31th EuroForth Conference

October 2-4, 2015

Pratts Hotel
Bath
England

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 31th EuroForth finds us in Bath for the first time. The two previous EuroForths were held in Hamburg, Germany (2013) and in Palma de Mallorca, Spain (2014). Information on earlier conferences can be found at the EuroForth home page (`http://www.euroforth.org/`).

Since 1994, EuroForth has a refereed and a non-refereed track. This year there were two submissions to the refereed track, and both were accepted (100% acceptance rate). For more meaningful statistics, I include the numbers since 2006: 19 submissions, 12 accepts, 63% acceptance rate. Each 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. Late papers will be included in the online proceedings (`http://www.euroforth.org/ef15/papers/`).

Workshops and social events complement the program. This year's EuroForth is organized by Peter Knaggs

Anton Ertl

Program committee

M. Anton Ertl, TU Wien (chair)
David Gregg, Trinity College Dublin
Phil Koopman, Carnegie Mellon University
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas

Contents

Hardware multitasking within a softcore CPU

Ulrich Hoffmann (FH Wedel University of Applied Sciences), Andrew Read

June 2015

uh@fh-wedel.de, andrew81244@outlook.com

Abstract

We have developed and implemented hardware multitasking support for a softcore CPU. The N.I.G.E. Machine's softcore CPU is an FPGA-based 32 bit stack machine optimized for running the FORTH programming language. The virtualization model that we have developed provides at least 32 independent CPU virtual machines within a single hardware instance. A full task switch takes place in only two clock cycles, the same duration as a branch or jump instruction. We have use the facility to provide a multitasking platform within the N.I.G.E. Machine's FORTH environment. Both cooperative multitasking, by means of the PAUSE instruction, and pre-emptive multitasking are supported.

1 Introduction

The N.I.G.E. Machine is a complete microcomputer system implemented on an FPGA development board [1]. It comprises a 32 bit softcore processor optimized for the FORTH programming language, a set of peripheral hardware modules, and FORTH system software (figure 1). The N.I.G.E. Machine was presented at EuroFORTH in 2012, 2013 and 2014 [2, 3, 4]. The N.I.G.E. Machine follows in the footsteps of a number of significant FORTH processors [6, 7, 8, 9, 10, 11], most especially the J-1 [6]. The N.I.G.E. Machine design files are are freely available with an open source license [5].

Most embedded systems, including those that control scientific instruments (such as the Open Network Forth system that controls the Munich particle accelerator [25]), require some level of multitasking. In this paper we explain how we have implemented multitasking in a novel manner on the N.I.G.E. Machine at the hardware level.

The development of the N.I.G.E. Machine has followed a path of utilizing FPGA hardware to enhance the performance and features of a softcore CPU. The first version of the N.I.G.E. Machine, presented at EuroFORTH 2012 [2], demonstrated the integration of a softcore CPU with a full set of peripheral modules (VGA adapter, DMA controller, I/O ports) within the same FPGA to create a standalone microcomputer system intended for the rapid prototyping of experimental scientific apparatus. The second version, presented at EuroFORTH 2013 [3], added a custom memory controller to facilitate faster and more flexible access to FPGA system memory. The third version, presented at EuroFORTH 2014 [4], introduced a sophisticated hardware return stack to allow the FORTH exception handling constructs, CATCH and THROW, to be implemented within the CPU as atomic machine language instructions. With the same philosophy in mind, the fourth version of the N.I.G.E. Machine described in this paper, adds the facility of hardware multitasking with resulting benefits for performance and reliability.

After a short overview the paper begins with a review of prior work that looks at the history of hardware support for concurrency, the history of multitasking FORTH systems and considers some notable examples of hardware designs used to assist multitasking. The following section sets the terms of reference for implementing a hardware multitasking scheme by noting the requirements for multitasking on a FORTH system in general, the requirements for the multitasking of a stack machine, and specific additional requirements that are applicable to the N.I.G.E. Machine. After that, the implementation of hardware multitasking on the N.I.G.E. Machine is described in detail at both the hardware and software levels, along with a description of how the N.I.G.E. Machine's multitasking functionality can be accessed by user applications. Finally there is a brief discussion of the advantages and limitations of our design and implementation.

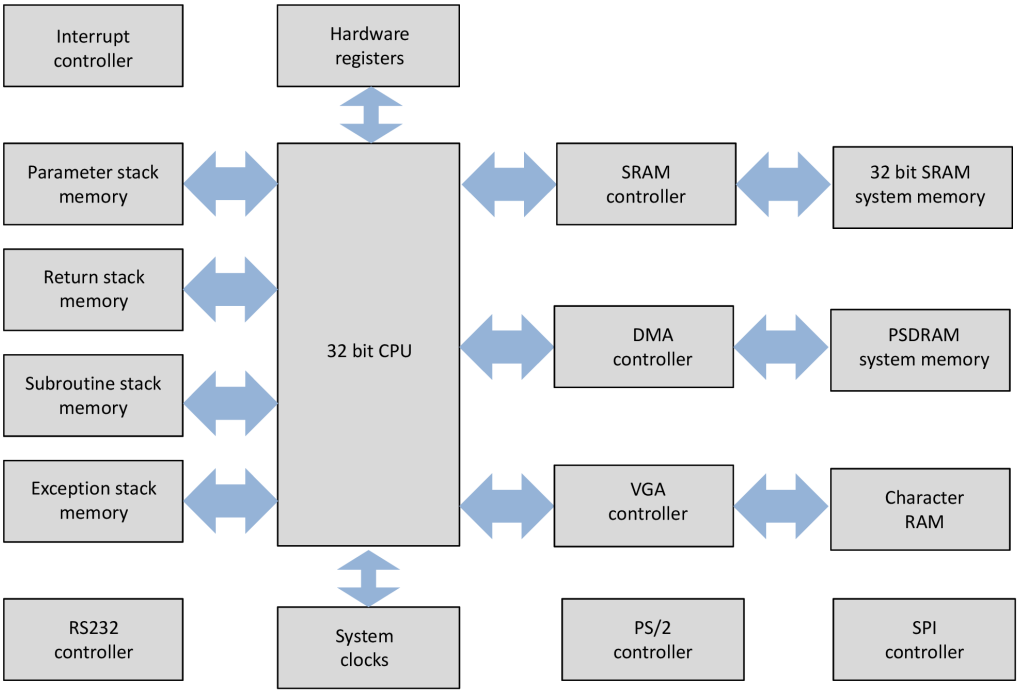


Figure 1: System diagram of the N.I.G.E. Machine

2 Outline of our model for hardware multitasking on the N.I.G.E. Machine

Our design provides 32 separate tasks hosted within a single softcore CPU instance. At any given point in time one task will be executing live on the CPU. The complete state of the other 31 tasks

that are not executing, including their program counters, stack pointers, and various registers are stored within a new sub-unit of the CPU, the virtualization unit. Because the multitasking unit is fabricated from FPGA logic within the CPU, a task switch (i.e. transfer of execution from one task to another) can be actioned with a single machine language instruction that executes very quickly, in two clock cycles in fact, the same duration as a branch or jump instruction. A second consequence of the fact that task switches are conducted entirely in hardware is that they are entirely atomic from the perspective of the flow of program code.

In addition, our design includes a lightweight task monitor that leverages the CPU hardware multitasking facility to provide multitasking capability at the FORTH software level.

3 Review of prior work

The IBM VM/370 as described by Love Seawright and Richard MacKinnon [12] offered hardware concurrency support in the form of virtualization. Rather than provide multitasking to software applications via additional functionality within the operating system, Seawright and MacKinnon's insight was to provide each application (or each operating system) with a virtual machine that was an exact copy of the underlying hardware. The software layer that provided these hardware replicas became known as the Virtual Machine Monitor (VMM).

At a lighter level than full virtualization are processor architectures that provide hardware support for context switching, but without the full resource isolation of virtualization. They are generally referred to as multithreading architectures of various types.

Barrel processors are processors that switch between n threads of execution on every cycle, thus guaranteeing that each processor will execute one instruction every n cycles. This has advantages for real-time threads operating with precise timing. The CDC 6000 range of supercomputers were pioneers of this architecture [29]. Barrel processors are an example of interleaved multithreading architectures.

Other processors such as the ARM [20] have multiple register banks to allow quick context switching for interrupt processing. Multiple register bank designs are examples of block multithreading architectures.

Another example of a block multithreading architecture is the Microcode Level Timeslicing architecture [28]. In this architecture CPU context information is held in hardware for a fixed number of tasks. Context switching overhead is eliminated since a task switch requires only the appropriate manipulation of select lines. A "stream control unit" performs the select line manipulation and coordinates context switching for the prefetch and execution units of the CPU according to the availability or otherwise of valid instructions in the prefetch registers.

Hardware multitasking support focused specifically on the efficient handling of exceptions has been tackled by two notable systems.

Klaus Schleisiek's microcore includes a hardware mechanism to support multitasking and is specifically focused on the problem of dealing with busy resources [7]. The microcore EXCEPTION mechanism allows routines to access external resources without having to query status bits to ascertain their availability/ readiness. This greatly simplifies the software needed for serial channels communicating with external devices or processes. It works as follows: when the processor intends to access a resource, the resource may not be ready yet. In such an event, the resource can assert the EXCEPTION signal before the end of the current instruction execution cycle. This disables storing the next processor state into any register except for the instruction register, which loads a special "exc" instruction instead of the next instruction from program memory. In the next processor cycle, exc will be executed calling the Exception Service Routine (ESR) at its fixed address. The ESR address will typically hold a branch to code that performs an operating system dependent task switch.

The INMOS Transputer [17], employed a rendezvous communication mechanism on external I/O ports that was used to perform a task switch entirely in hardware.

An alternative to virtualization or hardware multithreading is a multi-core processor architecture. In field of embedded design we note that most ARM Cortex [20] processors are now dual core or quad core. In addition the Parallax Propeller [21] is a low cost micro-controller with eight 32 bit cores that has a simple tool chain making it attractive for prototyping applications. Finally, the GreenArrays GA144 is a more specialized system with 144 polyFORTH execution units on a single chip [22]. Both the Transputer and the GA144 feature high speed connections between cores.

4 Hardware multitasking requirements

4.1 Requirements for a multitasking system in FORTH

The ultimate purpose of implementing hardware multitasking on the N.I.G.E. Machine is to provide a multitasking FORTH system. Brad Rodriguez's 1992 article, "FORTH Multitasking in a Nutshell" [15], provides a comprehensive review of the requirements. These are, in terms of memory: private stacks, private user areas and private buffers, and in terms of software: re-entrant FORTH system code, suitable mechanisms for the mutual exclusion of resources that cannot be shared, and suitably designed task switching functionality.

Many FORTH systems offer cooperative multitasking (where a call to the word PAUSE is required to yield the CPU to the next task) in preference to pre-emptive multitasking (where the CPU is automatically time sliced between tasks). In an embedded environment where all tasks are part of a single integrated system pre-emptive multitasking may not be necessary. In these cases cooperative multitasking may have some advantages for simplicity of design and testing, provided that all tasks truly cooperate.

4.2 Requirements for the multitasking of a stack machine

The general requirements for multitasking of a CPU are the ability to (a) switch the execution of the CPU from one task to another, (b) store the state of the task that is being "frozen" (i.e. preserve the "state vectors") and (c) restore the state of the task that is being "thawed". This requirement can be applied to a stack machine where the CPU state vector will in general comprise three elements (1) the program counter, (2) the stack pointers and (3) the stack memory space. (If stack memory space is global to all virtual machines then it is sufficient to switch only the stack pointers.)

In addition, a stack machine may utilize a number of registers. For example, top of stack values may be held in registers to enhance processing throughput, there may be flags such as arithmetic carry/overflow, or an interrupt processing indicator, and the internal state of the CPU is likely to be a finite state machine (FSM) with its own state register. For each of these registers a decision needs to be made as to whether (a) it will be included in the saved CPU state vector, (b) it will be discarded on each virtual machine switch, or (c) whether virtual machine switching will be arranged so that it is not necessary to save the value (e.g. switching can only occur when the FSM is in a single, predetermined state).

4.3 Specific requirements relating to the design of the N.I.G.E. Machine

The parameter and return stacks of the N.I.G.E. Machine are connected directly to the datapath through dedicated memory ports rather than being accessed via the general CPU system memory bus. This design leads to performance advantages because the datapath is always in a position to

read or write from stacks with no latency. However it also means that a "software only" multitasking implementation is not feasible on the N.I.G.E. Machine. Modifications must be made to the datapath itself in order to facilitate the switching of stacks for each task.

In addition to the parameter and return stacks, the N.I.G.E. Machine datapath utilizes internal subroutine and exception stacks that provide hardware support for subroutine calls, local variables and the FORTH exception handling words CATCH and THROW [4]. Although these stacks are not directly accessible to user applications, it is necessary for each task to have a private copies.

Lastly, the N.I.G.E. Machine has been designed specifically for embedded control and scientific prototyping. As EuroFORTH 2012 paper explained [2], short interrupt response times and deterministic execution are critical in these applications. Any hardware multitasking scheme employed needs to respect both of these objectives.

5 Hardware multitasking design on the N.I.G.E. Machine

5.1 General features

As explained in the introduction, the purpose of incorporating hardware multitasking into the N.I.G.E. Machine is to provide multitasking for the FORTH system software. (Since each task at the FORTH system level will run on its own virtual machine instance.)

In the default configuration of the N.I.G.E. Machine, 32 tasks are available, each with a parameter stack depth of 256 cells and a return stack depth of 128 cells. Cooperative multitasking is achieved with a PAUSE machine language instruction which executes a full task switch in 2 clock cycles (the same duration as the execution of a branch or subroutine call). A pre-emptive multitasking mode is also available that implements a task switch after a user definable count of executed instructions. The default task scheduling system is round-robin among active tasks.

There is a lightweight task monitor included in the FORTH system software that comprises words for starting, stopping, and otherwise managing tasks. The task monitor is not involved with task switching since this is handled entirely by hardware. Each task has a 2 KiB private user memory area that offers space for 245 longword user variables and holds private buffers and certain task specific system variables. The N.I.G.E. Machine's remaining 128 KiB of FPGA systems memory (including the FORTH dictionary) and all of the 16 MiB of PSDRAM is shared memory available to all tasks.

Simple semaphore based locks have been implemented in the FORTH system software to mediate access to shared I/O resources. The locks are arranged so that no FORTH system routine will ever attempt to lock more than one resource at any given time. In this way it is not possible for user applications to enter a deadlock situation if they only call system functions.

The N.I.G.E. Machine also provides a feature that we describe as "virtual interrupts". With a virtual interrupt, a task may cause another task to jump to a subroutine (typically a FORTH execution token) before returning to its prior point of execution. A virtual interrupt may be scheduled in advance at any time and will be actioned when a switch to the task in question next occurs.

5.2 Multitasking unit

The multitasking unit is a new component within the CPU alongside the control unit and the datapath (figure 2). The multitasking unit is responsible for two functions: firstly for storing the states of all of the tasks that are not currently executing, and secondly for managing the transition between executing tasks.

To achieve the first objective the multitasking unit relies on an internal RAM module termed the "freezer RAM". The freezer RAM module is 116 bits wide and 32 addresses deep. (The RAM modules within the multitasking unit are implemented with FPGA logic elements (registers) rather than BLOCK RAM - see section 6 for further explanation).

The state of each task is arranged as a 116 bit state vector as illustrated in table 1.

Task state vector component	**Number of bits**
Program counter	20
Top-of-stack register	32
Next-on-stack register	32
Parameter stack pointer	8
Return stack pointer	8
Subroutine stack pointer	8
Exception stack pointer	8
Total	116

Table 1: Storage of the task state vectors within the "freezer" RAM module

The freezer RAM module is dual ported with a single write port and a single read port. When a task switch is signaled, the state vector of the current (and now retiring) task is presented at the write port with a valid write enable signal, while the state vector of the next-to-execute task is taken from the read port. These state vectors are routed to and from the control unit (for the program counter) and the datapath (for all other elements). Figure 2 illustrates the place of the multitasking unit in relation to the control unit and datapath within the CPU.

The address input of the freezer RAM module's write port comes from a 5 bit wide register within the multitasking unit that holds the number of the currently executing task. The address input for the read port (i.e. the number of the next-to-execute task) is drawn from a second RAM module within the multitasking unit called the task control RAM. The signal to execute a task switch originates from the control unit (described more fully below).

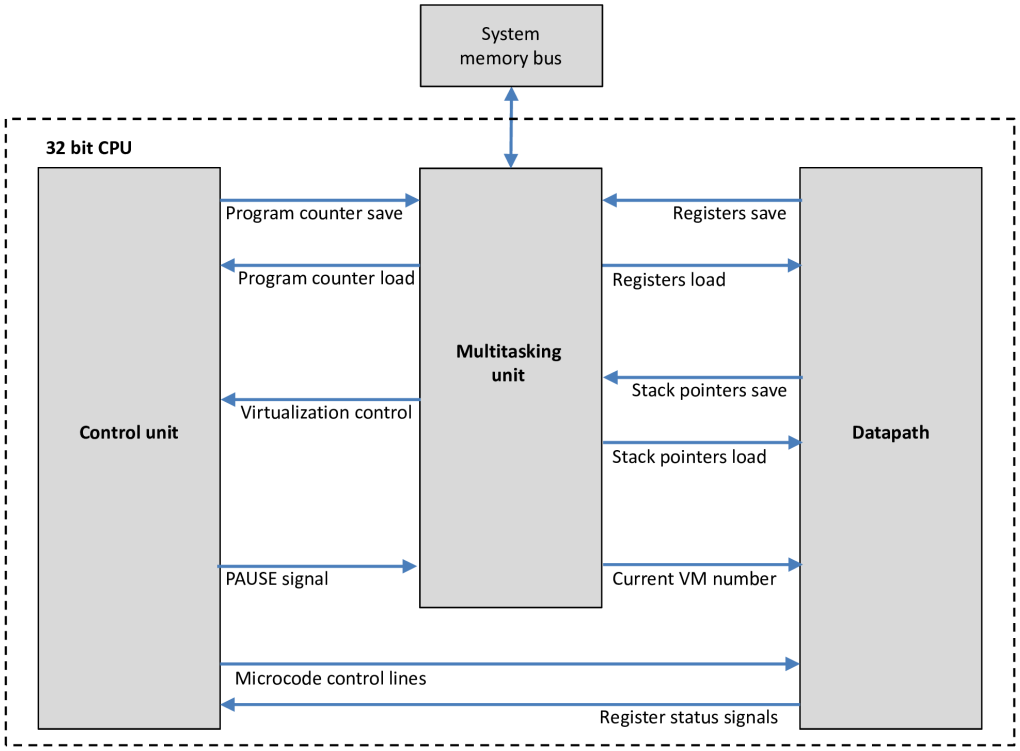


Figure 2: Relationship between the multitasking unit, control unit and datapath with the CPU

The task control RAM module is 16 bits wide and 32 addresses deep. It can be considered as a set of 32 x 16 bit registers, one belonging to each task. The lower 5 bits of each resister hold the number of the next-to-execute task following that task. For example if register 1 holds the value "00010", then task 2 is the next-to-execute task following task 1. The task control RAM module is dual ported. The first port is read-only and is addressed with the register holding the value of the currently executing task. The output from this port is therefore the number of the next-to-execute task (in the lowest 5 bits). It is used to address the read port of the freezer RAM as described above. The second port is a read/write port that is memory mapped to the system memory address space. The task monitor within the FORTH system software uses these memory mapped addresses to configure the order of task execution by appropriately setting the individual task control registers. The upper 11 bits of each task control register are not read by the multitasking hardware but are used by the task monitor as general storage for further task control purposes as described below.

In order to initialize a new task the task monitor needs to be able to configure the program counter of a task before it begins execution for the first time. A third dual ported memory block, the "PC override" RAM block, is used to provide this facility. The PC override RAM module is 20 bits wide and 32 addresses deep. It can also be thought of as 32 individual registers. Like the task control RAM module, each register is mapped to the system memory address space and can be written to by the task monitor. When a task switch occurs, if the PC override register of the next-to-execute task contains a non-zero value, then that value is sent to the control unit as the new program counter address in place of whatever value may have been held in the task's state vector in freezer RAM. The PC override register for that task is then automatically reset to zero

so that the following task switch will proceed without a second override.

The fourth RAM block (the "virtual interrupt" RAM block) provides functionality for virtual interrupts in a similar manner to the PC override RAM. However in the case of a task switch with a virtual interrupt, the control unit pushes the saved value of the PC from the state vector onto the subroutine and return stacks before setting the program counter register to the value from the virtual interrupt RAM.

The multitasking unit also contains a number of individual memory mapped registers that allow the task monitor to control the conduct of task switching. These are scheduled in table 2. Table 3 schedules the overall memory map of the multitasking unit as seen from the system memory bus.

Figure 3 illustrates the key operational features of the of the multitasking unit.

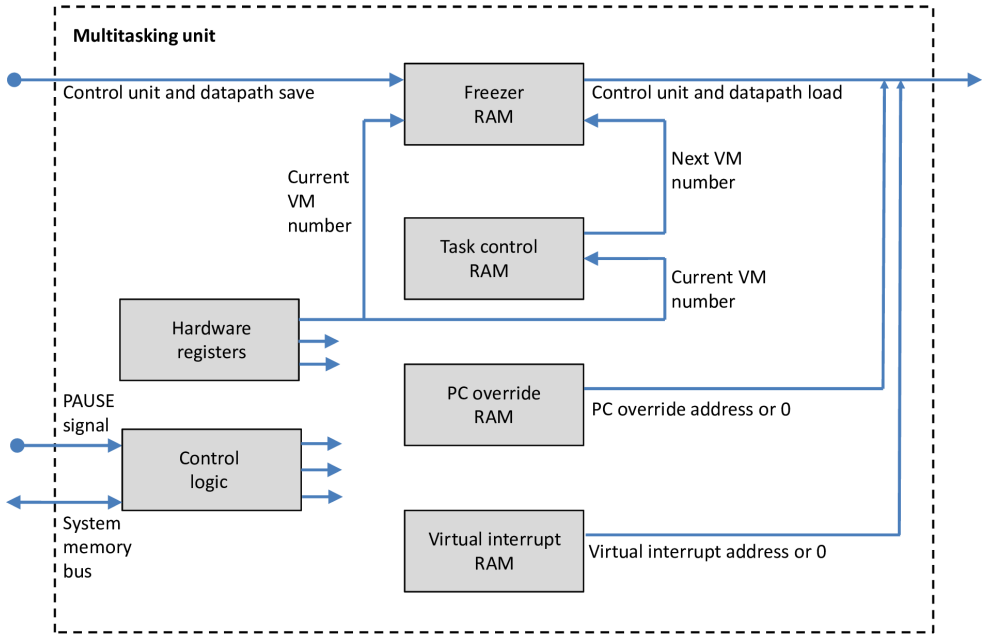


Figure 3: Illustration of the key operational features of the multitasking unit

Register name	Function	R/W	Width (bits)	System address
SINGLEMULTI	Enable ('1') or disable ('0') multitasking. If a PAUSE machine language instruction is encountered with multitasking disabled then it will be treated as a NOP	R/W	1	3F000
CURRENTVM	The number of the currently executing task. Tasks are numbered 0 through 31. At power-on task 0 will be executing the FORTH system software	R	5	3F004
INTERVAL	The interval for pre-emptive multitasking task switches, in count of instructions. If INTERVAL = 0 then pre-emptive multitasking is disabled.	R/W	16	3F008

Table 2: Multitasking control registers

Register name	Function	# registers	System address
Multitasking control registers	As table 2	3	3F000
Task control registers	Each virtual machine has an associated task control. Bits 4 down to 0 of this register specify the next-to-execute task. Bits 15 down to 5 are for task monitor usage	32	3F200
PC override registers	Writing a non-zero address to a PC override register will cause the task in question to continue from that address on the next occasion that it executes	32	3F400
Virtual interrupt register	Writing a non-zero address to a virtual interrupt register will cause the task to branch to a subroutine at that address on the next occasion that it executes	32	3F600

Table 3: Memory map of the multitasking unit as viewed from the system memory bus

5.3 Softcore CPU - datapath

Two updates were necessary to the CPU datapath in order to support hardware multitasking. Firstly, the parameter, return, subroutine and exception stacks were extended so that each task would have its own private stack. This was achieved by increasing the addressable width of each stack and allocating to each stack additional FPGA BLOCK RAM. Following this modification, each stack is addressed within the datapath by concatenating the number of the current virtual machine (higher 5 bits) with the current stack pointer (lower 8 bits).

Secondly, the values of the top-of-stack and next-on-stack registers and the values of the stack pointers were interfaced with the freezer RAM within the multitasking control unit. This was done simply by including additional multiplexers and extending the width of microcode instructions communicated from the control unit to 23 bits. Further details on the operation of the datapath and how the control unit uses microcode to choreograph the datapath multiplexers is given in the

EuroFORTH 2012 paper [2]. By way of illustration, table 4 shows how the exception stack pointer is updated each clock cycle according to the microcode signaled from the control unit.

Function	**Microcode bits 22 down to 20**
No change	000
Decrement	001
Increment	010
Set to zero	011
Load value from virtualization unit	100

Table 4: Control table for the exception stack pointer illustrating the extension for hardware multitasking. The exception stack pointer is a datapath register that is operated from the control unit by microcode bits 22 down to 20

5.4 Softcore CPU - control unit

The first modification made to the control unit was to specify a new PAUSE instruction to effect a cooperative multitasking task switch. No major "rewiring" of the control unit was required to accomplish this: the PAUSE instruction acts on the same level and in the same way as all machine language instructions within the control unit's finite state machine via microcode lookup. In fact the PAUSE instruction was implemented as a modified jump (JMP) instruction, but with the appropriate microcode to control the datapath registers, and with the next-instruction address read from the multitasking unit rather than from the parameter stack. It is because a task switch can be executed as a standard machine language instruction that the latency for a task switch is the same as for any other jump or branch (2 clock cycles). A task switch involving a virtual interrupt involves an additional stage to push the value of the program counter onto the subroutine stack and therefore executes in 3 clock cycles.

Secondly, a 16 bit counter was introduced to count the number of instructions executed since the last task switch. This counter is compared with the INTERVAL register of the multitasking unit and a pre-emptive task switch is triggered when the INTERVAL count has been reached or exceeded, if preemptive multitasking is enabled.

The control unit uses a common mechanism to handle a PAUSE instruction and a preemptive task switch. One important difference however is the value of the program counter that is saved to the multitasking unit. In the case of encountering a PAUSE instruction, the task should resume execution at the instruction following this PAUSE. In the case of a preemptive task switch, then the current instruction (whatever it is) will not be executed since the preemptive task switch has priority. In this case the task should resume execution at this instruction. This differentiation is similar to how the control unit selects the appropriate value of the program counter to push onto the subroutine stack in the cases of jump to subroutine (JSR) instructions and interrupts, and identical hardware logic was used.

5.5 Interrupts

With any multitasking design a decision is needed as to how interrupts will interact with task switching. There are two broad alternatives: either interrupts are synchronized with task switches so that interrupt handlers always run within their own tasks, or interrupts are handled by whichever task happens to be running at the time when the interrupt request occurs.

We did not examine the trade-off between these two approaches in great detail. The N.I.G.E. Machine takes the latter approach. For us the simplicity of design thus afforded and the avoidance

of any possible latency in interrupt responses were sufficiently compelling advantages in the absence of any obvious considerations to the contrary.

Given this design decision, the N.I.G.E. Machine system interrupts for RS232 and PS/2 I/O execute in whichever task is running when the interrupt request occurs. The RS232 and PS/2 interrupt service routines operate by transferring characters between the relevant hardware interface and memory buffers, updating the buffer counters accordingly. Tasks that need to wait for RS232 or PS/2 communications do so by polling for updates to the relevant buffer counters in a loop that includes a PAUSE statement. The FORTH word KEY? is implemented in this manner.

In order to avoid any performance or reliability impact in interrupt handling, it is necessary to ensure that interrupts cannot themselves be interrupted by task switches. Pre-emptive multitasking is automatically disabled by the control unit during interrupt processing. If the preemption instruction counter reaches INTERVAL while an interrupt is in progress then the task switch is postponed until immediately after the interrupt service routine has concluded. For cooperative multitasking, it is a N.I.G.E. Machine software design requirement that PAUSE machine language instructions should not be included within interrupt service routines.

5.6 Task monitor software

The task monitor software is a set of words within the FORTH system software available to initiate and control tasks. A description of some of the words is presented in section 7.

The task monitor's method to control the default sequencing and allocation of tasks is as follows: the lowest 5 bits of each task control register (there is one task control register for each task) indicate the number of the next-to-execute task. These 5 bits are utilized directly within the multitasking unit to sequence a task switch as described above. The remaining 11 bits do not directly control hardware but are utilized by the task monitor. Bit 15 is used to indicate whether a task has been allocated to a running task ('1') or is unassigned and therefore available for a new task upon request ('0'). Bits 9 down to 5 are used by the task monitor to indicate the number of the task that points execution to this task. In this way the lower 10 bits form the nodes of a doubly linked list that specifies the task execution order. Bits 14 down to 10 are reserved for expansion.

