTeX markup example



```
{ ρ | ρ ∈ ℰ ∧
   ∀x'.(x' ∈ choice(⟦s⟧^ν(ρ)) ⇒
      { ρ' | ρ' ∈ {ρ ⊕ ⦇ x ⇝ {x'} ⦈} ∧
         ⟦x⟧^ν(ρ ⊕ ⦇ x ⇝ {x'} ⦈) ⊆ choice(⟦t⟧^ν(ρ))
      } ≠ {}
   )
}
      ∩
{ ρ | ρ ∈ ℰ ∧
   ∀x'.(x' ∈ choice(⟦t⟧^ν(ρ)) ⇒
      { ρ' | ρ' ∈ {ρ ⊕ ⦇ x ⇝ {x'} ⦈} ∧
         ⟦x⟧^ν(ρ ⊕ ⦇ x ⇝ {x'} ⦈) ⊆ choice(⟦s⟧^ν(ρ))
      } ≠ {}
   )
}
```

$$\begin{array}{l}
\{\rho \mid \rho \in \mathcal{E} \wedge \\
\quad \forall x'.(x' \in \mathit{choice}(\llbracket s \rrbracket^{\nu}(\rho)) \Rightarrow \\
\qquad \{\rho' \mid \rho' \in \{\rho \oplus (\!| x \rightsquigarrow \{x'\} |\!)\} \wedge \\
\qquad\quad \llbracket x \rrbracket^{\nu}(\rho \oplus (\!| x \rightsquigarrow \{x'\} |\!)) \subseteq \mathit{choice}(\llbracket t \rrbracket^{\nu}(\rho)) \\
\qquad \} \neq \{\} \\
\quad) \\
\} \\
\qquad \cap \\
\{\rho \mid \rho \in \mathcal{E} \wedge \\
\quad \forall x'.(x' \in \mathit{choice}(\llbracket t \rrbracket^{\nu}(\rho)) \Rightarrow \\
\qquad \{\rho' \mid \rho' \in \{\rho \oplus (\!| x \rightsquigarrow \{x'\} |\!)\} \wedge \\
\qquad\quad \llbracket x \rrbracket^{\nu}(\rho \oplus (\!| x \rightsquigarrow \{x'\} |\!)) \subseteq \mathit{choice}(\llbracket s \rrbracket^{\nu}(\rho)) \\
\qquad \} \neq \{\} \\
\quad) \\
\}
\end{array}$$

Region-based Memory Allocation in Forth

M. Anton Ertl*
TU Wien

Abstract

Memory management has a pervasive effect on the way we program. In region-based memory allocation, objects with roughly the same life expectancy are allocated in one region, and in the end the whole region is freed at once. This avoids the need to keep track of the individual objects for `free`. Regions are simple to implement and compatible with real-time requirements and multi-threading, and seem to be ideal for Forth, except for one thing: The region id has to be passed to the allocation word, increasing the stack load. We propose using context wrappers to avoid that problem. This even allows to use existing `allocate`-based libraries with regions, but we then have to decide what `free` and `resize` inside these libraries do.

1 Introduction

The way that memory is allocated and deallocated has far-ranging consequences on program design.

For example, consider a string concatenation word. If you can allocate memory at will, and don't have to worry about deallocation (e.g., because you work on a garbage-collected system), you might use an interface like

```
astr+ ( c-addr1 u1 c-addr2 u2
        -- c-addr3 u3 )
```

By contrast, if memory is allocated once and for all ("static allocation"), you might go for an interface like

```
bstr+ ( c-addr1 u1 c-addr2 u2 c-addr3 u3
        -- c-addr3 u4 n)
```

(inspired by the Forth-2012 word `substitute`). `Bstr+` writes the resulting string in the buffer `c-addr3 u3`, with the length of the resulting string in `u4`, and `n` indicating whether the operation was successful (had enough buffer space).

If you need to free explicitly, you can use either interface, but if you use `astr+`, you have to keep track of c-addr3 and free it when you are done.

The usage of these words varies depending on how memory is allocated. E.g., consider wanting to build a file path from a directory name `dir ( -- c-addr u)` and a file name `file ( -- c-addr u )` and then using that file path for opening a file:

```
\ astr+ with garbage collection
dir s" /" astr+ file astr+ r/o open-file throw

\ astr+ with allocate/free
dir s" /" file astr+ over >r astr+ r> free throw
over >r r/o open-file throw r> free throw

\ bstr+ with preallocated buffers:
create buf1 200 chars allot
create buf2 200 chars allot
dir s" /" buf1 200 bstr+ 0< abort" buf1 short"
file buf2 200 bstr+ 0< abort" buf2 short"
r/o open-file throw
```

Garbage collection makes such things easy, and may be the decisive feature for distinguishing high-level languages from lower-level languages, but it seems like it does not quite fit Forth: Its implementation is complex, in particular in combination with lack of type information (a fundamental property of Forth), real-time requirements (relevant in significant numbers of Forth applications), and multiprocessing (becoming more and more important with the spread of multi-core CPUs). Nevertheless, there has been a garbage collection library for Forth available since 1999[1]; however, this library does not satisfy real-time requirements and is not designed for multiprocessing.

The Forth standard supports `allocate` and `free` (and `resize`) in the memory allocation wordset since Forth-94 (*heap allocation*). Unfortunately, this interface is cumbersome and error-prone:

- If you `free` too early, the system may allocate the memory for some other use and if you then try to access the (already-freed) object, you get the wrong data or change data in the new, unrelated object (*dangling reference*).
- If you fail to keep track of all allocations, you fail to `free` some, and you get a *memory leak*.[2]

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

[1] `http://www.complang.tuwien.ac.at/forth/garbage-collection.zip`

[2] Note that freeing everything just before leaving the sys-

There are various techniques to avoid these problems, but they tend to restrict the way you program, and they may cost performance; e.g., in the extreme you can make a new copy of the object every time you copy the address, and then you can be sure that you can free the object when you consume that address (because every object has only one live address) [Bak94], but all that allocating, copying, and freeing costs performance; also, this technique does not work for mutable objects.

This paper discusses region-based memory allocation, a technique in between `free` and garbage collection that might be a good fit for Forth. It describes what region-based memory allocation is (Section 2), presents Forth words for regions (Section 3), discusses how `allocate`/`free`/`resize` code can be used with regions (Section 4), outlines and implementation (Section 5) and discusses related work (Section 6).

2 Region-Based Memory Allocation

With region-based memory allocation, you can have several regions active at the same time. You allocate memory from one of these regions. When you no longer need any of the memory in a region, you free the region.

