32nd EuroForth Conference

September 9-11, 2016

Hotel Mein Inselglück
Insel Reichenau
Germany

Preface

EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 32nd EuroForth finds us on Reichenau Island on Lake Constance for the first time. The two previous EuroForths were held in Palma de Mallorca, Spain (2014), and in Bath, England (2015). 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: 21 submissions, 14 accepts, 67% 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/ef16/papers/`).

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

Anton Ertl

Program committee

Sergey N. Baranov, SPIIRAS, Russia
M. Anton Ertl, TU Wien (chair)
David Gregg, Trinity College Dublin
Ulrich Hoffmann, FH Wedel University of Applied Sciences
Jaanus Pöial, Estonian Information Technology College, Tallinn
Bradford Rodriguez, T-Recursive Technology
Bill Stoddart
Reuben Thomas

Contents

A synchronous FORTH framework for hard real-time control

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

June 2016

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

Abstract

Forth control programs are typically written in an event triggered style: events that take place in the environment interrupt the main control program. The interrupt handler either handles the event completely on its own (if that's simple enough or timing requires it) or it triggers a task from an underlying multitasking system to take care of the event (in a non timing critical way). Most Forth multitasking systems are cooperative thus offering high reliability and predictable timing behavior. The framework described here uses a synchronous approach to meet hard real-time requirements. The approach borrows from different sources, most notably from synchronous hardware design, where signals are updated at a fixed cycle rate, and program logic is implemented via finite state machines. Despite the fact that applications built with this framework follow hard real time constraints they may still retain interactivity through a FORTH interpreter. This is accomplished by means of an optional high level threaded code interpreter which can be executed in a step-wise way and will only progress as fast as necessary to still be within the real-time boundaries. The only requirement for this framework is a single free-running counter/timer with a known clock period. All other functionality is expressed in standard Forth and is thus portable to different standard systems.

1 Introduction

The outline of this paper is as follows: we first briefly review synchronous digital logic and finite state machines. Through this review we identify the essential concepts that we wish to abstract for our software framework. We then consider related work, most notably time triggered architectures for embedded systems and finite state machines in FORTH. Our synchronous FORTH framework for hard real-time control is then presented in a top-down fashion beginning with a conceptual overview and leading to implementation details. We go on to explain the general requirements for the implementation of this framework on a FORTH system and describe our specific implementation on a Texas Instruments Tiva-C development board using Mecrisp Forth. We present test measurements that we obtained on the Tiva-C board. Finally we discuss the potential advantages and limitations of our framework and suggest possible applications.

2 Synchronous digital logic

2.1 Background

The essential characteristic of synchronous digital logic is a clock to which all signal transitions are synchronized. By contrast, asynchronous signals update in their own time (fig. 1). Most commonly, signal transitions are synchronized to the rising edge of the clock although in dual data rate (DDR) interfaces signal transitions occur on both clock edges.

Synchronous digital logic is implemented in hardware using flip-flops, referred to from a logic design perspective as registers. A typical D-type flip-flop will have an input port, a clock port, an output port and a reset port (fig. 2).

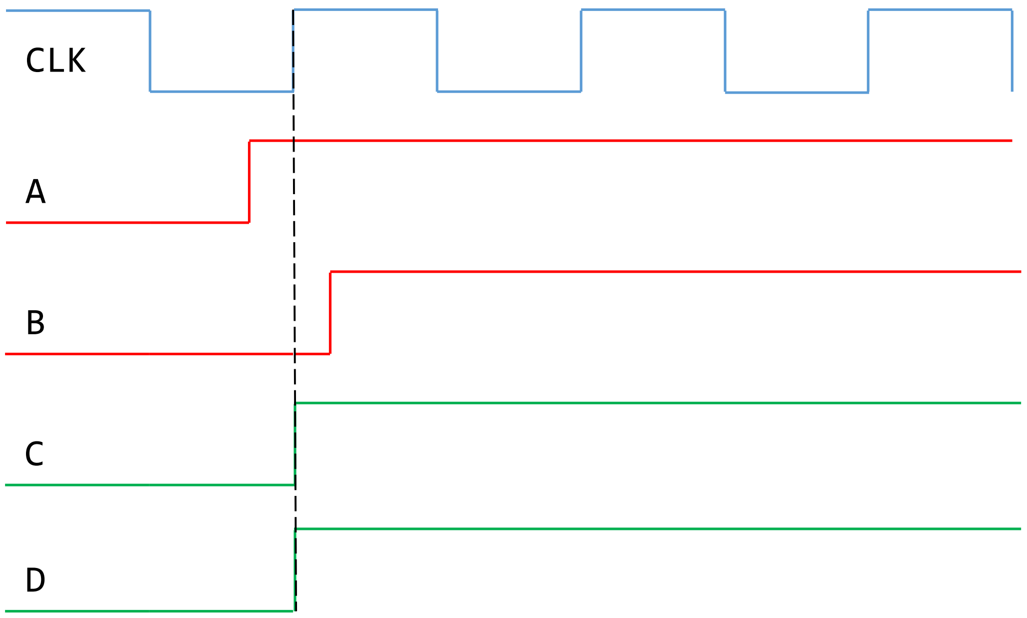


Figure 1: An example of asynchronous and synchronous signals. A and B update in their own time without reference to CLK. C and D update synchronously with each other and with the rising edge of CLK.

Whilst all signal transitions are conceptually synchronous to the rising edge of a clock, in reality certain timing constraints must be met if the physical devices are to operate correctly (fig. 3). Firstly the input signal must become stable some minimum time before the rising edge transition of the clock. This is the set-up time constraint. The input signal must also be held stable for some minimum time after the clock transition - the hold time constraint. Lastly, the output signal will not transition until some time after the clock transition. This is the clock output time constraint.

2.2 Multiple clock domains

A complex digital logic design is likely to have more than one clock domain (fig. 4), often because different peripheral interfaces must be clocked at different rates. A single design may also have clocks that run at the same frequency but at a fixed phase offset, for example to register data arriving from external peripherals with a phase delay.

2.3 Essential concepts for a software framework

The relative benefits of synchronous and asynchronous circuits continues to be debated, but in the present era the most common central processing units (CPU's) and peripheral integrated circuits used in real-time control applications are based on synchronous logic. A noteworthy exception is the asynchronous GreenArrays G144 Forth processor [1].

We do not attempt to reevaluate the merits of the synchronous and asynchronous approaches in this paper but summarize some simple notes as follows.

Metastability is a breakdown of the digital logic abstraction that allows signals (which are actually potential differences with respect to electronic ground) to be considered as exclusively 'high' or 'low'. The synchronous design approach deals with issues such as race-conditions and signal metastability by means of objective timing requirements that are validated during the place and route stage of circuit implementation. These timing constraints also impose a limitation on the overall circuit size. Asynchronous circuits may be arbitrarily large provided appropriate mechanisms are in place for completion detection, but logic still needs to be synchronized at the circuit borders in any application with deterministic timing requirements.

Digital logic circuits are expressed in hardware design languages (the most common being VHDL and Verilog) that synthesis tools translate into the physical layout of an integrated circuit. The framework that we present in this

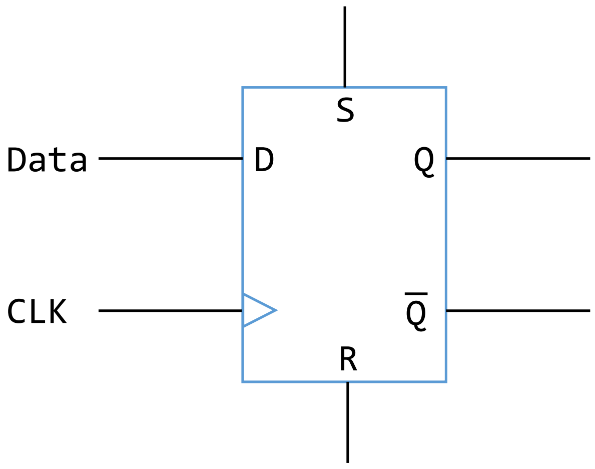


Figure 2: A schematic of a D-type flip-flop. The signal at input D is registered on the rising edge of CLK. The registered signal subsequently appears on outputs Q and inverted Q and hold steady until the next rising edge of CLK. Set (S) and reset (R) inputs are available to drive Q high and low respectively irrespective of D.

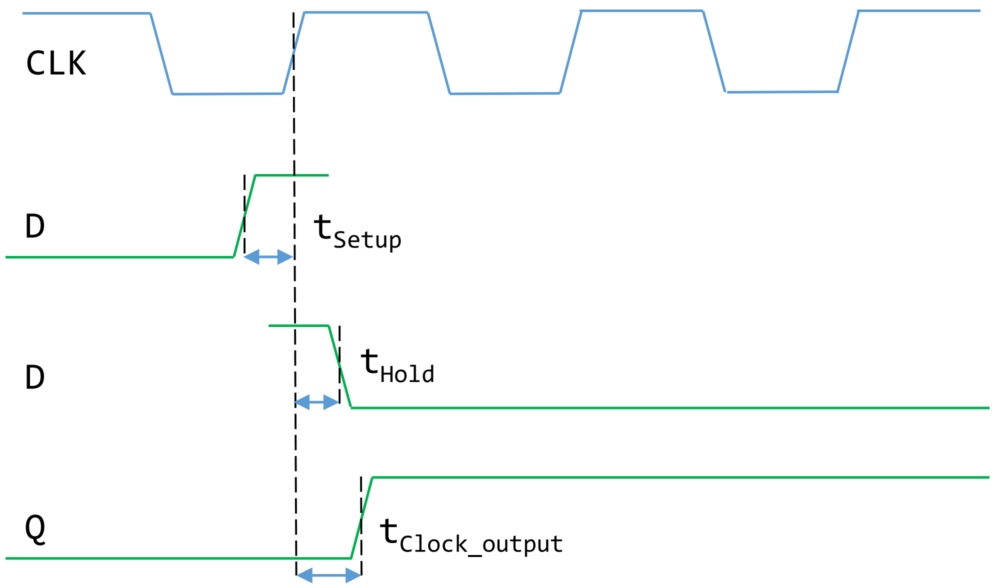


Figure 3: Timing characteristics of a D-type flip-flop. The data signal must be stable t_{Setup} before the leading edge of the clock. The data signal must remain stable t_{Hold} after the leading edge of the clock. The output signal is updates $t_{\text{Clock\_Output}}$ after the leading edge of the clock.

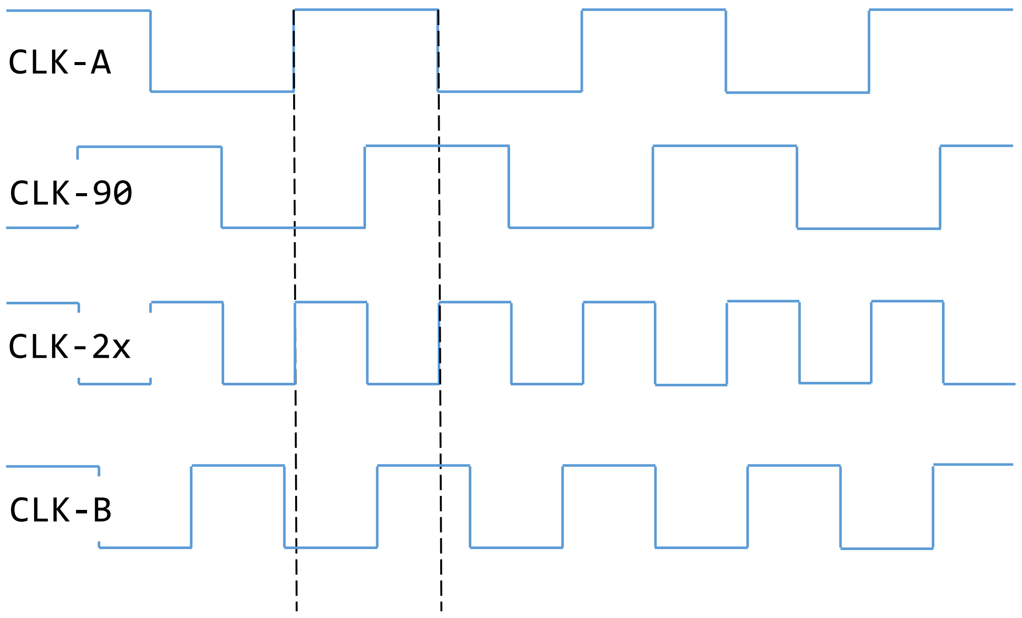


Figure 4: An illustration of multiple clock domains. Clock-A is the reference clock. Clock-90 has the same period as Clock-A but is retarded 90 degrees in phase. Clock-2x has twice the period of Clock-A but is in phase. Clock-B has a period somewhere between Clock-A and Clock-2x and does not have a fixed phase relationship with either

paper is a software approach that provides similar functionality to a hardware design language such as VHDL: that is the ability to read inputs, write outputs and update internal signals synchronously with one or more free-running clocks.