When a new task is requested (see RUN in section 7) the task monitor first searches the set of task control register to identify an unassigned task (indicated if bit 15 is clear). The new task is then inserted into the doubly linked list of executing tasks after the currently executing task by updating the nodes of the double linked list maintained within the lower 10 bits of the task control registers. For all newly initialized tasks, the program counter for the new task is directed to a common initialization routine. This brief (~60 byte) initialization routine is responsible for resetting the stack pointers of that task to zero, copying the set of initialization parameters specified by the RUN command to the stack via. intermediate storage in shared memory, initializing the user variable area, initializing the exception stack variables and then jumping to the specified execution token.

When a task is put to sleep (see SLEEP in section 7), it is removed from the doubly linked list of executing tasks but bit 15 of the task control register remains set to indicate that the task is still allocated. When a task is woken (see WAKE in section 7) it is re-inserted into the doubly linked list of executing tasks. When a task is stopped, then in addition to removing it from the list of currently executing tasks, bit 15 is cleared to indicate that it is now free for reallocation.

5.7 Controls over task switching

The multitasking unit operates two controls to limit task switching and thus avoid glitches:

Firstly, because the state-vector of the next-to-execute task is automatically transferred from the RAM blocks within the multitasking unit into the control unit and datapath when a task switch

occurs, it is necessary for these RAM blocks to have time to update properly between task switches. The minimum interval between task switches is 3 clock cycles due to this update requirement. A control device within the multitasking unit imposes a 5 clock cycle minimum interval between task switches. If a PAUSE instruction or pre-emptive task switch occurs within this limit then it will simply be ignored. This is a critical control since cooperative PAUSE instructions continue to remain effective even after pre-emptive multitasking has been enabled.

Secondly, multitasking is automatically disabled by the multitasking unit when there is only one active task (i.e. where the lower 5 bits of the currently active task's task control register references itself). This is necessary on account of the same update constraint.

5.8 User memory area

Each task has a private 2 KiB user memory area that is mapped to the system address space. The user memory areas are hosted within 64 KiB of FPGA BLOCK RAM. The upper 5 address bits are linked directly to the register that holds the number of the currently executing task. In this way each of the 32 user memory areas can be mapped to the same address range in the system address space. The currently executing task will always have guaranteed private assess to its own user memory area but no access to the user memory areas of other tasks. Table 5 is an outline memory map of the user memory area. Further details are given in the N.I.G.E. Machine documentation.

Usage	**# bytes**	**System address**
FORTH system task specific variables	44	3D000
Available for USER variables	980	3D02C
The FORTH PAD buffer	512	3D400
The FORTH ACCEPT buffer	256	3D600
Reserved for expansion	256	3D700

Table 5: Outline memory map of the 2 KiB user memory area

5.9 Memory management

We took the decision not to implement any form of memory management over the 128 KiB of system memory that holds the FORTH system dictionary and user applications, or over the 16 MiB of off-chip PSDRAM that holds the screen buffer and is available for application data storage. As a result, all of the system memory and PSDRAM is available to any task without restriction. The discussion in section 8 considers the merits and limitations of this approach.

5.10 Motivation for the design decisions

The principal motivations for our design decisions are discussed in section 8 by way of comparison to alternative concurrency strategies.

A hardware multithreading approach offers some advantages, as described in this paper, but necessarily also entails some fixed allocation of resources at design time. Our resource allocations were based on "educated guesses" rather than specific research, but this is a softcore design and flexibility is retained since the allocations can quite easily be adapted for individual builds, often simply by changing VHDL GENERIC declarations.

6 FPGA implementation

The N.I.G.E. Machine is implemented on a Digilent Nexys4 development board [18] which features a Xilinx Artix-7 FPGA (Xilinx part number XC7A100T-1CSG324C [19]). The design has been developed using the VHDL hardware description language and the Xilinx ISE development studio, version 14.6. Table 6 shows the FPGA utilization for the fully synthesized design. Table 7 analyzes the usage of BLOCK RAM.

FPGA resource	**Utilization**
Slice registers	4%
Slice look up tables (LUT's)	9%
FPGA BLOCK RAM	97%

Table 6: FPGA utilization

Design component	**BLOCK RAM count**
System memory	32.0
Task private user memory	16.0
Parameter stacks	7.5
Return stacks	4.0
Subroutine stacks (including space for local variable storage)[4]	68.0
Exception stacks [4]	5.0
VGA display interface	1.0
CPU microcode	0.5

Table 7: FPGA BLOCK RAM usage by design component. Each BLOCK RAM resource represents 4Kbytes.

The Xilinx XC7A100T is a latest generation FPGA in the Xilinx "value" range. The N.I.G.E. Machine utilizes less than 10% of the fabric logic on this device. On the other hand the BLOCK RAM is significantly utilized at 97%. The simple reason for the high utilization of BLOCK RAM is that the private stacks and user memory areas for all of the 32 tasks are pre-dedicated at synthesis time regardless of how many active tasks any given application will actually create. However it is also possible to synthesis the design with 16, 8, 4, or 2 virtual machines instead of 32 by adjusting the top-level VHDL GENERIC declarations and the ipCORE declarations of the relevant RAM blocks.

Not all of the RAM modules on the N.I.G.E. Machine are instantiated with BLOCK RAM. All of the RAM modules within the multitasking unit are instantiated using distributed FPGA logic elements for resource efficiency reasons. Xilinx Artix-7 BLOCK RAM units can be configured in a variety of formats between 32K x 1 and 512 x 36 (address depth x bit width), but the freezer RAM has a relatively wide but shallow format of 32 x 116 which would lead to poor utilization in BLOCK RAM.

We had concerns that the over utilization of BLOCK RAM would significantly impede place and route performance, since the design effectively requires that signals be routed to and from BLOCK RAM instances right across the FPGA. During development we found that ISE's simulated annealing placement algorithm was quite sensitive to design changes (meaning that small changes in the logic design could have significant impact on the place and route performance, presumably due to their implications for routing). ISE's SmartXplorer, which is a tool for automatically optimizing

placement using for example, different cost tables within the simulated annealing algorithm, was able to meet timing with a clock frequency of 100 MHz but significant search effort was required (8 out of 100 strategies succeeded). At a clock frequency of 95 MHz, SmartXplorer was able to meet timing with the vast majority (95 out of 100) of strategies. The N.I.G.E. Machine's 100 MHz clock frequency has been retained, but it would likely be easier to develop future projects with a clock frequency of 95 MHz and then use SmartXplorer re-optimize place and route for a clock frequency of 100 MHz as the final step.

7 N.I.G.E. Machine multitasking functionality

This section describes a selection of the FORTH words that are available to user applications to control multitasking on the N.I.G.E. Machine. A full list is give in the N.I.G.E. Machine system documentation [5]. The majority of words are directly analogous to those of the PolyFORTH multitasking system [16].

7.1 Multitasking configuration

`SINGLE ( --)`

Disable multitasking. PAUSE instructions will be treated as a NOP. Multitasking is enabled at power-on by default on the N.I.G.E. Machine.

`MULTI ( --)`

Enable multitasking. Note that if there is only a single active task then PAUSE will be treated as NOP.

7.2 Task initiation

`RUN ( p1 ... pn n XT -- TN true | false)`

Initialize a new task to take n stack parameters (p_1 ... p_n) and execute the code pointed to be execution token XT. Return the number of the task allocated to this task (TN) and true if successful, or false if all tasks are currently otherwise allocated. The newly created task will be positioned in the round-robin sequence immediately after the current task. Tasks are numbered 0 through 31. Note that the XT must either code an infinite loop or contain termination instructions to self-abort.

7.3 Task switching

`PAUSE ( --)`

Task switch. Yield CPU execution of the current task and switch CPU execution to the next-to-execute task.

`SLEEP ( n --)`

Put task n to sleep by removing it from the list of executing tasks. The task remains allocated and can be woken at a later time.

`WAKE ( n --)`

Wake task n by inserting it into the list of executing tasks immediately following the current task.

`STOP ( n --)`

Deallocate task n and remove it from the list of executing tasks. This task may now be recycled by RUN.

7.4 Pre-emptive multitasking

`PREEMPTIVE ( n --)`

Enable preemptive multitasking with period of n instructions between task switches. If n = 0 then preemptive multitasking is disabled. Preemptive multitasking is disabled at power on by default on the N.I.G.E. Machine.

7.5 Virtual interrupts

`VIRQ ( XT n --)`

Virtual interrupt. Cause task n to branch to the subroutine at XT and then return to its prior point of execution. The virtual interrupt will be actioned when task n is next scheduled to execute.

7.6 Mutual exclusion

`ACQUIRE ( sem --)`

Acquire the binary semaphore (sem) or yield until it becomes free. A semaphore can be any FORTH variable with global scope. Semaphores are minimum single byte in length (word or longword length variables may also be used). A semaphore contains the number of the latest successfully acquiring virtual machine XOR 255, or 0 if not acquired.

`RELEASE ( sem --)`

Release the binary semaphore (sem).

7.7 Inter-task communication

We have not attempted to provide hardware based support for inter-task communication. As noted above, each task's 2 KiB private memory is not accessible by other tasks but the rest of the 128 KiB system memory and all of the 16 MiB of PSDRAM on the N.I.G.E. Machine is accessible by all tasks. Application specific inter-task communication designs can utilize shared memory for data passing and may take advantage of the ACQUIRE and RELEASE words for mutual exclusion.

8 Discussion

8.1 Comparison with other hardware multithreading strategies

The N.I.G.E. Machine's approach to hardware multitasking has some similarities with the multiple register file / block multithreading architectures referenced in section 3 but as a whole is different from those strategies.

Multiple register file architectures essentially use control lines to select which instances of the CPU registers are to be updated each cycle. The N.I.G.E. Machine takes a similar approach in selecting the parameter and return stack for each task by extending the address width of each stack and concatenating the number of the currently executing thread at the high end of the address bus with the relevant stack pointer at the low end.

However the N.I.G.E. Machine does not apply this approach to the registers within the CPU and datapath (e.g. the program counter and top-of-stack register), rather they are saved and reloaded from an external store (the multitasking unit) each time a task switch occurs. The reason for doing

this is timing efficiency. The N.I.G.E. Machine operates at 100 MHz meaning that each clock cycle must complete within 10 ns if the design is to "meet timing". Each additional layer of logic in a signal path adds delay due to both the response time of the logic itself and the necessary signal routing. In terms of logic layout, a register file of 32 registers is essentially a 32 way multiplexer that must sit between say the ALU and the register to be updated. Even though modern FPGA multiplexers are highly optimized, a 32 way multiplexer would need to be implemented in 2 or 3 additional layers of FPGA logic[30]. By storing register information for each task in a separate unit, the N.I.G.E. Machine avoids the need for any extra FPGA logic on the signal path that updates each cycle. Rather the "update burden" is shifted to the cycles that occur between clock cycles, which in the case of the N.I.G.E. Machine is only 2 clock cycles in any case.

A final difference is that a task switch on the N.I.G.E. Machine switches more than the CPU context: the USER memory areas that are private to each task are switched concurrently. Hence we have termed the N.I.G.E Machine's architecture as hardware multitasking rather than hardware multithreading.

8.2 Comparison with multi-core processor strategies

In recent years the trend in processor development has been firmly towards multi-core CPU's, even in embedded applications [20]. However this trend is not without a number of difficulties imposed by the complications of multi-core software development [23]. Our focus on developing a virtualization model for the N.I.G.E. Machine has been to attempt to balance the pursuit of absolute performance with simplicity for the application programmer.

The key difference between programming a single core multitasking architecture such as the N.I.G.E. Machine as compared with a multi-core CPU is the elimination of possible asynchronous effects. The N.I.G.E. Machine used in cooperative multitasking mode will have absolutely deterministic behaviour (and timing) since there is a single execution path through all contexts. In a multi-core CPU multiple asynchronous execution paths must be modelled, programmed and debugged.

The N.I.G.E. Machine is intended for rapid prototyping applications where fast and easy software development should be a particular advantage. The certain absence of asynchronous effects is the programmer simplification motivating our preference for a single core rather than a multi-core approach.

8.3 Chosen approach to memory management

As described in section 5, aside from providing a 2 KiB private memory area, we decided not to implement any memory management mechanisms for the 128 KiB of main system memory and the 16 MiB of PSDRAM. We recognize that on a modern server or desktop based multitasking system it would be considered a fatal weakness not to provide memory protection mechanisms that prevent tasks from corrupting the memory used by other tasks. High-reliability computing is also in demand in the embedded space.

However the intended focus of the N.I.G.E. Machine is in relatively small scale deeply embedded applications. For that reason we envisage that in most situations, all tasks will be sub-modules of a single overall application and so inter-task protections may be a less critical factor in total application reliability.

Another reason for our decision is that since the overall system memory is only 128 KiB (albeit this is probably still a reasonable system memory size for a deeply embedded device [24]), it would not be feasible to subdivide this memory between tasks and still retain a sensible amount for each.

Finally, the FORTH language is dictionary based and it is typical for multitasking FORTH systems to have access to a common system dictionary. The FORTH system software on the N.I.G.E.

Machine implements the ANSI Search-Order word list which allows individual applications or tasks to extend or restrict the dictionary with private word lists if desired.

8.4 Alternative task scheduling models

The task scheduling model implemented by the task monitor is a simple round-robin scheme. That is, all active tasks take their turn to be executed once per round. This is the fastest task switching model that can be implemented using the N.I.G.E. Machine's hardware multitasking framework since the next-to-execute task is determined in advance and task switches take place atomically in only two clock cycles. Given the high performance of this scheduling model within the N.I.G.E. Machine, and it's ubiquity on multitasking FORTH systems [15], we believe that it would likely be the most effective choice for most embedded applications.

However many other priority based task scheduling models exist [13]. A priority based task scheduling model can be accommodated within the N.I.G.E. Machine's hardware multitasking framework by changing the way that task control registers are used by the task monitor. An outline of how this could be achieved is as follows: the task control registers of all tasks are set such that the next-to-execute task is always a common scheduling task (say task 31). The scheduling task would be responsible for maintaining a list of task priorities and determining the next-to-execute on a real time basis. It would conclude its operation by setting the value of its own task control register before executing PAUSE.

Paul Bennett has pointed out [26] that an alternative model for cooperative multitasking is the Time-Triggered Systems (TTS) approach [27]. TTS is typically based on just one interrupt (the system tick timer). It schedules cooperative tasks to run at intervals according to their order in the "tick list". There is no main loop of program execution aside from the tick list itself. All I/O is polled. TTS could quite likely be implemented on the N.I.G.E. Machine using this multitasking hardware with light adjustments to the task monitor software.

Although we have not further investigated in any detail, it was mentioned to us that this architecture might also be leveraged to support the high level language features of co-routines and continuations.

8.5 Limitation of the hardware multitasking approach

An obvious limitation of the N.I.G.E. Machine's approach to multitasking is that the number of tasks is limited to the 32 that are pre-instantiated in hardware. We did not conduct a feasibility study of the number of tasks typically required by an embedded system but would expect based on general experience that parallel programming complexities might become a constraint in an embedded application before the limit of 32 tasks had been reached.

Another limitation is that hardware resources (mainly FPGA RAM blocks) are pre-allocated to tasks that may never be used, in which case they are effectively wasted. With 32 tasks allocated there is 128Kbytes of system memory for FORTH applications and parameter and return stack depths of 256 and 128 cells respectively in each virtual machine. So sufficient resources are available for a meaningful FORTH system. As explained in section 6, it is possible to synthesize the N.I.G.E. Machine with 32, 16, 8, 4 or 2 tasks by adjusting the generics declarations in the VHDL code.

8.6 Advantages of the hardware multitasking approach

We suggest that there are a number of advantages to using hardware multitasking to provide a multitasking FORTH system as compared with traditional software-based multitasking.

Firstly although task switching on FORTH systems is typically very fast [15], the ability to complete a full task switch in the same duration as a jump or branch means that effectively multitasking has no performance overhead on the N.I.G.E. Machine.

One way this performance advantage could be put to use for enhancing reliability is by including PAUSE instructions within FORTH loop structure words (LOOP, +LOOP, UNTIL, AGAIN, REPEAT) [26]. Although that has not been done with the current version of the N.I.G.E. Machine's FORTH system software, the advantage of doing so would be to decrease the likelihood of tasks failing to cooperate due to their being insufficient PAUSE instructions within their routines.

Secondly, because pre-emptive multitasking is implemented directly by the CPU and not via interrupts, it is never necessary to disable interrupts during tasking switching, or even during the initiation of new tasks. This means that the N.I.G.E. Machine avoids any interrupt latency due to multitasking.

Lastly, since task switching is handled by a single machine language instruction, each task switch is atomic, i.e. the thread of execution is always with one task or another, never in-between tasks. This may have some reliability benefits since there are no task switching software routines that have the potential to become corrupted during program execution.

9 Conclusion

FORTH systems have offered multitasking since very early in the history of FORTH language and on very lightweight system [15, 16]. Now that the N.I.G.E. Machine includes multitasking capability with fast and efficient hardware support, we believe that the platform is sufficiently developed to be applied to its intended field in the rapid prototyping of experimental scientific apparatus. It is hoped that the next stage of development will focus on opportunities in this area. In addition there is the possibility to port the design to other FPGA development boards (e.g. the Diligent Nexys4-DDR) or enhance the range of input/output ports.

The authors wish to thank the anonymous academic reviewers for their comments, especially those relating to the terminology of the architecture. Their comments have significantly improved the clarity of the paper.

References

[1] Andrew Read, YouTube video demonstrations
https://www.youtube.com/channel/UCz_LqPfKT0r2rEID7Av-Chw

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

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

[4] Andrew Read, "Concept and implementation of an extended return stack to enhance subroutine and exception handling in FORTH", in *EuroFORTH*, 2014

[5] Andrew Read, Github open source repository
https://github.com/Anding/N.I.G.E.-Machine

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

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

[8] B. Paysan, “b16-small – Less is More,” in *EuroFORTH*, 2004.

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

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

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

[12] L. H. Seawright, R. A. MacKinnon: “VM/370-a study of multiplicity and usefulness”, IBM Systems Journal, 1979

[13] Andrew S. Tanenbaum, Albert S. Woodhull, “Operating Systems Design and Implementation”, Prentice Hall, 2nd ed., 1997

[14] Dawson R. Engler, “The Design and Implementation of a Prototype Exokernel Operating System”, MIT, 1995

[15] Brad Rodriguez, “Forth Multitasking in a Nutshell”, The Computer Journal #58, 1992

[16] GreenArrays Inc., PolyFORTH Reference Manual, 1986-2012

[17] INMOS Limited, The Transputer Reference Manual, 1988

[18] Digilent Inc, Nexys4 FPGA Board Reference Manual, 2013-2015

[19] Xilinx, Artix-7 FPGAs datasheet, 2014

[20] ARM, ARM Cortex Portfolio, 2014

[21] Parallax Semiconductor, Propeller P8X32A datasheet, 2011

[22] GreenArrays Inc, GA144A12 chip reference, 2011

[23] David Patterson and John Hennessey, “Computer Organization and Design, Fifth Edition: The Hardware/Software Interface”, 2013

[24] Atmel AVR 8-bit and 32-bit microcontrollers datasheet, 2015

[25] Open Network Forth Control System for the Munich Accelerator Facility, Ludwig Roher, Heinz Schnitter, Egmot Woitzel, at the FORML conference, 1998

[26] Paul E. Bennett IEng MIET, private correspondence, 2015

[27] Michael J. Pont, “The Engineering of Reliable Embedded Systems”, ISBN 978-0-9930355-0-0

[28] Daniel Curtis McCrankin, “The Microcode Level Timeslicing Processor Architecture”, McMaster University, 1988

[29] Control Data Corportation “CDC Cyber 170 Computer Systems; Models 720, 730, 750, and 760; Model 176 (Level B); CPU Instruction Set; PPU Instruction Set”, 1979 - 1981

[30] Xilinx, “Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources”, 2014

Use of Forth to Enable Distributed Processing on Wireless Sensor Networks

Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella and Daniele Peri
DICGIM, University of Palermo - Viale delle Scienze, Ed. 6 - 90128 Palermo, Italy
{salvatore.gaglio, giuseppe.lore, gloria.martorella, daniele.peri}@unipa.it

***Abstract*—Wireless Sensor Networks (WSNs) are composed of tiny sensor nodes able to monitor environmental conditions. Existing applications for WSNs usually adopt a centralized approach that exploit sensor nodes just for sensing, while data processing takes place on more powerful base stations. This can be considered a consequence of the common WSN programming practice that proves too rigid to support development based on distributed processing. In fact, local processing of complex data, such as symbolic information and rules, is an under explored aspect. The adoption of high level interpreters above general purpose operating systems is often unpractical since it implies the saturation of the available resources. In this paper, we detail the implementation of an alternative Forth-based approach that implements a minimal but extensible operating system featuring common WSN functionalities as well as advanced skills such as symbolic distributed processing. We show the definition of words and syntactic constructs that enable collaborative processing on WSNs and ease the development of complex applications even on resource constrained WSN nodes. To this purpose, our approach is based on an abstract mechanism enabling nodes to exchange directly Forth code. Cooperative behaviors, introducing dynamic computation into the network, are thus easily implemented, as we show in a few applicative examples. Moreover, using the same mechanism, remote nodes can be effortlessly reprogrammed even after their deployment. Finally, we show how our approach proves to be feasible and advantageous through a comparison, in terms of memory usage, with relevant interpreter-based software platforms for WSNs.**

I. Introduction

Wireless Sensor Networks (WSNs) are composed of tiny wirelessly interconnected sensor nodes that are equipped with a microcontroller, a radio interface subsystem, some sensor devices and an autonomous power supply, usually consisting in batteries [1]. Generally, such devices are characterized by quite constrained resources in terms of energy, communication and processing capabilities.

WSNs represent a very active research area as several applications have been proposed in literature in several contexts such as biomedical, healthcare, military, industrial and environmental fields [2].

The development of high level applications is typically supported by general purpose operating systems for WSNs such as Contiki and TinyOS [3], which primarily focus on reducing power consumptions while optimizing resource usage [4].

Mainstream programming practices involve the cross-compilation of specialized code with the thin layer operating system of choice, and the subsequent code uploading to the on-board ROM memory. Any modifications in the source code lead to retrace the same steps afresh.

Such practice strongly limits the development of more advanced applications than the static acquisition and transmission of sensory data that is then to be processed by a base station [5].

Sophisticated applications, such as those concerning Ambient Intelligence (AmI) scenarios, could instead be developed if the nodes were able to process cooperatively more complex data –e.g. symbolic data and rules– than the numerical values in rigid representations resulting from sensing. Such applications may in fact implement intelligent, autonomic, and self-organizing behaviors by distributed processing of symbolic and qualitative description of the observed phenomena. However, due to memory constraints of the available development methodologies, such kind of applications are too complex to be implemented on WSNs without recurring to centralized or Cloud-based infrastructures [6].

In order to give the network some adaptivity to changes of the environment as well as of the application goals after the nodes have been deployed, alternative WSN application development tools are thus strongly required [7].

To overcome the inflexibility of conventional programming methodologies, several interpreters targeting resource constrained Wireless Sensor Network (WSN) nodes have been presented in literature [8], [9], [6], [10]. Their primary goal is to support the application development as well as the retasking of already deployed nodes. However, node reprogramming affects just the application code, while the hardware-abstraction layer modifications require to upload the whole binary image or to replace just the modules to be updated [11].

In general, high-level language interpreters are designed as applications running atop the chosen general purpose operating system. Unfortunately, such a strategy dramatically increases the processing load on the on-board microcontrollers, and detaches the application from the hardware. Moreover, this solution often leads to high memory occupation that leaves insufficient memory resources to develop not trivial applications [8].

The choice of Forth in WSN AmI applications, which are characterized by realtime and resource constraints, seems thus quite natural and desirable [12]. Moreover, the interactive nature of Forth makes it easy to face the challenges of AmI development with experimental programming.

In this paper, we detail our experimentation on the use of Forth on WSN nodes as an operating system and development tool. We describe our ongoing implementation of a

Forth-based software platform that provides nodes with basic WSN capabilities such as networking, sensing, and actuation, accessible through expressive words, and easily extensible to support complex functionalities.

Distributed processing is one of the key goals of our platform that we addressed with a simple abstract mechanism based on the transmission of Forth code among nodes, even already deployed ones. In the next sections, we show how we have been able to implement this abstraction in a few dozen words, with a remarkably low resource usage with respect to other available interpreters.

The remainder of the paper is organized as follows. Section II details the wordset we have implemented to use Forth as an operating system for WSNs. In Section III, the primitives supporting distributed processing are presented. Section IV describes some working applications running on WSN nodes in order to demonstrate the feasibility of our approach and finally Section V reports our conclusions.

II. Forth as an Operating System for WSNs

Forth naturally provides an interactive environment with most of the functionalities of an operating system for common computers. In the case of WSN nodes the OS responsibilities include the management of networking as well as all the various on-board and optional sensors and devices.

Most WSN nodes –referred to as *motes* in the specific literature– are based on MCU with Harvard architecture with separate memories for data and programs. Several interfaces, e.g. digital I/O, analog inputs, I2C, SPI and UARTs enable the connection with external modules, such as the radio subsystem, sensing boards, and so forth. For instance, the IrisMote platform that we used as a testbed, which is one of the most adopted, especially for research purposes [13], is equipped with an IEEE 802.15.4 compliant radio transceiver, 128 KB of Flash memory, 8 KB of static RAM and a 4 KB EEPROM, and can be expanded with sensing and prototyping boards.

At the beginning of our experimentation, we sorted out all the available Forth environments targeting the AVR microcontroller used in the IrisMote platform. We chose AmForth [14], a simple indirect threaded code interpreter, as it proved mature enough to be used as a development tool, and as it also provided a usable interactive shell through a serial terminal. However, AmForth did not include natively the support for any WSN platform. This required us to patch AmForth for the IrisMote to include specific configuration settings, such as those concerning ports, clock generators, on-board radio registers, timers and so on.

In our efforts to build an operating system for WSNs we defined an essential collection of definitions for the basic functionalities needed by WSN applications, such as networking and sensing.

Not all the definitions are strictly related to the hardware. Instead, we defined some more generic and platform-independent words that are not tied to specific hardware implementation and can thus easily work on different platforms.

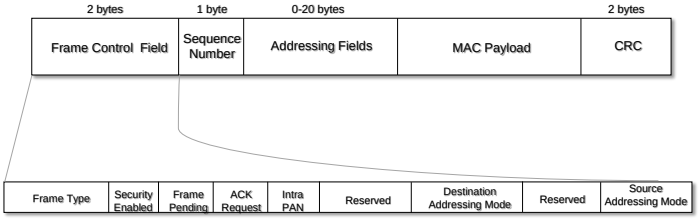


Fig. 1: Format of a valid MAC layer frame according to the 802.15.4 standard. The frame control field is detailed in the bottom of the figure.

As an example, hardware independent words are those used to create valid data frames according to the 802.15.4 standard. In our implementation, transmission and reception of valid 802.15.4 frames is based on two buffers:

- outbound: a memory area where the outgoing frame is stored before downloading it to the radio frame buffer for the transmission;
- inbound: a memory area where the received frame is stored after it is uploaded from the radio frame buffer.

The buffers are 128 bytes long. According to the 802.15.4-2003 standard (see Figure 1), we defined the words to create valid data frames and to set the frame fields appropriately, e.g. short/long destination addressing mode field, frame type, frame length, and so on. In particular, Listing 1 shows the word definition to create a default frame with the following settings:

- short addressing mode (source and destination);
- intra-pan bit set to 1;
- 0xabcd pan address;

Listing 1: Forth word to create a valid 802.15.4 data frame with fixed settings

```
: default-pkt ( -- packet )
outbound dup erase
data frame_type
pan_compr
dest short mode! src short mode!
dest pan $abcd s_addr! src addr id @ s_addr! ;
```

A. Forth Words for Input Redirection to the Radio Module

Our Forth-based implementation supports interactive development on already deployed devices. This feature permits adding new words on remote nodes even if they are not physically connected to a serial terminal. Interactivity, symbolic processing and executable code exchange are the pivotal characteristics of our system.

The code exchanged among nodes and received from the radio channel is interpreted by the system, provided that the default input –the USART, at boot– has been redirected to the radio transceiver. Each incoming frame triggers the interrupt invoking the text interpreter on the frame payload.

Listing 2 shows the word definitions to enable the interpretation of incoming frames from the radio channel, by switching the input from the USART to the radio.

Listing 2: Forth words to redirect the standard input device to the radio

```
variable old-key
variable old-key?
variable payld-addr
variable payld-size
variable payld-in
$200 constant usart_rx_in
$201 constant usart_rx_out
$202 constant usart_rx_data

: payld-reset
    0 payld-size !
    0 payld-in ! ;

: payld-set ( addr n -- )
    payld-size !
    payld-addr !
    0 payld-in ! ;

: radio-key?
    payld-size @ dup 0 > swap
    payld-in @ > and ;

: radio-keyin
    payld-in @ dup payld-addr @ +
    c@ swap 1 + payld-in ! ;

: radio-key
  begin pause radio-key? until radio-keyin ;


: +radio-input
  payld-reset
  ['] key defer@ old-key !
  ['] key? defer@ old-key? !
  ['] radio-key is key
  ['] radio-key? is key? ;

: -radio-input
  old-key @ is key
  old-key? @ is key? ;

: usart_inject
  usart_rx_in c@ usart_rx_data + !
  1 usart_rx_in +! ;
```

Essentially, the input redirection makes the deferred words `key?` and `key` point to `radio-key?` and `radio-key` respectively. The word `radio-key?` is used to assess if there are unread characters in the frame payload by checking either if the variable `payld-size` is greater than 0 and the current pointer to the payload `payld-in` is lower than `payld-size`. The word `radio-keyin` fetches the next character in the frame payload and advances the current payload pointer `payld-in`. Finally, the word `radio-key` executes `radio-key?` and `radio-keyin` until all the characters in the frame payload have been read. To redirect the input to the radio, the word `+radio-input` is typed in the node shell. The execution causes the AmForth shell to be lost, until a data frame containing the word `-radio-input` is received. This word restores the input to the USART.