The way regions are typically used is: As application programmer you know that a bunch of things are guaranteed not to be needed beyond a certain point, so you introduce a region for these things, and allocate memory for these things from this region. In between, you can allocate things from longer-lived or shorter-lived regions. Typical examples for this kind of pattern are:

- A web server typically has a lot of things that don't survive the HTTP request. These things could be allocated in a region that is freed when servicing the request is completed.
- A compiler could have regions for the basic block (straight-line code sequences), and the definition. As soon as it is done with one basic block, it frees the basic block region and starts a new basic block region for the next basic block. Likewise for definitions.
- A text formatting program could have regions for a line, a paragraph, a page, a section, and the whole document.

Regions give programmers a wide range of control over memory management. E.g., you could start out with few regions (e.g., in the compiler only have regions for definitions); when you notice that this consumes more memory than you want, you can introduce additional regions for more fine-grained control (but with the potential for more bugs).

Regions are relatively easy to implement (about the same difficulty as `allocate/free`), even in the presence of real-time requirements and multiprocessing. So they appear to be a good fit for Forth. Why have they not caught on?

3 Forth interface for regions

A straightforward region interface works with region IDs passed on the stack:

```
new-region ( -- region-id )
region-alloc ( usize region-id -- addr )
free-region ( region-id -- )
```

The disadvantage of this kind of interface is that it requires passing the region-id around. E.g., for our string concatenation example, we would have a word

```
cstr+ ( c-addr1 u1 c-addr2 u2 region-id
        -- c-addr3 u3 )
```

The region-id would have to be passed around on the stack inside `cstr+`, and we would have to pass the region-id to `cstr+`. For our file path example this could look as follows:

```
new-region >r
dir s" /" r@ cstr+ file r@ cstr+
r/o open-file throw
r> free-region
```

This works passably in this case, but we consumed the top-of-return-stack for the region-id, and cannot use it for something else anymore. In any case, this kind of region interface increases the stack load by one item.

This has deterred me from using regions for a long time, but recently I have thought about how to use stack load reduction techniques [Ert11] to avoid this problem. I settled for using context wrappers, because this allows writing general-purpose words. `Region-alloc` is split into:

```
ralloc ( usize -- addr )
with-region ( ... region-id xt -- ... )
\ xt: ( ... -- ... )
```

So you pass the region-id to `with-region`, which executes the xt, and while executing the xt, every `ralloc` allocates from region-id (unless it is executed in a nested `with-region` context).

Let's look at our string concatenation example again. We can now use the `astr+` interface instead fo `cstr+`:

tem is counterproductive; it may page in stuff that would just be freed (without paging) by the operating system as part of terminating the process.

```
new-region dup
[: dir s" /" astr+ file astr+
   r/o open-file throw ;] with-region
free-region
```

This example uses the syntax `[: ... ;]` for nestable unnamed definitions (quotations). The example is not shorter than the cstr+ one, but the return stack is now free for other uses (within the quotation).

But there is still a stack item passed from new-region to free-region. We can also have a wrapper that replaces these two words:

```
do-region ( ... xt -- ... )
\ xt stack effect: ( ... region-id -- ... )
```

With that, our example looks as follows:

```
[: [: dir s" /" astr+ file astr+
      r/o open-file throw
   ;] with-region
;] do-region
```

For cases like this example where do-region and with-region work together, we can also have

```
do-with-region ( ... -- ... )
\ xt stack effect: ( ... -- ... )
```

which combines the effects, resulting in:

```
[: dir s" /" astr+ file astr+
   r/o open-file throw ;] do-with-region
```

4 Allocate/free/resize

With the region passed implicitly, we can use an interface that is compatible to the standard word

```
allocate ( usize -- addr ior )
```

instead of ralloc. Indeed, we can even redefine allocate to allocate from the current region when called inside a with-region context. This allows to use words or libraries written for the standard memory-allocation wordset with regions.

To make this idea work, we also need to determine what free and resize should do when called inside a with-region context.

For free this is relatively straightforward: if the memory has been allocted from a region, free should not free anything (the memory will be freed when the region is freed); if the memory has been allocated from the heap, then free should perform the standard free.

Resize is more complicated. One can see it as allocating memory from the current region, and freeing the original memory as described above. However, that would not always reflect the intent of the programmer who wrote the resize, and may lead to too-early freeing.

So how is resize used in practice? In my experience resize is used in two ways:

- To simulate statically allocated buffers of unlimited size. The program first allocates a small buffer (or stores 0 as buffer address), and grows the buffer with resize when necessary. These buffers are never freed.
- For temporary growing structures. These structures are freed when the program no longer needs them.

Given that, one approach for dealing with resize is to always treat it as working on the heap. If the memory was first allocated from a region, the resize should be treated as allocating from the heap. People who want to write code for regions should not use resize.

One problem with these ideas is that it sometimes requires determining whether a piece of memory was allocated from the heap or from a region. Determining this can require quite a bit of code and can be slow (depending on the implementation of regions and the heap).

The following assumptions would get rid of this need:

- Resize only gets 0 or previously resized memory as a-addr1 parameter. With this assumption resize does not need to see if the memory was allocated from a region (it wasn't). Unfortunately, the standard does not specify that resize works for a-addr1=0 (Gforth does), so this assumption will not hold for standard programs that use resize.

  An alternative, less restrictive assumption is that the resized memory was allocated from the heap, but that would restrict the usage of with-region in combination with code that uses resize for temporary growing structures. To avoid programs that don't get this right, it would be useful to check this assumption, but that again requires determining whether memory was allocated from the heap or from a region.

  If this assumption is made, but does not hold (i.e., region-allocated memory is resized), the result is unpredictable and depends on the heap implementation.

  The other alternative is to assume that the memory is either from a region or previously resized. Then, if it is not previously resized, we just heap-allocate new memory, copy the old memory there, and do not free the old memory. If the old memory was actually heap-allocated, this will lead to a memory leak.