In addition to synchronous signal update and multiple clock domains, our framework requires a suitable model for the computation. The natural choice is the finite state machine (FSM), arguably the most common computation device in digital logic design and a familiar construct in software applications [2].

3 Finite state machines

Finite state machines need little or no introduction in this paper. Proponents of the FSM approach argue that the FSM model provides a systematic approach for designing computations that lead to optimal or near-optimal implementations [2]. Essentially a finite state machine is a device that must always in one of a finite number of predefined states. Transitions between states occur according to the rules of a state transition diagram. The next state is always a function of the current state and of the FSM inputs (which may include internal registers such as counters). In the most simple finite state machines outputs are a function of only the current state (Moore Machines). Alternatively outputs may be a function of the current state and of the inputs (Mealy Machines). Finally, in recursive finite state machine designs outputs may also be a function of state transition and output history since reset [2].

4 Review of related work

Many embedded systems are developed in an *event triggered architecture* style: whenever an external asynchronous event occurs the embedded system is interrupted and a handler is invoked to take care of the event. Once the handling is completed the embedded system continues its previous work. Depending on the number of different external events and their timing properties it might be quite difficult to build reliable real time systems this way, especially when events can occur simultaneously or while the handling of other events is underway.

An alternative approach for real time system design is *time triggered architecture*: handling of external events takes place at regular intervals. Michael Pont [4] developed a time triggered architecture and implemented it for LPC-1769 arm processors based on a single timer interrupt. In [13] Kopetz and Bauer give a comprehensive summary of their research on time triggered architectures with an exhaustive bibliography on the topic. Event triggered and time triggered architectures are also described in Peter Hintenaus book on embedded system engineering [5]. He also discusses implementing real time systems with multiple clock domains and finite state machine implementations in soft- and hardware.

Our approach is also a time triggered architecture, but we avoid interrupts completely and synchronize multiple clock domains by means of a free running counter. (If one is not directly provided in hardware, a timer interrupt can be used for a straightforward implementation.)

Finite state machines are a contemporary way to model system behavior that is especially popular in hardware design as finite state machines are easy to define and fit well to synchronous system architectures [2]. In their book "Structure and Interpretation of Signals and System" [6]Lee and Varaiya describe the use of finite state machines for system design and implementation from a computational point of view. They discuss how to combine state machines in order to model complex systems and define so called linear time-invariant systems as special state machines with beneficial signal processing properties.

There have been numerous Forth implementations of finite state machines, only few of them were published: Basile [7] gives a short implementation in Forth-79, Rawson[8]- in polyForth. Nijhof [12] calls his implementation "Goto in Forth". Starling [11] discusses a state machine hard/software co-design and the processing of external events with Forth. Carter [10] describes Forth implemented FSMs for robotics. The most elaborated discussion of state machines in Forth has been done by Noble [9]. These approaches focus on Forth as sequential language and make use of its extensibility to add new state machine defining structures that allow for easy definition. However, real-time considerations are not either not taken into account or else they are not well-documented.

Embedding a slow pace Forth inner interpreter into real-time applications has been best practice for many years with Microchip field engineers.

Our work reuses elements, including those cited above, that have been commonplace in embedded programming for many years. What we present in this paper is a different combination in a novel framework. We avoid interrupts, we implement finite state machines and multiple clock domains in software, we retain the interactivity of FORTH, and bring everything together in a systematic framework with an elegant syntax borrowed from digital logic design.

5 FORTH framework

5.1 Overview

The Forth framework is illustrated in figure 5. The framework comprises five entity types: CLOCKs (or clock domains), SIGNALs, INs, and OUTs and FSMs (finite state machines) together with an operational loop.

The first step in establishing an application is to define one or more clock domains. If there is only a single clock domain then the only parameter that needs to be specified is the clock period. If more than one clock domain is in use then phase offset between each is also specified. Signals are added to each clock domain. A signal is effectively a value-type variable but with an important distinction. A signal can be updated at any time during a clock cycle, but it will not assume its new value until the following clock cycle. All signals within a clock domain are updated synchronously at each clock cycle. A signal may be updated directly by the application logic or, alternatively, they it may be connected to an IN port. An IN port specifies a memory address (which may be a memory-mapped register) that provides the value with which to update the signal at each clock cycle. A signal may also be connected to an OUT port. An OUT port provides a memory address to which the value of a signal is written at each clock cycle. Finally, a single finite state machine which reads and updates signals at each cycle in accordance with the application requirements is associated with each clock domain.

As a high-level illustration, the following Forth listing gives an example of the establishment of a clock domain and related entities. More complete explanations of the framework entities are given throughout the remainder of this section. The discussion of usage and applications continues at a higher level in the next section.

```
FCPU               \ FCPU is the CPU frequency
10000              \ desired clock domain frequency in Hz
/                  \ number of CPU cycles in a clock domain period
```

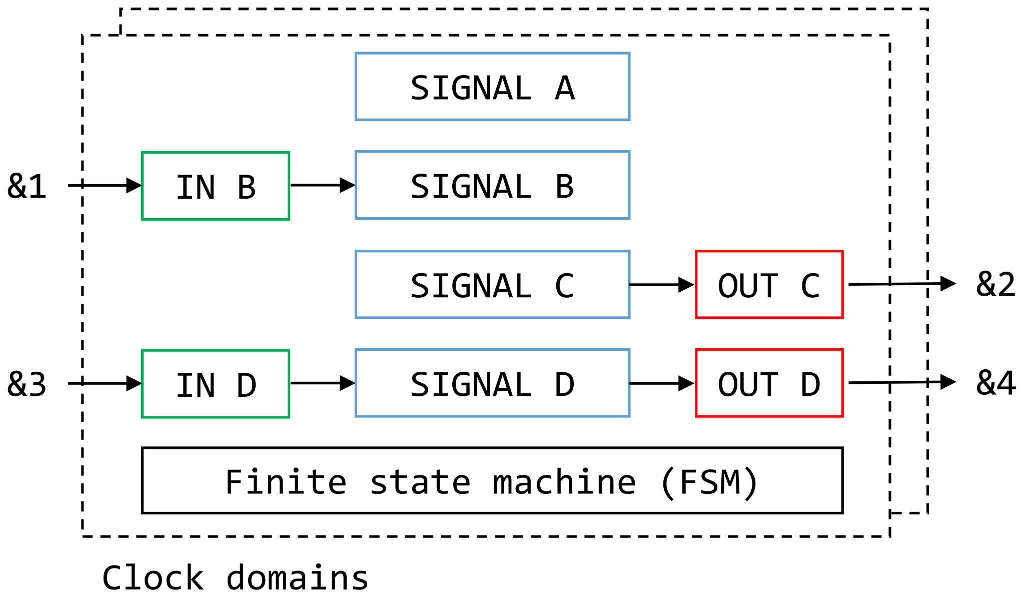


Figure 5: Schematic overview of the FORTH framework. Entities are organized within CLOCK domains that have a defined frequency and phase. Within each CLOCK domain the primary construct is a SIGNAL which is essentially a variable with clock-synchronous update. IN's link memory-mapped addresses to signals with a synchronous read relationship. OUT's link memory-mapped addresses to SIGNALs with a synchronous write relationship. There is a single finite state machine (FSM) within each clock domain that contains the FORTH code to inspect and update all SIGNALS each clock cycle.

```
0                    \ example phase offset
CLOCK 10kHz          \ define a new clock domain labeled '10kHz'

10kHz 0 SIGNAL s0    \ add a signal named 's0' with a reset value of 0 to the clock domain
10kHz 0 SIGNAL s1    \ add another signal 's1'

10kHz $1000 IN s0    \ tie the input of signal s0 to memory address $1000

10kHz $2000 OUT s1   \ tie the output of signal s1 to memory address $2000

: our-code           \ a simple (single-state) FSM
    s0 not => s1     \ invert s0 and update s1 with the result each clock cycle
;

' our-code 10kHz FSM     \ attach the FSM to this clock domain
```

5.2 Framework requirements

The framework is written entirely in ANSI Forth. The only requirement of the underlying hardware is a free-running timer counter of known clock period accessible through Forth. The framework expects the following two words to be available.

```
: CPU-time ( -- n) ;
        \ return the value of the free-running timer counter
        \ assumed to be unsigned and full-cell width (e.g. 32 bits in a 32-bit cell)

: init-CPU-time ( --) ;
        \ the framework calls this word (which may be empty) at initialization
```

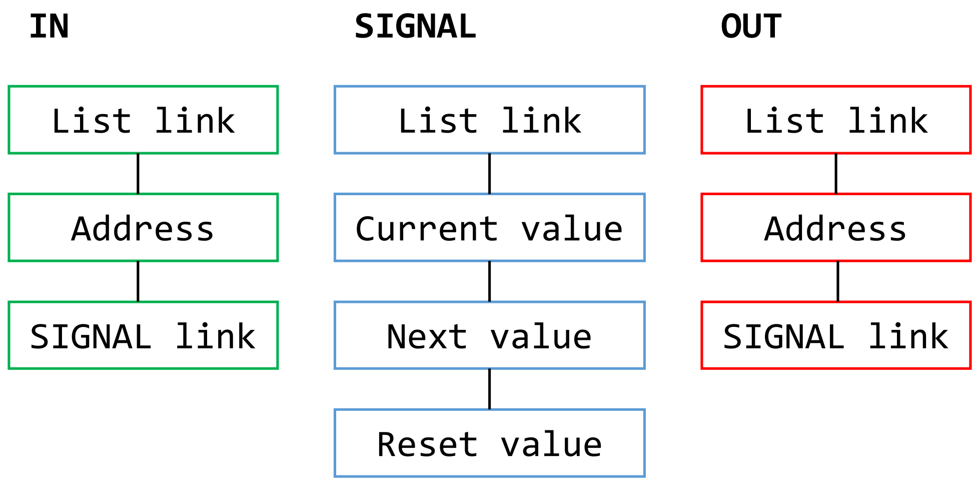


Figure 6: Structure of the SIGNAL, IN and OUT entities. Each entity type is organized within a linked list

5.3 SIGNAL, IN, and OUT entity structures

Figure 6 illustrates the structures of the signal, in and out entities. All of the entities are organized as linked lists with the anchor node in the clock data structure (see section 5.5). The SIGNAL entity reserves space for three values, each of cell size: the current value, the next value that will be assumed at the following clock cycle, and a reset value that was established when the signal was defined. The IN and OUT entities each hold a memory address and a link to the SIGNAL to which they are attached to.

5.4 Programmed and synchronous updates

Figure 7 illustrates the mechanisms by which a SIGNAL is used and updated. 'Programmed', refers to usage of the signal within the finite state machine. At any time when the signal is read the current value field is returned. If the signal is written to within the finite state machine, the next value field is updated. The framework itself synchronously updates all SIGNALS once each clock domain cycle. This synchronous update copies the next-value field to the current-value field. The application may also instruct a reset, in which case the reset-value field will be copied to the current-value field

Figure 8 illustrates the update relationship between SIGNALs, INs and OUTs. The framework follows the following sequence when a clock domain is triggered to perform a synchronous update. Firstly the INs are processed in turn. The memory address specified by each IN is read and the value is written to the next value field of its associated signal. Next the SIGNALs are processed in turn. As described above, the next value field of each signal is copied to its current value field. Finally the OUT's are processed. In each case the current value of the associated signal is written to the memory address.