Code processing takes place directly in the interpreter loop as the last character of each incoming frame is required to be a carriage return. Such an event triggers the interpretation of the payload. However, in real use, interacting with networked devices through a wired line is unpractical. In fact, to redirect the input to the radio system without any wired connection, we defined a special frame containing just the character $17, which is the ASCII code for the non-printable character ETB.

Once a frame is received, the node uploads it from the radio frame buffer and checks whether the frame payload is equal to ETB. If so, it executes `+radio-input`. Actually, to switch the input, the word `usart_inject` must be executed to exit the system blocking loop waiting for characters from the USART that in the current AmForth implementation cannot be preempted in other ways.

B. Support to the Radio Operations

In order to support the communication among nodes, we defined a number of words to drive the radio of IrisMotes. The low-power AT86RF230 transceiver [15] is connected to the master SPI interface of the microcontroller and to additional control signals, i.e. IRQ and GPIO signals. Essentially, the SPI is used for frame buffer and register access operations, according to the SPI protocol. Although AmForth already provides the words `spi!` and `spi@` for writing and reading a character on the SPI bus, further efforts were needed to configure ports for the specific target device.

We also defined word sets to support the functional specification of the radio device. For instance, in Listing 3 the words `reg_rd` and `reg_wr` specify the operations to be undertaken for reading and writing the radio registers. Similarly, we defined the words `to_framebuf` and `from_framebuf` to upload incoming frames, and download outgoing frames, respectively. Word choices reflect the nomenclature of the radio datasheet.

Uninterruptible code, such as that implementing SPI operations, is enclosed within critical sections. The words `ss_l` and `ss_h` set the SS line of the SPI interface respectively low and high.

Listing 3: Some words of the radio driver

```
: reg_rd ( register_address -- register_value )
      reg_addr_mask and reg_rd_command or
      critical[
      ss_h ss_l spi! spi@ ss_h
      ]critical
;

: reg_wr ( register_value register_address -- )
      reg_addr_mask and reg_wr_command or
      critical[
      ss_h ss_l spi! spi! ss_h
      ]critical
;

: to_framebuf ( packet_to_send -- packet )
      dup critical[
      ss_h ss_l framebuf_wr_command spi! length spi!
      dup length 0 ?do dup I + 1 + c@ spi! loop
      ss_h ]critical
;

: from_framebuf ( packet -- packet )
      critical[
      ss_h ss_l framebuf_rd_command spi!
      spi@ over c! dup length 0 ?do
      spi@ over I + 1 + c! loop ss_h ]critical
;
```

The radio transceiver operating modes and its transitions can be represented by the state diagram in Figure 2.

To permit a plain alignment between specifications and implementation, the same diagram can be completely ported into Forth definitions as shown in Listing 4, which includes just a restricted number of defined words.

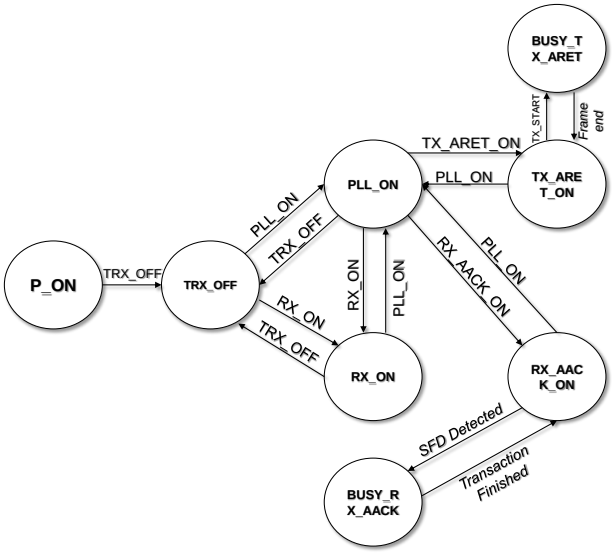


Fig. 2: Part of the state diagram representing the set of operating modes of the AT86RF230 according to the datasheet [15]. The label on the arrows represents the command that writes to the TRX_STATUS register causing the transaction to another state. Events are indicated with labels in italics.

Listing 4: Definitions for radio operating modes and transitions

```
: pll_on ( -- )
      pll_on_state cmd-wr  ;

: trx_off ( -- )
      st-reset ;

: tx_start ( -- )
      tx_start_cmd cmd-wr ;

: rx_on ( -- )
      rx_on_state cmd-wr  ;

: rx_aack_on ( -- )
      pll_on rx_aack_on_state cmd-wr ;

: tx_aret_on ( -- )
      pll_on tx_aret_on_state cmd-wr ;

: transmit ( packet -- )
     -int IRQ low
      idle? if green led blink else
      trx_off outbound process-tx drop to_framebuf drop
      tx_aret_on tx_start
      then +int
;

: switch-input? ( inbound -- )
      dup payld addr nip c@ $17 = if
      +radio-input $17 usart_inject
      then
;

: received ( inbound -- )
      trx_off
      from_framebuf dup process-rx switch-input?
      payld addr nip swap payld size payld-set
;
```

In particular the words `process-rx` and `process-tx` are deferred words that may be used to process `inbound` and `outbound` buffers. A possible use may be for encryption and decryption purposes. The last three definitions implement frame transmission, input redirection and frame reception. Transmission and reception of frames are signaled by interrupts on the Timer1 Input Capture Trigger. The Interrupt Service Routine in AmForth is also a defined word. Therefore, we defined a word acting as the handler routine and we stored its address as interrupt vector. Our interrupt handler routine reads the IRQ STATUS register and acts as a dispatcher. Since the AT86RF230 differentiates between six interrupt events, it calls the appropriate interrupt handler, according to the interrupt source. For instance, the interrupt generated by either a frame transmission/reception causes the execution of the word `trx_end_isr`. If a correct transmission triggers the interrupt, the radio enters the `rx_aack_on` state, otherwise the frame is downloaded to the `inbound` buffer.

```
: trx_end_isr
      red led blink state?
      tx_aret_on_state = if
      else inbound received
      then trac_status trac !
      rx_aack_on ;
```

C. Supporting Sensing and Actuating Tasks

The acquisition of sensory data is the main functionality of WSN nodes. Typically, expansion boards are required to provide the nodes with several sensors simultaneously. As in the case of the radio transceiver driver implementation, we have extended the WSN node dictionary with a number of words to drive sensor boards. Moreover, word sets composed of high level words enable the data sensory acquisition through the different available sensors. For instance, a program to make a node sense the temperature may consist of the single word `temperature` that leaves at the top of the stack the required sensory value. Similarly, the word `luminosity` activates the light sensor, puts the sensory reading atop the stack, and finally disables the sensor. Although the code is concise and expressive, the execution of these words involves low level aspects as reading from the ADC and returning the raw data on the stack. However, we choose high level word names to make the description of a task in natural language and the implementation as similar as possible. The words we defined to support sensing tasks are summarized in Table I.

WSNs may also include some actuator nodes to change the environmental conditions. An IrisMote can behave as an actuator when connected, for instance, to the MDA300 expansion board that includes two relays, one of which normally opened and the other one normally closed. We defined words to drive the relays and developed a light control application by connecting a LED to the expansion board, as detailed in Section IV.

III. A Forth-based Approach to Enable Symbolic Distributed Processing for WSNs

To implement sophisticated AmI applications, even resource constrained nodes may need to exchange complex information that is not rigidly structured and that may differ from numerical values such as symbolic descriptions and rules. Conventional programming methodologies impose to define in advance the format of the message to be exchanged as well

TABLE I: Summary table of words used for sensing tasks. Words are indicated together with their "stack effect"

Word name	Description
temperature (-- temp_value)	Measure temperature and push the numeric value onto the stack
luminosity (-- light_value)	Measure light and push the numeric value onto the stack
mic (-- mic_value)	Measure sound level and push the numeric value onto the stack
accx (-- accx_value)	Measure the acceleration along the X axis and push the numeric value onto the stack
accy (-- accx_value)	Measure the acceleration along the Y axis and push the numeric value onto the stack
+sounder (--)	Activate the buzzer
-sounder (--)	Disable the buzzer

as to fix the packet fields where given information must be placed. To overcome this rigidity, interpreters targeting WSN nodes, such as Maté [9], T-RES [6] and TakaTuka [10], have been presented. However, such solutions are based on bytecode transmission and interpretation. Not only the source code expressivity gets lost as the source code is translated into bytecode but also the translation process is, in all effects, a cross-compilation.

In order to retain expressiveness without sacrificing compactness, we let our nodes able to directly exchange and execute Forth code. Indeed, we implemented an abstract mechanism to handle the transmission of code among nodes, and from the terminal shell to nodes. The implementation cost of such an abstraction is quite low in Forth and, at the same time, the support to distributed applications is straightforward, as shown in Listing 5.

Listing 5: Forth words for executable code exchange

```
variable current_pay
variable nest
variable current_buf
variable buf $80 cells allot

: 2dup over over ;
: buf-reset buf current_buf ! ;
: pay-reset outbound payld addr nip current_pay ! ;

: (write) \ i*x addr len  dest_addr -- j*y
 swap cmove
;


: num>str ( number -- string_addr string_len )
      hex 0 <# #s [char] $ hold #> ;

: space+ ( pay_ptr -- pay_ptr+1)
      bl p+
;

: cr+ ( pay_ptr -- pay_ptr+1)
      $0d p+
;

: nest+ ( -- )
      nest @ 1 + nest ! ;
: nest- ( -- )
      nest @ 1 - nest ! ;

: [tell:]? ( addr len -- f )
      s" [tell:]" icompare ;

: [:tell]? ( addr len -- f )
      s" [:tell]" icompare ;

: tell:? ( addr len -- f )
      s" tell:" icompare ;

: :tell? ( addr len -- f )
      s" :tell" icompare ;


: >buf ( addr1 n -- )
      dup >R current_buf @ dup >R (write)
      R> R> + space+ current_buf ! ;

: >pkt ( addr1 n -- )
      dup >R current_pay @ dup >R (write)
      R> R> + space+ current_pay ! ;

: c>pkt ( value -- )
      current_pay @ swap ( c@ -- ) p+ current_pay ! ;

: char>buf ( value -- )
      current_buf @ swap over c! 1 + current_buf !
;


: subst
      0 do buf I + c@ dup ( c@ -- )
      case
      $0e of drop num>str >pkt endof
      $0f of drop num>str >pkt endof
      $10 of drop >pkt endof
      c>pkt
      endcase loop
;

: [endtell] ( flash-addr flash-count -- )
      dup >r buf imove r>
      subst outbound dup current_pay @ cr+
      endpayld transmit
      pay-reset
;

: endtell ( buf buf-len -- )
      nip subst outbound dup current_pay @ cr+
      endpayld transmit
      pay-reset
;

: subst? nest @ 0 = if
         2dup s" ~" icompare if drop drop $0e true else
         2dup s" ~~" icompare if drop drop $0f true else
         2dup s" ~s" icompare if drop drop $10 true else
         drop drop 0 then then then
         else drop drop 0 then
;


: parse-tell ( -- buf buf-len )
   buf-reset
   begin bl word count
        2dup :tell? if
          nest @ 0 > if nest- >buf 0
                   else true
                   then
        else
        2dup tell:? if nest+ >buf 0
        else 2dup [tell:]? if nest @ 3 + nest ! >buf 0
        else 2dup [:tell]? if nest @ 3 - nest ! >buf 0
        else 2dup subst? if char>buf drop drop 0
        else >buf 0
        then then then then then until drop drop
        buf current_buf @ over -
```

```
;

: [parse-tell]
   buf-reset
   begin bl word count
         2dup [:tell]? if
           nest @ 0 > if nest @ 3 - nest ! >buf 0
                    else true
                    then
         else
         2dup [tell:]? if nest @ 3 + nest ! >buf 0
         else 2dup tell:? if nest+ >buf 0
         else 2dup :tell? if nest- >buf 0
         else 2dup subst? if char>buf drop drop 0
         else >buf 0
         then then then then then until drop drop
         buf current_buf @ over -
;

: reply ( -- dest_addr)
      inbound src addr @ nip ;
;

: pkt-init 0 nest ! outbound erase
      default-pkt dest addr rot s_addr! drop
      pay-reset
;

: tell:
     pkt-init parse-tell endtell
;


: [tell:]
      postpone pkt-init
      [parse-tell] postpone sliteral
      postpone [endtell]
; immediate
```

Our programming environment and experimental setup is composed of some nodes wirelessly deployed and a wired node that behaves as a bridge to send user inputs to the network.

The syntactic construct for the code exchange among nodes is based on the word `tell:` that parses the input until `:tell` is encountered and sends a default data frame, according to IEEE 802.15.4 standard, to the node holding the MAC address placed on top of the stack.

To tell all the nodes in the radio range to turn their green LED on, a simple line of code is all that it needs to be typed on the bridge node shell:

```
bcst tell: green led on :tell
```

As a consequence, the microcontroller on the bridge node interprets the text typed by the user and creates a default data frame with the broadcast address as destination, containing the program to be sent, `green led on`, as payload.

A recursive usage of code exchange, through nested `tell:` *<code>* `:tell` constructs, permits commands to hop form one device to another before reaching the final destination, as shown in Figure 3. From a mere semantic standpoint, the sense is *"to tell a node to tell another node to do something"*.

The code to be remotely executed may contain syntactic placeholders that are substituted at runtime with the content of the top of the stack using a hexadecimal representation. For the sake of clarity, our implementation consists in a two pass parsing process. An intermediate substitution of such special markers takes place in the first pass, while the items on top of

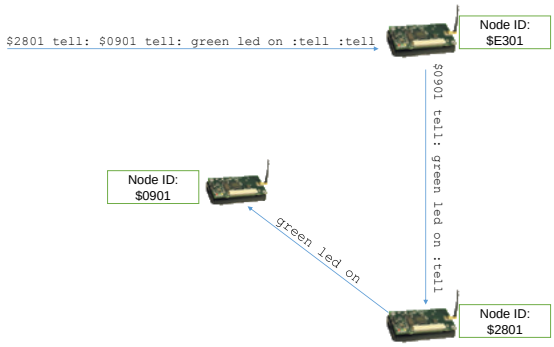


Fig. 3: Recursive employment of the `tell: <code> :tell` `primitive`. Once the node with ID E301 encounters the first `tell:` it parses all the following symbols until the last `:tell` and a frame containing `0901 tell: green led on :tell` is sent to node with ID 2801. The payload interpretation of the payload on the receiving side leads to the sending of a new frame destinated to node 0901 with `green led on` as payload. Once received, node 0901 turn its green LED on.

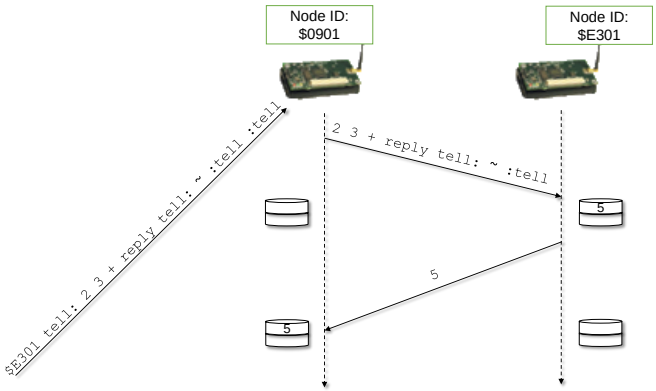


Fig. 4: The node with address $0901 interprets the code it receives and tells the node with address $E301 to perform the sum between 2 and 3, and then to reply with the value on top of its stack. Even though the reply message consists only in a literal value, it is interpretable Forth code and it is simply executed by node $0901 leaving 5 on top of its stack.

the stack definitively replaces placeholders during the second pass.

Such special markers are:

- ~ for a single cell value
- ~ ~ for a double cell value
- ~s for strings

Instead of implementing state-smart words for code exchange [16], we defined the compile-time construct `[tell:]`*<code>*`[:tell]`.

An example of code exchange between two nodes is described in Figure 4. Incorporating such high level abstraction on resource constrained devices leaves plenty of room for the development of WSN applications that natively support distributed processing. The word sets composing our software platform are reported in Figure 5 along with their size in terms

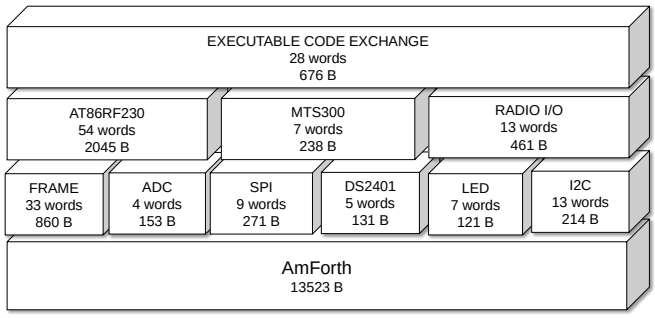


Fig. 5: A comprehensive view of the main word sets we defined and that compose our software platform. For each word set, the number of words and the Flash usage in bytes are indicated.

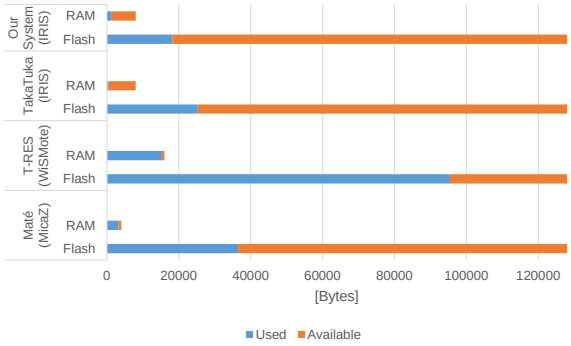


Fig. 6: Memory footprint of our software platform along with some representative interpreter-based architectures.

of number of words and Flash memory occupation.

Besides providing an on-board interpreter that does not need cross-compilation, our approach compares favorably with the aforementioned interpreter-based architectures with respect to memory usage, as we assessed with tests in our experimental setup. Where possible, as for TakaTuka and Maté, we compiled the software platforms for the IrisMote or for the quite similar MicaZ hardware. T-RES, instead, only runs on the WiSMote hardware platform.

Results, reported in Figure 6, confirm that the implementation of the interpreter above a general purpose operating system occupies much of the available memory, as in the case of Maté and T-RES. As scripts are stored in RAM, not enough space is left for the development of complex applications, even in the presence of the double-sized RAM of WiSMotes. Our Forth-based approach, instead, compactly keeps application code in the relatively abundant Flash, while RAM just holds temporary data as variable values, buffers and stacks.

More in detail, the memory footprint of all the platform word sets is 5170 bytes of Flash, as reported in Figure 6, and 1026 bytes of RAM. Including the underlying AmForth, the overall footprint of our platform is 18693 bytes of Flash memory and 1321 bytes of RAM memory.

IV. Application Development on WSNs

We have developed different applications for WSNs to test both our approach and our software platform. As a first step, we designed and implemented a working telnet-like remote shell on the bridge node to be actually used as a development tool [17]. Using the remote shell application on the bridge node through a serial terminal, the programmer can interact with a remote node that is reachable by the bridge node.

Besides debugging and node reprogramming, this application can serve different purposes such as the inspection of the state of a remote node or the acquisition of sensory readings as if the remote node were physically connected through the serial line.

The code is fully functional, and has been extensively used in our experimentations. We defined a number of words to redirect the output to the outgoing message, to display incoming messages from the inspected node, and to implement the remote shell loop. The resulting implementation is quite readable and understandable. The almost complete remote shell application code is shown in Listing 6. Although few additional words are omitted, their description can be found in Table II.

Listing 6: Code for a simple remote shell application

```
80 constant cmd-maxlen
variable cmd cmd-maxlen cells allot
variable cmd-len
variable node_id
variable timeout

: input-send ( -- )
    cmd cmd-len @
    node_id @ [tell:] ~s [:tell] ;

: rshell-task (  -- )
    payld-reset
    input-send
    timeout @ wait-answer if
    payld-print then ;

: user-input ( -- )
    cmd cmd-maxlen accept ( -- len )
    cmd-len ! ;

: close ( -- )
    node_id @ [tell:] -radio-output
    [:tell] quit ;

: rshell-loop ( -- )
    begin
     cr ." rsh>" user-input
     close?
     if close
     else rshell-task
     then
    again ;

: on-timeout ( -- )
    ." Connection timeout." cr ;

: welcome-msg
    ." Welcome to the remote shell
 application!" cr
    ." Enter 'close' to close the
 application" cr ;

: rshell ( id -- )
    welcome-msg
    2000 timeout  !
    dup node_id !
    [tell:] +radio-output [:tell]
```

```
    rshell-loop  ;
```

A desirable use of WSN nodes is monitoring the environment to react to undesired events. A role-based working implementation differentiates the words defined on remote nodes on the basis of their role in the network.

For instance, the actuator node dictionary could include words to trigger an alarm if the luminosity value exceeds a predefined threshold. A node provided with a light sensor may regularly perform the luminosity measurement and tell its neighbor to forward it to the actuator.

Such words explicitly make use of the syntactic construct for distributed code exchange. However, the designer may interactively set the topology, the threshold, the actuator node and may start the event detection application by interacting with the bridge node shell.

The code to implement such an application is provided in Listing 7 and consists in few words defined on the three kinds of nodes. With respect to the baseline of our platform, the increment of RAM usage on the sensor node for the application is just 6 bytes, while additional 156 Flash bytes are required to store the word definitions for timer3 management and the application. Since the routines for timer3 are not needed on the forwarder and actuator nodes, Flash and RAM increments for both are just 86 and 4 bytes respectively.

More importantly, the overall Flash and RAM memory footprint of the application and the software platform, even considering 21 additional bytes for the turnkey definitions, is 18714 bytes of Flash memory and 1321 bytes of RAM memory. Such result is even lower than the baselines of the other platforms –that is without any application– as can be deduced from Figure 6.

Listing 7: Distributed event detection application on WSN nodes

```
\ Defined on the sensor node
variable neighbor
variable threshold

: luminosity-check
        luminosity
        threshold @ > if
        neighbor @ [tell:] alarm [:tell]
        then ;

: light-monitoring
        ['] luminosity-check
        5seconds timer3.init timer3.start ;

\ Defined on the forwarder node
\ and on the actuator node

variable actuator
variable neighbor

: same ( -- outbound)
        outbound inbound over length copy ;

: message ( dest_addr outbound --outbound)
         dest addr rot s_addr! ;

: propagate
        transmit ;

: actuator?
        actuator @ 1 = ;

: alarm
        actuator? not if
        neighbor @ same message propagate
        else +sounder 1000 ms -sounder
        then ;
```

Furthermore, instead of an alarm, the actuator may directly switch the light off once the luminosity exceeds the threshold value through a redefinition of the word `alarm` (Listing 8).

Listing 8: Redefinition on the actuator node of the word that triggers the alarm

```
: alarm
        actuator? not if
        neighbor @ same message propagate
        else     light off then ;
```

In previous work [18] we have also showed how to support smart applications that exploit symbolic reasoning. We enriched a Forth formalism for Fuzzy Logic by VanNorman [19] with the possibility to exchange definitions and evaluations among nodes. Instead of reasoning about crisp values, resource constrained nodes process the fuzzy variables `temp` and `lightexp` that can be easily defined on deployed nodes. Further words such as `fvar` define the related membership functions. By exploiting executable code exchange, a fuzzy variable definition can be easily distributed among nodes even after their deployment. After defining the membership functions `lightexp.low`, `lightexp.medium`, `lightexp.high` and `temp.low`, `temp.medium`, `temp.high`, the following code makes a node measure and fuzzify light exposure:

```
lightexp measure apply
```

while the code:

```
lightexp.low @
```

pushes onto the stack the truth value resulting from the fuzzification phase. For instance, rather than through a thresholding process, a device can establish if it is close to the window through the evaluation of fuzzy rules in the form:

```
temp.high @ lightexp.high @ &
=> close-to-window
```

A node can request the others to update their fuzzy temperature values as follows:

```
bcst temp fvar-remote-update
```

The word `fvar-remote-update` evaluates `temp` and broadcasts a message containing the Forth code to update the three membership values. The frame payload includes a repetition of the structure:

```
<truth> <membership func> fvar-update
```

for each membership function of the argument fuzzy variable. When a node receives the message, it interprets the command updating the truth values of its local membership functions. The combination of symbolic reasoning with executable code exchange makes even resource constrained devices able to process and exchange qualitative information about the physical phenomenon.

TABLE II: Summary table of additional words used in the remote shell application

Word (before -- after)	**Description**
+radio-output (--)	Redirect the output to the radio. This word is part of the Radio I/O word set
-radio-output (--)	Redirect the output to the UART. This word is part of the Radio I/O word set
radio-input? (timeout -- flag)	Check for incoming radio messages. If no message arrives before `timeout` milliseconds leave false on the stack, otherwise leave true
payld-reset (--)	Set to 0 the incoming payload length and its current pointer
payld-print (--)	Display the incoming frame content (i.e. the payload)
wait-answer (timeout -- flag)	Wait for incoming radio frame for a predefined period of time specified by the `timeout` variable. If the timeout expires without receiving any answer message, an exception handled by `on-timeout` occurs
user-input (--)	Wait for user input and store its content in the `cmd` buffer and its length in the `cmd-len` variable for further processing by `close?` and `input-send`

V. Conclusions

As remarked in literature, common programming methodologies for WSNs lack proper programming abstractions for the development of distributed applications. The standard practice consists in linking the application, written in C-derived programming languages, with a general-purpose operating system at the end of a cross-compilation process. All this proves rigid and time consuming. To overcome these limitations, the adoption of interpreters for high-level languages to be run on established operating systems has been proposed. Nevertheless, existing approaches consist in several software layer implementations that collide with the resource constraints of nodes.

In this paper, we detailed the implementation of an alternative Forth-based approach that implements a minimal but extensible operating system featuring common WSN functionalities along with symbolic distributed processing through executable code exchange. The Forth-based software platform we have implemented is quite compact. Indeed, we showed how a symbolic distributed AmI event detection application can be implemented with a total memory usage that is less than the mere baselines of relevant interpreter-based software platform for WSNs. In further experimentations we will compare our Forth environment to other existing interpreter-based platforms for WSNs in terms of efficiency, interpretation overhead and energy consumption.

References

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communication Magazine*, vol. 40, no. 8, pp. 102–114, 2002.

[2] E. Gilbert, K. Baskaran, and E. E. Blessing, "Research Issues in Wireless Sensor Network Applications: A Survey," *International Journal of Information and Electronics Engineering*, vol. 2, no. 5, pp. 702–706, 2012.

[3] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011.

[4] A. M. V. Reddy, A. P. Kumar, D. Janakiram, and G. A. Kumar, "Wireless sensor network operating systems: a survey," *Int. J. Sen. Netw.*, vol. 5, no. 4, pp. 236–255, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1504/IJSNET.2009.027631

[5] L. M. Oliveira and J. J. Rodrigues, "Wireless Sensor Networks: a Survey on Environmental Monitoring," *Journal of communications*, vol. 6, no. 2, pp. 143–151, 2011.

[6] D. Alessandrelli, M. Petracca, and P. Pagano, "T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs," in *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*, May 2013, pp. 337–344.

[7] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922656

[8] L. Evers, P. Havinga, J. Kuper, M. Lijding, and N. Meratnia, "SensorScheme: Supply chain management automation using Wireless Sensor Networks," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, Sept 2007, pp. 448–455.

[9] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," in *ACM Sigplan Notices*, vol. 37, no. 10. ACM, 2002, pp. 85–95.

[10] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rhrup, and Z. Uzmi, "Optimized Java Binary and Virtual Machine for Tiny Motes," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, R. Rajaraman, T. Moscibroda, A. Dunkels, and A. Scaglione, Eds. Springer Berlin Heidelberg, 2010, vol. 6131, pp. 15–30.

[11] W. Munawar, M. Alizai, O. Landsiedel, and K. Wehrle, "Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks," in *Communications (ICC), 2010 IEEE International Conference on*, May 2010, pp. 1–6.

[12] B. Watts, "FORTH, a Software Solution to Real-time Computing Problems," *Behavior Research Methods, Instruments, & Computers*, vol. 18, no. 2, pp. 228–235, 1986.

[13] "Iris Datasheet," 2013, available online at http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf.

[14] "Amforth documentation," 2013, available online at http://amforth.sourceforge.net/amforth.pdf.

[15] "At86rf230 datasheet," 2013, available online at http://www.atmel.com/images/doc5131.pdf.

[16] M. A. Ertl, "State-smartness— Why it is Evil and How to Exorcise it," *EuroForth98*, 1998.

[17] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Sept 2014, pp. 1–8.

[18] S. Gaglio, G. L. Re, G. Martorella, and D. Peri, "High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices," in *Accepted at The First International Conference on Cognitive Internet of Things (COIOTE 2014)*, October 2014.

[19] R. VanNorman, "Fuzzy Forth," *Forth Dimensions*, vol. 18, pp. 6–13, Mar. 1997.