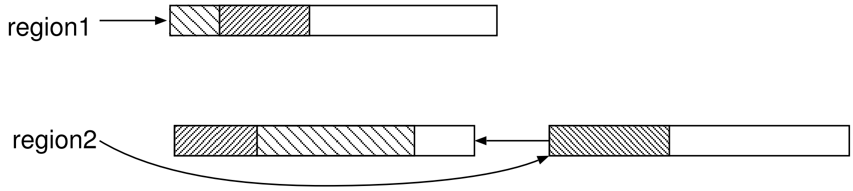


Figure 1: Implementation based on `allocate`

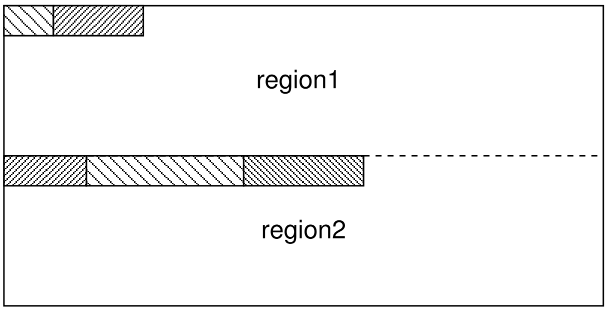


Figure 2: Implementation based on one big memory block

- `Free` within a region only refers to region-allocated memory, except possibly `resize`d memory. With this assumption, `free` needs to check only if memory is `resize`d, which is cheaper to check. Ideally `resize`d memory is always freed with a separate word, then we can do with a placebo `free` inside a region. If this assumption is made, but does not hold (i.e., heap-allocated memory should be `free`d in region context), there will be a memory leak.

It is unclear which of the various options in this design space is best. So it is probably best to use the simplest option at first, build in checking to make users aware of the restrictions, and ask users for feedback.

5 Implementation

This section sketches two implememtation approaches.

5.1 Based on `allocate`

Each region is represented by a linked list of blocks. Each block has a standard size (e.g., 16KB) and is `allocate`d. Within each block, there is a pointer to the first free byte, and a new allocation in the region is made there. If the rest of the block is too small for the allocation, a new block is started (see Fig. 1). If an allocation is bigger than the standard block, it gets its own private block of the appropriate size.

When a region is freed, the linked list is traversed and all the blocks in the linked list are `free`d. For real-time requirements, one could arrange to delay the freeing, such that only one block is freed per region allocation.

For checking whether an address is allocated with `resize`, one could have a simple array of resize addresses. If there are only few resize addresses at the same time, this is sufficient. A more scalable data structure (inspired by a sparse set representation [BT93]) would have an extra cell before the resized memory that points to the array; if this address points within the bounds of the array, and the place where it points to points back to the address we are looking at, the address has actually been allocated with `resize`.

For checking whether an address is allocated in a region or on the heap, we would have to walk all the blocks of all the heaps, and check whether the address is contained there.

The benefits of this kind of implementation over one that uses one `allocate` per `region-alloc` and links all the allocations together is less memory overhead for links, and less time overhead in allocation and deallocation.

5.2 Based on one big memory block

In an embeded system with full control over memory we may prefer to reserve one big block of memory for regions. Similarly, if we are working on a decent virtual memory system, we could mmap a big chunk of address space for regions (say, as big as the physical memory of the machine).

This implementation is based on buddy memory allocation. The first region starts out at the bottom of the big block. When starting another region, the block is divided into two parts (see Fig. 2). If the part of one region runs out of space, one can split the part of a region with more free space, and continue there.

When freeing a region, all the parts it has are freed, possibly regrowing parts of other regions.

Checking for `resize` addresses is the same as for the other implementation.

Checking whether an address is allocated in a region or on the heap is very easy: If the address is within the big block, it is in a region.

Overall this implementation approach is similar to the other one, but you implement the base memory allocator yourself (as buddy allocator) instead of using the system's `allocate`. The benefits are that you can use your knowledge of the base allocator's implementation to simplify some of the operations of the region allocator (e.g., checking whether something is in a region).

6 Related work

Region-based memory allocation is an old idea, that has appeared under different names: regions [GA98], arenas [Han90], pools (Apache), memory contexts (PostgreSQL), obstacks (glibc). "Region" is the name used in most recent papers and in Wikipedia[3].

Glibc's obstacks extend the usual capabilities of regions by allowing to grow allocations, and deallocate from an obstack in a stack-based way, i.e., a very dictionary-like behaviour, except that you can have several obstacks, and a growable object is not addressable while it is still growable.

The regions implementation based on `allocate` is the same as that described by Hanson [Han90], and as described in the obstacks documentation of glibc.

Gay and Aiken [GA98] evaluate regions empirically, and find that regions are either best or close to the best alternative in both run-time and memory consumption. They also propose and evaluate a safe version of this technique, based on reference counting (references into a whole region).

Because regions and their implementation are so simple, there is little academic literature on them themselves, but rather on more complex ideas like region inference, where the compiler tries to determine regions for allocations automatically.

Context wrappers are one of the techniques for reducing the stack load [Ert11]. They were inspired by Jenny Brien, who proposed a wrapper for dealing with the input stream on comp.lang.forth `<8s7mkl$4ql$1@news6.svr.pol.co.uk>`.

7 Conclusion

Region-based memory allocation offers a more convenient memory allocation model than `allocate`/`free`, while avoiding the problems of garbage collection: regions are much simpler to implement, especially in combination with multi-threading and real-time requirements.

So regions seem to be a good fit for Forth. However, they have not caught on yet, because they require passing the region id around, thus increasing the load on the stack. By using context-wrappers we can reduce this stack burden.

This opens up the possibility to use existing, `allocate`-using code with regions, often avoiding the need to keep track of each piece of allocated memory for `free`. But one then has to do something about the `free`s and `resize`s in this code. We have discussed this issue here, but are not sure what the best approach is.

References

[Bak94] Henry Baker. Linear logic and permutation stacks — the Forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.

[BT93] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, 1993.

[Ert11] M. Anton Ertl. Ways to reduce the stack depth. In *27th EuroForth Conference*, pages 36–41, 2011.

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[3] `http://en.wikipedia.org/wiki/Region-based_memory_management`

Doing C-style structs on cell addressed uCore

Klaus Schleisiek - kschleisiek@wauland.de

Last year, the technical high-school of Windisch (FHNW - Fachhochschule Nordwest-Schweiz) realized a uCore back end for LCC (Little C-Compiler). LCC was enhanced by Markus Knecht to become FCC (Forth C-Compiler) integrating advanced stack allocation techniques into the front end. This substantially reduced the number of local variables on the return stack and turns uCores dual stack architecture into a performant C engine.

Another problem with C is its fundamental byte orientation. I took this problem lightly for a long time proposing to declare a byte to be any number of bits as long as it is more than eight. Unfortunately, this way of looking at things does not help in the case of C at all: Unions may be defined to access a quadruple of bytes as one 32-bit integer.

Therefore, bytes need to be accessible within larger memory cells - of which only even multiples of eight make sense at all. So lets discuss a 32-bit word width architecture. Integer (32-bit **i@**, **i!**), word (16-bit **w@**, **w!**), and byte (8-bit **c@**, **c!**) accesses within a 32-bit cell are needed.