5.5 CLOCK entity structure

Figure 9 illustrates the structure of the CLOCK domain entity. An application may define multiple clocks and they are organized in a linked list. For each clock its phase and period are specified. The period is the number of CPU clock cycles (as returned by the free-running counter timer `CPU-time`) between each round of synchronous updates of within the clock domain. The phase is also specified in the number of CPU clock cycles. If an application includes multiple clock domains then the phase parameters may be used to specify a phase relationship between the clock domains. The clock entity also anchors the linked list of SIGNALs, INs, and OUTs that have been defined for that clock domain. The execution token of the finite state machine associated with the clock domain is held in

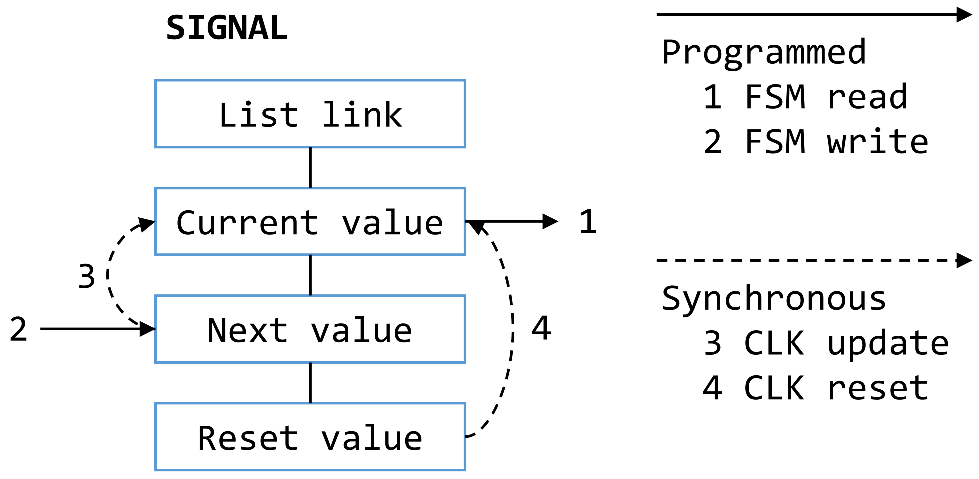


Figure 7: Schematic of the programmed and synchronous update of SIGNAL's

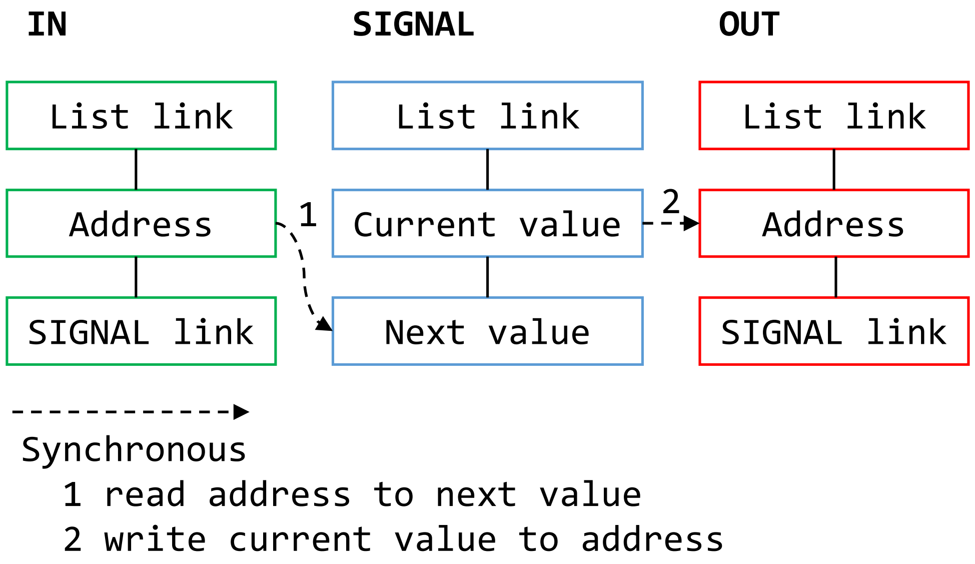


Figure 8: Schematic of the synchronous updates of IN's and OUT's

the XT field of the CLOCK domain entity. The due and flags fields are used by the framework during operation (see the next section).

5.6 Timing framework

Figure 10 presents a flowchart of the timing framework that acts to coordinate the clock domains and their respective elements. It comprises an outer loop (`super-loop`) and two routines (`check-clocks` and `run-FSMs`).

First we examine `check-clocks`. This routine proceeds once through the linked list of all clock domains. For each clock domain the number of CPU cycles until that clock domain is due to update synchronously is computed. This is done by reference to the due field and a call to `CPU-time`. If the CPU-cycles-until-due value is zero or negative then the clock domain proceeds to a synchronous update in the manner described above. All of the INs, SIGNALs and OUTs in that clock domain are updated in turn. The clock-domain's due-field is updated by an increment equal to the 'period' and a flag is set to indicate that the FSM of that clock domain is also now due for execution. The FSM is not executed at this point. Regardless of whether a clock domain proceeds through a synchronous update, the value of 'least-slack' is examined and potentially revised. For each clock domain, the 'slack' is the number of CPU cycles until that domain is due for a synchronous update. The 'least-slack' is the number of CPU cycles until the earliest of the clock domains is due to update. If the 'least-slack' is below the 'minimum-slack' threshold, then FSM processing is skipped in favor of synchronous update.

Returning to `super-loop`, after `check-clocks` has been run the final value of 'least-slack' is compared with the value of 'minimum-slack', which provides a threshold as described below. If the 'least-slack' exceeds the 'minimum-slack' then execution proceeds to `run-FSMs`. If not then `check-clocks` is run again until 'least-slack' exceeds the threshold. `run-FSMs` proceeds through each clock domain in turn. If the ready-to-execute flag has been set by `check-clocks` then the FSM is executed at this time by calling its XT and the ready-to-execute flag is cleared.

Figure 11 presents two examples of the operation of the framework with a single clock domain. In both cases the clock domain is due for synchronous updates at t_0 and t_1. Consider first case A. The period labeled CLK-A indicates the time during which the entities of this clock domain are being updated (this occurs within `check-clocks` when the due time is reached). During this period the updating of SIGNALS and OUTS leads to the update of OUT-A. After CLK-A has been completed the read-to-execute flag will have been set for this clock domain. When `run-FSMs` is called the FSM of this clock domain is executed. For illustration purposes we present a simple device in which the only action of FSM-A is to invert the SIGNAL driving OUT-A, so that OUT-A toggles between logical high and low levels and produces a square wave of twice the period of the clock domain. In case B there is a problem. The run time duration of the FSM-B is too long for the specified clock domain period and execution is not completed in time for the synchronous update expected at t_1. This constraint places a practical lower limit on the period that may be specified for a clock domain that depends on the underlying hardware and the host Forth platform.

5.7 Multiple clock domains

If an application defines only a single clock domain then the 'minimum-slack' threshold has no impact on the operation of the framework and the default value of zero applies. Where an application has more than one clock domain then the 'minimum-slack' threshold influences the sequence of operations, as illustrated in figure 12. In this example clock domain A is assumed to have twice the frequency of clock domain B and the two clock domains are (approximately) in phase. Here the 'minimum-slack' parameter was set to some non-zero value. After CLK-A has proceeded through a signal update at t_0, super-loop (figure 10) computes that 'least-slack' is less than 'minimum-slack' and so the flow of execution returns to `check-clocks` rather than proceeding to `run-FSMs`. As a result CLK-B is able to proceed to a signal update immediately following CLK-A. Subsequently the FSMs of both clock domains are called and the cycle repeats at t_1.

The benefit of using the 'minimum-slack' mechanism is that it enables priority to be given to the synchronous signal update process (which is assumed to be hard real-time critical since it drives the OUT signals) at the expense of the FSM calls, which are not time critical, subject to each FSM completing execution at some time before the next synchronous update is due.

We have not analyzed the effect of 'minimum-slack' from a theoretical standpoint and leave it to be engineered on the application basis. There naturally is a trade-off in setting the value of 'minimum-slack' since if a value is specified that is too high then execution of the FSM may be unnecessarily delayed and a failure case may result as illustrated in figure 11 (case B).

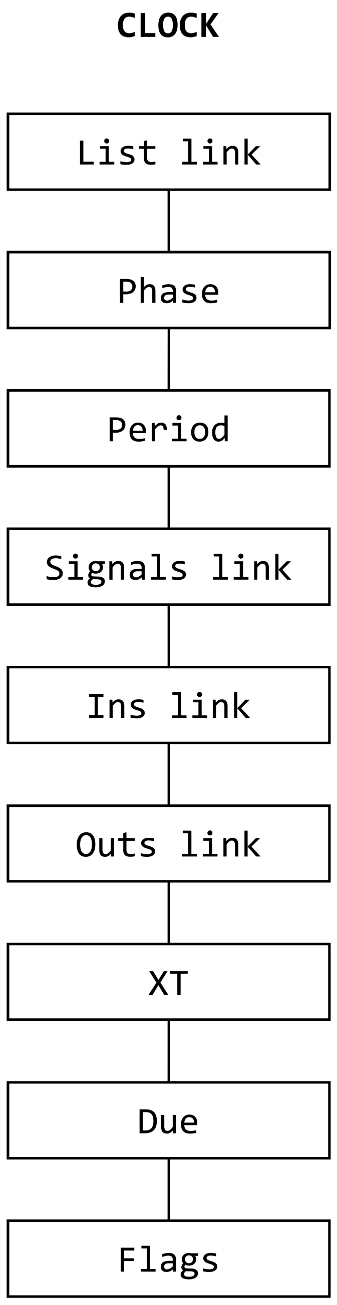


Figure 9: Structure of the CLOCK domain entity

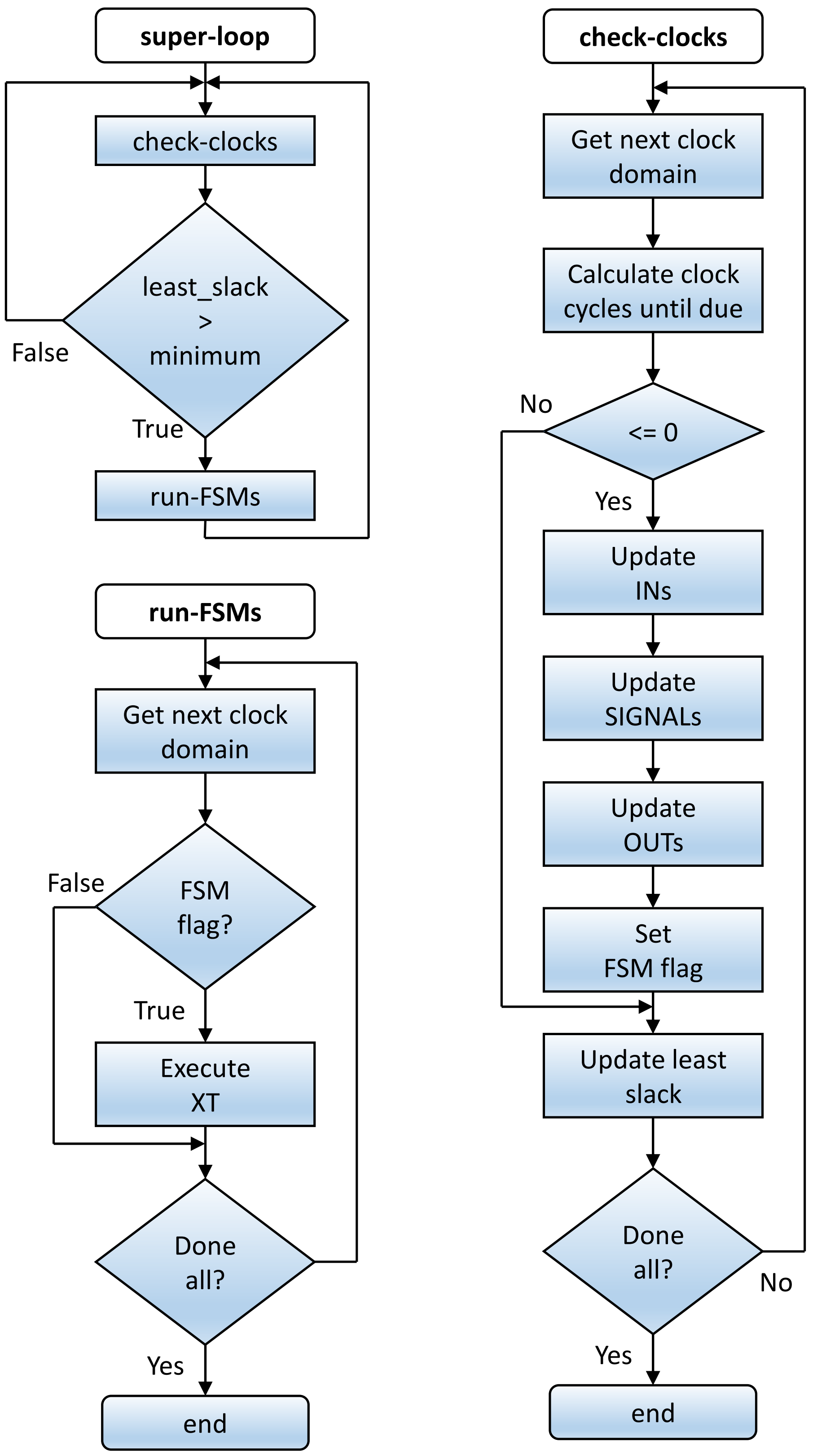


Figure 10: Flowchart of the FORTH software for the framework.