A Forth-Simulator of Real-Time Multi-Task Applications

Sergey Baranov
St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences (SPIIRAS), ITMO University[1]
SNBaranov@gmail.com

Introduction. Software applications for real-time systems (RTS) are usually built as cooperative complexes of communicating *tasks* $\tau_1, \tau_2, \ldots, \tau_n$, which share common computational and informational resources and whose behavior is impacted by *system events*, occurring from time to time according to a particular *scenario*. Each task is a sequential program, closed in itself with respect to control flow, and is activated in response to external events within some timing intervals not less than some value called its period and is expected to elaborate some response as a result of its activation and the following run. A j^{th} activation of the task τ_i ($1 \le i \le n$) means generation of a j^{th} *instance* of this task τ_i; i.e., a respective *job* denoted as ${}_j\tau_i$ for subsequent execution. When this execution terminates, it means that a respective result has been provided in response to the system event which caused activation of this particular job.

A characteristic feature of an RTS is the requirement for *on-time execution*, usually expressed as a requirement that for each task τ_i the longevity $r({}_j\tau_i)$ of any of its jobs ${}_j\tau_i$ shall not exceed some pre-defined deadline value D_i: $\forall i,j\ r({}_j\tau_i) \le D_i$. With the notion of the task *response* time $R_i = \max\{r({}_1\tau_i), r({}_2\tau_i), \ldots\}$ this may be reformulated as $\forall i\ R_i \le D_i$ with any allowable scenario of system events and is often interpreted as the property of *feasibility* of the given multi-task application. To check application feasibility, various *structural models* of its tasks are built and analyzed to provide reliable estimates for the response times of the application tasks, taking into account all impacting factors.

Software simulation is an acknowledged method to check feasibility of real-time multi-task applications. This paper describes an experience of constructing such simulator in Forth with the VFX Forth for Windows [1] as a development platform. Forth was selected as the implementations language due to the flexibility it provides for implementing programming solutions. The simulator employs a simple model of a multi-task application under study which may use several *scheduling modes* with various task priorities for allocation of the processor computational resource and several *access protocols* to access shared informational resources. The simulator helps to study multi-task application behavior and check whether a given combination of the scheduling mode and access protocol guarantees application feasibility under the given processor performance and system event scenarios. It may also identify the minimal processor performance which still ensures application feasibility under the given conditions.

By now, the nomenclature of scheduling modes and access protocols implemented in the simulator consists of two classical scheduling modes – RM (*rate monotonic*) and EDF (*earliest deadline first*) – and three access protocols – NI (*no inheritance*), BI (*basic inheritance*), and PI (*priority inheritance*). However, it may be further extended to simulate systems with other scheduling modes on a multi-processor and/or multi-core platform and other protocols of access to shared informational resources [2].

Source Data. Simulation is based on components of four kinds: *resources*, *tasks*, *jobs*, and *events*. Resources and tasks are entities of the application under study; jobs and events are entities created and operated on by the simulator. Resources and tasks are also represented within the simulator with respective entities. The application is assumed to run on a single processor platform with a certain processor performance P in terms of "the number of standard operations per second", which a particular scaling factor determining the actual processor speed is related to. Each application task τ_i is characterized by its timing period T_i – the minimal timing interval

[1] This work was partially financially supported by Government of the Russian Federation, Grant 074-U01.

between two consecutive activations of τ_i determined by the current scenario of system events, its priority $Prio_i$ – which descends with increase of i, its weight W_i – the amount of processor time needed to accomplish this task, its deadline D_i – the maximal time period for the task to be completed, and its phase Ph_i – the offset of the first activation of this task from the simulation starting moment (by default Ph_i=0). Like the processor performance P, the task weight W_i is specified in the number of standard operations, and may be converted into seconds: $C_i=W_i/P$. Obviously, $\forall i\ C_i \leq T_i$. The values T_i, D_i, and Ph_i are specified in absolute timing units (e.g., seconds) and do not depend on the processor performance P.

Application tasks may access shared informational resources identified with their unique ID numbers; however, at any moment of time a shared resource may be accessed by only one task. Tasks which do not share any informational resources are considered to be *independent* with respect to each other. To prevent simultaneous access of 2 or more tasks to a shared resource, *critical intervals* within the task code are established and guarded with special constructs of the *mutex* type, which is a particular case of Dijkstra semaphores.

With this in mind, the structure of each task τ_i is represented in the simulator as a finite series of *k(i)* segments, each segment performing some computation within a certain period of time S_j>0 and terminating with one of the following system events: "*Lock m*", "*Unlock m*", or "*End*", *m* being the resource ID number. The duration of processing a system event is assumed to be negligibly small. A correct application should neither unlock a resource not locked by this task earlier, nor lock it again without preceding unlocking it, nor leave it locked upon task termination, and each task should terminate with the segment "*End*". Obviously, the task weight W_i equals to the sum of time periods of all its segments: $W_i = \Sigma_{j=1..k(i)} S_j$.

Task τ_1

```
<task name="t_1" prio="1"
      phase="5" period="15">
  <segment length="1"
           interface="m_1"
           op_type="lock"/>
  <segment length="1"
           interface="m_1"
           op_type="unlock"/>
  <segment length="1"
           op_type="end"/>
</task>
```

Task τ_2

```
<task name="t_2"   prio="2"
         phase="5" period="35">
  <segment length="9"
           op_type="end"/>
</task>
```

Task τ_3

```
<task name="t_3" prio="3"
      phase="3" period="25">
  <segment length="1"
           interface="m_1"
           op_type="lock"/>
  <segment length="2"
           interface="m_2"
           op_type="lock"/>
  <segment length="1"
           interface="m_2"
           op_type="unlock"/>
  <segment length="1"
           interface="m_1"
           op_type="unlock"/>
  <segment length="1"
           op_type="end"/>
</task>
```

Task τ_4

```
<task name="t_4" prio="4">
                 period="45">
  <segment length="2"
           interface="m_2"
           op_type="lock"/>
  <segment length="4"
           interface="m_2"
           op_type="unlock"/>
  <segment length="1"
           op_type="end"/>
</task>
```

Fig. 1. Four tasks sharing 2 resources

An example of an application description in an XML-type fashion [3] is provided in Fig. 1. Here are 4 tasks τ_1, τ_2, τ_3, and τ_4, which share 2 informational resources m_1 and m_2. The code of the highest priority task τ_1 consists of 3 segments of 1 time unit each. Its first segment ends with the operation lock for resource m_1; the next segment ends with unlocking this resource and the third segment terminates the task. The code of the task τ_2 consists of the only segment of 9 time

units while task τ_3 consists of 5 segments with two critical intervals to access the resources m_1 and m_2, the intervals being embedded in one another. The least priority task τ_4 consists of 3 segments and accesses only the resource m_2.

Task periods T_1, T_2, T_3, and T_4 for task activations are 15, 35, 25, and 45 time units respectively with the phase shifts 15, 35, 25, and 45; deadlines are assumed to be equal to task periods: $D_i=T_i$. Tasks and resources are rendered by objects of the type *task* and *resource* respectively and are created by respective Forth words during simulator initialization when reading an input file with the task descriptions:

```
: CreateTask ( -- task-addr)
: CreateResource ( n -- resource-addr)
```

Output Data. For each task τ_i the derivative characteristics are defined: its utility load $U_i=C_i/T_i$ and its hardness $H_i=T_i/D_i$ which characterize tasks execution. If $H_i \leq 1$ then the existence intervals of consecutive jobs ${}_j\tau_i$ and ${}_{j+1}\tau_i$ created from two consecutive activations of the task τ_i do not intersect. The reverse condition $H_i > 1$ means that they may intersect. An important metric – the density of the whole application: $Dens=max_P(\Sigma_{i=1..n} U_i)$ – may be calculated too, in order to compare different application structures and implementations on their efficiency [4].

The ultimate purpose of simulation is to obtain data on efficiency of various combinations of scheduling modes and access protocols in various scenarios of system events. In particular, the dual problem to calculating the application density – to determine the minimal processor performance which still ensures the feasibility of the application (i.e., that $\forall i\ R_i \leq D_i$) under given conditions – may be solved as well.

To calculate the application density, the initial interval $[a,b]$ for selecting the scaling factor $f \in [a,b]$ for the task weights and processor performance is established. Prior to the simulator run, the source values of task segment durations S_j (and therefore, the task weights W_i) in task descriptions and the processor performance P are multiplied by this factor. Obviously, if the inequality $R_i \leq D_i$ is violated for some i at the end-values a and b of the interval, it is violated for all intermediate values. However, for $f=a=0$ (which means an infinitely high processor performance) these inequalities do hold for all i. Therefore, the initial values are set to $a=0$ and $b=\Sigma_{i=1..n} U_i$ with the standard processor performance $P=10^6$ standard operations per second. Then the first simulation iteration is performed with the scaling factor $f=(b-a)/2$. If no violations of $R_i \leq D_i$ occurred, then a is set to f, otherwise b is set to f and simulation is reiterated until the scaling interval shrinks to just one value $[a, a+1]$ in which case the scaling factor equals to this found value a, the application density is calculated accordingly, and the minimal processor performance P which still ensures the application feasibility is $P=a\times10^6$ operations per second. It usually takes from 5 to 15 simulations to reach the resulting values.

Data Structures. The simulator uses ordered chained lists whose elements consist of 3 cells: the link to the next list element or NULL, the ordering value and the data specific to the list. Elements in a list are ordered with respect to the ordering value, starting with the smallest one. Lists are defined with the defining word `List`:

```
: List ( list-element-size, max-list-length -- )
```

and use respective "methods" to add and retrieve elements in lists created by this word:

```
: >List ( new-elem-addr, list-addr -- )
        Place a new element into the ordered list
: List@ ( list-addr-- elem-addr)
        Get the first (heading) element of the list
: List> ( list-addr-- elem-addr)
        Delete the first element from the list
: List>> ( ordering-value, list-addr--)
        Find and delete a list element with this ordering value
```

Static objects (tasks and resources) are created at the simulator initialization from the task description file and are modified during simulation.

A resource is rendered with an object of 4 cells: its ID number, its priority (reserved for future use), its status (either NULL if the resource is currently unlocked, or a reference to the job description, which currently owns this resource and locked it), and a possibly empty ordered list of job descriptions, currently waiting for this resource to become unlocked. Resources are stored in a special pool which allows to easily enumerate them and to add a new one.

Tasks are represented with objects of various length which depends on the number of task segments. It starts with 10 cells followed by a series of 4 cells for each task segment. The initial 10 cells contain: task unique ID number *i*, task period T_i, task weight in the number of standard operations W_i, task weight in seconds C_i (depends on the scaling factor *f*), task priority $Prio_i$, task response time R_i (is calculated during simulation), task deadline D_i, task phase Ph_i, the number of executed task activations , and the number of task segments. The 4 cells for each task segment are: segment type (*Lock*, *Unlock*, or *End*), segment parameter (the resource ID for *Lock*/*Unlock* and zero for *End*), segment weight in the number of standard operations S_j, and the segment time in seconds (recalculated while scaling the task data with the scaling factor *f*).

Dynamic objects (jobs and events) are created during simulation sessions as needed with the words `CreateJob` and `CreateEvent` :

```
: CreateJob ( task-addr--job-addr)
: CreateEvent
( resource-addr, job-addr, task-addr, event-type, event-time --
  event-addr)
```

The job object is represented with 10 cells: the job unique ID, its current priority (it may change with the priority inheritance scheduling mode), current segment number which specifies the segment begin executed, current segment expected termination time, current segment start time, current segment used time, current segment time yet to be used, reference to the respective task, number of references to the job description, and a reference to a resource which this job is waiting for or NULL if the job is not waiting for a resource. Jobs waiting for the processor form a chained list `JobList` in the order of their current priorities. The first job in this list owns the processor and is considered as the current one. When this list is empty, the processors stays idle.

System events are characterized by the time when they occur. Events with the same timing form a group of time-sake events. Four types of system events are considered: *to activate* a task (i.e., to form a job for this task and add it to the list `JobList` of active jobs waiting for the processor), *to terminate* the current job (and pass the processor to another job in list `JobList`, if any), *to lock* a resource, or *to unlock* a resource – and these activities are performed with respective Forth words:

```
: TaskActivate ( task-addr--)
: JobTerminate ( job-addr--)
: ResourceLock ( resource-addr, job-addr--)
: ResourceUnlock ( resource-addr, job-addr--)
```

The event object which represents a system event consists of 6 cells: the event unique ID, the scheduled time for this event to occur, the type of the event (*Activate*, *Lock*/*Unlock*, or *End*), a reference to the task object to be activated or NULL, a reference to the job object to be ended or NULL, and a reference to the resource object to be locked/unlocked or NULL. The chained list `EventList` of system events ordered with respect to their time moments when they scheduled to occur is maintained by the simulator.

The Simulator. Simulator initialization consists in selecting the desired combination of the scheduling mode and access protocol, setting the respective simulator constraints, reading the task description file, and forming the respective resource and task objects. Then the initial list of system events `EventList` is formed which consists in activation of the all tasks at the moments of system time defined by their phase shifts. Counts for their maximal response times are set to zero and all resources are set to be unlocked.

The major simulator loop does the following. The first group of time-sake events in the `EventList` is considered, the simulator system time is set to this time moment and all system

events from this first group are processed one-by-one. Processing depends on the event type: activate a task, terminate a job, or lock/unlock a shared resource.

Activating a task. A new job is created from this task referred to by the event with its planned starting time equal to the current system time and is added to the `JobList` with its priority, while a new event is added to the `EventList` – to activated the next copy of this task at the moment of time not less than the current time plus the task period T_i.

Terminating a job. The response time of the task referred to by the respective job object is updated: the difference between the current system time and the moment when this job was created and added to `JobList` (the response time which consists of the time when the job owned the processor plus the time it waited for it) is calculated and the maximum of this value and the response time already stored in the task referred to is stored as the new value of the task response time. If this exceeds the task deadline D_i, then a violation of the task feasibility is registered. The considered job is deleted from the `JobList`.

Locking a resource. If the resource is unlocked, then it becomes locked by this task; otherwise, the job is moved from the `JobList` to the ordered list of jobs waiting for unlocking of this resource.

Unlocking a resource. If the ordered list of jobs waiting for unlocking of this resource is not empty, then the first job form this list is moved from it back to the `JobList` according to its priority and the resource becomes locked by this job; otherwise, the resource becomes unlocked.

Upon completion of the event processing, the considered event is deleted from the `EventList`. After all time-sake events have been processed, the `JobList`, which may have changed as a result of previous event processing, is considered unless it is empty.

If the `JobList` is not empty then the first job from it (which currently owns the processor) is selected and the residue of the processor time not yet consumed by its current segment is considered. This value determines the moment of the segment termination. If this value is greater than the time of the next time-sake group of system events in the `EventList` then this residue is decremented by the remaining time till this event group; otherwise, a new event corresponding to this segment is added to the `EventList` for this moment of segment termination and the next job segment if any becomes its current segment.

Emptiness of the `JobList` means that the processor is idle from this moment till the next time-sake event group in the `EventList`. Upon completion of processing the first job of `JobList` (if any) the major loop is reiterated. The loop terminates upon exhausting the time limit of the simulation session or when a specified number of created jobs is reached (which of these conditions occurs earlier, if both limits are specified).

TimeLimit=25 JobLimit=0 ViolationLimit=1	TimeLimit=25 JobLimit=0 ViolationLimit=1
SchedulingMode=RM InheritanceMode=NI	SchedulingMode=RM InheritanceMode=BI
Configuration file name: c:\MPE\App_4t2r.txt	Configuration file name: c:\MPE\App_4t2r.txt
Time=0 Proc=0 for 0 A 4.1	Time=0 Proc=0 for 0 A 4.1
Time=2 Proc=4.1 for 2 L 4.1 of 2	Time=2 Proc=4.1 for 2 L 4.1 of 2
Time=3 Proc=4.1 for 1 A 3.2	Time=3 Proc=4.1 for 1 A 3.2
Time=4 Proc=3.2 for 1 L 3.2 of 1	Time=4 Proc=3.2 for 1 L 3.2 of 1
Time=5 Proc=3.2 for 1 A 1.3 A 2.4	Time=5 Proc=3.2 for 1 A 1.3 A 2.4
Time=6 Proc=1.3 for 1 W 1.3 of 1	Time=6 Proc=1.3 for 1 W 1.3 of 1
Time=15 Proc=2.4 for 9 E 2.4	Time=7 Proc=3.2 for 1 W 3.2 of 2
Time=16 Proc=3.2 for 1 W 3.2 of 2	Time=10 Proc=4.1 for 3 U 4.1 of 2 L 3.2 of 2
Time=19 Proc=4.1 for 3 U 4.1 of 2 L 3.2 of 2	Time=11 Proc=3.2 for 1 U 3.2 of 2
Time=20 Proc=3.2 for 1 U 3.2 of 2	Time=12 Proc=3.2 for 1 U 3.2 of 1 L 1.3 of 1
Time=21 Proc=3.2 for 1 U 3.2 of 1 L 1.3 of 1	Time=13 Proc=1.3 for 1 U 1.3 of 1
Time=22 Proc=1.3 for 1 U 1.3 of 1	Time=14 Proc=1.3 for 1 E 1.3
Time=23 Proc=1.3 for 1 E 1.3	Time=23 Proc=2.4 for 9 E 2.4
Time=24 Proc=3.2 for 1 E 3.2	Time=24 Proc=3.2 for 1 E 3.2

Time=25 Proc=4.1 for 1 E 4.1 Time=25 Hardness=1,0000 1/Hardness=1,0000 Density=0,6056 ScalingFactor=1,0000 **ERROR: Deadline violation in Task 1** ok	Time=25 Proc=4.1 for 1 E 4.1 Time=25 Hardness=1,0000 1/Hardness=1,0000 Density=0,6056 ScalingFactor=1,0000 ok

Fig. 2. Logs of two simulation sessions as they are output by the simulator

The results of simulation – task maximal response time, number of deadline violations, the application density, and other statistics data are displayed. A simulation log may also be displayed. When any system event is processed, the respective time and other accompanying data are printed-out. All these data may be easily copied into MS Excel for a graphical representation of the obtained results and execution log.

There are the two logs of simulator runs in Fig. 2 – for two different protocols of access to shared resources: NI (no inheritance) and BI (basic priority inheritance) as they are recorded by the simulator. The number after "Time=" is the time of an occurring system event denoted by one of the letters: A – activate, E – end, L – lock, U – unlock, or W – wait to lock an already locked resource, followed by the event parameter. The job ID is displayed as two numbers (the task number and the unique job number separated with a period). The section "of" is followed by the resource number to be locked or unlocked, while a number after "for" is the activity duration terminated with this event. Same logs are presented in Fig. 3 in a more readable graphic form.

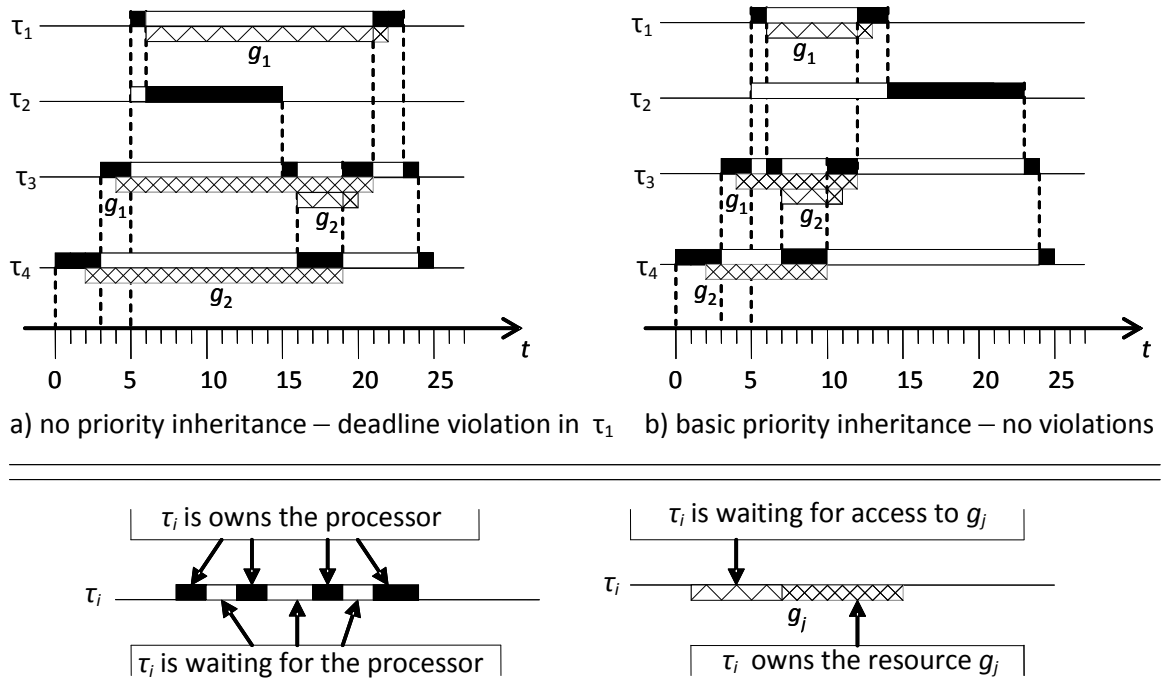
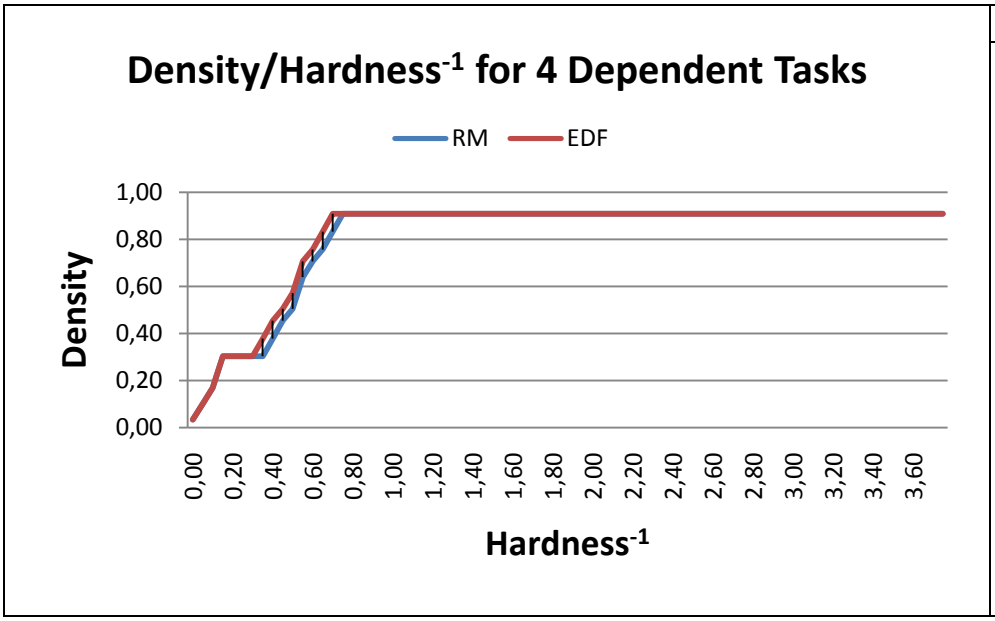


Fig. 3. Simulated execution of 4 tasks sharing 2 resources with different access protocols

This application, when simulated twice with different access protocols, demonstrates two different behaviors: a violation of the specified deadline 15 for the highest priority task τ_1 under the protocol NI – Fig. 3a, and correct work with no violations under the protocol BI – Fig. 3b.

Fig. 4 compares two scheduling modes for the same application of 4 tasks and 2 shared resources defined in Fig. 1. The output simulation data were copied into an Excel file to obtain these charts. Data for application hardness and respective density values for the two scheduling modes are in the right columns of the chart. As one can see, there's no big difference in the application density between the two scheduling modes RM and EDF for this application. Density as a function of $hardness^{-1}$ grows nearly linearly with two plateaus and then the growth stops after $hardness^{-1}$=0.75. As one can see, this application cannot reach 100% density – its maximum is 0.9083 with the application *hardness*=1/0.75= 1.33 and it does not change with

further decrease of hardness (i.e., increase of hardness^{-1}), which means that the processor would be inevitably idle for at least ≈10% of time while executing this application.



1/H	RM	EDF
0.00	0.0336	0.0336
0.05	0.1009	0.1009
0.10	0.1682	0.1682
0.15	0.3028	0.3028
0.20	0.3028	0.3028
0.25	0.3028	0.3028
0.30	0.3028	0.3028
0.35	0.3028	0.3784
0.40	0.3784	0.4541
0.45	0.4541	0.5046
0.50	0.5046	0.5719
0.55	0.6392	0.7065
0.60	0.7065	0.7568
0.65	0.7568	0.8326
0.70	0.8326	0.9083
0.75	0.9083	0.9083

Fig. 4. RM vs. EDF for same application of 4 tasks with 2 resources

Four Dining Philosophers. This classical puzzle, first proposed by E.Dijkstra as "Five Dining Philosophers" [5], demonstrates the situation of mutual blocking under certain scenarios of dependent task behavior with any number $n \geq 2$ of the respective processes. Let's consider 4 iterative processes, each with two alternate activities called "think" and "eat", the latter assuming simultaneous access to 2 of 4 shared resources (called the left and the right fork for this philosopher) for a certain period of time. Access to the resources is performed via critical intervals guarded with respective mutexes.

With the proposed technique this may represented as 4 tasks τ_1, τ_2, τ_3, and τ_4 (the philosophers), which share 4 informational resources r_1, r_2, r_3, and r_4 (the forks). Task phases are 10, 7, 4, and 1 respectively; 2 units after its start the task τ_1 locks the resource r_1 and after 4 units more it locks the resource r_2. Then after 20 time units it unlocks r_1 and in 68 units more it unlocks r_2. After 1000 time units or more since its start, the task τ_1 reiterates. Other tasks behave similarly with 73, 79, and 85 time units rather than 68 for unlocking their second resource (left fork). In the formalism of Fig.1 the behavior of task τ_1 may be specified as (others are similar):

```
<task name="t_1" phase="10" period="1000">
   <segment length=2 interface="r_1" op_type="lock"/>
   <segment length=4 interface="r_2" op_type="lock"/>
   <segment length=20 interface="r_1" op_type="unlock"/>
   <segment length=68 interface="r_2" op_type="unlock"/>
   <segment length=2 op_type="end"/> </task>
```

With the specified phases and timings for locking/unlocking resources, a clinch occurs at time=25, as Fig.5 displays this with the log obtained by the simulator.

System Log	Interpretation/Comments
TimeLimit=1000000 JobLimit=0 ViolationLimit=0	
SchedulingMode=RM InheritanceMode=PI	Rate Monotonic with Priority Inheritance
Configuration file name: c:\MPE\App_4PhD.txt	
Time=1 Proc=0 for 1 A 4.1	Task 4 (job 4.1) is activated at time=1
Time=3 Proc=4.1 for 2 L 4.1 of 4	Task 4 (job 4.1) locks resource 4 at time=3
Time=4 Proc=4.1 for 1 A 3.2	Task 3 (job 3.2) is activated at time=4
Time=6 Proc=3.2 for 2 L 3.2 of 3	Task 3 (job 3.2) locks resource 3 at time=6

System Log	Interpretation/Comments
Time=7 Proc=3.2 for 1 A 2.3	Task 2 (job 2.3) is activated at time=7
Time=9 Proc=2.3 for 2 L 2.3 of 2	Task 2 (job 2.3) locks resource 2 at time=9
Time=10 Proc=2.3 for 1 A 1.4	Task 1 (job 1.4) is activated at time=10
Time=12 Proc=1.4 for 2 L 1.4 of 1	Task 1 (job 1.4) locks resource 1 at time=12
Time=16 Proc=1.4 for 4 W 1.4 of 2	Task 1 (job 1.4) waits for resource 2 at time=16
Time=19 Proc=2.3 for 3 W 2.3 of 3	Task 2 (job 2.3) waits for resource 3 at time=19
Time=22 Proc=3.2 for 3 W 3.2 of 4	Task 3 (job 3.2) waits for resource 4 at time=22
Time=25 Proc=4.1 for 3	Clinch detected for task 4 (job 4.1) when it tried
Mutual clinch for job 4.1 on resource 1 ok	to lock resource 1 at time=25

Fig. 5. System log for the 4 philosophers puzzle

The resource status displayed by the word `.resources` confirms this clinch. As one can see there's a vicious circle of locked resources with mutually waiting jobs:

```
Resource_1 Prio=0 Status=Job 1.4 JobsWaiting=NULL
Resource_2 Prio=0 Status=Job 2.3 JobsWaiting=Job 1.4
Resource_3 Prio=0 Status=Job 3.2 JobsWaiting=Job 2.3
Resource_4 Prio=0 Status=Job 4.1 JobsWaiting=Job 3.2
```

Conclusions. The simulator was written in Forth with VFX Forth for Windows, version 4.70, provided to the author at the courtesy of MPE [6], and is just 985 lines of code under the respective coding standards. It uses only fixed-point arithmetic and works remarkably fast on a PC. To avoid memory overflow, the simulator uses its own simple subsystem for memory allocation and reuse for chained list elements, jobs and events. Further work will be focused on improving the user interface, extending the nomenclature of scheduling modes and access protocols of this simulator, and transition to simulation of multi-core and multiprocessor platforms, as well as running more experiments with models of real-time multi-task applications.

References.