For fetches this is easy. **i@**, **w@** and **c@** can be realized in a single cycle, perhaps followed by the word **signed** that takes care of appropriate sign extension. Without signed, the most significant bits will be zero filled. Stores are more complicated requiring an un-interruptible dual cycle read-modify-write cycle. We fetch the appropriate 32-bit cell, modify the byte or word to be written and write the result back to the cell.

This leaves us with two more problems: 1) how to do byte/word addressing and 2) what to do when access happens to a "misaligned" address. The answer to 2) is classical: We raise an exception and execute a call to the "misaligned address trap" address. More on this later.

1) is more tricky and there have been two approaches to addressing bytes on cell based machines. In a 32-bit machine, we need two additional bits to locate a byte. We observe that this reduces the address space of the word addressed machine by a factor of four, which is not a real limitation on a 32-bit machine, and if it is, upgrade to 64-bits.

This leaves the question: Where do we put the additional address bits? The most intuitive solution is to shift the word address two times to the left and use the two new least significant bits for selecting a byte within a 32-bit cell. Byte address arithmetic is trivial - normal 2scomplement arithmetic will do. But unfortunately, under this approach a 32-bit integer address is a completely different number than a 32-bit cell address accessing the same memory cell.

Therefore, another solution turns out to be more efficient over all: The two additional bits are placed in the two most significant bits of a byte address. This way, **i@** as well as @ operating on the same numerical value will access the same memory cell as long as the two most significant bits are zero. But how do we do byte address computation? All we need is just one operator **byte+ (caddr n -- caddr')** that adds n, a signed number of bytes, to the byte or cell address on the stack. All the pains of doing weird arithmetic on a number whose least significant bits are kept in the two most significant bit positions are encapsulated in the **byte+** operator. On uCore, this is a single cycle instruction.

Now the last problem to be solved is the behaviour of the "misaligned address trap". A call to this trap temds to be the result of a software bug. Most of the time, we could just replaced the misaligned address by the nearest properly aligned address and the software will work as expected. Therefore, a basic misaligned trap handler should correct the address and re-execute the trapped memory access instruction. On the side, it can do statistics so the programmer is able to learn about his software bugs after the program executed.

These are the new words introduced:

i@ (caddr -- 32b)
fetches a 32-bit number from memory address caddr. If the two most significant bits of caddr are non-zero, the misaligned address trap will be called.

w@ (caddr -- 16b)
fetches a 16-bit number from memory address caddr. The 16 most significant bits of 16b will be zero. The most significant bit of caddr determines, which 16-bit section of the cell located at the equivalent cell address of caddr will be selected. If the second but most significant bit of is non-zero, the misaligned address trap will be called. As a side effect, the "word" status flag will be reset to zero.

c@ (caddr -- 8b)
fetches an 8-bit number from memory address caddr. The 24 most significant bits of 8b will be zero. The two most significant bits of caddr determine, which 8-bit section of the cell located at the equivalent cell address of caddr will be selected. As a side effect, the "word" status flag will be set to one.

signed (u -- n)
Depending on the state of the "word" status flag, u will be sign extended. If the "word" status flag is set, bit 7 of u will be copied into bits 8 to 31 of n. Otherwise, bit 15 of u will be copied into bits 16 to 31 of n.

i! (n caddr --)
stores n into the memory cell at caddr. If the two most significant bits of caddr are not zero, the misaligned address trap will be called.

w! (16b caddr --)
stores 16b into the memory cell at caddr. This is an uninterruptible read-modify-write cycle, because 16b has to be merged with the 32-bit content of the memory cell at caddr. If the most significant bit of caddr is set, 16b will be stored into bits 16 to 31 of the memory cell at caddr. Bits 0 to 15 will not be affected. If the second but most significant bit of caddr bit is non-zero, the misaligned address trap will be called.

c! (8b caddr --)
stores 8b into the memory cell at caddr. This is an uninterruptible read-modify-write cycle, because 8b has to be merged with the 32-bit content of the memory cell at caddr. The two most significant bits of caddr determine, which 8 bit section of the memory cell at caddr will be modified; the remaining bits will not be affected.

MSB setting	destination for 8b
00	bits 0 to 7
01	bits 8 to 15
10	bits 16 to 23
11	bits 24 to 31.

byte+ (caddr n -- caddr')
performs byte address arithmetic on caddr. This is different from standard +, because the two least significant bits of the byte address are located in the two most significant bits of caddr.

Forth - The Next Generation

Gerald Wodni

September 16, 2014

Abstract

To attract the next generation of Forth programmers, new tools are needed. The Forth Net should serve as a single point of entry to get them started.

1 Introduction

The Forth Net[1] is in the process of being changed to a meta-repository which can host an optional git repository for each project, but can also link to other repository websites like GitHub[2]. The main features remain to provide a single point of entry for Forth-related projects, declaring dependencies between projects, and the ability to specify additional tags for each project to find similar ones or specify groups.

To make the Forth Net attractive for new programmers I investigated the Node.js community.

2 Related Work

Node.js[3], a platform for running JavaScript outside the browser environment is one of the fastest growing communities on the web. To find out what the next generation of Forth programmers want and need, I investigated the community to identify its main pillars.

NPM, Node Packaged Modules[4] is the main repository for sharing JavaScript source. It has a small and easy to learn interface based on a simple file in each project and the NPM program itself. To use NPM for a new project one adds a package.json file, which specifies the dependencies. This file also contains project meta data like name and author, making the project itself a valid NPM package.

GitHub has no fancy website for each project, but just displays a README file different formats, most prominent ones are MarkDown or plain text. This makes the user interface required to setup a project description website even smaller.

3 Flink

Copying these features is not doing justice to Forth, I wanted to emphasize Forth's unique features like the interactive compiler interface. An emulated Forth System inside the browser is not of much use for serious projects, so the system is laid-out as follows:

Server A web server capable of handling WebSockets[5] used as a broker between the other parties.

Flink An interactive browser IDE, build as responsive website running on every major browser which supports HTML5 and WebSockets.

Uplink A tiny implementation of the WebSocket interface, which is only necessary until the target Forth system understands the Flink-WebSocket protocol. As the protocol is a work in progress, please consult the repository for the latest command set[6].

Flink consists of an interactive console[7] and an editor[8] which can load and save source code to the project's repository. Once the programmer is logged in, and has a target system attached via uplink, the console behaves like a line-buffered Forth. To compile the code from the editor window, it is transfered a line at a time waiting for the Forth's "ok" or an error messages. If the Uplink is connected directly to a system with no Internet access (i.e. over a serial line), Flink enables this device to a rich IDE and allows inclusion of other files and even projects.