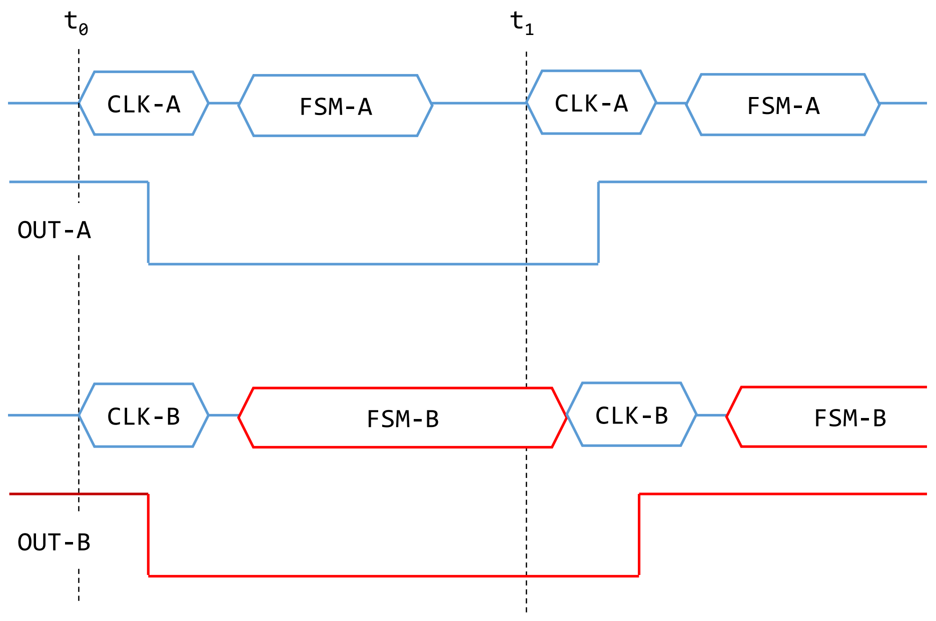


Figure 11: Timeline diagram of the timing framework in operation. A and B are separate cases and are presented on the same diagram only for comparison purposes. Case A illustrates typical operation of the framework with a single clock domain. Case B illustrates that there is a practical lower limit that can be specified for the period of a clock domain and that the computation requirement for the FSM computation is a relevant factor to be taken into account

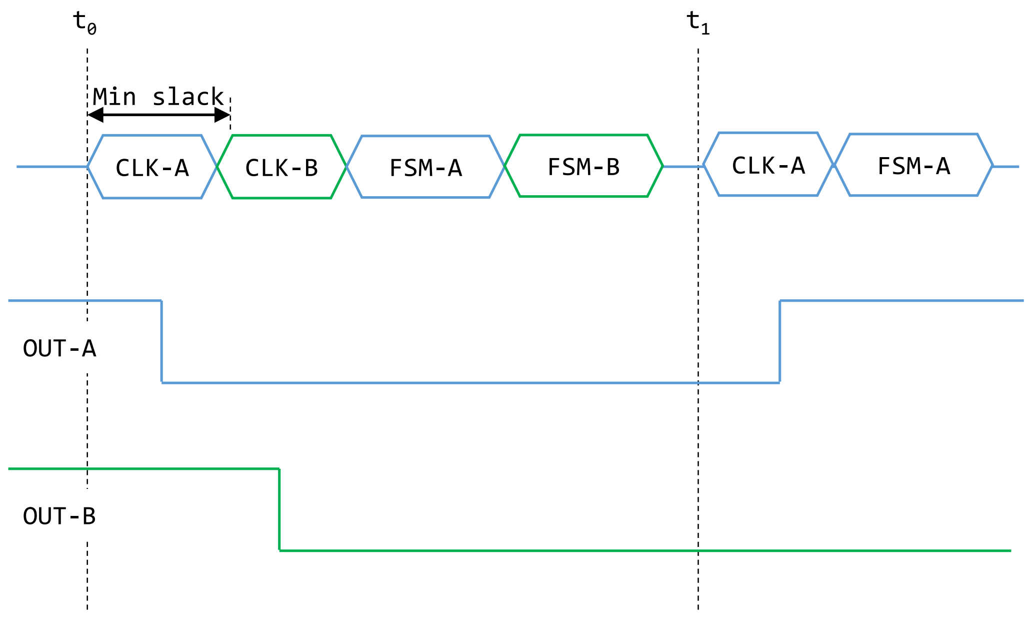


Figure 12: Multiclock timeline. In this example it is assumed that clock domain A has exactly half the period of clock domain B and that the two clock domains are (approximately) in phase.

5.8 Finite state machine logic

Argued by analogy with typical digital logic design practices, finite state machines are a natural complement to the synchronous logic entities that we present within our framework. However our framework does not insist that the executable attached to each CLOCK domain be a finite state machine. It could be arbitrary FORTH code, but in that case the system will no longer be within the scope of this paper. In particular the success of the design in meeting timing requirements will be implementation dependent.

The SIGNAL construct itself also provides a simple mechanism to implement a finite state machine in Forth, as illustrated in the following listing. The benefit of using a SIGNAL as the FSM state variable is that it may be updated at any point in the program flow, but the next value will not take effect until the time of the synchronous update which is guaranteed to be after completion of the entire FSM executable. Hence next state and output calculation logic may be cleanly divided between control structures.

```
0 constant state_init
1 constant state_A
2 constant state_B

1kHZ state_init SIGNAL state
1kHZ 0 SIGNAL s0

: my-fsm

    \ next state logic
    state CASE
        state_init OF state_A => state ENDOF
        state_A OF ( some next state logic) ENDOF
        state_B OF ( some next state logic) ENDOF
    ENDCASE

    \ output logic
    state CASE
        state_init OF 0 => s0 ENDOF
        state_A OF ( some output logic) ENDOF
        state_B OF ( some output logic) ENDOF
    ENDCASE

;

\ add this FSM to the clock domain
' my-fsm 1kHz FSM   \ attach the FSM to this clock domain
```

5.9 Interactivity via an additional interpreter

One of the advantages of embedded programming in Forth is the interactive use of the interpreter during development. We have retained this facility without compromising our framework by implementing a threaded code Forth interpreter as a finite state machine. The interpreter is activated by setting up a new clock domain with a suitable time period and attaching to it the interpreter's finite state machine. With the interpreter in place, the usual debugging capabilities such as inspecting variables or memory locations, making interventions in stored values, running diagnostic routines, etc. are all available. In addition the interpreter can be used to make on-the-fly changes to clock domains and their associated finite state machines.

The interpreter is described in a separate technical report [14] by Ulrich Hoffmann so here we give just a rough overview of its working principles.

Whereas many of today's Forth systems compile definitions to machine code, traditionally Forth has been implemented by means of an address interpreter (the so called *inner* interpreter to contrast it with the source code analyzing text (*outer*) interpreter). For this, the systems implement a small interpreter loop usually named `NEXT` that is highly optimized for overall system speed (and thus often no longer recognizable as a loop).

The interpreter loop can certainly also be implemented in high-level Forth. The current implementation adheres to ANSI Forth. It manages an instruction pointer that traverses arrays of execution tokens. Words are executed by invoking these execution tokens. Special primitives for instruction pointer manipulation (control structure primitives) are defined. This address interpreter is the basis of an entire new Forth system (including a text interpreter of its own), the *Guest* system. It shares some functionality with the surrounding *Host* system but can be different in any aspect. Defined words of the Host System are primitives of the Guest system.

The Guest has the beneficial property that its address interpreter can be invoked to just carry out a single interpretation step and then transfer control back to the caller. This allows to implement the Guest as a state machine where each transition performs just a single address interpreter step. Choosing an appropriate clock domain allows to fine tune the Guest execution speed. The Guest is much slower than the Host as its `NEXT` is not optimized for speed, but it can use Host primitives. We found the interpreter to be fast enough for reasonable interactive use. So here we have a slow but interactively usable Forth systems that fits our framework.

6 Implementation on specific platforms

We now discuss the practical issues arising from the implementation of this framework on a number of Forth platforms

6.1 General considerations

As mentioned, we sought the widest applicability of our framework by minimizing the requirements of the underlying system. Our framework expects only ANSI Forth and a free-running timer counter. It has been successfully implemented and tested in VFX Forth, G-Forth and Mecrisp on the Texas Instruments Tiva-C.

All of the above platforms allow the Forth dictionary to be hosted entirely in RAM as opposed to FLASH. Having the dictionary entirely in RAM freed us from needing to consider any implications that would arise from the separation of the dictionary into executable elements and dynamic data elements. The framework code would require modification on systems where the dictionary is not entirely hosted in RAM. Our preliminary analysis has convinced us that the changes required to deal with a mixed FLASH/RAM dictionary structure would not be major. However we prefer to omit further discussion of that issue in the current paper as we consider it to be a side topic relevant to certain implementations only.

6.2 VFX Forth

Implementation on VFX Forth was very straightforward since we could rely on full ANSI Forth compatibility. We defined the necessary support words as follows

```
: CPU-time ( -- n)
        ticks                   \ 1ms increments
;

: init-CPU-time ( --) ;
```

6.3 G-FORTH

Implementation on G-FORTH was likewise straightforward.

```
: CPU-time ( -- n)
        ntime drop 10 /         \ 10ns scale - increments will be larger
;

: init-CPU-time ( --) ;
```

Parameter	Result	Notes
Jitter	1.75%	Relative standard deviation of the period of the generated square wave
$t_{\mathrm{Clock\_Output}}$	10 microseconds	Incremental time delay for each additional OUT to be updated
f_{max}	5 kHz	Highest CLOCK period achieved using an illustrative synthetic test

Table 1: Summary of results on the Tiva-C at 16MHz with Mecrisp Forth

6.4 Mecrisp on the Tiva-C

We were very pleased to have the Mecrisp platform available on the Texas Instruments Tiva-C to implement our framework and conduct real time measurements in hardware. Mecrisp is not a completely ANSI compatible system, as a result we prepared an ANSI compatibility layer to support our framework. Otherwise implementation on the Mecrisp was also straightforward. Rather than present a detailed report of our ANSI compatibility layer within this paper we intend to make our notes available as a separate technical report.

7 Measurements

Following implementation on the Mecrisp Tiva-C platform we conducted a number of practical investigations using the framework. We used the generation of a square wave by SIGNALs within a CLOCK domain as our synthetic test for measurement purposes. This synthetic test has the merits of simplicity and convenience and we intend it only for illustration purposes. We recognize that a square wave would likely generated microcontroller PWM (pulse wave modulation) facilities in an actual application.