1. VFX Forth for Windows. User manual. Manual revision 4.70, 19 August 2014. – Southampton: MicroProcessor Engineering Limited, 2014. – 429 p.
2. Andersson B., Baruah S., Jonsson J. Static-Priority Scheduling on Multiprocessors // Proc. of 22nd IEEE Real-Time Systems Symposium. – London, 2001. – P.193-202.
3. Nikiforov V.V., Shkirtil V.I. Specification of interfaces in real-time software applications by XML forms. // SPIIRAS Proceedings, 2009, issue 11. – P. 159-175. (In Russian.)
4. Baranov S.N., Nikiforov V.V. Density of Multi-Task Real-Time Applications // Proceedings of the 17th Conference of Open Innovations Association FRUCT, Yaroslavl, Russia, 20-24 April 2015. – P.9-15.
5. Dijkstra E.W. Hierarchical ordering of sequential processes. Acta Informatica 1(2), 1971. – P.115-138.
6. MicroProcessor Engineering Limited. Company site http://www.mpeforth.com .

About the Author. Sergey N. Baranov graduated with honor the Leningrad State University in 1972, worked at this University, at SPIIRAS, Motorola, St.Petersburg State Polytechnic University; PhD since 1978, Doc.Sci since 1991, Professor since 1993. Currently he works at SPIIRAS as a Chief Research Associate and a lecturer at 3 major St. Petersburg Universities. His major scientific interests are software engineering, compilers, analysis and verification of software specifications, formal methods, symbolic computations, and Forth. He is an active member of the international Forth community after publishing in 1988 the monograph "The Programming Language Forth and its Implementations", the first one on Forth to appear in Russian.

From `exit` to `set-does>` A Story of Gforth Re-Implementation

M. Anton Ertl* Bernd Paysan
TU Wien

Abstract

We changed `exit` from an immediate to a non-immediate word; this requires changes in the deallocation of locals, which leads to changes in the implementation of colon definitions, and to generalizing `does>` into `set-does>` which allows the defined word to call arbitrary execution tokens. The new implementation of locals cleanup can usually be optimized to similar performance as the old implementation. The new implementation of `does>` has similar performance similar to the old implementation, while using `set-does>` results in speedups in certain cases.

1 Introduction

Over the years there were several complaints about not being able to tick `exit` in Gforth. In July 2015 we decided to do something about this. In combination with other innovations, this led to a number of further changes in the implementation, and eventually to a generalization of `does>`.

The story of these changes and the other implementation issues they touch on should be interesting and instructive for readers interested in Forth implementation techniques, and is told in Section 2. These changes were not performed for performance reasons, but performance should not suffer from them. In Section 3 we evaluate the performance impact with microbenchmarks. Section 4 discusses an implementation caveat for locals cleanup on native-code compilers.

2 The Story

While Forth-94 and Forth-2012 systems are allowed to implement `exit` as an immediate compile-only word, we have received a number of complaints about Gforth implementing `exit` this way, so we decided to change the implementation of `exit` into a non-immediate word in July 2015.

2.1 Locals cleanup

Now we had implemented `exit` as immediate compile-only word for a good reason: When `exit`ing a definition with locals, we need to remove the locals before exiting. In the following contrived example:

```
: foo { a } exit ;
```

the original immediate `exit` compiles `lp+ ;s`, where `lp+` increments the locals-stack pointer lp to remove `a` from the locals stack and `;s` returns to the caller of `foo`.

Our new, non-immediate `exit` is just an alias for `;s`, so we have to clean up the locals in some other way. We took the established approach of pushing additional data and an additional return address on the return stack. In our case the additional data is the depth of the locals stack at the start of the colon definition, and the return address points to a code fragment equivalent to

```
r> lp! ;s
```

except that we have a single primitive `lp-trampoline` that does what this sequence would do; the (sub-optimal) AMD64 code[1] for this primitive is:

```
mov  %rp,%rax
mov  0x8(%rp),%ip
lea  0x10(%rp),%rp
mov  (%rax),%lp
add  $0x8,%ip
mov  -0x8(%ip),%rdx
mov  %rdx,%rax
jmpq *%rax
```

So, an `exit` inside a colon definition with locals jumps to this code fragment, sets lp to its old value, and finally returns to the calling definition (see Fig. 1).

This solution for the clean-up problem poses the problem of where these additional return-stack

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

[1] Register names for virtual machine registers are replaced, as follows: `ip=rbx, rp=r13, lp=rbp, sp=r15, tos=r14, cfa=rcx`.

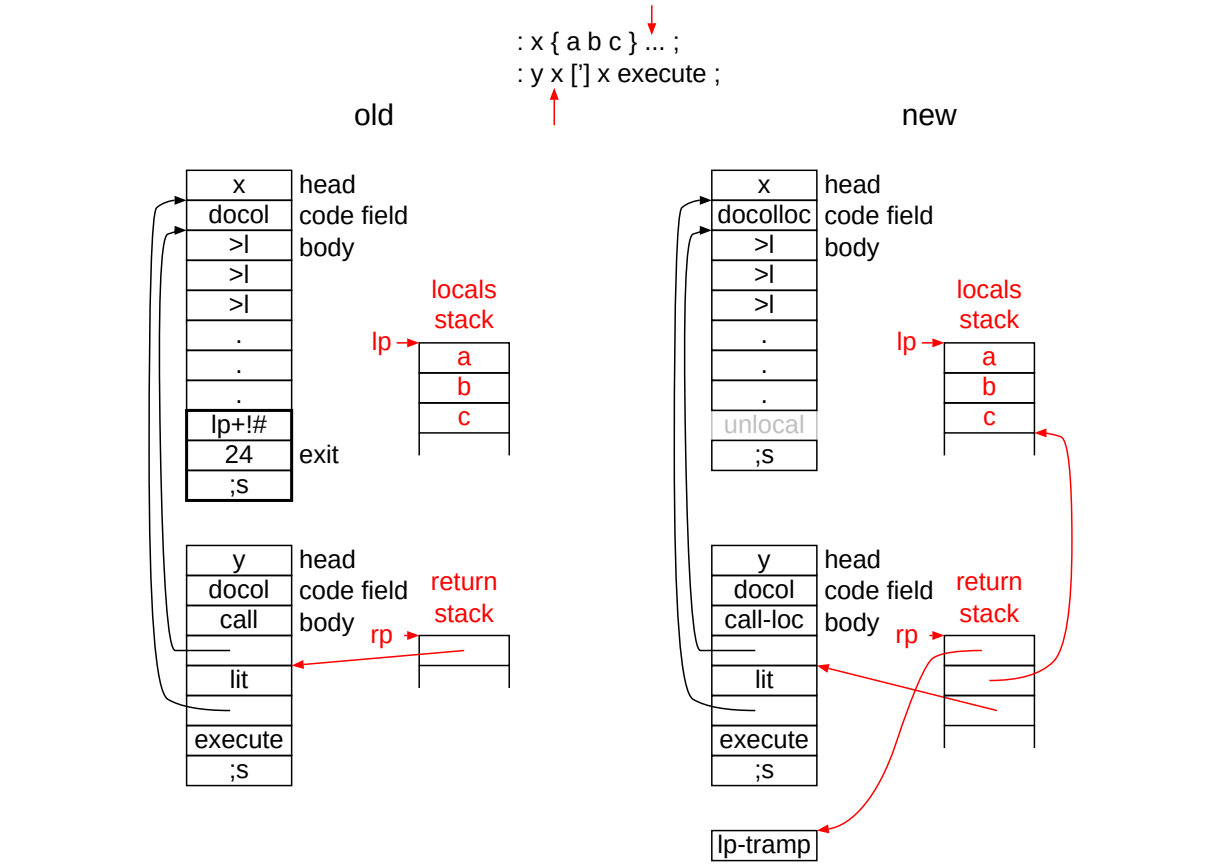


Figure 1: Old and new implementation of cleaning up locals; the state of the return and locals stack corresponds to execution being at the red arrows in the code.

items are pushed. A classical solution would be to do it at the first definition of locals. However, in Gforth, locals can be first defined inside control structures, e.g.:

```
: foo ?do { a } i loop ;
```

Either we push the return-stack items before the control structure, or we have to pop them off the return stack at the end of the loop[2].

We decided to push them before the control structure, on entering the colon definition, by changing the code field to point to a new routine `docolloc` instead of the ordinary `docol` routine. `Docolloc` peforms all the work that `docol` does, but in addition pushes the current value of lp and the address of the code fragment pointing to `lp-trampoline` on the return stack. Here you see both routines for the AMD64:

```
docol                   docolloc
mov  %rp,%rax           mov  %rp,%rax
mov  %ip,%rdx           mov  %ip,%rdx
lea  -0x8(%rp),%rp      lea  -0x18(%rp),%rp
mov  %rdx,-0x8(%rax)    mov  %rdx,-0x8(%rax)
                        lea  0x80(%rsp),%rdx
lea  0x18(%cfa),%ip     lea  0x18(%cfa),%ip
                        mov  %lp,-0x10(%rax)
                        mov  %rdx,-0x18(%rax)
mov  -0x8(%ip),%rdx     mov  -0x8(%ip),%rdx
mov  %rdx,%rax          mov  %rdx,%rax
jmpq *%rax              jmpq *%rax
```

Gforth uses primitive-centric threaded code [Ert02], so the routines `docol` and `docolloc` are executed only when the word is `execute`d or called through a `defer`red word. When calling the word directly from a colon definition (about 99% of the calls), Gforth uses the primitives `call` and `call-loc` that take (the body address of) the called definition from the next cell in the threaded-code:

[2] In general, whenever the locals stack becomes empty.

```
call                    call-loc
mov  %ip,%rdx           mov  %ip,%rdx
mov  (%ip),%ip          mov  (%ip),%ip
mov  %rp,%rax           mov  %rp,%rax
add  $0x8,%rdx          add  $0x8,%rdx
lea  -0x8(%rp),%rp      lea  -0x18(%rp),%rp
                        mov  %lp,-0x10(%rax)
mov  %rdx,-0x8(%rax)    mov  %rdx,-0x8(%rax)
                        lea  0x80(%rsp),%rdx
add  $0x8,%ip           add  $0x8,%ip
                        mov  %rdx,-0x18(%rax)
mov  -0x8(%ip),%rdx     mov  -0x8(%ip),%rdx
mov  %rdx,%rax          mov  %rdx,%rax
jmpq *%rax              jmpq *%rax
```

Gforth has an intelligent `compile,` that produces the appropriate primitive for the word, and it generates `call` for `docol` words, and `call-loc` for `docolloc` words, plus (in the next cell) the body address of the colon definition.

Some Forth programmers like to use code like `r> drop exit` to return to the next-but-one surrounding definition instead of the next one. If the next one uses locals, the programmer has to force a cleanup, and we provide the word `unlocal` to achieve this. So if the calling word uses locals, the sequence above has to be modified to `r> drop unlocal exit`. `Unlocal` just removes the additional return stack data and removes the locals from the locals stack:

```
unlocal                 unlocal-;s
mov  %rp,%rax           mov  0x10(%rp),%ip
lea  0x10(%rp),%rp      mov  0x8(%rp),%lp
add  $0x8,%ip           add  $0x18,%rp
mov  0x8(%rax),%lp      add  $0x8,%ip
mov  -0x8(%ip),%rdx     mov  -0x8(%ip),%rdx
mov  %rdx,%rax          mov  %rdx,%rax
jmpq *%rax              jmpq *%rax
```

The sequence `unlocal ;s` is more efficient than `;s` jumping to `lp-trampoline` (see Section 3), especially if we combine the sequence into a superinstruction `unlocal-;s`. So, if the current definition contains locals, and if we know that `exit` returns from the current definition, we can compile `exit` into `unlocal ;s` as an optimization (through the intelligent `compile,`). If the definition performs return-address manipulation (so that the `exit` may return from a different definition), it first has to clean up the locals with `unlocal`. So, if the word contains `unlocal`, we disable this optimization.

2.2 does>

In addition to colon definitions, words defined with `does>` also call code that may define locals. We will use the following running example:

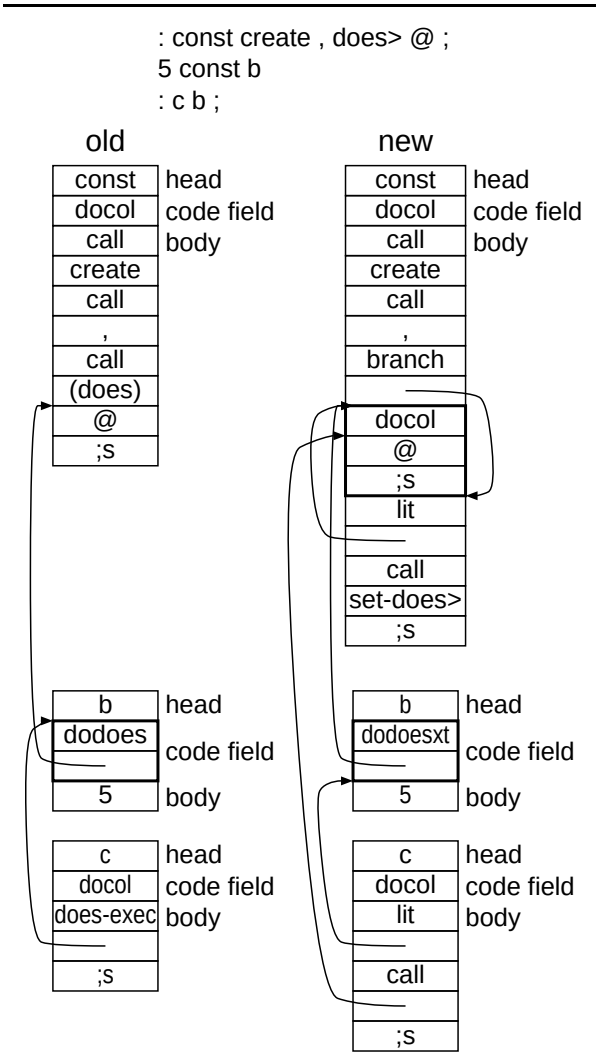


Figure 2: Old and new implementation of `does>`

```
: const create , does> ( A ) @ ;
5 const B
```

When running B, the code A after `does>` is called with either the primitive `does-exec` (when B is `compile,`d) or with the code-field routine `dodoes` (when B is `execute`d).

Now the `does` part may also define locals and contain `exit`, so we have to push the additional stuff on the return stack in these cases, too. Our first idea was to add `dodoesloc` and `does-exec-loc`, but that would have resulted in complications, so we soon came up with the following, better idea (see Fig. 2):

The code after `does>` (A in our example) is a full-blown colon definition with its own code field and execution token. Instead of using `dodoes`, which (after pushing the body address of B) calls the code at A as `call` does, we have `dodoesxt`, which `execute`s the xt of A. Here is the code for `dodoes`, compared to `dodoesxt` followed by `docol`:

```
dodoes                  dodoesxt, docol
mov  %tos,(%sp)         mov  %tos,(%sp)
lea  0x10(%cfa),%tos    lea  0x10(%cfa),%tos
mov  %ip,%rdx
mov  0x8(%cfa),%ip      mov  0x8(%cfa),%cfa
sub  $0x8,%sp           sub  $0x8,%sp
                        mov  (%cfa),%rdx
                        mov  %rdx,%rax
                        jmpq *%rax
mov  %rp,%rax           mov  %rp,%rax
                        mov  %ip,%rdx
lea  -0x8(%rp),%rp      lea  -0x8(%rp),%rp
                        lea  0x18(%cfa),%ip
mov  %rdx,-0x8(%rax)    mov  %rdx,-0x8(%rax)
add  $0x8,%ip
mov  -0x8(%ip),%rdx     mov  -0x8(%ip),%rdx
mov  %rdx,%rax          mov  %rdx,%rax
jmpq *%rax              jmpq *%rax
```

The advantage for our locals problem is that no additional work is needed: the first locals definition in A changes the A colon definition into a `docolloc` colon definition, and there is no need to change the `dodoesxt`.

Again, `dodoesxt` is only used when B is `executed` or called through a deferred word. When we `compile,` B, the intelligent `compile,` compiles the body address of B as literal, followed by `compile,`ing the xt of A, resulting in `call` or `call-loc` followed by the body address of A. We have added static superinstructions for `lit call` and `lit call-loc` to eliminate the overhead of executing two primitives instead of one. Here is the code for `does-exec` compared to that for the `lit-call` superinstruction:

```
does-exec               lit-call
mov  %tos,(%sp)         mov  %tos,(%sp)
                        mov  %ip,%rax
mov  (%ip),%tos         mov  (%ip),%tos
mov  %ip,%rdx           mov  0x10(%ip),%ip
mov  %rp,%rax           mov  %rp,%rdx
add  $0x8,%rdx          add  $0x18,%rax
lea  -0x8(%rp),%rp      lea  -0x8(%rp),%rp
sub  $0x8,%sp           sub  $0x8,%sp
mov  0x8(%tos),%ip
mov  %rdx,-0x8(%rax)    mov  %rax,-0x8(%rdx)
add  $0x10,%tos
add  $0x8,%ip           add  $0x8,%ip
mov  -0x8(%ip),%rdx     mov  -0x8(%ip),%rdx
mov  %rdx,%rax          mov  %rdx,%rax
jmpq *%rax              jmpq *%rax
```

Given that our `does>` is now based on taking an xt, we can make another interface to this functionality available: `set-does> ( xt -- )` changes the last defined word to first push its body address, then execute the xt. There are two benefits to `set-does>`:

First, when there is only one word between `does>` and `;`, one can pass that word (instead of a colon definition containing just that word) to `set-does>`, saving one call-exit pair at run-time. E.g.:

```
: const create , [’] @ set-does> ;
5 const B
```

When compiling B, this produces `lit @` (without additional effort), and saves a `call` and `;s` around the `@` at run-time.

The other advantage is that `set-does>` can be used more flexibly than `does>`, e.g., inside control structures; e.g, `struct.fs` contains

```
: dofield ( -- )
does> ( name execution: addr1 -- addr2 )
    @ + ;

: dozerofield ( -- )
    immediate
does> ( name execution: -- )
    drop ;

: field ( align1 off1 align size "name"
          --  align2 offset2 )
    2 pick >r create-field r> if \ off1<>0
        dofield
    else
        dozerofield
    then ;
```

In the usual case, a field should perform the `does>` part of `dofield`, but if the field has offset 0, then it should not compile anything, so it is defined as immediate word that does nothing (not even `0 +`, to avoid stack underflow at compile time). The properties of `does>` force this factoring, which I don’t consider particularly conducive to understanding. With `set-does>`, we can define this as

```
: field ( align1 off1 align size "name"
          --  align2 offset2 )
    2 pick >r create-field r> if \ off1<>0
        [: @ + ;] set-does>
    else
        [: ;] set-does> immediate
    then ;
```

Another use case of `set-does>` is optimization:

```
: const ( n "name" -- )
  \ you must not change the body of "name"
  create ,
  [’] @ set-does>
  [: >body @ postpone literal ;] set-opt ;
```

`set-opt ( xt -- )` sets what happens when the created word is `compile,`d (it is the basis for the intelligent `compile,`). In this case it optimizes the

word such that, instead of looking up the value at run-time, the lookup happens at compile, time and the resulting value is compiled as literal.

This can only happen after does> or set-does>, because does> and set-does> change what the word does, and that also overwrites any earlier set-opt. It is possible to implement the above with does>, but, like the field example, it would be more cumbersome.

Continuing onwards from set-does>, we could also define a defining word does-create (xt --) that combines the functions of create and set-does, used as follows:

```
: const ( n "name" -- )
  ['] @ does-create , ;
```

The advantage of does-create would be that the defined word gets the final code field right from the start, instead of being first created with a dovar code field, and later overwritten with dodoes and (in our example) A; the current two-step approach leads to problems on Forth systems compiling to flash memory; while Forth implementors have found workarounds for these problems, it's better to provide an interface that does not need such workarounds. Does-create is not (yet?) implemented in Gforth.

Note that, while the new does> implementation makes the new locals-cleanup implementation simpler, the reverse is not true: You can do the new does> implementation (and set-does> and does-create) just fine in combination with the old style of locals-cleanup implementation.

3 Performance Impact

For a realistic evaluation of performance we would need a number of application benchmarks that spend a lot of time calling to and returning from definitions containing locals, and application benchmarks performing lots of calls to does>-defined words.

Unfortunately, we are not aware of benchmarks with these characteristics, so we use microbenchmarks here to evaluate the performance. Real applications may see much smaller performance differences than we see in these microbenchmarks.

Fig. 3 shows the results, and they will be explained in the following.

We call ten different words, with results from the old implementation shown in reddish colours and new implementations in bluish colours:

baseline An empty colon definition (without locals) that gives us a baseline.

0-locals A colon definition that is empty except that it contains the overhead of cleaning up locals (plus, for the new implementation, putting the additional stuff on the return stack). We measure three ways to clean up the locals: *old* is the old way of cleaning up locals (using lp+!#); *lp-trampoline* is the new implementation without using unlocal, so ;s jumps to lp-trampoline; *unlocal* is the optimized variant of the new implementation that performs unlocal before ;s, thus skipping lp-trampoline. Both new versions incur the overhead of pushing the additional data on the return stack with call-loc or docolloc.

3x0-locals If there are several words with locals, our new implementation (without unlocal) calls the same instance of lp-trampoline from each of these words, and the NEXT inside lp-trampoline then jumps to different code; this can lead to mispredicting this branch (depending on the indirect branch predictor of the CPU). 0-locals just has one such word and should not have problems with the branch predictor. For contrast, we also have 3x0-locals, where we have three instances of a word like the one used in 0-locals; for the *lp-trampoline* variant, this leads to a mispredicted NEXT in lp-trampoline on CPUs that use a branch target buffer (BTB) for predicting indirect branches. The number of calls and the number of loops is the same, so there should be no other differences from 0-local (except for the execute variants, where there is more overhead for handling three xts instead of one, and additional mispredictions, see below).

does A does>-defined word that just drops the address that dodoes (or its replacement) pushes on the stack. There are no locals in this set of words (the new does> implementation can also be implemented without changing the locals, and the performance should be independent). Here we also have three variants: the *old* one using does-exec and dodoes; the *new* one generating lit call (as a superinstruction) and dodoesxt; and finally a variant defined with ['] drop set-does> that saves the call and return overhead.

We call these words in a loop in two ways: We compile, them into the loop, or we call them in a loop with dup execute (a little more complicated for the three-copy-variant). We also measure an empty loop and subtract its instructions, cycles, and branch mispredictions from the results to get an approximation of the pure cost of executing just that one word.

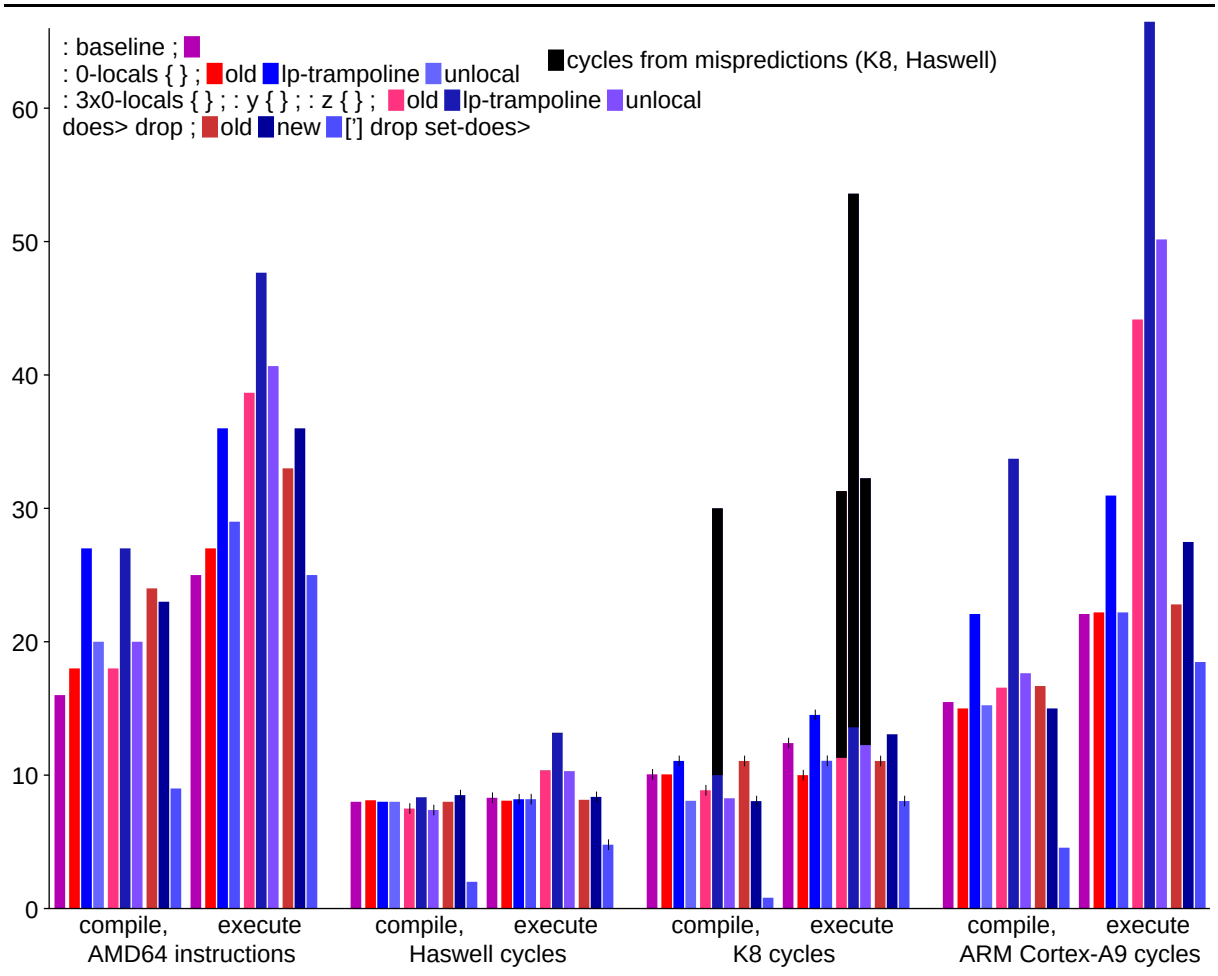


Figure 3: Instructions and cycles for performing a word (one invocation)

We used three different machines in our experiments: A Core i7-4790K (Haswell) based machine, an older (2005) Athlon 64 X2 4400+ (K8), and an ARM Cortex-A9 based PandaBoard ES. All machines ran Linux. We used the same binaries for the two machines with AMD64 architecture (Haswell, K8). On the Haswell and K8 we measured instructions, cycles, and branch mispredictions using performance counters; on the Cortex-A9 we measured CPU time with `time`, and computed cycles from that.

3.1 Locals performance

The first set of columns shows the AMD64 instructions executed when running the words. We see that, in the old implementation, cleaning up the locals stack takes two additional instructions over the locals-less baseline, and the *lp-trampoline* implementation takes 9 more instructions than the old one; the *unlocal* implementation costs only 2 instructions more than the old implementation. The same differences are seen in both the `compile,`d variant and in the `executed` variant, but the baseline is higher for the `executed` variant.

These differences in instruction count are not reflected in the Haswell cycles; this processor apparently manages to execute most of the additional instructions in parallel to the instructions that it already performs in the baseline. But why can it not extract more parallelism from the baseline? There is probably a data-dependence chain having to do with the Forth VM instruction pointer (ip).[3] In more realistic code there is more code inside the loops and definitions, so ip-based dependency chains probably do not usually determine performance in realistic code. Therefore, the instructions counts may be a better indicator of the performance impact of our changes on real code than the Haswell cycle counts.

The Haswell has a very good branch predictor [RSS15], so branch mispredictions don't play a significant role on Haswell, even for 3x0-locals.

The K8 also mostly shows few performance differences between the implementations of the locals,

[3] Save ip to the return stack, load it back, load the target of the `(loop)` primitive, and perform a few additions in between.

except that there are big differences in some cases coming from branch mispredictions (the K8 predicts indirect branches with a BTB). We estimate 20 cycles penalty per misprediction, and have coloured the corresponding part of the bars in black; comparing the non-black part of the 3x0-locals bars with the 0-locals bars, this estimate is about right. The `compile,`d 3x0-locals benchmark causes one misprediction with the *lp-trampoline* variant, from `lp-trampoline`, as discussed above. Optimizing the new locals implementation with `unlocal` eliminated this slowdown.

The `execute`d 3x0-locals benchmark has an additional branch misprediction, in `docol/docolloc`, with all implementations.

The ARM Cortex-A9 timing results seem to be influenced by instructions counts (which are probably be similar to the AMD64 counts), and (comparing 0-locals with 3x0-locals) also by mispredictions in a way similar to the K8 results, so the Cortex-A9 probably also has a BTB. Unfortunately, we do not have performance counter results for this CPU, so we cannot present misprediction results (nor instruction counts).

Concerning the difference between the old and the new locals cleanup implementation, we see that, on the Cortex-A9, *lp-trampoline* is quite a bit of slower than the old implementation, but the `unlocal` implementation has similar performance as the old implementation.

3.2 DOES> performance

When `compile,`d, the new implementation of the `does>`-defined word uses one instruction less (with the `lit call` superinstruction) than the old implementation. There are also corresponding small differences in the cycles on the K8 and Cortex-A9; on the Haswell the new implementation takes 0.5 cycles more than the old one.

When `execute`d, the new implementation takes three additional instructions; on the K8 and Cortex-A9 this is also reflected in the number of cycles, while there is little difference on the Haswell.