4 Further Steps

Package Format A simple format for the Forth Net which provides similar functionality like NPM's package.json . An alternative would be to parse the forth source code for "finclude ...".

User Interface A HTML5 user interface which simplifies API to the Forth System drastically by having a full-blown GUI on the front end, and a simple text interface to Forth.

M2M Communication As Flink is based on the Websocket Protocol, it also works behind most firewalls and allows for remote machine maintenance as well as indirect machine to machine communication

References

[1] The Forth Net. URL http://theforth.net.

[2] GitHub. URL https://github.com.

[3] node.js. URL http://nodejs.org.

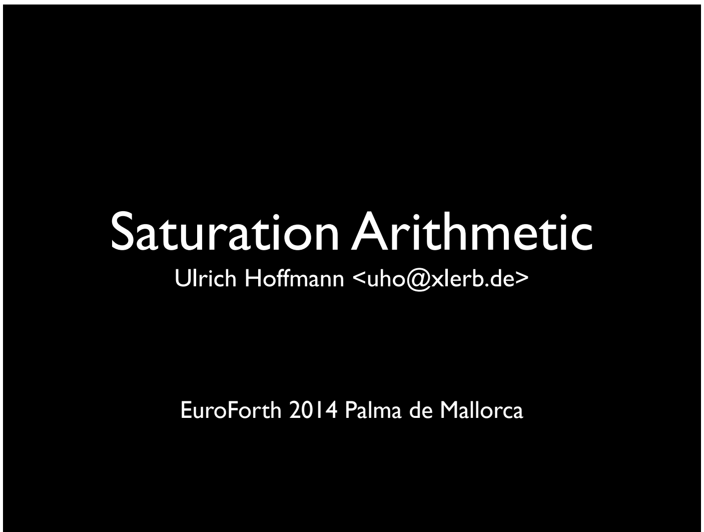
[4] NPM. URL https://www.npmjs.org.

[5] I. Fette and A. Melnikov. RFC6455 The WebSocket Protocol.

[6] Uplink. URL https://github.com/GeraldWodni/uplink.

[7] jq-console. URL https://github.com/replit/jq-console.

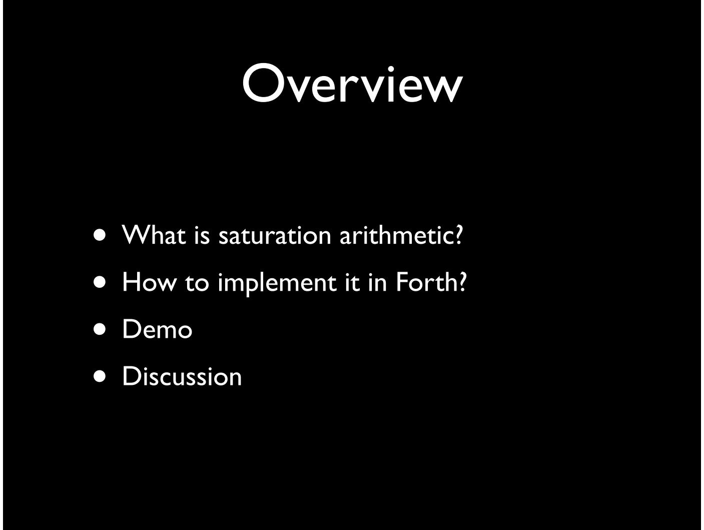
[8] Ace cloud 9 editor. URL http://ace.c9.io/.



Saturation Arithmetic

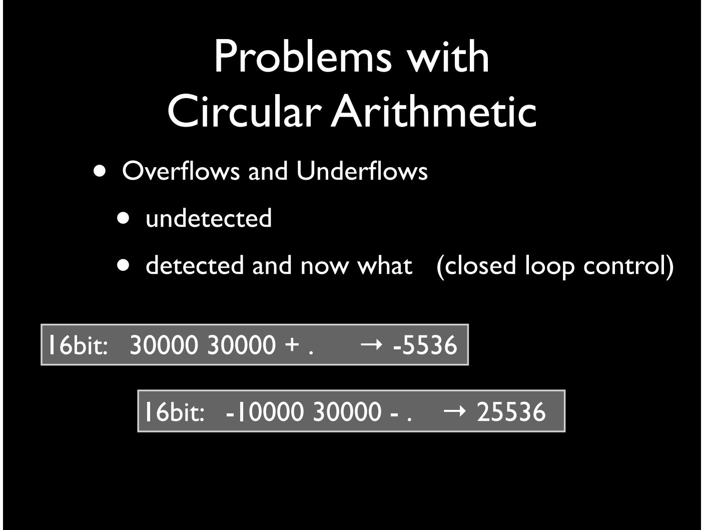
Ulrich Hoffmann <uho@xlerb.de>

EuroForth 2014 Palma de Mallorca



Overview

- What is saturation arithmetic?
- How to implement it in Forth?
- Demo
- Discussion

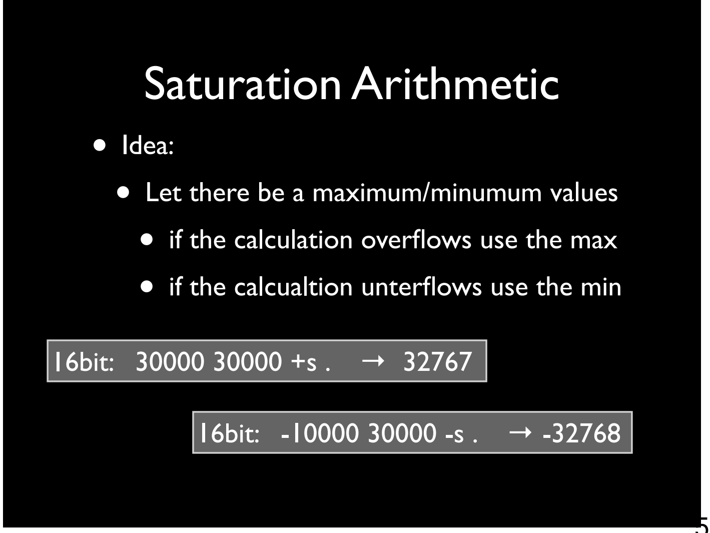


Problems with Circular Arithmetic

- Overflows and Underflows
  - undetected
  - detected and now what (closed loop control)

```
16bit:  30000 30000 + .    → -5536
```

```
16bit:  -10000 30000 - .   → 25536
```