Table 1 summarizes the quantitative measurements. These and additional qualitative tests are described in the following sections.

7.1 Jitter

Jitter is commonly defined as the deviation of our synchronous signal updates from true periodicity. We measured the period of 30 individual square wave periods using a PC oscilloscope and determined the standard deviation of the clock period expressed as a percentage of the mean. This is the relative standard deviation, which we measured at 1.75%. Figure 13 is an oscilloscope trace of the actual output measured on the Tiva-C. The file jitter.fs in the test listing section shows the code used to generate it.

7.2 $t_{\mathrm{Clock\_Output}}$

We examined the delay that our framework requires to "synchronously" update each additional OUT signal in a clock domain after the first OUT signal (fig. 14). In a digital logic design, parallel logic elements would be responsible for updating all outputs simultaneously, but with a software framework outputs are necessarily updated one by one. We define the delay to be the incremental $t_{\mathrm{Clock\_Output}}$ times of our framework (i.e. the marginal delay for each additional OUT signal). It was measured at approximately 10 microseconds.

7.3 Multiple clock domains

We conducted qualitative tests to assess the ability of our system to support multiple clock domains (figs. 15, 16). Although we did not make specific measurements we observed that the signals from two clock domain appeared to be stable over a period of approximately six hours.

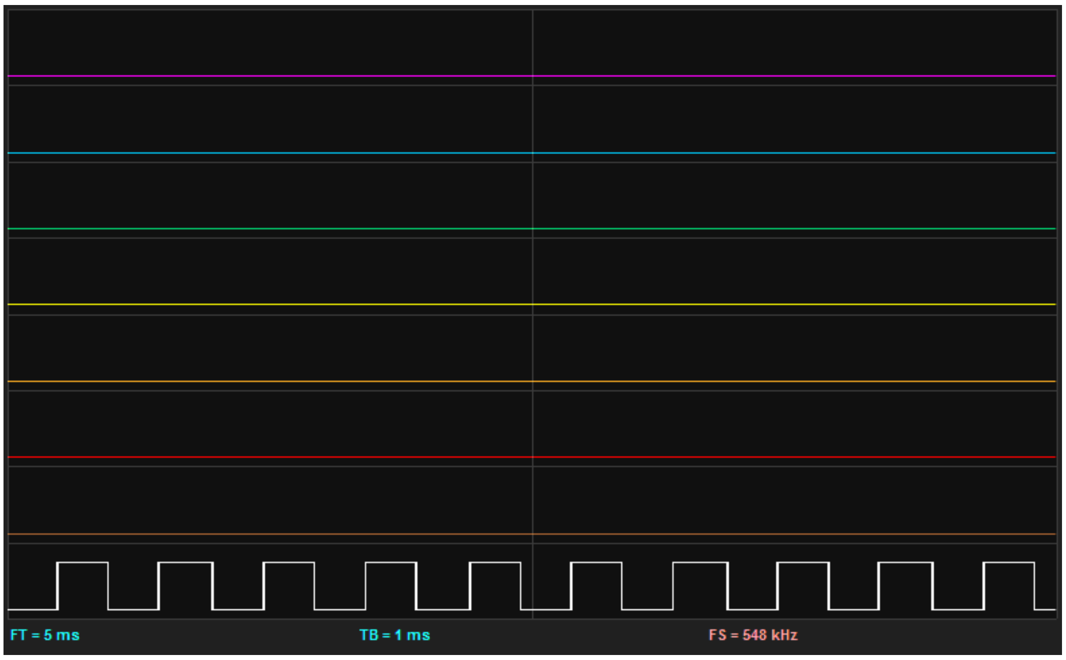


Figure 13: Measurement of jitter. A single square wave is being generated. The duration of each cycle from rising-edge to rising-edge was measured using the on-screen cursor of a PC oscilloscope. In all 30 cycles were measured.

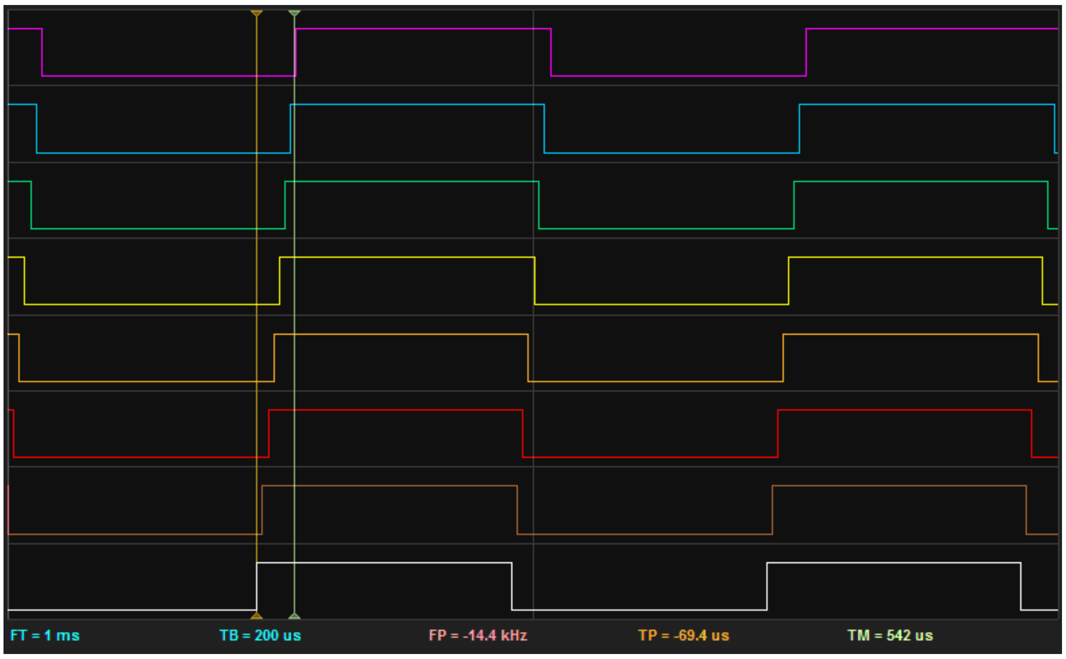


Figure 14: Measurement of $t_{\text{Clock\_Output}}$. A series of signals have been generated. The lowest signal is taken as the leading edge of the clock and the delay in the output of the other signals was measured.

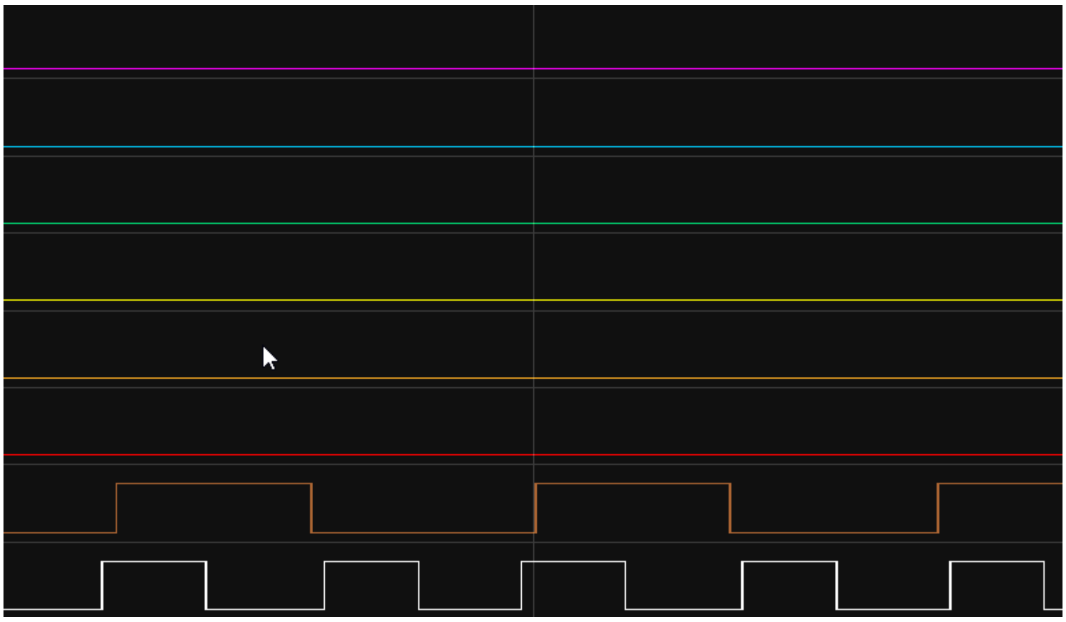


Figure 15: Two synchronous clock domains. The lower signal is a 1 kHz square wave (generated in a 2 kHz clock domain that inverts the output each cycle). The upper signal is a 0.5 kHz square wave specified with no phase offset to the lower signal. The transition edges of both signals should theoretically coincide exactly along the time axis, but there is an offset due to the inherent limitation of using a single CPU to generate both outputs.

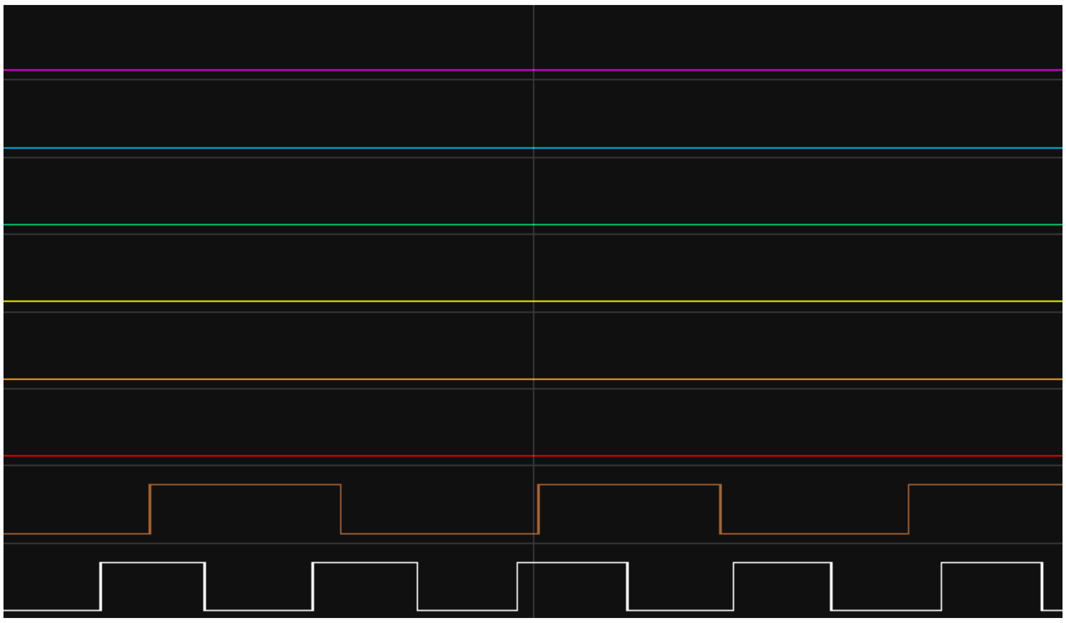


Figure 16: Two asynchronous clock domains. In this case the lower signal is a 1 kHz square wave and the upper signal is a 0.75 kHz square wave. The two clock domains are asynchronous in the sense that their transition edges can occur at any point relative to the phase of the other. Visual inspection of the signals in the above figure and over a period of several hours with our live oscilloscope trace confirmed that the framework produced stable outputs in spite of the changing phase relationships.

8 Discussion

8.1 Postulated advantages of this approach

This article presents a novel framework for real-time control in FORTH. At this stage we have evaluated the framework on the Mecrisp platform with various test measurements but we have not implemented an actual application. Nevertheless we postulate some advantages compared to traditional multitasking based approaches.

Firstly, whilst the finite state machine methodology is a more constrained programming model than synchronous processes, it renders the system capable of being conceptualized at a more abstract level and, in principle, systematic techniques suitable for the evaluation of FSMs can be applied to this framework. The SIGNAL construct itself provides a convenient approach for implementing FSMs in Forth.