The `['] drop set-does>` variant saves 15 instructions for the `compile,`d version and 8 instructions for the `execute`d one compared to the old implementation. It also gives good speedups on all CPUs; this time this even includes the Haswell, because this variant shortens the dependence chain.

4 Native-code Caveats

If implemented naïvely, the additional return address can have a high cost on native-code systems that (unlike Gforth) use the architecture's return instruction for implementing `exit`. Return instructions on modern CPUs have a special branch predictor that is called *return stack* (yes, the same name as the Forth return stack, and it also contains return addresses, but it's not programmer-visible). A return to the address of the corresponding call normally predicts correctly, and a return to a different address causes a misprediction (about 20 cycles penalty on a modern CPU). Therefore each return address should be produced by a call instruction, and not manipulated. One way to achieve this in a native-code system is to push the additional (Forth) return stack items as follows:

```
push data needed for cleaning up locals
call rest-of-definition
clean up locals
ret
rest-of-definition:
...
ret
```

5 Conclusion

If we require that `['] exit execute` works, we have to clean up locals in a compatible way. The popular technique of pushing extra data and the return address of a cleanup code fragment on the stack works, but has some performance caveats. Fortunately, we achieve performance similar to the old implementation in most cases by optimizing `exit` to perform `unlocal ;s`.

The new locals cleanup implementation also led to a new `does>` implementation (but the new `does>` implementation can be implemented without the new locals cleanup). The performance for code using `does>` is comparable to the old implementation; but the new implementation also makes it possible to use `set-does>`, which allows more flexibility in structuring words, and may save a call-return pair, increasing performance.

References

[Ert02] M. Anton Ertl. Threaded code variations and optimizations (extended version). In *Forth-Tagung 2002*, Garmisch-Partenkirchen, 2002.

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *Code Generation and Optimization (CGO)*, 2015.

EuroForth 2015

A Forth Programmer Jumps Into The Python Pit

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

Abstract
A unique opportunity arose to compare two similar applications on closely related platforms, one written in Python and one written in Forth.

1. Introduction

In the past two years, the economics of display devices in industrial automation has been transformed by the introduction of very low cost micro-PCs. These can be regarded as circuit boards with an RJ45 Ethernet connector at one end, and an HDMI digital video output at the other end. In our industry, several applications were immediately suggested. Initially, no Forth compiler was then available, so the first application was written using Python. Shortly afterwards, a version of Forth became usable, and therefore a second, similar application was written using Forth, giving an excellent means of comparing the efficiency of constructing new applications in each language.

2. Economics

An industrial display application (excludes screen etc. common to both solutions)
BUYING IN COST (Not sales price)

a) Before Micro-PCs

Industrial fanless PC, including disc & PSU	£475
PC mounting brackets	£23
Operating system (Windows OEM license)	£49
Replication time (approx. 1 engineer-hour)	£45
TOTAL	**£592**

b) Using Micro-PC

Micro-PC	£26
Steel protective enclosure	£11
PSU	£7
SD card	£10
Replication time (approx. 5 engineer-minutes)	£4
TOTAL	**£58**

3. Hardware overview

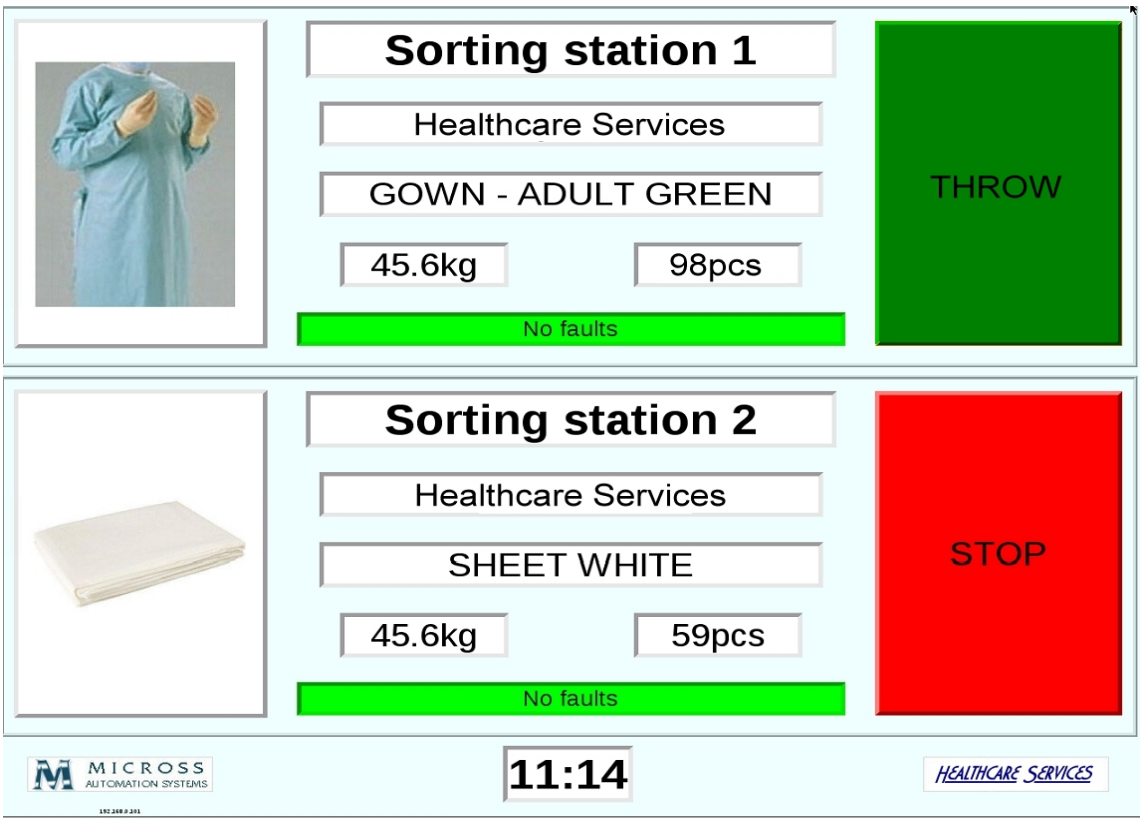
There is a wide variety of micro-PCs now available, all with similar function. By far the best-selling of these devices is known as the "Raspberry Pi". It consists of a circuit card 85mm x 56mm with a highly integrated ARM-based CPU, memory, and a variety of ports. The "disc" consists of a plug-in micro-SD card.

4. Operating System

The recommended operating system for the Raspberry Pi is a version of Linux which is very similar to Debian. However, when used in industrial applications, this has a very serious drawback, in common with many other versions of Linux. In the event of an unplanned power interruption, there is a high chance of corrupting the "disc". If this happens in a conventional implementation, with a keyboard and mouse, there are repair programs that can be run. However, in a standalone display only application, it is not practical to have to repair the disc on a regular basis. To overcome this limitation, we have constructed our own implementation of Linux that makes the SD card "read-only". The systems are now highly reliable and maintenance free. Development work takes place on a standard Linux, and on completion of development, is copied onto the reliable Linux system, which can be switched between read-only and read-write modes.

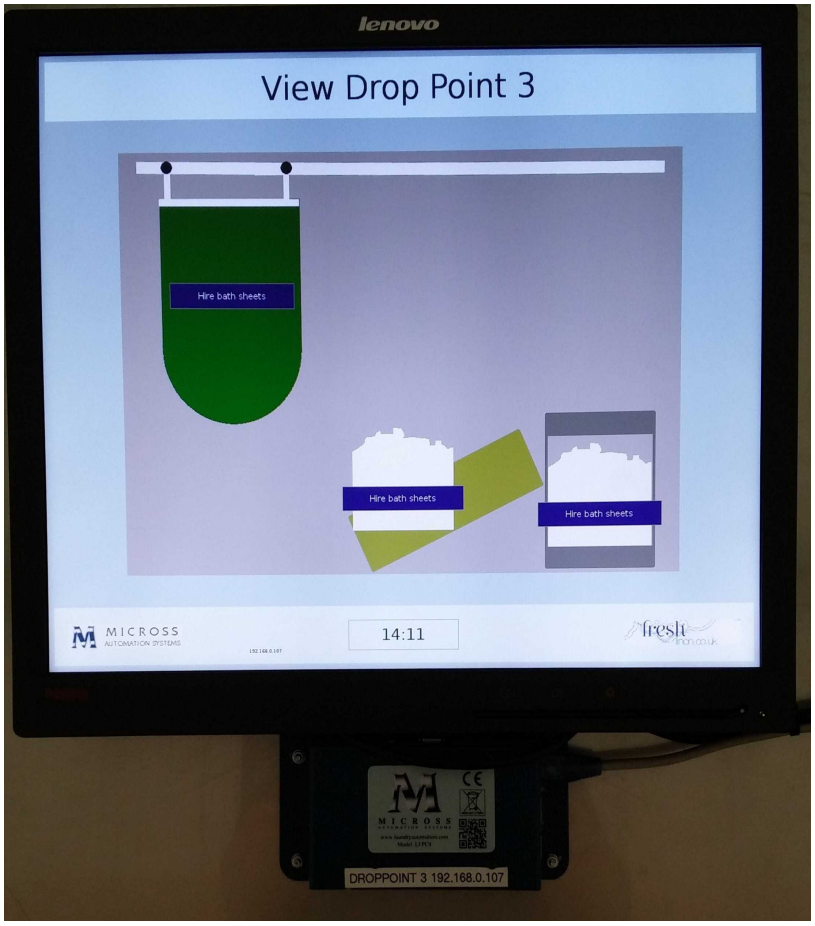
5. Application overview

Typical applications in our industry are for the purpose of displaying dynamic data to a shop floor operator. The display device is in constant communication with a central control PC, which sends out information to be displayed, typically every 0.5s, using UDP messages over the Ethernet network.



In the example shown on the previous page, the display unit shows the status of two healthcare laundry sorting stations. The customer, category, weight and piece count are shown. The category is illustrated for quick identification. Indicator beacons show the operator clearly when to sort items. This application was implemented using Python.

In the example shown below, the display unit shows loads of towels approaching a towel folder feed station. The operator can clearly see the classification of the towels in the overhead rail bag approaching, on an intermediate belt conveyor, and in the feed bin. The photograph also shows the Micro-PC protective enclosure, just below the screen. This application was implemented using Forth.



6. Program structure

The structure of both programs is exactly the same. On startup, the graphical user interface and the UDP port are initialised. A thread is started to process the communications, and a GUI timer is started to keep the graphics up to date.
The communications thread accepts UDP messages from the central control PC, indicates to the GUI that new data is available, and replies to the central control PC.
The timer compares each item of display data, and updates each part of the display as required. It also displays error information if no new data has been received recently.

7. The Python Experience

Although I have very extensive experience of programming in Forth, and quite wide experience of several other languages, this was my first foray into Python. In addition, I had not worked very much with Linux before. So there was quite a big learning process.

a) Ease of learning
Python is often suggested as a beginner's language, and indeed it is possible to write elementary programs after only a few hours study. However, some of the concepts of the language are extremely subtle, and in order to produce a serious and reliable commercial program, a long period of study is required. It is very unfortunate that there is currently no good book available which covers the latest version of the language. The online documentation is complete and well organised but can be somewhat terse.

8. The Forth Experience

Being already very familiar with Forth, I had only to master a new version, and the interface with Linux.

a) Ease of learning
Like Python it is possible to write elementary programs after only a few hours study. However, equally, in order to produce a serious and reliable commercial program, a long period of study is required. In this case, a good and up to date book is readily available.

b) Dialects and variations
A Forth programmer will be shocked to learn that the Python language is highly regulated. There is essentially only one version of the language that is current at any time. Language development follows a formal process, and the originator of the language is regarded as a "Benevolent Dictator For Life" and is assumed to have a power of veto.

c) Programming paradigm
Although Python bills itself as having multiple programming paradigms, in practice you are compelled to use an object-oriented model, because all of the really useful library functions assume this.

d) Standard libraries
The best recommendation for the Python language is the very comprehensive standard library support, covering almost every eventuality. Python bills itself as "batteries included". In practice, there are some important libraries that have not been updated to conform to the last major release, which came out in 2008.

e) Graphical user interface
Python strongly encourages the use of the TCL/Tkinter GUI. This is extremely unfortunate because it lacks the flexibility of GUIs that are more regularly maintained.

b) Dialects and variations
A Python programmer will be shocked to learn that the Forth language is unregulated to the point of anarchy. There are as many versions of the language as there are programmers. Any "Benevolent Dictator" would be instantly overthrown. From here downwards, this paper will describe the VFX Forth for Linux by MPE.

c) Programming paradigm
Forth is its own paradigm.

d) Standard libraries
VFX Forth comes with a fair range of extensions covering many frequently needed functions. It is in the nature of Forth that these are regarded as suggestions only, and are frequently modified to suit a particular application.

e) Graphical user interface
VFX Forth provides elementary wrappers for the GTK+ GUI. Unfortunately the wrappers support only the older principal version 2 of GTK+, rather than the current principal version 3. In Forth, it is not difficult to create upgraded wrappers using the older code as a model. GTK+ is highly complete, very flexible, regularly updated, and far superior in every respect to the Tkinter GUI preferred by Python.

f) Style and compactness
Python is the only language I have come across that approaches the compactness of Forth. In my view this is a very important feature of any language because it enables complete functions to be read in one glance, which greatly assists in bug detection. But, this compactness is achieved by the use of indentation to delimit blocks. This prevents free formatting which we use regularly to improve code readability. In addition, multiple statements per line are discouraged.

g) Readability
All languages can be used to produce more, or less, readable code. But Python programs when well written are definitely easier to read than most other languages.

h) Community support
Python has a very large user base, and as one might expect there is a variety of community forums. In practice these tend to be clogged with elementary queries from beginner programmers, and it can be hard to get good advice on the very subtle difficulties of the language.

f) Style and compactness
Forth is still the most compact language to code, primarily due to its low structuring overhead and concatenative programming model.

g) Readability
In a language that contains such "bad" key words as -ROT and PICK, it is of course easy to write obfuscated code in Forth. However, with careful naming, use of Forth's completely free formatting, and careful structuring, Forth code can still be the best language for readability and maintainability.

h) Community support
Forth has a rather small active base of practising commercial programmers, and therefore the chance of finding anyone else working in the same dialect and in a similar application area is slight. On the other hand, there is the opportunity (possibly after liquid bribery) to consult the actual author of the compiler.

i) Difficulties

In addition to the problems with the obsolete default GUI, and the out of date libraries, we encountered the following problems with Python.

i. Scope of variables

There is an assumption in Python that all variables are as local as possible. Python is extremely averse to global variables, and for anyone used to the opposite assumption, this leads to frequent misunderstandings. In fact the whole subject of scoping of variables is so tricky in Python that it fills the community forums with obscure problems.

ii. Structures

The whole approach to structures in Python is completely different from that in C or Forth. The Python approach is really quite clever, but it is also quite complex and hard for a beginner to grasp. This makes it very difficult when writing a communications protocol, in which the data is normally defined as a C-like structure.

iii. Garbage collection

This is the biggest weakness of Python, and one which occupies reams of forum discussion about the difficulty of debugging. Extreme coding care is needed to avoid either memory leaks (caused by uncollected garbage) or miraculously dereferenced variables (due to over-zealous binmen).

i) Difficulties

In addition to the problems with the obsolete GUI version, we encountered the following problems with VFX Forth for Linux.

i. No initial support for floating point

Unfortunately this meant that for the first application, it was not possible to use Cairo for drawing. The floating point is now working and will be used on the next application.

ii. Difficulties with cache flushing in Linux

This means that perfectly correct code will sometimes fail to compile. It is a temporary irritation only, as a second (sometimes third) compilation attempt will succeed.

9. Equivalent codes examples

The communications thread, with almost identical function in both applications:
a) Python

```
def coms(app):
    pcsock=socket_init()
    while apprunning:
        if socket_ready(pcsock):
            rbytes, raddress = pcsock.recvfrom(2000)
            app.mq.put_nowait(rbytes)
            sbytes = struct.pack('B', 0)
            pcsock.sendto(sbytes, raddress)
        time.sleep(0.05)
```

b) Forth

```
: COMACTION ( --- ) \ Communications task action
  SOCKET-INIT                          \ Initialise socket
  NEWFLAG OFF                          \ Clear new data available
  BEGIN
    SOCKET-READY IF                    \ Message ready
      SOCKET-GETPACKET LFPC4TX1 = IF   \ Correct message length
        SOCKET-PROCESS                 \ Process received packet
        SOCKET-TRANSMIT                \ Reply
      THEN
    THEN
    50 ms PAUSE
  AGAIN
;
```

9. Conclusion

Each language has both advantages and disadvantages. Full mastery of either language could only be achieved by constant practice. Since all our principal applications are written in Forth, we will continue to use it for future applications in Linux on Micro-PCs.

NJN
September 2015

Using System Hyper Pipelining for a Multi-Threaded FORTH Compatible Stack Processor Mapped on an FPGA

Tobias Strauch

***Abstract*— FORTH compatible stack processors are still subject to research and development. They also face new multi-processing and multi-threading challenges. Most notoriously the multicore processor G144 from the inventor of FORTH Charles Moore. This paper outlines an alternative concept to generate a mutli-threaded FORTH compatible stack processor by using System Hyper Pipelining (SHP), which overcomes the limitations of classical multi-threading methods by adding thread stalling, bypassing and reordering techniques to better cope with the challenges of Symmetrical Multi-Processing and Simultaneous Multi-Threading. SHP is ideal for FPGAs with their high number of registers and their flexible memory usage. The paper gives results for a FORTH compatible stack processor mapped on an FPGA.**

Keywords—FORTH, C-Slow Retiming, Symmetrical Multi-processing, Simultaneous Multi-threading, Multi-processor Systems

I. Introduction

Charles Moore, the inventor of FORTH introduced a multicore array chip based on a FORTH compatible stack processor in [1]. This design uses the traditional approach of generating a multicore chip based on an array-like orientation. This paper now proposes an alternative approach of generating a multi-threaded stack processor by using System Hyper Pipelining, which is a successor of C-Slow Retiming.

C-Slow Retiming (CSR) provides C copies of a given design by inserting registers and reusing the combinatorial logic in a time sliced fashion. CSR therefore improves the performance per area factor. Leiserson et. al. introduced the concept of C-Slow Retiming (CSR) in [2]. In section II, the System Hyper Pipelining technology is shown and how it differs from CSR. Section III outlines a thread controller which copes with a high number of threads. Section IV outlines, how a FORTH based stack processor can benefit from SHP. A system is proposed in section V before the results are given in section VI.

II. CSR and SHP Technology

System Hyper Pipelining (SHP) has been introduced by Strauch in [3]. This paper gives a 2-page introduction for the readers' convenience again. SHP is based on C-Slow Retiming (CSR). It enhances CSR with thread stalling, bypassing and reordering techniques by replacing the original registers of the design with memories and by adding a thread controller (TC). In the remainder of this paper, the word "thread" (T) is used synonym for the execution of a program or algorithm.

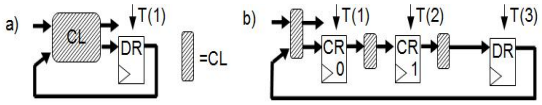


Figure 1: a) Simplified single clock design. b) Applying CSR technique.

Figure 1a shows the basic structure of a sequential circuit with its inputs, outputs, combinatorial logic (CL) and original design registers (DR). The sequential circuit handles one thread T(1). Figure 1b shows the CSR technique. The original logic is sliced into C (here C=3) sections. This results in C functionally independent design copies T(C=1..3) which use the logic in a time sliced fashion. Each thread has its own thread ID (TID). For each design copy it now takes C "micro-cycles" to achieve the same result as in one cycle (called "macro-cycle") of the original design. The implemented registers are called "CSR Registers", (CR) and are placed at different C-levels (CRn).

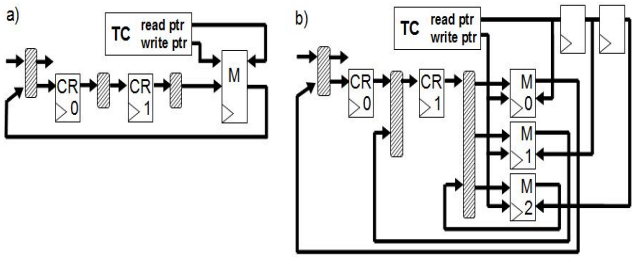


Figure 2: a) SHP-ed design with thread controller, memories and CRs. b) Advanced SHP.

Figure 2a shows the modifications of a CSR-ed design towards SHP. Assuming the DRs are now replaced by a memory (M). The incoming design states / threads are stored at the relevant address (write pointer) based on the TID. D is the number of threads which the memory can hold (memory depth). The outgoing thread can now be freely selected within D available threads (read pointer), except the threads already passing through the design logic. A CSR-ed design has usually many shift registers. DRs are followed by a series of CR

registers. In the SHP-ed version, many memory data outputs are connected to CRs. In this case, the shift registers at the outputs can be replaced by registers at the read address inputs of the memories (Figure 2b). The memory is sliced into individual sections (M0, M1, M2) and each section has a delayed read of the thread. The outputs can now be directly connected to the relevant combinatorial logic and the shift registers can be removed. The same trick can be applied on the shift register chains at the inputs of the memory.

$$Fcsr = Forig * C * r^{C} \text{ with } r \sim 0.93 \quad (1)$$

$$0\ Hz <= Ft <= Forig * r^{C} \quad (2)$$

$$Fshp = \Sigma\ Ft <= Fcsr \quad (3)$$

We define Forig as the maximal speed of the original design. The maximal speed of a CSR-ed design can be estimated by using Equation 1. Fcsr is C times the original speed Forig reduced by a correction factor r^{C}, which considers the delay inserted on the critical path by the CRs. r is technology dependent. Based on empirical data, r is roughly 0.93 for a Virtex-6 FPGA. Equation 2 says, that in an SHP-ed design, a single thread can now run at any speed (over a long period) between 0 Hz (stalled) and Forig * r^{C}. The maximal speed of an SHP-ed design Fshp is the sum of all active threads (Equation 3). Fshp cannot be greater than Fcsr.

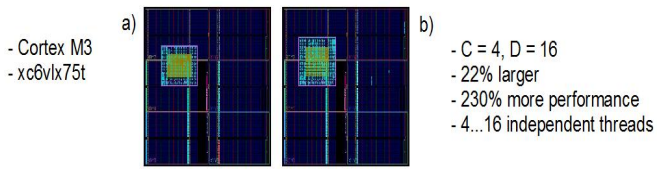


Figure 3: SHP based performance per area improvement based on a Cortex M3 example.

Figure 3a shows a Cortex M3 (as it can be found in [4]) implementation on a a Virtex-6. With C=4 and D=16, the SHP-ed version (Figure 3b) is just 33% larger (occupied slices) but can achieve 230% more performance (overall 330%) compared to the original implementation. In other words, SHP improves the performance per area factor if the application can utilize this performance gain by using at least 4 independent threads.

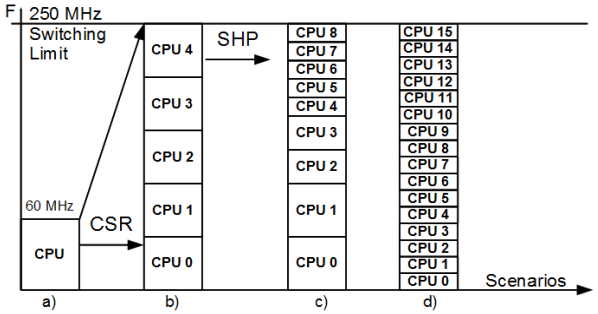


Figure 4: Histogram of different scenarios (a-d) of running CSR and SHP.

Figure 4 shows the advantages of CSR and SHP over the original design. The x-axis shows different scenarios. Assuming a single CPU runs at 60MHz on an FPGA (Figure 4a). It can be seen, how CSR improves the system performance of the original system implementation, (Figure 4b). When using CSR, the system performance is not necessarily limited by the critical path of the original design, but - for instance - by the switching limit of the FPGA (e.g. 250MHz) or the external memory access instead.

There are two key observations when SHP is used on a design. First, for executing multiple programs on multiple CPUs (symmetrical multi-processing (SMP)) or for executing multiple threads on a CPU (simultaneous multi-threading (SMT)), SHP allows a more efficient usage of the system resources. It adds the possibility to dynamically scale the system performance over a minimum (C, Figure 4b), and a maximum (D, Figure 4d) set of design copies, whereas any solution in-between can be realized (Figure 4c). This load balancing is handled by a thread controller (TC).

Secondly, threads don't interact with each other. There is no register dependency between the individual threads. The runtime of each thread is therefore deterministic. The variable latency that the execution per thread may experience due to different behavior in if-branches for instance is not an issue, because all threads work independent of each other.

III. The Thread Controller

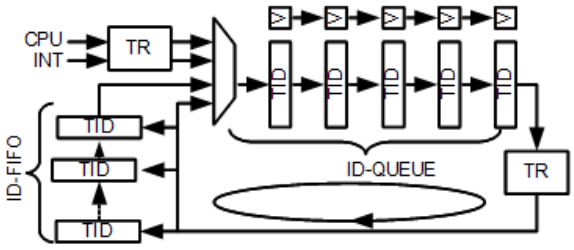


Figure 5: Thread Controller Mechanism.

A TC is used, which is controlled by a special function register set TCRS (Thread Controller Register Set). It is accessible by all active threads (T). Each thread has its own thread register (TR). Figure 5 shows how the Thread ID (TID) is provided for the SHP memories. When a thread is executed, its TID passes through the ID-Queue (IDQ). It is reinserted into the IDQ or into the ID-FIFO, if the relevant bit in the TR shows that the thread is still active and not on hold or killed. When less than C threads are valid, active threads need to be re-executed, but the valid bit V of the IDQ indicates, that its state copies should not be stored. When more than C threads are executed or an additional thread is inserted, then the TID is parked in the ID-FIFO.

Threads can be added to the active thread list by writing their program start address to the “Activate” register. This sets the thread-specific active A bit in the TR. Threads can be hold by setting the hold H bit or killed by clearing the active A bit in the TR. When the thread priority bit P is set in the TR, then

a thread execution has a higher priority than the threads stored in the ID-FIDO or threads resulting from an interrupt or the CPU. It is therefore directly re-inserted into the IDQ again and not stored into the ID-FIFO.

To cope with **fork-join** queueing, the following mechanism is implemented. A set of Ts an be started from a single main thread (MT) by successively writing the individual start addresses of the Ts to be started to the TCRS called "Activate and Count" (AC). By doing that, the number of Ts called (CT) by the MT is stored in the AC register. Optionally the MT stalls itself after that process. Each CT saves the MT's SID in the "forked thread register" (FT). When a CT is killed, it checks the FT and decrements the AC of the MT. If this number gets 0, the MT stalling bit is cleared by default and the MT continues. Alternatively the MT can read it's AC register to continue execution.

The TCRS can be programmed to set a group of consecutive threads into dynamic length instruction word (**DLIW**) mode. By doing that, a given number of threads are executed in parallel. The concept is similar to the very long instruction word (VLIW) concept, except the fact that the number of threads running in parallel can be dynamically defined (but must be lesser or equal to D).

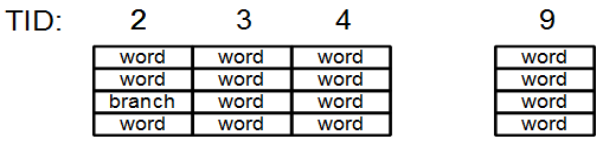


Figure 7: Grouping threads to run dynamic length instruction words.

Figure 7 shows an example. The threads with the TID 2,3 and 4 are running as a group using DLIW. Thread 9 is listed to show that alternative threads can still be actively running. The first thread in the DLIW (TID 2) starts the parallel execution when the number of subsequent threads (here 2) is written into its DLIW register. Branch instructions (words) are only consider by the initial thread (here TID = 2), the program counter of the subsequent threads are always derived from its trailing thread (by adding the value of 2 for instance).

This outlined TC has a low complexity (see result section). It can stall and bypass individual threads and it is capable of handling fork-join queues. By default, a thread runs completely independent of the other threads when its priority thread is set and when only less or equal number of C threads have the priority bit set. It can also group threads to run dynamic length instruction words. The DLIW method is accompanied by the message passing implementation, which is outlined in the sequel of this paper.

IV. Benefits of a Multi-Threaded Stack Processor

A. ... compared to its single core implementation

This sections lists some of the benefits of an SHP based multi-threaded stack processor. First, it is compared to a single core implementation. It is assumed, that the application can be partitioned into individual threads to a certain degree. This certainly has its limit (Amdahl's Law), but applications in the field of automation and controlling for instance can usually be partitioned into individual tasks, whereas most of them can be executed in parallel.

The key benefit of SHP when applied on a single core is the increase of the performance-per-area (PpA) factor. This has been shown in [3] based on FPGAs and it is demonstrated in the result section of this paper again. The performance of a system is already increased when a second thread can be used. All threads of an SHP-ed processor can share the same data. The time sliced access to the program memory increases the memory bandwidth and reduces potential memory-wall problems.

B. ... compared to a multicore implementation

Charles Moore has introduced a multicore array stack processor GA144 [1]. A functional identical SHP-ed version can be realized on a smaller die size due to the increased ppa factor. Alternatively more performance could be realized on the same area when SHP is used.