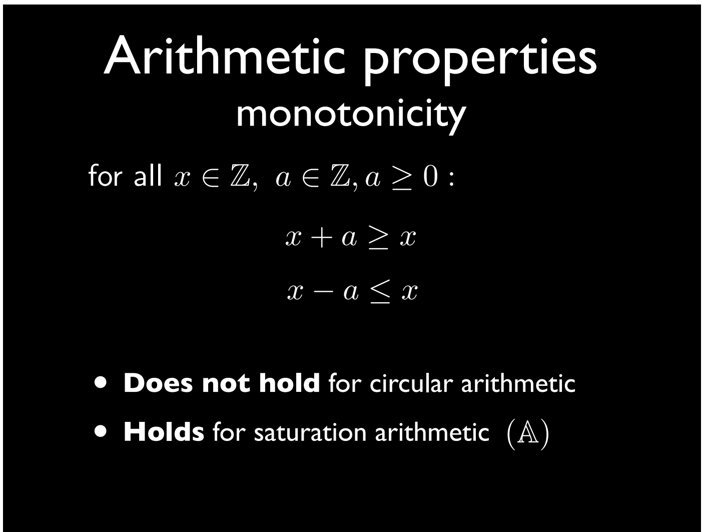


Saturation Arithmetic

- Idea:
  - Let there be a maximum/minumum values
    - if the calculation overflows use the max
    - if the calcualtion unterflows use the min

```
16bit:  30000 30000 +s .   →  32767
```

```
16bit:  -10000 30000 -s .   → -32768
```



Arithmetic properties

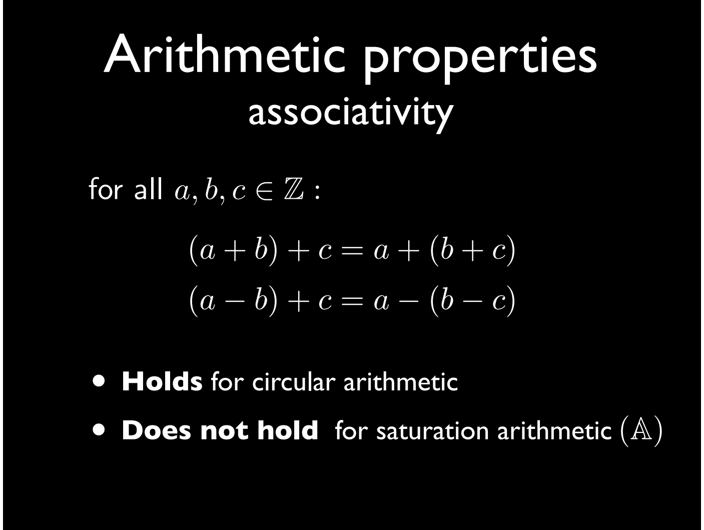
monotonicity

for all $x \in \mathbb{Z},\ a \in \mathbb{Z}, a \geq 0:$

$$x + a \geq x$$

$$x - a \leq x$$

- **Does not hold** for circular arithmetic
- **Holds** for saturation arithmetic $(\mathbb{A})$



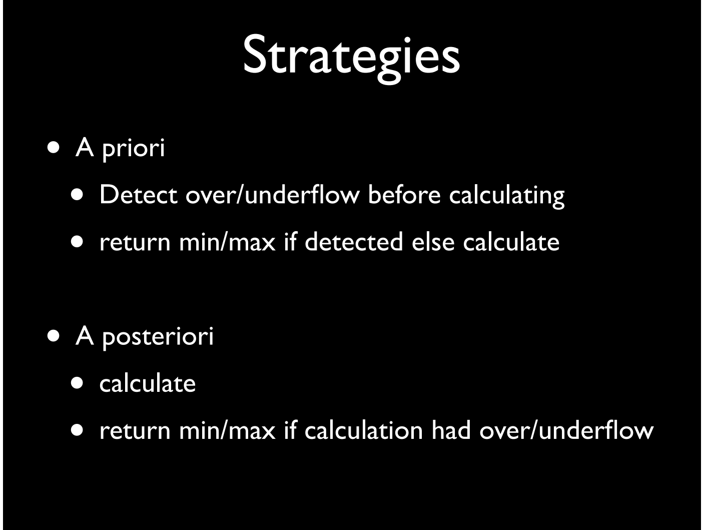
Arithmetic properties

associativity

for all $a, b, c \in \mathbb{Z}:$

$$(a + b) + c = a + (b + c)$$
$$(a - b) + c = a - (b - c)$$

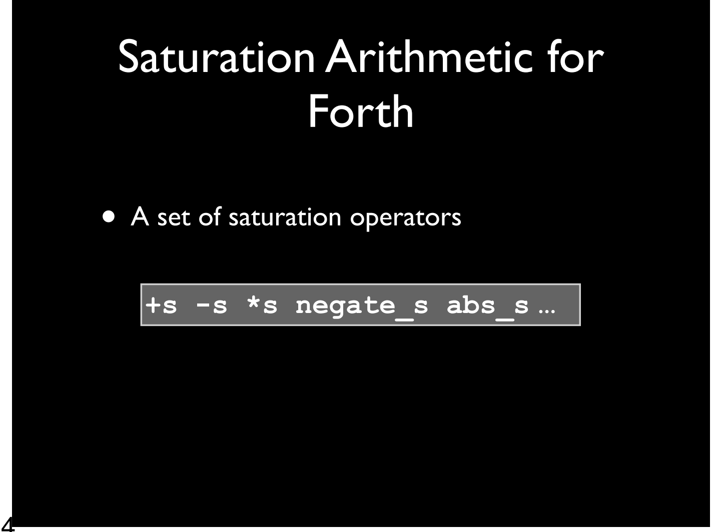
- **Holds** for circular arithmetic
- **Does not hold** for saturation arithmetic $(\mathbb{A})$



Strategies

- A priori
  - Detect over/underflow before calculating
  - return min/max if detected else calculate

- A posteriori
  - calculate
  - return min/max if calculation had over/underflow



Saturation Arithmetic for Forth

- A set of saturation operators

```
+s -s *s negate_s abs_s ...
```

What about unsigned numbers?

What about unsigned numbers?

- Another set of unsigned saturating operators?

```
16bit:   30000 30000 +us u.    → 60000
16bit:   40000 40000 +us u.  → 65535
16bit:   10000 30000 -us u.        → 0
```

Too many operators!