Secondly, because our framework separates the reading and writing of external registers from the computational code and application control flow, we are able to specify the timing of signal updates more predictably and also come closer to a true synchronous system.

Thirdly, because our framework is running on a single CPU, there are no meta-stability issues with the signals of different clock domains, since signal update is effectively atomic from the perspective of the FSM logic.

Finally, compared to other time-triggered architectures that focus in the main just on the timing of code execution, we argue that our framework is possibly a more complete approach because it adds SIGNALS, INS and OUTS as well as CLOCK domains for triggering events.

8.2 Limitations

We naturally also recognize a number of limitations to our framework.

Firstly, not all embedded systems developers are attracted to develop applications using the construct of finite state machines. This is a matter of programming model preference. Our framework imposes the additional constraint on the FSM design since in order to meet the timing requirements of a certain clock frequency the FSM update logic must complete within a limited amount of time (or the FSM must be split into simpler sub-units of computation).

Secondly, any layer that sits between application code and the CPU will naturally consume resources and limit maximum performance compared to the potential performance of hand-crafted assembler. We achieved a maximum CLOCK frequency of 5kHz on the 16MHz Tiva-C microcontroller board in our synthetic test.

On a related note, the use of a single CPU rather than true parallel logic also introduces latency into the update of output signals. We measured a 10us delay between the update of consecutive signals. By comparison, with logic circuits implements in commonly available FPGA's, we would expect that such latencies could easily be constrained within half a clock cycle.

Perhaps most seriously of all we do not offer any method for computing whether an application will be able to meet hard real time requirements using our framework [3]. Allied to this point we do not offer a systematic procedure for setting the min_slack, which is critical to the operation of multiple clock domains. Instead it is left for trial and error tuning during application testing.

As a next step it might be useful to instrument the FSM engine to track the actual number of CPU cycles spent in `run-FSM` and record the incidence of timing glitches such as those illustrated in case B of figure 11.

8.3 Possible applications

At present our framework has been presented at a conceptual level with a small number of trial implementations and synthetic measurements offered as evidence of practical feasibility. To be properly relevant to embedded systems developers our framework would prove its worth in a real-world application. We are considering possible robotics applications in this regard.

Within the field of test and measurement our framework could also be a platform for the rapid development of an FSM-based system aided by the interactivity provided by the FORTH terminal prior to translation into FPGA's or ASIC's.

9 Proposal for an alternative implementation approach

One of our objectives in developing this framework was to minimize the requirements that we demand from the underlying system. Hence our choice to require only a free-running timer counter from the underlying system and use what is effectively a busy loop to schedule synchronous updates. The disadvantage of this approach is the lesser precision of a busy-loop in calling the synchronous updates as compared with a timer-driven interrupt. For completeness we present an interrupt driven model for framework operation in figure 17. At the present time we have not implemented this model and offer it as a proposal for next steps.

10 Conclusion

We have presented a novel framework using Forth in applications with hard real-time requirements. Our framework is directly inspired by the methods of synchronous digital logic design and we have introduced constructs intentionally borrowed from VHDL, such as clock domains, SIGNALs, INs and OUTs. Our framework is compatible with any ANSI Forth system that includes a free running timer/counter. We have implemented the framework in VFX Forth, GForth and Mecrisp on the Texas Instruments Tiva-C. We devised some synthetic tests on the Tiva-C platform and made some measurements to give a qualitative sense of the performance of our framework and offer evidence of its practical feasibility. For our framework to become relevant to embedded developers we recognize that its effectiveness in real world applications would need to be demonstrated. Nevertheless we suggest a number of advantages to the use of our system in hard real time applications. In essence our framework moves application design along the spectrum from the relative freedom of a pure software approach to the more constrained (and therefore arguably more reliable) approach of synchronous digital logic design. We are currently considering possible applications, potentially in robotics.

The authors are grateful to Matthias Koch for the availability of the Mecrisp platform and his assistance to us during discussions, and to the anonymous academic reviewers for their helpful comments.

References

[1] Architecture datasheet PB002-100822, GreenArrays, Inc., 2010

[2] Finite State Machines in Hardware, Volnei Pedroni, MIT Press, 2013

[3] Philip Koopman, “Better Embedded System Software”, Drumnadrochit Press, 2010

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

[5] Hintenaus, Peter, “Engineering Embedded Systems: Physics, Programs, Circuits”, Springer, 2015

[6] Edward A. Lee and Pravin Varaiya, “Structure and Interpretation of Signals and Systems”, Second Edition, LeeVaraiya.org, 2011

[7] James Basile, “A Forth Finite State Machine”, The Journal of Forth Application and Research 1,2, 1982

[8] E. Rawson, “State Sequence Handlers”, The Journal of Forth Application and Research 3,4, 1986

[9] Julian V. Noble, ”Finite State Machines in Forth”, The Journal of Forth Application and Research 7,1, 1995

[10] Everett F. Carter Jr, “Robots and Finite-State machines”, Dr. Dobb’s Journal, February 1997

[11] Starling, M. K, "A Hardware/Software Finite State Machine Implementation", 1983 Rochester Forth Applications Conference. Rochester: Institute for Applied Forth Research, 1983.

[12] Albert Nijhof, “Goto in Forth”, http://home.hccnet.nl/anij/c/c213a.html, last access 2016-06-23

[13] Hermann Kopetz and Günther Bauer,”The Time-Triggered Architecture”, Proceedings of the IEEE, 2003

[14] Ulrich Hoffmann, “Implementing the Forth Inner Interpreter in High Level Forth”, Technical Report, EuroForth 2016

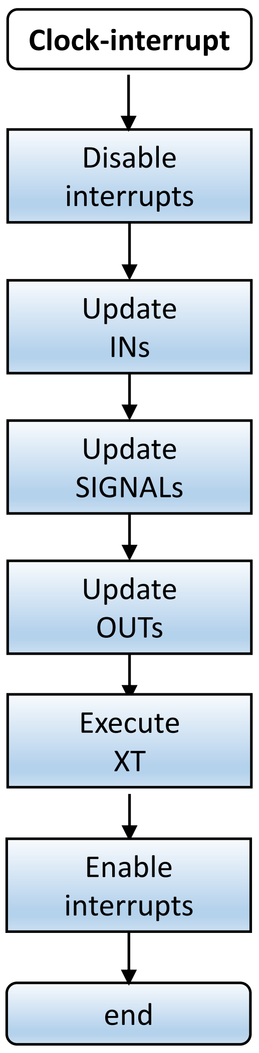


Figure 17: Alternative scheme for triggering each clock-domain with an interrupt

Source code listing

```
: field:
\ Create and use fields in a structure
    Create ( offset size -- offset' )
        over ,                  \ save the current value of the offset
        +                       \ increment the offset by this field's size
    Does> ( structure-base -- field-address)
        @ +                     \ add this field's offset to the structure-base
;

\ CLOCK domain data structure
0
1 cells field: >link            \ link field to prior CLOCK or zero if the first CLOCK
1 cells field: >phase           \ relative phase offset of this CLOCK in CPU clock cycles
1 cells field: >period          \ period of this CLOCK in CPU clock cycles
1 cells field: >signal-link     \ linked list of signals operated by this CLOCK
1 cells field: >in-link         \ linked list of IN's operated by this CLOCK
1 cells field: >out-link            \ linked list of IN's operated by this CLOCK
1 cells field: >xt              \ execution token of this CLOCK's FSM
1 cells field: >due             \ count in CPU clock cycles when this CLOCK is next due to execute
1 cells field: >flags           \ boolean flags bit0: 1 = alive, 0 = sleeping
drop

\ SIGNAL data structure
1 cells \      >link            \ link field to prior signal or zero if the first signal in this clock domain
1 cells field: >current         \ current signal value
1 cells field: >next            \ becomes this value at UPDATE
1 cells field: >reset           \ becomes this value at RESET
drop

\ IN port / OUT port data structure
1 cells \      >link            \ link field to prior IN or zero if the first IN in this clock domain
1 cells field: >addr            \ memory mapped register referenced by this IN
1 cells field: >signal          \ signal referenced by this IN
drop

variable clock-link 0 clock-link !  \ pointer to linked list of CLOCK domains

: {nothing} ( --)
\ dummy FSM
;

: CLOCK
\ create a new clock domain
    Create ( period phase <NAME> --)
        here clock-link @ ,         \ link
        clock-link !                \ save this clock's location to the global clock-link variable
        ,                           \ phase
        ,                           \ period
        0 ,                         \ signal-link
        0 ,                         \ in-link
        0 ,                         \ out-link
        ['] {nothing} ,             \ XT
        0 ,                         \ due
        0 ,                         \ flags
    Does> ( -- structure-base)
                                    \ return the address of the clock structure
;

: FSM ( XT clock-domain --)
\ set the finite state machine associated with a clock-domain
    >xt ! ;

: SIGNAL
\ create a new signal
    Create ( clock-domain default-value <NAME> --)
        swap here swap              ( default-value structure-base clock-domain)
        >signal-link dup @          ( default-value structure-base signal-link last-signal)
        ,                          ( default-value structure-base signal-link) \ link
        !                          ( default-value) \ save this signal's location to signal-link
        dup ,                      \ current
        dup ,                      \ next
        ,                          \ reset
    Does> ( -- current value)
        >current @                 \ return the current value of the signal
;

: IN ( clock-domain addr <SIGNAL> --)
\ create a new IN port
    swap here swap             ( addr structure-base clock-domain)
    >in-link dup @             ( addr structure-base in-link last-in)
    ,                          ( addr structure-base in-link)
    !                          ( addr)
    ,                          \ addr
    ' >body ,                  \ signal
;

: OUT ( clock-domain addr <SIGNAL> --)
\ create a new OUT port
    swap here swap             ( addr structure-base clock-domain)
    >out-link dup @            ( addr structure-base in-link last-in)
    ,                          ( addr structure-base in-link)
    !                          ( addr)
    ,                          \ addr
    ' >body ,                  \ signal
;

: => ( n <name> -- )
\ store a value in the next field of a SIGNAL (better not to redefine TO?)
     ' >body >next  state @ IF postpone literal  postpone !  EXIT THEN ! ; immediate

: {update-signal} ( 'signal -- )
\ update a signal to its next value
     dup >next @  swap >current ! ;

: {reset-signal} ( 'signal -- )
\ update a signal to its default value
    dup >reset @  swap 2dup >current ! >next ! ;

: {update-in} ( 'in -- )
\ read an IN port and write to the >next field of its SIGNAL
    dup >addr @ @ swap >signal @ >next ! ;

: {update-out} ( 'out -- )
\ write to an OUT port, the >current field of its SIGNAL
    dup >signal @ >current @ swap >addr @ ! ;

: do-list ( xt link -- )
    BEGIN                      ( xt link)
        @ dup                  ( xt 'item)
    WHILE                      ( xt 'item)
        over over >r >r swap   ( 'item xt)
        execute                ( --)
        r> r>                  ( -- xt 'item)
    REPEAT
    drop drop ;

: do-clocks ( xt --)
\ apply XT to all clocks in turn
\ XT must have signature (i*x 'clock -- j*x)
    clock-link do-list ;

: do-signals ( i*x clock-domain xt -- j*x )
\ apply XT to each signal in turn in a given clock-domain
    swap >signal-link do-list ;

: do-ins ( i*x clock-domain xt -- j*x )
\ apply XT to each IN in turn in a given clock-domain
    swap >in-link do-list ;

: do-outs ( i*x clock-domain xt -- j*x )
```

```
\ apply XT to each IN in turn in a given clock-domain
    swap >out-link do-list ;

: UPDATE-SIGNALS  ( clock-domain -- )
\ update all signals synchronously to their next values
    ['] {update-signal} do-signals ;

: RESET-SIGNALS ( clock-domain -- )
\ update all signals synchronously to their default values
    ['] {reset-signal} do-signals ;

: UPDATE-INS  ( clock-domain -- )
\ read all IN addresses and write to the >next fields of their associated signals
    ['] {update-in} do-ins ;

: UPDATE-OUTS  ( clock-domain -- )
\ read all IN addresses and write to the >next fields of their associated signals
    ['] {update-out} do-outs ;

: .clock ( 'clock --)
\ print the fields of a clock
    ." [CLOCK@" dup                         0 u.r
    ."  name=" dup body> >name .name
    ." , link=" dup >link @                 0 u.r
    ." , phase=" dup >phase @               0 u.r
    ." , period=" dup >period @             0 u.r
    ." , signal-link=" dup >signal-link @   0 u.r
    ." , XT=" dup >xt @                     0 u.r
    ." , due=" dup >due @                   0 u.r
    ." , flags=" >flags @                   0 u.r ." ]" cr
;

: .signal ( 'signal --)
\ print the fields of a signal
    ." [SIGNAL@"    dup                     0 u.r
    ."  name=" dup body> >name .name
    ." , link=" dup >link @                 0 u.r
    ." , current=" dup >current @           0 u.r
    ." , next=" dup >period @               0 u.r
    ." , reset=" >reset @                   0 u.r ." ]" cr
;

: .in ( 'in --)
\ print the fields of an in
    ." [IN@"    dup                         0 u.r
    ." , addr=" dup >addr @                 0 u.r
    ." , signal=" >signal @                 0 u.r ." ]" cr
;

: .out ( 'in --)
\ print the fields of an in
    ." [OUT@"   dup                         0 u.r
    ." , addr=" dup >addr @                 0 u.r
    ." , signal=" >signal @                 0 u.r ." ]" cr
;

: .signals ( clock-domain --)
\ print all of the signals in a clock domain
    cr
    ['] .signal do-signals
;

: .clocks ( --)
\ print all of the clock domains
    cr
    ['] .clock do-clocks
;

: .ins ( clock-domain)
\ print all of the IN's in a clock domain
    cr
    ['] .in do-ins
;
```

```

: .outs ( clock-domain)
\ print all of the IN's in a clock domain
    cr
    ['] .out do-outs
;


variable slack
\ slack is updated by check-clocks, it contains the number of CPU cycles until the next clock domain due time

variable min-slack  0 min-slack !
\ min-slack is fine-tuned by the designer. If slack < min-slack then super-loop will wait
\ for the next clock rather than proceed with FSM execution

: {initialize-clock} ( t0 'clock -- t0)
\ initialize a clock given the t0 value of CPU-time
    >R dup R@ >period @ R@ >phase @ + + R@ >due !       \ set due time
    0 R@ >flags !                                       \ clear flags
    R> reset-signals                                    \ reset all signals
;

: initialize-clocks ( --)
\ initialize all clock domains
    CPU-time ['] {initialize-clock} do-clocks drop ;


: {check-clock} ( 'clock --)
\ check if this CLOCK is due and if so update the SIGNALS, set flags = alive, and schedule the next clock
    dup >due @                  ( 'clock due)
    CPU-time -                  ( 'clock cycles-until-due)
    dup 0 <= IF                 ( 'clock cycles-until-due)
        drop                        ( 'clock)
        dup update-ins                  \ all IN's, SIGNALs and OUTs now update synchronously
        dup update-signals
        dup update-outs
        dup >flags dup @            ( 'clock 'flags flags)
        1 OR swap !                     \ set flags so to indicate that the FSM is due to run
        dup >period @ dup >R        ( 'clock period R:period)
        over >due @                 ( 'clock period last-due R:period)
        +                           ( 'clock next-due R:period)
        over >due !                     \ determine and save the next due time
        R>                          ( 'clock period)
    THEN                            ( 'clock cycles-until-due/period)
    slack @ MIN slack ! drop    \ update the slack
;

: check-clocks ( -- slack)
\ check all clocks, update SIGNALS and flags where clocks are due, and update slack
    100000000 slack !                   \ initial dummy value
    ['] {check-clock} do-clocks
    slack @
;

: {run-FSM} ( 'clock --)
\ check if this clock is now active to run and if so, run the FSM
    dup >flags @                        ( 'clock flags)
    1 AND IF                            ( 'clock )
        \ run the FSM logic
        dup >r >xt @ execute r>
        \ set flags so that this task will now sleep)
        dup >flags dup @                ( 'clock 'flags flags)
        254 AND swap !                  ( 'clock)
    THEN
    drop
;

: run-FSMs ( --)
    ['] {run-FSM} do-clocks ;

: super-loop
```

```
    BEGIN
        BEGIN
            check-clocks    ( slack)
            min-slack @ >   ( flag)
        UNTIL
        run-FSMs

        key? drop           \ VFX FORTH needs this to facilitate Windows refresh
    AGAIN
;

: main ( -- )
        reset \ threaded code interpreter
    initialize-clocks
    super-loop
;
```