The GA144 was used by Schneider et al. in a research project [5]. In their work, an application is partitioned into individual tasks, whereas each task is assigned to an individual core on the GA144. It is easy to understand, that in this case, tasks have to stall sometimes, because they need to wait on data from other tasks to be processed. It has been outlined in section II of this paper, that SHP allows the dynamic scaling of individual threads on an SHP-ed processor. If a task can be stalled (because it is waiting for data from other tasks for instance), then its associated thread can be stalled and therefore frees performance for other threads. A task on the SHP-ed processor array does not necessary consume performance nor logic area when stalled.

On an SHP-ed based processor array, multiple threads (D) on each element can share the same memory. On a traditional multicore array (GA144), data has to be transferred to the individual core/thread.

C. ... when used in a safety critical environment

FORTH compatible stack processors can be used in a safety critical environment. Most notoriously is the controlling of the Philae landing process, using FORTH and a radiation hardened processor [6]. A C-slow retimed processor can be used to generate a time redundant system, as outlined in [7]. It enables the detection of single event upsets (SEUs) and allows an on-the-fly recovery. The same technique can be applied on almost the same area on an SHP-ed stack processor.

As an alternative concept to detect malfunctions of an application running on a stack processor is the usage of different software tasks, which are aiming to deliver the same results. If the results differ, at least one task did not execute the code as expected. The "free" additional threads that come

with SHP and its increased PpA factor enable the execution of redundant tasks on almost the same die size without loosing a reasonable amount of system performance.

D. ... and what turned out to be not very beneficial

The idea of combining a FORTH compatible SHP-ed processor with OpenMP [8] was not very successful. Although some OpenMP concepts can be used in the FORTH language, the restriction comes from the write/read policies of private/shared variables used by individual tasks. Still, the programmer can use the TCRS to benefit from the implemented fork-join mechanism.

Another intriguing idea when working with an SHP-ed FORTH processor is that each thread can access the stack of alternative threads. It turned out that the resulting logic is too complex and therefore inefficient. Alternatively, threads can be synchronized using the TC's DLIW technique as well as the message passing method, which is outlined in the next section.

V. The Proposed System

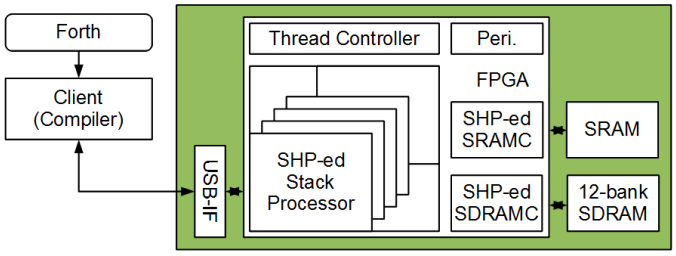


Figure 8: Overview of the Proposed System.

A. Overview

Figure 8 gives an overview of the proposed system. It is based on the diploma-thesis of G. Hohner, which is released on the OpenCores webpage [9]. A FORTH compatible program can be compiled by the original client (compiler). The program is then downloaded through an USB interface to a 12-bank wide SDRAM block. The original stack processor is enhanced by the SHP technology. A thread controller is added which can be programmed by the processor using special function registers (SFR). There are also some standard peripherals like GPIO, UART and Timer. The design is mapped on an FPGA. An external SRAM provides enough memory for data access.

B. The FORTH Stack Processor and its SHP-ed Version

The CPU is a FORTH compatible stack processor with 6 stages and two 32-bit wide stacks. It supports all common FORTH commands (see [9] for more details).

The CPU is slightly optimized so that it can be used for an automatic transformation process towards an SHP-ed version, which is done by a tool called CoreMultiplier [10]. Based on empirical data of other CPUs of comparable complexity, the parameter C was set to 4, which results in a good performance-versus-area (occupied slices) trade-off. In other words, 3 (C-1) registers are automatically inserted into each path in the CPU by a timing driven algorithm (, whereas some of the registers are merged into their adjacent memory blocks again, see section II). The parameter D was initially set to 16. Less than 16 threads do not reduce the number of occupied memory resources. Due to the high registers count of the CPU, D was then reduced to 8 so that the stacks can share FPGA memory resources.

The CPU accesses an external SRAM using an SHP-ed SRAM controller for data transfer and a external SDRAMs using an SHP-ed SDRAM controller for the program code. The external SDRAM is based on 16-bit wide devices so that a pair generates a 32-bit wide interface. Three individual SDRAM pairs build an SDRAM list with 12 banks. This allows individual threads to access individual banks. Each thread can access the complete SRAM range and the complete SDRAM range as program memory.

C. The Message Passing Extension

Before SHP was applied on the original design, one additional coprocessor register COR is added. A new FORTH instruction CTC (copy to coprocessor) writes the stack value into COR. An additional instruction CFC (copy from coprocessor) writes the COR value back onto the stack. The SHP-ed version was then modified so that each thread writes the stack's value into the COR of the thread with the next TID. This adjacent thread can then read this value one cycle later by using the CFC word. This mechanism is very helpful when the DLIW method is used and a message needs to be passed to another thread.

D. The FPGA Board

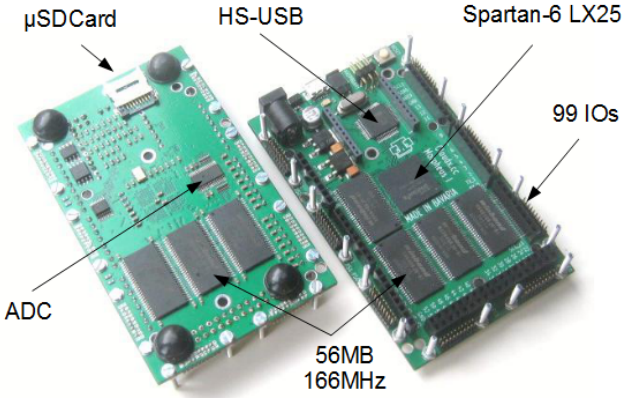


Figure 9: FPGA Board with unique SDRAM access structure.

The proposed system is mapped on a Spartan 6 LX25 FPGA board with a unique SDRAM access structure (see Figure 9). It allows the individual threads on the CPU to access up to 12 individual SDRAM banks. Without that infrastructure, the memory bandwidth would be a major bottleneck (memory wall).

E. The Software Compiler and Client

The FORTH compiler and the client are taken over from the original source [9], which is written in Java. The two

additional instructions (CTC and CFC) are added to the compiler code. The communication with the board was enhanced for HS-USB using an FTDI [11] DLL and an FTDI USB chip.

VI. Results

In this section, the original design variations are compared to the system hyper pipelined version of its single core implementation. The original work includes a feature to automatically generate a multicore solution by modifying a parameter called "core". By increasing the core parameter, the number of CPUs and CPU stacks increases accordingly.

Table 1. Comparing Performance per Area of the Original Design and the SHP-ed Version.

	unit	original, core =				SHP, C = 4, D = 8
		1	2	3	4	
occS		1377	1865	2380	3143	1927
Perf.	MHz	45,44	45,44	45,44	45,44	159,12
PpA	kHz/occS	32,99	24,36	19,09	14,45	82,57
ΔPpA	%	100	73,83	57,86	43,81	250,23

Table 1 compares the results for the different implementations mapped on a Spartan 6 LX25 FPGA. The occupied slices (occS) of the original design with increasing core factor (1, …, 4) is shown. The performance remains stable for all 4 core variations at 45,44 MHz. This decreases the performance-per-area factor (PpA) due to the increased number of occS. The SHP-ed achieves a performance of 159,12 MHz on 1927 occS, which results in a PpA of 82,57 kHz/occS. This is a PpA increase of 250% compared to a single core implementation of the original core. The occS number of the SHP-ed version includes the design of the TC, which consumes just 226 slices of the FPGA.

Further performance tests are not conducted, because they heavily depend on how the algorithm can be partitioned into multiple independent threads. The runtime of the original program and the runtime enhancements when using the multithreaded SHP-ed version can easily be derived from the numbers given in Table 1.

VII. Conclusion

This paper showed how C-Slow Retiming (CSR) and parallel programming can be combined to a new method called System Hyper Pipelining (SHP). SHP benefits from the higher performance per area (PpA) factor, which can be achieved when using CSR. Additionally, SHP offers also flexible thread stalling, bypassing and reordering features which are used by multi-threading methods to improve the system performance.

SHP is applied on a FORTH compatible stack processor. This stringent transformation process can be automatically accomplished within seconds and results in a multi-threaded version of the stack processor. Fig. 4 shows, how the increased system performance can be distributed among multiple design copies by using a thread controller. Individual threads can run at different speeds and can even be completely stalled without consuming relevant power anymore. The paper shows how a thread controller enables fork-join operations by accessing its special function registers. Also very large instruction words (VLIW) can be executed by running consecutive threads. In the proposed system the VLIW can also have a dynamic length.

The system is mapped on an FPGA. The increased system performance though requires an enhanced memory access method to reduce the potential memory bottleneck. A hardware solution with 12 SDRAM banks is proposed. The time shared memory access works in-line with the time-shared mechanism used to duplicate the functionality of the FORTH compatible stack processor.

References

[1] GreenArrays. B001 - F18A Technology Reference. Available online: www.greenarraychips.com/home/documents/greg/DB001-110412-F18A.pdf, as of April 21, 2013.

[2] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry", Algorithmica, vol. 6, no. 1, pp. 5-35, 1991.

[3] T. Strauch, "The Effects of System Hyper Pipelining on Three Computational Benchmarks Using FPGAs", 11th Intern. Symposium in Applied Reconfigurable Computing, ARC 2015, 13-17 April 2015, Bochum, Germany, pp. 280 – 290

[4] Atmel, "AT91SAM ARM based Flashed MCU", Available online: http://www.atmel.com/Images/doc11057.pdf

[5] T. Schneider, I. Von Maurich, and T. Guneysu, "Efficient implementation of cryptographic primitives on the GA144 multi-core architecture", 24th Intern. Conf. on Application-Specific Systems, Architectures and Processors (ASAO), IEEE 2013, 5-7 June 2013, Washington, pp 67-74.

[6] MPE Microprocessor Engineering, "Comet Landing – a triumph for Forth in Hardware and Forth in Software", Press Release, 13th November 2014, Southampton, UK

[7] T. Strauch, "Using C-Slow Retiming in Safety Critical and Low Power Applications", First Intern. Workshop on FPGAs in Aerospace Applications, FASA 2014, 5th September 2014, Munich, Germany, pp. Tpd.

[8] OpenMP, "The OpenMP API Specification for Parallel Programming", Available online: www.openmp.org

[9] Opencores, Stockholm, Sweden, 2007, Available online: www.opencores.org/projects

[10] Edaptix, CoreMultiplier, Munich, Germany, Available online: www.edaptix.com/coremultiplier.htm

[11] FTDI, Avalable online: www.ftdichip.org

uCore goes floating point

Klaus.Schleisiek at spacetech-i.com

As every good Forth programmer I despised floating point. Real men use fixed point. Until I was supposed to compute the following expression in the Merlin project, which needs to stabilise laser frequencies to 20ppm precision in order to hit the methane absorption maximum. That implies that the laser temperature has to be stabilised as well, using a peltier element as the actor and an NTC resistor as the sensor.

$$T = \frac{B}{\ln(R/r_\infty)} \qquad R = r_\infty e^{B/T}$$

Therefore, I was confronted with these equations to compute the temperature from the resistance and to find the resistance set point for a certain temperature. (r_∞ is a pre-computable constant.)

At first, I used Matlab to compute a 3rd order polynomial function that fits "reasonably well" in the temperature range of interest. Nonetheless it was not easy to get the scaling right using */. The precision was disappointing and to make things worse, the error distribution of the two functions were inconsistent.

This was the starting point to rething my resistance to floating point. As a motivation, I will show you what I ended up with to solve the problem:

```
&3892  float Constant B-factor
-&298  float Constant -T0
&10000 float Constant R0
&27300       Constant 0_degC

B-factor -T0 f/ R0 fln f+ fexp Constant R_lim

: R>T   ( Ohm -- degC*100 )
   float R_lim f/   fln   B-factor swap f/   &100 float f* integer 0_degC -
;
: T>R   ( degC*100 -- Ohm )
   0_degC + float &100 float f/  B-factor swap f/   fexp   R_lim f*   integer
;
```

That's the code needed for the conversion functions that have fixed point numbers as inputs and outputs, scaled to Ohm and centidegC. To add even more to the motivation: All the floating point code needed cross-compiles into just 500 instructions (bytes).

Design principles

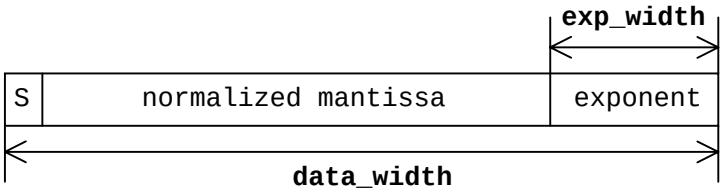
uCore has a configurable data word width and of course, the floating point representation must be able to cope with it. Therefore, IEEE-754 serves as a guideline and interchange format, but not as an implementation standard.

Nowadays, even on uCore, the **data_width** is at least 24 bits wide. Therefore, floating point numbers will fit on the stack. That already saves the code for a separate floating point stack.

Real number string input and output: Having written a floating point package for the RTX-2000 some 20 years ago, I remembered that about a third of the code dealt with proper number input and output. Not desirable. I have chosen a much simpler solution: integers can be converted to floating point using FLOAT, floating point numbers to integers using INTEGER. The floating point numbers can be properly scaled to engineering values using KILO, MEGA, MILLI, and MICRO in such a way that they properly scale using standard Forth number input and output.

Floating point representation

Floating point numbers are characterised by **exp_width**, the width of their exponent field. Whereas **data_width** is a more general parameter that specifies the native cell size of a uCore instantiation.



Exponent: I chose to put the exponent at the far right in order to cope with the variable data width easily. For an exp_width of 8 bits IEEE-854 adds a bias value of $7F to the exponent. That involves an add. uCore is just flipping the sign of the exponent using $80 xor, which consumes fewer logic resources.

Mantissa: IEEE-854 stores the mantissa as an absolute value preceeded by its sign. No rational is given but I suppose it has to do with the representation of + and - zero. It also avoids the singularity of the most negative number of 2s-complement representation. You don't know what I mean? - Just do "$80000000 abs u." in any 32-bit Forth. But computing the absolute value also involves an add and carry propagation and therefore, uCore stores the mantissa in 2s-complement representation.

I am still not sure whether there are more important reasons for IEEE's choices, but I have not seen any numerical misbehaviour due to uCore's non-standard representation.

Zero: uCore also has a positive and a negative zero. Given above choices, the positive floating point number zero is just - zero. Which is nice. When this zero has its sign set, it is a negative zero, making good use of the $80000000 sigularity. The check for floating zero is simple:

```
: f0= ( real -- flag )  2* 0= ;
```

There are only very few situations where the negative zero has to be explicitly handled and therefore, I believe it is a good choice.

Over/underflow: On overflow, the ovfl status bit will be set, and the largest positive or negative number will be returned depending on the expected sign of the result. On underflow, the unfl status bit will be set, and a positive or negative zero will be returned. For simplicity, there are no NaNs (Not-a-Number).

Hardware support

Four words have been implemented as uCore instructions for speed of execution:

`*. ( n u -- n' )`

It is used to compute mathematical functions based on polynomial expressions according to Horner's scheme: $(...((c_n * x + c_{n-1}) * x + c_{n-2}) * x + ... + c_0)$.

`normalize ( man exp -- man' exp' )`

The mantissa and the exponent are on the stack, both as 2s-complement numbers. Normalize shifts the mantissa to the left until only one single "leading" sign bit remains. The exponent is adjusted accordingly. This instruction can take several cycles depending on the magnitude of the mantissa. It gave rise to a uCore invention: Interruptible auto repeat instructions, which are simplifying uCore's instruction set considerably: um/mod, m/mod, um*, m*, sqrt, log2, shift, ashift can now be implemented as single instructions.

`>float ( man exp -- real )`

Amalgamates the mantissa and the exponent on the stack into a real taking care of the configurable floating point format. It also takes care of over/underflows, because the 2s-

complement exponent on the stack may not fit into the exponent field. The leading digit of a normalized mantissa will be dropped, because by definition its value is the inverted sign bit.

`float> ( real -- man exp )`

Converts the configurable real number into the mantissa and the exponent, both as full 2s-complement numbers.

Host support

I want to be able to compute floating point numbers "on the fly" during compilation. Therefore, a matching set of floating point words must be present on the host gforth system to support the uCore cross-compiler.

It is implemented in such a way that it takes care of the potential cell size difference of the host and the target. The host is characterized by constant **`cell_width`**, which can be determined automatically, and the target is characterized by constant **`data_width`**.

I have appended the code for gforth in the appendix and it can be downloaded at http://www.forth-ev.de/repos/microcore/trunk/Microcore/floating_point

Mathematical functions

Some functions can be computed using bit step algorithms: These are - besides multiply and divide - square root and logarithm (see **`log2)`**. In principle, the **`exp2`** function could be computed bit wise as well, but it needs to take the square root in each step and therefore, a polynomial approximation is more efficient.

The other functions will have to be approximated to sufficient precision. In general, a real number will be split up into an integer (before the decimal point) and a fractional part (after the decimal point) after appropriate scaling. Then the fraction will feed the approximation function and the integer part will be handled in a function specific way. Most functions can be approximated by polynomials, which are evaluated using Horner's scheme and the **`*.`** operator.

My bible for function approximations is "Computer approximations" by John F. Hart, Wiley&Sons. It discusses the methods needed and presents coefficient sets for different precisions.

In the implementation for **`exp2`** and **`sin`** I have used the original coefficients from Hart and an on-the-fly scaling scheme to adapt to different data_widths. More functions will be added as the need arises.

Numerical precision

It is possible to compile the gforth code for different **data_width** and **exp_width** settings. Using

```
: ntc-test ( -- )
   &1000 &50000 bounds DO  cr I . I r>t dup . t>r . &1000 +LOOP ;
```

we get a good impression how the numerical precision of the **`fln`** and **`fexp`** code degrades when cutting down on the data_width. A **data_width** of 23 and an **exp_width** of 6 still produces results, which are far better than the initial integer based 3rd order polynomial approximation. Smaller data_widths damage the exp2 function, a smaller exp_width can not cope with the dynamics of the expressions any more.

This is a satisfying result, because in small systems I am usually using a 24 or 27 bit data_width.

Immenstaad, 1-Oct-2015

```
Only Forth also definitions

: shift  ( n1 quan -- n2 )   dup 0< IF  abs rshift  EXIT THEN  lshift ;
: ashift ( n1 n2 -- n3 )     dup 0< IF  negate 0 DO 2/ LOOP EXIT THEN  0 ?DO 2* LOOP ;
: u2/    ( u1 -- u2 )        1 rshift ;
: 2**    ( n -- 2**n )       1 swap lshift ;
: m/mod  ( d n -- rem quot ) fm/mod ;
: \\     ( -- )              source-id IF  BEGIN  refill 0= UNTIL  THEN  postpone \ ;

: cell_width ( -- u )  \ cell width of the host Forth system
   0 1 BEGIN  swap 1+ swap  2* ?dup 0= UNTIL
;
&32                         Constant data_width  \ cell width of target system
8                           Constant exp_width   \ width of exponent field
cell_width data_width -     Constant delta_width \ cell width >= data width !

data_width 1- 2**           Constant #signbit
exp_width 2** 1-            Constant #exp_mask
#exp_mask invert            Constant #man_mask
#exp_mask 2/ invert         Constant #exp_min
#exp_mask #exp_min and      Constant #exp_sign
#signbit                    Constant #fzero_neg
0                           Constant #fzero_pos
#signbit #exp_mask or       Constant #fmax_neg
#signbit invert             Constant #fmax_pos
-1 delta_width negate shift Constant #data_mask

Variable underflow  0 underflow !
Variable overflow   0 overflow !

Variable Scale  \ used for optimal scaling of a set of polynomial coefficients
: scaled     ( n -- n' )  s>d data_width 1 - 0 DO  d2*  LOOP Scale @ fm/mod nip ;
: scale_factor  ( n -- )  Scale ! ;

: round   ( dm -- m' )
   over 0< 0= IF  nip  EXIT THEN   \ < 0.5
   swap 2* IF  1+  EXIT THEN       \ > 0.5
   dup 1 and +                     \ = 0.5, round to even
;
: *.  ( n1 u -- n2 )  over 0< IF  swap negate um* round negate  EXIT THEN  um* round ;

: normalized?  ( m -- f )   dup #signbit and swap #signbit u2/ and 2* xor ;

: normalize    ( m e -- m' e' )
   over normalized? ?EXIT
   over 0= IF  drop   #exp_min  EXIT THEN
   BEGIN  dup #exp_min = ?EXIT
          1 -   swap 2* swap   over normalized?
   UNTIL
;
: >float  ( m e -- r )
   overflow off   underflow off
   normalize   swap #man_mask and swap
   over #fzero_neg =   over #exp_min =   and >r
   over #fzero_pos =   r> or
   IF  drop  #exp_mask invert and  EXIT THEN    \ leave floating +/-zero.
                                                \ For +zero irrespective of exponent
   dup #man_mask 2/ and
   dup 0< IF  #man_mask 2/ xor  THEN            \ exponent over/underflow?
   IF  0< IF  underflow on   0< IF  #fzero_neg  EXIT THEN  #fzero_pos  EXIT
        THEN   overflow on   0< IF  #fmax_neg   EXIT THEN  #fmax_pos   EXIT
   THEN
   dup #exp_min =
   IF  drop #man_mask and  EXIT THEN     \ smallest exponent => denormalized
   #exp_mask and   #exp_sign xor   swap  \ flip sign of exponent => bias = #exp_min
   dup 2* [ #signbit invert #exp_mask invert and ] Literal and
   swap 0< IF  #signbit or  THEN  or
;
```

```
: float>  ( r -- m e )
   dup #exp_mask and   ?dup 0= IF  #exp_min  EXIT THEN   \ de-normalized
   dup #exp_sign and IF  #exp_mask 2/ and
                     ELSE  #exp_mask 2/ invert or
                     THEN  swap                          \ flip sign and extend
   dup 0< IF  #exp_mask 2/ or  2/                        \ add 0.5 for rounding
              [ #signbit #exp_sign or u2/ invert ] Literal and
         ELSE  #man_mask   and u2/                       \ add 0.5 for rounding
              [ #signbit #exp_sign or u2/ ] Literal or
         THEN  swap
;
: int.frac  ( r -- frac int )  \ split float number into integer and fractional part
   float> [ data_width 2 - ] Literal +
   dup 0< IF  invert 0 ?DO  u2/  LOOP  2* 0  EXIT THEN
   0 swap [ delta_width 2 + ] Literal + 0 DO  d2*  LOOP
;
data_width &32 = [IF]

: >ieee ( r -- ieee )  \ only valid for 32-bit data_width
   float> $80 xor $7F + $FF and                    \ exponent
   over 0< IF  $100 or  THEN  &23 shift swap       \ sign
   abs -&7 shift $7FFFFF and or                    \ mantissa
;
: ieee> ( ieee -- r )  \ only valid for 32-bit data_width
   dup   dup 0< IF  negate $7FFFFF and $1000000 or
                ELSE  $7FFFFF and $800000 or
                THEN  7 shift
   swap -&23 shift $7F -  dup $80 and IF  $7F and  ELSE  $7F invert or  THEN  >float
;
[THEN]

: f+   ( r1 r2 -- r3 )
   float>   rot float>   rot 2dup -                                  \ m2 m1 e1 e2 e1-e2
   dup 0< IF  swap >r nip  ELSE  rot >r nip >r swap r> negate  THEN \ m> m< diff_e1-e2
   1- dup [ data_width exp_width - negate ] Literal u< IF  drop 0 swap  THEN
   over IF  ashift  ELSE  drop  THEN  swap 2/ +   r> 1+ >float
;
: f*   ( r1 r2 -- r3 )
   float>   rot float>         \ m2 exp2 m1 exp1
   rot + data_width + -rot     \ exp3 m2 m1
   m* delta_width 0 ?DO  d2*  LOOP
   nip swap >float
;
: f/   ( r1 r2 -- r3 )   overflow off
   dup 2* 0= IF  invert xor #signbit and invert  overflow on  EXIT THEN
                                          \ leave +/- largest number on / by zero
   float>   rot float>
   data_width - rot - -rot
   0 swap delta_width 2 +  0 ?DO  d2/  LOOP
   rot m/mod nip   swap 2 + >float
;
: fnegate  ( r -- -r )
   dup 2* IF  float> 1+ swap 2/ invert #exp_sign 2/ + swap >float  EXIT THEN
   0< IF  0  EXIT THEN  #signbit               \ handle + and - zero
;
: fabs    ( r -- |r| )      dup 0< IF  fnegate  THEN ;
: f-      ( r1 r2 -- r3 )   fnegate f+ ;
: f<      ( r1 r2 -- f )    f- 0< ;
: f>      ( r1 r2 -- f )    swap f- 0< ;
: f<=     ( r1 r2 -- f )    f> 0= ;
: f>=     ( r1 r2 -- f )    f< 0= ;
: f0=     ( r -- f )        2* 0= ;
: f0<     ( r -- f )        0< ;
: f2*     ( r1 -- r2 )      float> 1+ >float ;
: f2/     ( r1 -- r2 )      float> swap 2/ swap >float ;

: float   ( n -- r )        0 >float ;
```

```
: integer ( r -- n )
   dup 2* #data_mask and 0= IF  2*  EXIT THEN  \ +/- zero
   1 float f2/ f+                              \ add 0.5 for rounding
   float> ashift
;
: 1/f     ( r1 -- r2 )      1 float swap f/ ;

: fscale  ( r1 n -- f2 )    dup 0< IF  abs float f/  EXIT THEN  float f* ;
: milli   ( r1 -- r2 )      -&1000 fscale ;
: micro   ( r1 -- r2 )      -&1000000 fscale ;
: kilo    ( r1 -- r2 )       &1000 fscale ;
: mega    ( r1 -- r2 )       &1000000 fscale ;

\ ******************************************************************************
\ logarithm, exponential
\ ******************************************************************************

: log2 ( frac -- log2 )   \ Bit-wise Logarithm (K.Schleisiek/U.Lange)
   delta_width 0 ?DO  2*  LOOP
   0   data_width 0
   DO  2* >r   dup um*
      dup 0< IF  r> 1+ >r  ELSE  d2*  THEN     \ correction of 'B(i)' and 'A(i)'
      round   r>                               \ A(i+1):=A(i)*2^(B(i)-1)
   LOOP  nip
;
: ?fzero ( r -- r / rdrop !! )  \ careful: manipulates rstack!
   dup 2* #data_mask and ?EXIT  drop #fmax_neg   overflow on   rdrop ;

: flog2  ( r1 -- r2 )  \ only defined for positive values
   ?fzero float> [ data_width 2 - ] Literal + 0 >float   swap
   abs 2* log2 u2/ [ data_width 1 - negate ] Literal >float f+
;
: exp2  ( ufrac -- uexp2 )
            \ Hart 1042, 23 bit precision, 1 > ufrac > 0, 1 = 2**(cell_width-1)
   [ &001877576 &008989340 + &055826318 +
     &240153617 + &693153073 + &999999925 + scale_factor ]
   >r [ &001877576 scaled ] Literal   r@ *.
      [ &008989340 scaled ] Literal + r@ *.
      [ &055826318 scaled ] Literal + r@ *.
      [ &240153617 scaled ] Literal + r@ *.
      [ &693153073 scaled ] Literal + r> *.
      [ &999999925 scaled ] Literal +
;
: +fexp2 ( r1 -- r2 )
   int.frac 2** float   swap exp2 [ data_width 2 - negate ] Literal >float f*
;
: fexp2  ( r1 -- r2 )   dup f0< IF  fnegate +fexp2 1/f  EXIT THEN  +fexp2 ;

&1442695 float micro Constant log2(e)

: fln  ( r1 -- r2 )  ?fzero flog2 log2(e) f/ ;

: fexp ( r1 -- r2 )  log2(e) f* fexp2 ;

\ ******************************************************************************
\ sine, cosine
\ ******************************************************************************

: sin ( ufrac --- usin )
           \ HART 3341  27 bit precision, pi/2 > frac >= 0, 1 = 2**(cell_width-2)
   [ &000151485 -&004673767 + &079689679 +
     -&645963711 + &1570796318 + 2* scale_factor ]
   dup >r  dup *. >r
   [  &000151485 scaled ] Literal   r@ *.
   [ -&004673767 scaled ] Literal + r@ *.
   [  &079689679 scaled ] Literal + r@ *.
   [ -&645963711 scaled ] Literal + r> *.
   [ &1570796318 scaled ] Literal + r> *.
;
&1570796327 float milli micro Constant fpi/2
```

```
: +fsin  ( r1 -- r2 )
   fpi/2 f/   int.frac >r
   r@ 1 and IF  invert  THEN  sin
   r> 2 and IF  negate  THEN
   [ data_width 2 - negate ] Literal >float
;
: fsin  ( r -- r' )  dup f0< IF  fnegate +fsin fnegate  EXIT THEN  +fsin ;

: fcos  ( r -- r' )  fpi/2 f+ fsin ;

: degree ( fdeg -- frad )   [ fpi/2 &90 float f/ ] Literal f* ;

\ ***************************************************************************
\ Converting NTC resistance to temperature and vice versa
\ ***************************************************************************

&3892  float Constant B-factor
-&298  float Constant -T0
&10000 float Constant R0
&27300       Constant 0_degC

B-factor -T0 f/ R0 fln f+ fexp Constant R_lim

: R>T   ( Ohm -- degC*100 )
   float R_lim f/   fln   B-factor swap f/   &100 float f* integer 0_degC -
;
: T>R   ( degC*100 -- Ohm )
   0_degC + float &100 float f/  B-factor swap f/   fexp   R_lim f*   integer
;
\ : ntc_test  ( -- )   &1000 &50000 bounds DO  cr I . I r>t dup . t>r . &1000 +LOOP ;
```

Components for Certification.