- Just two new words:

```
sat ( x -- x | max )      signed saturation
usat ( x -- x | umax )   unsigned saturation
```

- Let + – * set (internally) enough information so that `sat` and `usat` can work.

```
16bit:   30000 30000 + sat .     → 32767
16bit:   -10000 30000 - sat .    → -32768
16bit:   30000 30000 + usat u. →  60000
16bit:   40000 40000 + usat u. →  65535
16bit:   10000 30000 - usat u.  →       0
```

Has saturation happened?

- `usat` and `sat` set a flag **`usatq`** when saturation took place.
- Applications can check it to see if the results are exact.
- Applications must explicitly reset `usatq`.

Demo

Implementation

- 4e-Forth

```
;C +        n1/u1 n2/u2 -- n3/u3     add n1+n2
        HEADER  PLUS,1,'+',DOCODE
        ADD     @PSP+,TOS
        MOV     SR, &SRSAVE
        BIS     #1000h, &SRSAVE
        NEXT
```

- Implementation of – similar.

Implementation

- 4e-Forth

```
; SAT      x -- x
    HEADER  SAT,3,'SAT',DOCODE
            BIT     #100h,&SRSAVE     ; was overflow bit set?
            JZ      nosat
            BIT     #1h,&SRSAVE       ; check carry for over or underflow
            JZ      satovl
            MOV     #8000h,TOS
            jmp satsetq
satovl:     MOV     #7FFFh,TOS
satsetq:    MOV     #-1, SATQ
nosat:      NEXT
```

- Implementation of `usat` similar.

Discussion

- Fewer error handling code as you can just continue to run.
- What to do with division by zero?
- Adding more tasks to + and – slows them down, even if you don't need saturation but
- Overall system-impact low
- As a kernel option or code generator configuration when saturation arithmetic is required

¿Questions?

net2o: Command Language

A universal language for structured data and RPC

Bernd Paysan

September 26, EuroForth 2014, Palma de Mallorca

Overview

Motivation

Object Oriented Forth Code as Data

A Few Examples

Forth–Style Communication

Requirements for secure communication (secure as in "no exploitation through misinterpretation")

- Extremely simple interpreter
- Extensible, but extensions must be allowed by the receiver
- Universal, i.e. only one interpreter to audit and verify
- Triviality makes it difficult to explain

Basics

- Five data types: Integer (64 bits signed+unsigned), flag, string (generic byte array), IEEE double float, objects
- Instructions and data encoding derived from Protobuf (7 bits per byte, MSB=1 means “data continues”, most significant part first)
- Four stacks: integer, float, objects, strings
- `endwith` and `endcmd` for ending object message blocks and commands
- `oswap` to transfer the current object to the object stack, to be inserted in the outer object
- `words` for reflection (words are listed with token number, identifier and stack effect to make automatic bindigs possible)

Why binary encoding?

- Faster and simpler to parse (simpler means smaller attack vector)
- Ability to enter commands on the fly in text form through a frontend interpreter still exists
- Debugging with a de–tokenizer is also very easy
- Object–oriented approach makes writing application–specific logic extremely simple

Why a programming language as data?

Lemma: every glue logic will become Turing complete

- Implement only the things you need — but you shouldn’t have to implement more than *one* generic interpreter
- Typical idea of sending remote procedure calls: serialize the entire object (with subobjects), and call a function on that object
- Net2o idea (derived from ONF): Keep the entire object synchronized by sending only the changes to it — these changes are simple messages (setters)
- This allows multi–message passing, and reduces latency

Security

Lemma: every sufficiently complex format can be exploited

Therefore stick to a very simple format, i.e.: simplify and factor the code

Interpreter

```
: cmd@ ( -- u )
  buf-state 2@ over + >r p@+ r> over - buf-state 2! 64>n ;
: n>cmd ( n -- addr )  cells >r
  o IF   token-table  ELSE  setup-table  THEN
  $@ r@ u<= IF  net2o-crash  THEN  r> + ;
: cmd-dispatch ( addr u -- addr' u' )  buf-state 2!
  cmd@ n>cmd @ ?dup IF  execute  ELSE  net2o-crash  THEN
  buf-state 2@ ;
: cmd-loop ( addr u -- )
  BEGIN  cmd-dispatch dup 0<= UNTIL  2drop ;
```

Reading Files

reading three files

```
0 lit, file-id "net2o.fs" $, 0 lit,
open-file <req-file get-size get-stat req> endwith
1 lit, file-id "data/2011-05-13_11-26-57-small.jpg" $, 0 lit,
open-file <req-file get-size get-stat req> endwith
2 lit, file-id "data/2011-05-20_17-01-12-small.jpg" $, 0 lit,
open-file <req-file get-size get-stat req> endwith
```

Reading Files: Reply

reading three files: replies

```
0 lit, file-id 12B9A lit, set-size
    138D607CB83D0F06 lit, 1A4 lit, set-stat endwith
1 lit, file-id 9C65C lit, set-size
    13849CAE1F3B6EA8 lit, 1A4 lit, set-stat endwith
2 lit, file-id 9D240 lit, set-size
    13849CAE2643FDCC lit, 1A4 lit, set-stat endwith
```

Messages

messages

```
msg 13977C927BF7F1AA lit, msg-at  "Hi Bob!" $, msg-text
    85" Z(&3*>qxl*bWM*DUCA-Mf9N~u;<ddcW0C<XR)ezh?=jmn7zq4RFduAe=a(
    $, msg-sig endwith
85" e}&3&Kep3Im`T3?tIU=8fs>4=(C`Uic<rhs{(J`k&c5k8{H2^0*}`rV0(F3e"
$, push-$ push' nest 0 lit, ok?
```

Structured Text a la HTML

HTML–like structured text

```
body
   p "Some text with " text
       bold "bold" text oswap add
       " markup" text
   oswap add
   li
       ul "a bullet point" text oswap add
       ul "another bullet point" text oswap add
   oswap add
oswap add
```

Literature&Links

Bernd Paysan

http://fossil.net2o.de/net2o/

Forth in Education

Spreading the word

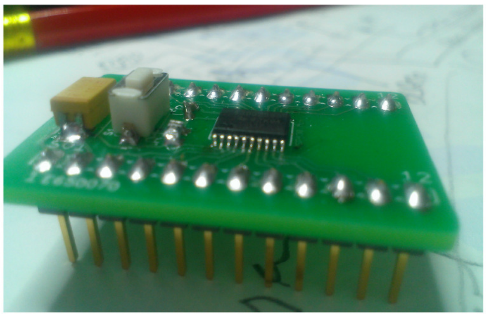
http://www.hidecs.co.uk/

Email: Paul_E.Bennett@topmail.co.uk

Aims

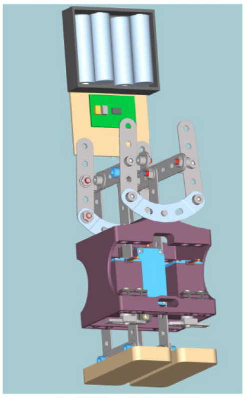
- To get Forth known in every school and college
- To Enable students to explore more involved areas of science and technology
- To unleash imagination

Some tools

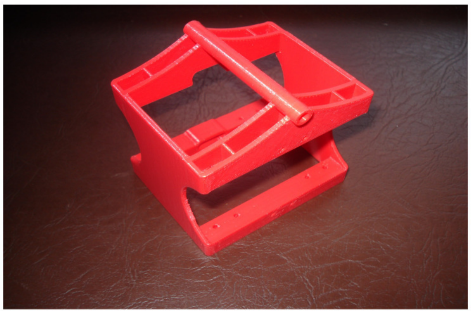


MSP430-Min 24 Pin DIL board with MPE's MSP430VfX-Lite installed.