Test listings

```
jitter.fs

erase-clocks

FCPU 2000 / 0 CLOCK 2kHz

2kHz 0 SIGNAL s0

: toggle
    s0 not => s0
;

' toggle 2KHz FSM

2KHz LOGIC0 OUT s0

: test
    main
;
```

```
max_freq.fs

erase-clocks

10000 constant freq  \ frequency in Hz

FCPU freq / 0 CLOCK CLK
CLK 0 SIGNAL s0

0 variable t0

: toggle
    s0 not => s0
    100 0 DO i t0 ! LOOP
;

' toggle CLK FSM

CLK LOGIC0 OUT s0

: test
    main
;
```

Simulating Recurrent Neural Networks in Forth

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

Introduction. Recurrent neural networks (RNN) [1] regained attention of researchers as an instrument for data recognition with a great variety of strategies for transmitting information (usually binary images, video, and audio frames) among their network layers and ways of information transformation and processing. After a certain boom in instrument creation a number of software tools appeared [2, 3] which helped researchers to study various aspects of RNN-based solutions, Matlab [4] probably being among mostly widespread ones. However, most of these tools look like "dinosaurs" – they are huge and inflexible for running sophisticated experiments with carefully carved parameters and features.

An alternative approach based on the "small is beautiful" paradigm [5] was successfully used to overcome some of these hurdles, tools based on the Python language [6, 7] being quite successful and thus encouraging to try other options.

This paper describes a program, called the Associative Intellectual Machine (AIM) [8], for simulating the behavior of a multi-layer RNN under a particular scenario of input signals incoming and the given structure of their further propagation among RNN layers. An input signal considered as a binary image is converted into a matrix of pulses that propagate through the RNN and may produce a series of output signals in the same form. AIM is a prototyping tool for studying the associative memory mechanism modeled through such an RNN and establishing its characteristics (e.g., the precision of recognition of presented binary images) in comparison with conventional memory under various space-time structures for pulse propagation [9]. The ultimate goal of this preliminary research is to design an AIM as an analog associative memory device of the RNN architecture on highly parallel memristors [10] as its base elements.

In the classical McCalloch-Pitts discrete neural network model with N neurons ν_i (i=1..N) (often referred to as the perceptron model), the N binary inputs $x_j(t)$ of each neuron ν_i received at the time moment t are converted into its single binary output $y_i(t+1)$ at the next time moment t+1 according to the formula:

$$y_i(t+1) = max(0, signum(\Sigma_{j=1..N}(w_{ij} \times x_j(t) + w_{i0}))).$$

Here w_{ij} are the weights of the input synaptic links to the neuron ν_i and w_{i0} is the so called threshold value for this neuron.

In a more general Hebb model, the weights of synaptic links are updated at each step of the network functioning, thus performing *training* of the network:

$$w_{ij}(t+1) = w_{ij}(t) + \eta \times y_i(t) \times y_j(t).$$

Here η is the so called "training factor" usually selected from the interval [0.7..0.9]. A number of different training strategies were proposed; however, they all comply with the known Widrow-Hoff rule:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t+1), \quad \Delta w_{ij}(t+1) = x_j(t) \times (d_i(t+1) - y_i(t+1)),$$

$d_i(t+1)$ being the expected output value at the next time moment t+1. After a series of trainings on the provided set of M inputs $X=\{X_1,X_2,...X_M\}$ (where each X_j is a vector of N binary values $X_j=\{x_{1j},x_{2j},...,x_{Nj}\}$) and expected outputs $D=\{d_1,d_2,...d_M\}$ the weights are not changed anymore and the network continues its performance with thus obtained weights of the synaptic links.

In an RNN of the classical Hopfield model the output $y_i(t+1)$ also serves as an input to all other neurons at the next time moment t+1.This *feedback* link extends the network capabilities

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

for training and self-learning of the RNN. In this model pulse propagation is relatively straight forward (even for recurrent networks) and homogeneous for all constituting neurons.

Existing Forth implementations of neural networks, like [11], [12], and [13], follow the above mentioned classical models of one or more layer networks with discrete timing cycles of 1 unit. In contrast to these, AIM uses various timing delays in pulse (or excitation) propagation and elaborates further the *routes* or *paths* of such propagation among the network layers. The neuron layers are split into equal fields and a path is specified through enumerating them as they occur in this path [14]. Neurons in two adjacent fields of a pulse propagation path are called neighboring or twin neurons if they are located in the same places in these adjacent fields. While in general case each RNN neuron is linked to all other neurons located in the adjacent layers and thus may receive/send pulses from/to them, the neighboring neurons have priority in pulse propagation over other neuron pairs: the effect of a pulse between them is much stronger than that of a pulse between non-twin neurons.

The main distinguishing feature of the AIM is how the neuron output and the weight of the respective synaptic link are recalculated when a pulse comes through this link, this may change the neuron potential which is its output and the weight of the link, both considered as integers. The new potential $U_\nu\ (t+\Delta t)$ of a neuron ν at the next time moment $t+\Delta t$ equals to the sum of all its inputs, not exceeding some U_{max}:

$$U_\nu(t+\Delta t) = min(U_{max}\,,\ max(0, U_\nu(t) + \Sigma_{\eta\in\{\nu\leftrightarrow\eta\}}(U_\nu(t) - U_\eta(t)) \times w_{\nu\leftrightarrow\eta}))),$$

where $\{\nu\leftrightarrow\eta\}$ is a set of all synaptic links connecting neurons ν and η, and $w_{\nu\leftrightarrow\eta}$ is the weight of this link $\nu\leftrightarrow\eta$ which is recalculated accordingly:

$$w_{\nu\leftrightarrow\eta}(t+\Delta t) = w_{\nu\leftrightarrow\eta}(t) + f(\nu,\eta)$$

where f is a function of these two neurons which tends to zero very fast as the distance between these neurons increases. It's noteworthy that this function may be programmed with fixed point arithmetic only (using tables for expressions like $y=e^{-x}$) thus avoiding floating point arithmetic with related issues and trade-offs. This approach assumes that weights and potentials are treated as integers multiplied by some scaling factor (e.g., 10000 for precision of up to 4 digits).

This AIM program was developed in compliance with the Forth 200x standard [15] on the VFX Forth for Windows IA32 [16] platform to be portable and run on any 32-bit Forth system compliant with Forth 200x, including freeware platforms like gForth [17]. Its core is an event-driven engine. AIM allows the user to specify various cases and combinations of "experiment parameters", to specify pulse propagation paths, timing delays, etc., and to visualize the obtained results of RNN behavior simulation as well as the very process of their development in order to objectively estimate and compare them. Fig. 1 presents a typical AIM consisting of a two-layer RNN with one propagation path (its projection on the upper layer is marked with a dotted line).