Paul E. Bennett IEng MIET
Systems Engineer, HIDECS Consultancy

Abstract

Even the humble hexagonal nut has a data-sheet that describes its functionality, performance factors, interfaces and limits beyond which its continued performance is not guaranteed. Electrical and Electronic Components also have data-sheets that describe their functionality, interfaces, performance and limitations. Why should software be any different? Yet, much of the software in existence has not carried through providing this useful artefact of the rest of the engineering world.

Some consider software as a quite different aspect of creative development and so it has, for many, become a black art for devotees of a specific programming language to get to understand the hieroglyphics that they use. Yet software is being used in a wider range of products, some of which are becoming even more mission, security or safety critical, and sometimes, all three aspects simultaneously.

There have been some attempts at Component Oriented Development[3] with artefacts like .NET, and CORBA. Huge system modelling tools, built mainly for the software industry, have grown up that will churn out code from the model, all without the real feel of whether the model was correct or whether the translation of that model to code was correct. In such circumstances it becomes very difficult to be certain about any assessment of the final products fitness for purpose and absence of hidden faults.

In this paper we will take a look at what is required for Component Oriented Development that can be proven to be fully trustworthy to perform as its data-sheet implies.

1 What is a component?[4]

Across all engineering disciplines, the author considers that the following features should be common to all components. In fact Components

- have a unique reference identifier
- have Surfaces by which other components are interfaced.
- have been specified for operation within given environmental constraints
- have data-sheets that describe all functionality, features, performance and limitations of guaranteed performance.
- can be used and re-used many times over.
- can be inspected, tested and certified individually without impact on other components.
- conform to standards relevant to its functionality and performance.
- can, once certified, be used without being re-certified for every new situation, provided the new situation does not exceed the expectations of its published data-sheet.

However, using a certified component will not imply that the whole system is certified just by using it. To certify a whole system, the whole system needs to be constructed from known certified components throughout, have an audit trail that has logged all component certification, and itself be tested against its own statement of requirements.

For software components, there is a need for a development environment that allows the easy inspection and testing for individual components, preferably without having to write special test stubs to implement the testing. Where test stubs have to be created to perform

the test these should be logged with the component for subsequent confirmation testing and should receive as much attention to their correctness as the component itself.

2 Component Specification

Specifications of components grow out of the specification of the system to which they will ultimately belong. Such specification will mention aspects like the operational environment, lifetime expectations, MTBF (Mean Time Between Failures), Maximum and Minimum expectations of operation, Nominal Operating Regions and perhaps some notes on intended methods of use. Specifications, whether for the entire system or just a single component, should always adhere to the precept that they are Clear, Complete, Concise, Coherent, Correct and Confirm-able. Any lesser adherence to the principal 6 Cs[5] of specification will detract from the ability to fully assess the quality and robustness of the eventual product.

3 Component Management

Having created a component, all the artefacts, such as designs, data-sheets, inspection, test reports and other ancillary information relevant to the components use (like application notes) should be stored in a secure archive for which there is strong version control and strict change management procedures in place. This ensures the longevity of information about the component and its inspection and testing.

Regular auditing of the archive ensures that versioning and change management processes are being carried out properly and that the security of the information remains unsullied. The version control and change management becomes a very important aspect to development processes where the expected outcome of a development is a safe, secure, mission critical product.

4 Component Inspection, Testing and Certification

The Requirements of High Integrity Systems, especially in the Safety Critical[1] world, are:-

Arg1 - the system has been specified to be safe - for a given set of Safety Criteria, in the stated operational environment

Arg2 - the resulting system design satisfies the agreed specification

Arg3 - the implementation satisfies the system design

In examination in accordance with these three arguments, those who inspect the component (and system) will need to see robust evidence that the material presented is valid. Such demonstration is given by provision of:-

Direct evidence - which provides actual measures of the attribute of the product (i.e. any artefact that represents the system), and is the most direct and tangible way of showing that a particular assurance objective has been achieved.

Backing evidence –which relates to the quality of the process by which those measures of the product attributes were obtained, and provides information about the quality of the direct evidence, particularly the amount of confidence that can be placed in it.

The references to inspection and testing, above, have specific connotations in the light of components. For the mechanical world, there will be certificates on the material being used to assure that it is of the appropriate quality for the intended purpose. Physical viewing of the component to confirm its identity as the right component for the task, and measurements of

the final component to ensure that it conforms to its design data (as in the case of the nut, checking all the components dimensions to ensure a match to the drawings). There may even be a destructive stress test conducted on a small sample of the component to ensure the design criteria has been met.

For software, whilst we will still need an inspection and testing method to ensure that the design criteria is met, the methods are slightly different. Below, we will cover the three aspects of inspection and testing, namely the Fagan Style Inspection, Functional Testing and Limits Testing.

4.1 Inspection of Software

The author recommends the Fagan Style Inspection[2] as the best technique to perform a rigorously intense examination of the software itself. Getting to the point of inspecting a component for certification will have already initiated a series of inspections and reviews to ensure that the specifications on which the specification of this component relies are sound in principle and capable of compliance. Aspects that need to be observed during this inspection are:-

- Each component shall have a full statement of specification in which the functionality, performance characteristics, methods and limitations of the software component are fully described (references to specific clauses in standards or other document relied upon for the component are permitted but have to be made available to the inspection team).
- Any components on which this component relies already has certification in place as evidenced by the availability of that components certificate of conformity.
- All logical pathways through the code are checked individually to ensure that there are ways in which all pathways can be executed, and that the logic used is sound. Preferences are for simple decision structures or non decisions at all.
- The logical pathways in the code implement precisely the logic demand by its specification. Disparities should be recorded in the inspection and test report and regarded as a failure.
- The component has exactly one entry and one exit point.
- The Cyclomatic complexity is as low as is reasonably practical to the intended task described in the specification.

As you will detect, a lot of reliance is placed on having the specification and code closely allied during the inspection process. Fagan Inspections are, essentially, a style of static analysis but conducted with close attention to the details of implemented intent.

4.2 Functional Testing[6]

Functional testing, for certification, has to operate the component in its normal mode function but ensure that all logical pathways are fully exercised. The requirement is 100% logical pathway coverage. Running a functional test with a copy of the source code to hand and a marker to indicate when the pathway is taken and under what conditions. The function performed should precisely match the description in the component specification. Any deviation from the functional specification is seen as a failure of the functional test and should be noted in the test report.

4.3 Limits Testing[6]

Much of software may operate without encroaching any limitations whatsoever.

However theoretical the limitless possibilities might be, all implementations of a software component will exhibit limits with respect to the cell-width of the machine on which it will operate. So long as such limitations are understood by the user of the component there is usually no real concern.

However, some software components implementing specific algorithms, will exhibit a limitation of their accuracy or performance outside certain bounds. Hence, the specification should make it clear where such limits theoretically lie in order that testing against such limits can be undertaken to ensure the component continues remains to remain stable despite exceeding such limits (ie: takes the appropriate actions when limits are exceeded). An example of such a limitation is the divide by zero error in routines that use division. For such errors, an appropriate means of managing the error needs to be put in place and tested to ensure that in all cases where the limitation is achieved, the proper course of action is always taken.

Implementing such testing often requires quite wild imaginations to accomplish but the intention is to actively try and destroy the software component, much like you would destroy the test sample of a mechanical component.

5 Summary

This paper has been but a brief run-through of the Component Oriented approach to software development. We have briefly mentioned the need for all components to have a data-sheet in which its functionality, interfaces, performance and limitations are fully described. Additionally, we have covered a brief overview of the necessary inspection and testing regimes by which component certification can be accomplished. Treating the development of software components similarly to the development of any hardware component, with a specification, inspection, and testing regime that is fully explorative of the component properties, will improve overall quality of the delivered system. Finally, that attention to detail is beneficial to the outcome and re-usability of the components developed by this means.

That the above implies an increase in documentation should not be seen as any reason to reject such an approach, as this increase in documentation is substantiated by the ease with which certification of components can be achieved.

This usually leads to an eventual saving of development costs for those developing the higher integrity systems which will ensure our continued safety and security.

6 References

[1] **Functional Safety by Design – Magic or Logic?** Derek Fowler; Safety-critical Systems Symposium, Bristol UK, February 2015.

[2] **A History of Software Inspections** Michael Fagan, sd&m Conference 2001, Software Pioneers Eds.: M. Broy, E. Denert, Springer 2002 <http://www.mfagan.com/pdfs/software_pioneers.pdf>

[3] **Component Software** <http://www.webopedia.com/TERM/C/component_software.html>

[4] **Component** <https://en.wikipedia.org/wiki/Component>

[5] **High Integrity Systems CODE** Paul E. Bennett IEng MIET, EuroForth 2014.

[6] **Software Testing – Goals, Principles, and Limitations** S.M.K Quadro & Sheik Umar Farooq International Journal of Computer Applications <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.206.4616>

The author is willing to advise on and oversee any software component oriented development process leading towards full system certification against any standard. He is also a member of the IET, Safety Critical Systems Club and has been using Forth, in a component oriented manner, for high integrity systems since the mid 1980's.

Paul E. Bennett IEng MIET <Paul_E.Bennett@topmail.co.uk>

Tel: +447822639972

Minimal Forth

Peter Knaggs Paul E. Bennett

September 27, 2015

Now that the Forth 2012 document has been published, it is time to review the direction on the standards effort. Both the '94 standard and the '12 document are directed at the professional Forth programmer. Providing a set of expectations that allows both the programmer and the program to be portable between different standard systems.

We argue that it is time to consider the niche market where Forth is generally used, that is embedded control systems. Such systems are often small will little memory and no or limited user Input/Output. The Core word set includes many words that are not relevant to such systems (around 100 words). Such embedded control systems require a small number of words.

However, even a minimal Forth system would tend to provide a full implementation of both the Core and Core-Ext word sets.

The current standard has separated into several sections within the same document, some of the words that one would expect in a minimal system are spread across several different word-sets. To this end, we would like to start a discussion over what would be the minimal word-set we would expect of any Forth. The aims of this discussion is two-fold:

First, from a professional standpoint for those developing critical controls, having a small system footprint that can be fully verified and validated is beneficial. The workload in knowing you have a good basis from which to springboard the application is reasonably simple as it will not take a long time to perform certification confirmation efforts. You can only certify from a know surface (surfaces are the interface between lower level and upper level words).

Second, from an educational standpoint, a Forth with 400+ words is rather daunting for a those wanting to learn a new language. Even knowing that they do not have to do so all at once still leaves many confused. Therefore, a less cluttered dictionary at the start will help the beginners learning process. This will be especially the case for someone who has never programmed before. Table 1 is a comparison of items to lean for different languages. It shows Forth as being well out it front with around 100 more items to lean than other languages, and that assumes the student only looks at the core word set.

Both of these aims could be met with a minimal Forth word set provided there is a reasonable number of useful words to aid in application creation and debugging. The only question then becomes what words and how many?

Frank Sergeant proposed a Forth with just 3 words (`XC@`, `XC!`, and `XCALL`). However, that is only useful in umbilical development and probably only by those who are already well experienced with creating such systems. Table 1 shows us that keeping a minimal word set to around 70–80 words would be acceptable from an educational standpoint. Staying away from desk-top aspects, and just sticking to controllers, it is expected around 50 would be reasonable for use on smaller controllers and provide a capable enough basis to grow control applications. At this end of consideration we are dealing with controllers the likes of the MSP430 and low end ARM systems used for embedded control applications and robotics. Such systems do not tend to have disk or graphics screens but will usually have a low-tech

Langauge	Words	keywords	operators	functions
Forth '79	129			
Forth '83	131			
Forth '12	182	133 (core)	49 (core-ext)	450 (overall)
Minimal Forth	69			
C	96	32	39	25
Java	85	50	39	—
C#	116	77	39	—
Ada	85	73	12	—

Table 1: Number of "words" in different languages

terminal communications capability and enough resource to run a minimal Forth system.

Proposal

We propose the standard be reduced to a basic description of the abstract machine that represents the language, complete with a limited minimal word set. The composition of this word set is a topic of debate, and appendix A gives a list of 69 words that we propose as a starting point for that debate.

Such a language description would allow people to grasp the language without being daunted by the size of the language. It would also allow for a possible formal description of the language, leading to certified compilers, and thus confidence in the program code.

In the description of the abstract machine it may be useful to allow for a number of more advanced topics. However, it would not be necessary to include words to access these functions in the minimal word set. Such topics could include:

- exception handling
- interpretation
- Multitasking/threading

The words necessary to access these features can be given in a number of supplements to the base description. These supplements may extend the abstract machine, or simply provide additional word sets.

A number of proposed supplements include, but are by no means limited to, the following:

- IEEE Floating Point
- String
- Double numbers
- Local variables
- Heap / User memory
- File access
- Networking / Sockets
- Internationalisation
- Multitasking (cooperate)
- Multitasking (pre-emptive)
- Security (cryptography)
- Exceptions

A Proposed minimal word set

1 Memory Access

6.1.0010	!	store	6.1.0850	C!	c-store
6.1.0150	,	comma	6.1.0860	C,	c-comma
6.1.0650	@	fetch	6.1.0870	C@	c-fetch
6.1.0705	ALIGN			CALIGN	c-allign
6.1.0706	ALIGNED			CALIGNED	c-alligned
6.1.0880	CELL+	cell-plus	6.1.0897	CHAR+	char-plus
6.1.0890	CELLS		6.1.0898	CHARS	chars

2 Arithmetic

6.1.0120	+	plus	6.1.0160	-	minus
6.1.0090	*	star	6.1.0230	/	slash
6.1.0320	2*	two-star	6.1.0330	2/	two-slash
6.1.0110	*/MOD	star-slash-mod	6.1.1890	MOD	

3 Logic

6.1.0270	0=	zero-equals	6.1.0530	=	equals
6.1.0480	<	less-than	6.1.0540	>	greater-than
6.1.0720	AND		6.1.1980	OR	
6.1.1720	INVERT		6.1.2490	XOR	x-or
6.2.2298	TRUE		6.2.1485	FALSE	
6.1.1805	LSHIFT	l-shift	6.1.2162	RSHIFT	r-shift

4 Stack

6.1.1290	DUP	dupe	6.1.1260	DROP	
6.1.2260	SWAP		6.1.1990	OVER	
6.1.0580	>R	to-r	6.1.2060	R>	r-from
6.1.2070	R@	r-fetch	6.1.2160	ROT	rote

5 Flow Control

6.1.1700	IF		6.1.1310	ELSE	
6.1.2270	THEN		6.1.0760	BEGIN	
6.1.2430	WHILE		6.2.0700	AGAIN	
6.1.2140	REPEAT		6.1.2390	UNTIL	
6.1.1240	DO		6.1.1800	LOOP	
6.1.1680	I		6.1.1730	J	
6.1.0070	'	tick	6.1.1370	EXECUTE	

6 Definitions

6.1.0450	:	colon	6.1.0460	;	semicolon
6.1.0950	CONSTANT		6.1.2410	VARIABLE	
6.1.1000	CREATE		6.1.1250	DOES>	does

7 Device

6.1.1750	KEY		10.6.1.1755	KEY?	key-question
6.1.1320	EMIT		6.1.0990	CR	c-r

8 Tools

6.1.0080	(	paren	6.2.2535	\	backslash
15.6.1.0220	.S	dot-s			

HIDECS
High Integrity Distributed Embedded Control Systems

Forth in Education – A Report

By Paul E. Bennett IEng MIET

Peak 2015 – IET Time of Your Light

- IET TV were present at the event and the three most involved in the event organisation (Stephen Powley, Juergen Pintaske and myself) were interviewed. Forth got prominent mentions during the interviews.
- The head of the IET Education group found our intention to expand out on the activity of interest and a meeting is being held this month between the IET and the above three persons to establish further promotion prospects and support.

The last year

- Signed up as a STEM Ambassador
- Production of the design of a small board for easy assembly by young people. This was based around the MSP430 & Vfx-Forth-Lite.
- Attending the Peak2015 Scout and Guide Camp for the IET “Time of Your Light” activity.
- Preparation for a follow-on meeting with the IET Education Team to explain further development proposals.
- Beginning of the development of a simpler means of teaching Forth to young people (even without the aid of a computer).
- Considering potentially work with the BBC-Micro:Bit.

Teaching Forth to the Young

- It does not need a computer to explain the principles. Just a few simple props.
- The young people can be in primary education but will need, at least, to read with cognition (so mostly 7+ or some very advanced 5+).
- Role play is seen as key to embedding the principles.
- Fun projects are needed to cultivate enthusiasm.
- We need to rebuild the maker generation.

STEM Ambassadors

- Approved to work directly with young and vulnerable people (Police checks are performed on the applicant).
- Provide career and opportunity advice to young people
- Engender Enthusiasm for learning STEM subjects through relating real-world experiences of Engineers, Technicians, Scientists and Mathematicians.
- Voluntary Time and Effort contribution needed but some expenses may get paid (travel etc if claimed)

The BBC Micro:Bit

- Languages the consortium considered were:-
  - Logo, Scratch & Python

Sadly Forth was not amongst those considered but that is our poor marketing as a community.

Micro:Bit method of programming is a long chain of events (see next slide)

The Micro:Bit distribution is currently delayed due to technical issues.

Peak 2015 – IET Time of Your Light

- It was the Year of Time so the activity had to do something with time in a visibly demonstrable way.
- We aimed to get more Forth based systems out in the wild
- A Kit of parts, donated by approximately 26 organisations found us with plenty of components and PCB's for the activity.
- 457 Scouts and Guides built their own micro-controller board complete with a battery supply that they could take home with them (Juergen calls this a microbox).
- The board was programmed with Vfx-Forth Lite and is accessible by a simple terminal programme (most would probably use a USB to serial converter cable like the FTDI one).

BBC Micro:Bit

- Programming Micro:Bit is via a web-based application that runs a simulator of the board.
- When the programmer is ready to commit, the program is uploaded (via a Micro-soft server) for it to be compiled (hence all programmers need a log-in)
- The uploaded programme, once compiled is provided back to the programmer via a flash utility to send the code to the Micro:Bit.

It could probably do with taking a Forth on-board to enable a more direct and simpler means of programming.

Conclusion

- Getting Forth in the minds of the younger generation will take some time and effort
- Becoming a STEM Ambassador and forming closer associations with schools and colleges will help get the message across here.
- It needs exciting project suggestions to raise enthusiasm of the young.

Recognizers
Why and How

M. Anton Ertl
TU Wien

How to deal with literals

Recognizers	Parsing Words
`123`	`s" abc"`
`$ff`	`H# ff`
`'a'`	`[char] a`
`1.2e3`	

- No way to define new recognizers
- No good way to define parsing words
  non-default non-immediate compilation semantics
  `State`-smartness and the like
  Not just an implementation problem
- ⇒ user-defined recognizers

Ideal

- Recognized literal acts like a normal word
- `: 123 123 ;`
- Interpret
  Compile
  `Postpone?` `]] a 123 b [[` vs. `]] a [[ 123 ]] literal b [[`
  `'?`
  `find`
  `find-name name>string ?`

How to specify and implement recognizers

- Specify interpret, compile, and postpone actions
  Advantage: Optimization possible
  Disadvantage: Bugs can hide, especially for postpone
- Specify parse-time, run-time, and data-shifting actions
  interpret: parse-time run-time
  compile: parse-time shift]] run-time [[
  postpone: parse-time shift]] shift]] run-time [[[[
- Define a temporary word
  Advantages: Allows ticking etc.
  Conceptual simplicity
  Disadvantage: Optimization?

Temporary words

- Separate dictionary pointer (like ELF section)
- Should be inlined if compiled. *But how?*
- Becomes permanent if `postpone`d or ticked
- Other permanent uses need explicit permanence
- Recognized string as name?
  Decompiler
  `name>string`

Coding example

```
: usingle ( c-addr u -- f )
  0. 2over >number 0= if
    drop 2drop 2drop false exit then
  drop drop rot rot ['] constant execute-parsing
  true ;
```

Inline when compiled

- Require using an intelligent `compile,`
  Quite elegant
  But `set-opt` is unlikely to be standardized
- Or specify parse-time and compile-time action
  For compilation, perform these actions
  In other cases, build the word

Performance with many recognizers

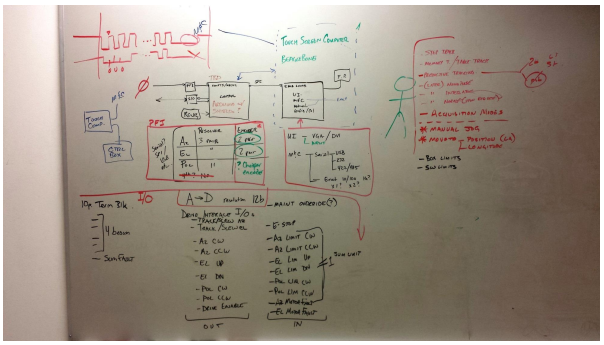
- Linear search through recognizer stack?
- Or fast pre-selection
- Pre-selection may accept invalid strings
  but must not reject valid strings
- prefix pre-selectors ⇒ trie
- regexp pre-selectors ⇒ NFA/DFA

Conclusion

- User-defined words are great!
  Let´s also allow user-defined recognizers
- New implementation approach:
  Define temporary words
  How to inline?
- Pre-Selectors for performance

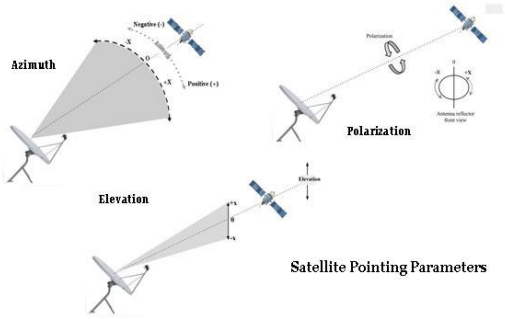
Radeus 8200 Series Antenna Controller

January 2015



Earth Station 11m Antenna

Antenna Axes



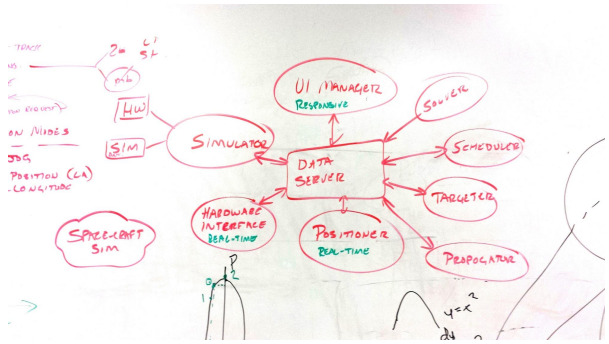
Ancient History

- Vertex (now General Dynamics) Model 7200
- Developed in the early 1990s
- Simple keypad/display UI

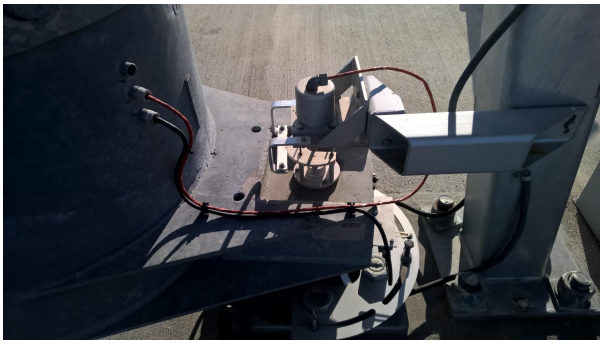
Inputs

- Azimuth Position Transducer
- Elevation Position Transducer
  - Resolver (0.01˚), Optical encoder (0.001˚)
- Polzarization Transducer
  - Resolver or optical (0.1˚)
- Limit Switches (Both ends of each axis)
- Beacon receiver (power level in dBm)

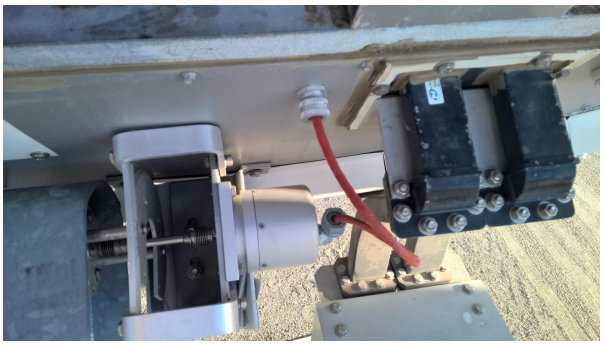
January 2015



Azimuth Position Transducer



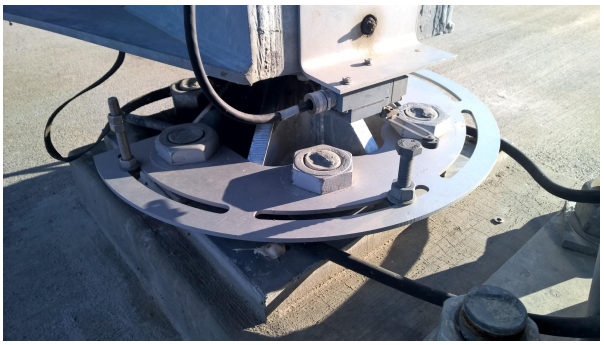
Elevation Position Transducer



Three Computers

- Transducer Interface (Encoders/Resolvers)
  - Cortex-M4, SwiftX-ARM
- Controller (positioner, fault detection)
  - Cortex-M4, SwiftX-ARM
- UI (touch-panel PC)
  - 10.2" Panel PC, Windows 7 Embedded,
  - SwiftForth, SWOOP

Azimuth Limit Switch



User Interface

Outputs

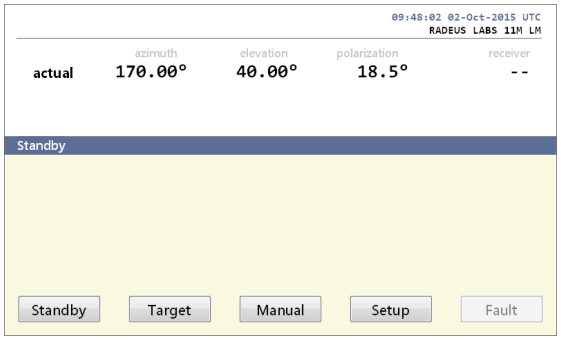
- Azimuth Motor Controls
  - On/off, Speed (slew/track), Direction (CCW/CW)
- Elevation Motor Controls
  - On/off, Speed (slew/track), Direction (Down/Up)
- Polarizatrion Motor Controls
  - On/off, Direction (CCW/CW)
- Alarm (actually "no alarm")

Touch-Panel PC

Motor Controllers

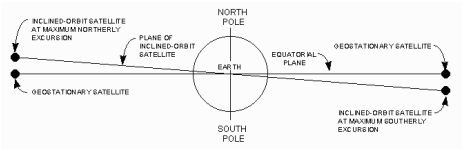
Rear Panel (Inside Chassis)

Standby



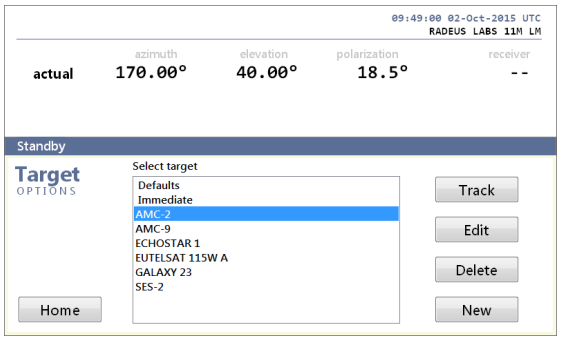
FORTH, Inc.

Inclined Orbit



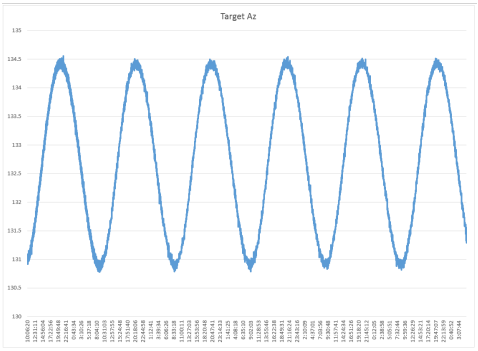
FORTH, Inc.

Select Target



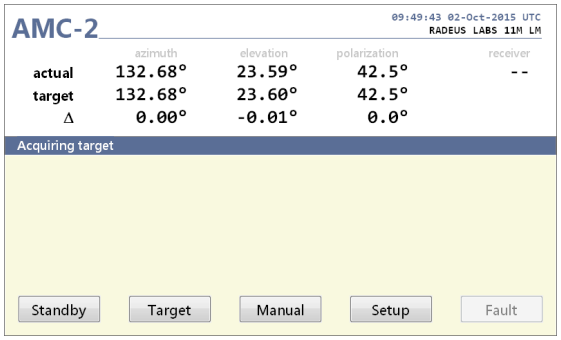
FORTH, Inc.

Log Data



FORTH, Inc.

Acquire Target



FORTH, Inc.

Test Site

FORTH, Inc.