Some CAD

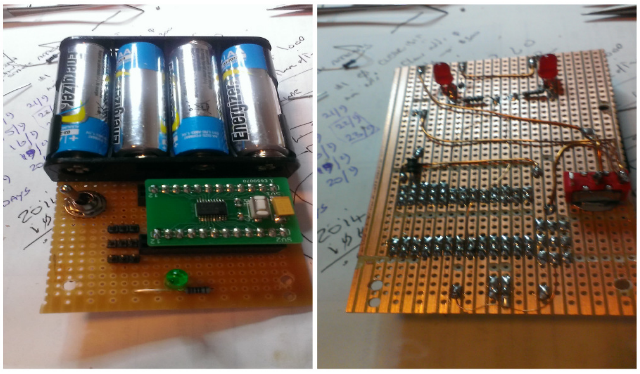


Some 3D Printing

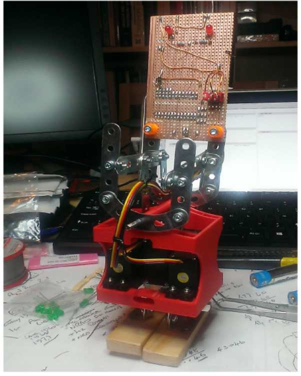


Some Mechanical Assembly

Some Electronic Assembly



...and about 50 lines of Forth later we have a Walking Robot



Hip Motion

```
: (DDA)    \ S: x\y -- 'x\'y
\ G: Starting with a value for x and y, calculate
\    the next step values 'x and 'y (Sine and
\    Cosine respectively) using the DDA algorithm
\    as published in several papers on the topic
\    [references included].
   TUCK OVER          \ y\x\y\x
   256 / - -ROT       \ 'y\y\x
   SWAP 256 / + SWAP  \ 'x\'y
;

Staring seed x=0 y=32768 (Maxneg on 16-bit)
```

PWM to Servos

```
: PWM  \ “<spaces> name”
\ G: Create an active array with the identity of "name" in
\    which is reserved two cells of data space. Each pass
\    through "name" shall decrement the second cell and return
\    a TRUE flag. If the second cell reaches zero the returned
\    flag shall be FALSE, the second cell is reloaded with the
\    value contained in the first cell during a PWM-RESET that
\    is aware of the storage structure. The first cell is the
\    desired value of delay for the channel.
   CREATE 0 , 0 ,            \ S: “ spaces name”
   DOES> DUP CELL+ @ ?DUP    \ S: -- flag
   IF    1- SWAP CELL+ ! TRUE
   ELSE  DROP FALSE
   THEN
;
```

PWM to Servos

```
Setting up the Servos
PWM Left-Leg
PWM Right-Leg
PWM Hip

\ Then to centre all servos
$80 ' Left-Leg PWM!
$80 ' Right-Leg PWM!
$80 ' Hip PWM!
```

That is but one example....

- My target so far has been the Schools and Colleges that teach the theory and practices in all STEM subjects (UTC's, Technical Secondary Schools, 6th Form colleges and beyond).
- A web-site is to be created where this and other ideas will be published to help others get started with projects that excite them. A forum will also be run where signed up members can post their questions and help answer others.

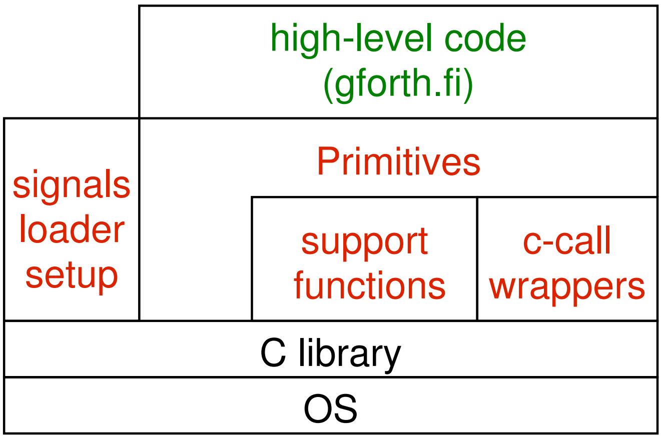
How to get rid of C

M. Anton Ertl
TU Wien

Problem: C has become unreliable

- 186 undefined behaviours in C standard
- every real-world program has them
- C compiler maintainers focus exclusively on programs without undefined behaviours benchmarks (SPEC)
- bug reports are not taken seriously
- ⇒ We want to get rid of C

Gforth components



Primitives

- replace with native-code compiler on popular platforms
- keep existing primitives on other platforms
  ⇒ we cannot get rid of C
  remove non-standard usage when gcc acts up
  no longer work around performance problems
  ⇒ slowdown
- Or maybe some primitives in assembly language
  high-level replacement for others

Native-code compiler

- Still want to use image files
- Compiler from image files to native code
- For interactive use:
  Compiler from threaded-like code to native code
  threaded-like code allows storing image files
- For bootstrapping:
  Compiler from image files to assembly language

Support functions

- Called by primitives
  e.g. mixed division
- replaced by native-code compiler
- or high-level code

Calling C

- For system calls
  Alternative: direct system calls
  additional system-specific stuff to implement
  CPU-specific optimizations
- For library calls
- use wrappers like now?
- teach calling convention to native-code compiler
  Use `extern:` for specifying C functions

Setup, loader, signals

- Could be replaced with Forth code
  on systems with native-code compiler
- But: two versions to maintain
- not performance-sensitive
  Slowdown from C standards compliance should not be noticable

Conclusion

- Getting away from C is a long-term effort
- Is it worthwhile to get rid of C completely?