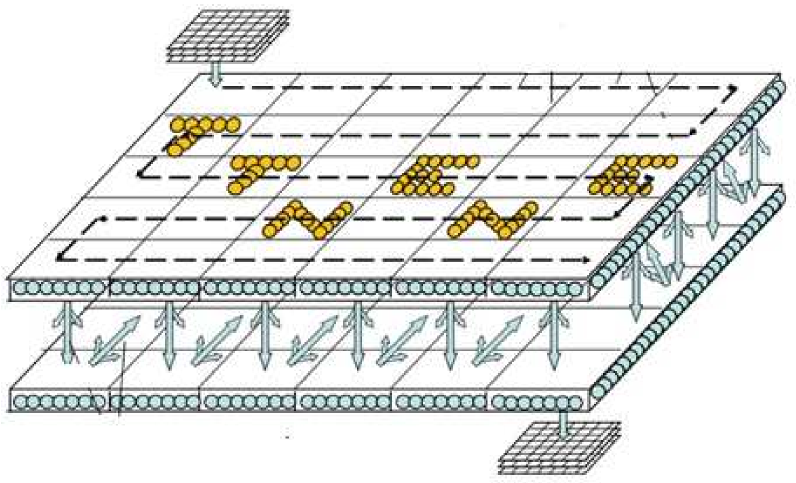


Fig. 1. A two-layer RNN of 5×6=30 fields with 6×7=42 neurons each

The simulator size is around 2 KLOC in Forth and employs a simple model of a multi-layer RNN with a user-defined interface. It relies on other advanced tools for further analysis and visualization of simulation results. The AIM source code is planned to be uploaded to an appropriate open-source repository and is currently available from the author on request.

To minimize the number of various code patterns in the code this Forth implementation intensively uses iterators, which perform the same parameterized action on each of homogeneous elements of a data structure composed by them in form of an array or a vector. They remind the standard word `TRAVERSE-WORDLIST` in their effect but rely on user-defined data structures, other than the implementation dependent list of words in a Forth vocabulary.

Representing RNN in Forth. The AIM program represents its subject RNN with special data structures and reuses an approach developed earlier [18] with a discrete counter of system time and a list of all simulated events `EventList` attached to the time axis with respective time-stamps.

Each neuron ν of the network may be in one of two states: *excited* or *unexcited*, and is characterized by the value of its current potential $U_ν$: $0 \le U_ν \le U_{\max}$ which changes with time. An unexcited neuron becomes excited when its potential reaches some value U_{min}: $0 < U_{min} \le U_{\max}$.

$F_0[0,0]$	$F_0[0,1]$	$F_0[0,2]$	$F_0[0,3]$	$F_0[0,4]$	$F_0[0,5]$
$F_0[1,0]$	$F_0[1,1]$	$F_0[1,2]$	$F_0[1,3]$	$F_0[1,4]$	$F_0[1,5]$
$F_0[2,0]$	$F_0[2,1]$	$F_0[2,2]$	$F_0[2,3]$	$F_0[2,4]$	$F_0[2,5]$
$F_0[3,0]$	$F_0[3,1]$	$F_0[3,2]$	$F_0[3,3]$	$F_0[3,4]$	$F_0[3,5]$
$F_0[4,0]$	$F_0[4,1]$	$F_0[4,2]$	$F_0[4,3]$	$F_0[4,4]$	$F_0[4,5]$

Upper neuron layer #0

$F_1[0,0]$	$F_1[0,1]$	$F_1[0,2]$	$F_1[0,3]$	$F_1[0,4]$	$F_1[0,5]$
$F_1[1,0]$	$F_1[1,1]$	$F_1[1,2]$	$F_1[1,3]$	$F_1[1,4]$	$F_1[1,5]$
$F_1[2,0]$	$F_1[2,1]$	$F_1[2,2]$	$F_1[2,3]$	$F_1[2,4]$	$F_1[2,5]$
$F_1[3,0]$	$F_1[3,1]$	$F_1[3,2]$	$F_1[3,3]$	$F_1[3,4]$	$F_1[3,5]$
$F_1[4,0]$	$F_1[4,1]$	$F_1[4,2]$	$F_1[4,3]$	$F_1[4,4]$	$F_1[4,5]$

Lower neuron layer #1

Fig. 2. Representation of a two-layer RNN of 30 fields as two matrices of 5 × 6 elements

Neurons form *K layers* of the same size, represented as a matrix of *M* × *N fields*, each field being itself a matrix of *m* × *n* neurons, where $1 \le K,M,N,m,n \le 32$. Thus, the total number of RNN neurons equals to the product $K \times M \times N \times m \times n$. This representation of a sample RNN in Fig. 1 is rendered in Fig. 2. The above mentioned RNN parameters *M*, *N*, *m*, and *n* are represented by Forth variables `m'`, `n'`, `m`, and `n`: respectively:

```
VARIABLE m' 5 m' ! \ The number of rows in the field layer matrix
VARIABLE n' 6 n' ! \ The number of columns in the field layer matrix
VARIABLE m  6 m !  \ The number of rows in a neuron field matrix
VARIABLE n  7 n !  \ The number of columns in a neuron field matrix
```

Fields within a layer with the number *l* ($0 \le l \le K–1$) are numbered as elements of the matrix $F_l[i,j]$, $0 \le i \le M–1$, $0 \le j \le N–1$, all numbering starting from zero as is common in Forth; neurons inside each field, in their turn, are numbered as elements of the matrix *Neu*[*i*,*j*], $0 \le i \le m–1$, $0 \le j \le n–1$. Thus, the full address of a neuron consists of 5 items (5 bits each) packed in one cell value: its layer number, 2 indices of the neuron field in this layer, and 2 indices of the neuron in this field. Fields reside in their layer matrix elements. Due to the above constraints on the index values, the described neuron address may be packed in one integer of the cell size which requires 5 × 5 = 25 bits; that's why a 32-bit Forth platform is anticipated for this AIM implementation.

Neurons are represented with data structures of `NeuronSize` cells each, fields are just vectors of neurons, and layers are vectors of fields. Upper case words are standard Forth words, while words with low shift characters are the user-defined words of this implementation.

```
5 CELLS CONSTANT NeuronSize \ The size of the data structure for a neuron
BEGIN-STRUCTURE Neuron
       CELL +FIELD Neuron.ID \ Unique ID of the neuron
       CELL +FIELD Neuron.State \ Current status of the neuron: 1 excited, 0 unexcited
       CELL +FIELD Neuron.Potential \ Current potential multiplied by scaling factor
       CELL +FIELD Neuron.SynDn \ Address of a vector of references to the upward layer
       CELL +FIELD Neuron.SynUp \ Address of a vector of references to the downward layer
END-STRUCTURE
```

Iterators simplify performing a particular action upon each neuron in a field:

```
: IterNeuField ( fieldaddr,cfa--) \ Iterator on all neurons of this field
       {: Action :}            \ Action to be executed for each neuron
       ( fieldaddr) DUP n @ m @ * NeuronSize * + ( NeuFirstAddr,NeuLastAddr) SWAP
       DO
              I ( NeuAddr) Action EXECUTE  NeuronSize
       +LOOP ;
```

E.g., in order to unexcite all neurons of a particular field (i.e., to assign zero to the neuron status stored in the second cell of the neuron structure) with the word `Unexcite` one may write:

```
: Unexcite ( NeuAddr--) Neuron.State 0! ; \ Unexcite this neuron
    ... ( ...,fieldaddr) ['] Unexcite  IterNeuField ( ...) ...
```

The address of the given field is placed on the stack prior to this code and the iterator enumerates all neurons of this field executing the `Unexcite` word for each such neuron. Iterator on each field within a layer is defined in a similar way with the only difference that the field size is represented as a variable rather than a constant because it depends on the actual number of neurons in a field.

```
VARIABLE FieldSize   m @ n @ * NeuronSize * CELL 2* + FieldSize !
: IterFieldLayer ( l#,cfa--) \ Iterator on all fields of this layer
       {: Action :}          \ Action to be executed for each field
       CELL * Layers + @ CELL+ CELL+ ( field00addr)
       DUP ( field00addr,field00addr)
       m' @ n' @ *  FieldSize @ * + SWAP
       DO
              I ( FieldJIaddr) Action EXECUTE  FieldSize @
       +LOOP ;
```

As the field structure starts with two auxiliary cells; that's why `CELL+ CELL+` is needed to obtain the address of the first field $F_l[0,0]$ in this vector. E.g., one may print out the current status of the RNN under simulation with the word `.RNN` defined as follows:

```
: .RNN ( --) \ Print-out the RNN status
       #Layers @ 0
       DO
              CR ." Layer "  I .
              I ( l#) ['] .Field IterFieldLayer
       LOOP ;
```

Here `#Layers` is a variable which stores the number of layers in the RNN and `.Field` is a word which prints out a field which address is provided to it on the stack.

Data flow (in form of pulse or excitation propagations) in an RNN occurs between its layers, as each neuron in a layer is connected to all other neurons in adjacent layers via special channels called synapses. Thus, the total number of synapses: $(K - 1) \times (M \times N \times m \times n)^2$ is rather large. However, it may be reduced with the notion of *synapse length* defined as the distance between its two neurons considered as points in a 3D space. A two-way synapse $\nu\leftrightarrow\eta$ is established between neurons ν and η from adjacent layers (considered as points in a 3D space), if and only if the distance $d_{\nu\eta}$ between them (i.e., the length of the synapse $\nu\leftrightarrow\eta$) does not exceed some predefined constant D_{max}. This distance is calculated as:

$$d^2_{\nu\eta}=D^2_z+D^2_{xy}\times((|x_\nu-x_\eta|+|x_F-x_{F*}|\times n)^2+(|y_\nu-y_\eta|+|y_F-y_{F*}|\times m)^2),$$

where D_z, and D_{xy} are scaling factors specified while configuring the AIM program and sub-indices F and $F*$ refer to coordinates of the respective field as an element of a layer matrix. However, for twin neurons the distance is equal to D_z and thus is the shortest possible. In order to minimize run-time computations, all these distances are stored as their squared values.

Each synapse is rendered with a 4 cell data structure, which stores references to its both neurons in two adjacent layers (the upward and downward ones), the synapse length and weight. The last serves as the RNN memory and may change in the process of RNN functioning if the respective flag `Training` is turned `TRUE` (such updates of synapse weights realize the so called "unsupervised training" of the RNN). Positive weight means storing data and negative weight means erasing (forgetting) it. Further research is needed to better control this "forgetting" feature of the AIM [19]. Each neuron refers to 2 sets of synapses connecting it to other neurons in the upward and downward adjacent layers (the upmost RNN layer has no upward neighbor, as well as the RNN bottom layer has no downward one), implemented as vectors of references to respective synapse structures.

```
4 CELLS CONSTANT SynapseSize \ The size of the data structure for a synapse
BEGIN-STRUCTURE Synapse
       CELL +FIELD Synapse.Weight \ The current weight multiplied by scaling factor
       CELL +FIELD Synapse.Length \ Synapse length squared
       CELL +FIELD Synapse.NeuUp  \ Reference to one neuron of the two
       CELL +FIELD Synapse.NeuDn  \ Reference to the other neuron
END-STRUCTURE
```

Each vector starts with a cell containing the number of its elements. A respective iterator is used to perform a particular action on each element referred to by such vector:

```
: IterRefVect ( VectRefAddr/NULL,cfa--) \ Iterator on all elements referred to
       {: Action :}  \ Action to be executed for each vector element
       ( VectRefAddr/NULL) ?DUP \ Check for the NULL parameter
       IF ( VectRefAddr)
              DUP @ CELL * ( VectRefAddr,VectLen) OVER + SWAP
              ?DO                \ The vector may have zero elements
                     I CELL+ @ ( RefAddr) Action EXECUTE  CELL
              +LOOP
       THEN ;
```

As was mentioned before, thus defined iterators are similar to the Forth 200x word `TRAVERSE-WORDLIST` in its semantics, but rely on different user-defined data structures in the AIM.

To reduce the number of colon definitions used only once in the respective iterator, one may use mechanism similar to quotations [20] or the word `:NONAME` of the Forth 200x standard.

With iterators, basic operations on neurons look quite straight-forward and transparent demonstrating the flexibility and power of Forth. E.g., passing excitation from an excited neuron to all unexcited neurons connected to it via synapses looks as:

```
: PassNeuExcit ( neu-addr--) \ Excite the neuron if its potential is high
\ and recalculate potentials of all neurons connected to it by synapses
       DUP Neuron.State 2@ ( neu-addr,potential,state) 0= SWAP Umin @ >= AND
       IF ( NeuAddr) \ The neuron is unexcited and its potential is high
              DUP Neuron.State 1+! \ Excite this neuron!
              DUP Neuron.SynDn @ ['] PulseDn IterRefVect
              DUP Neuron.SynUp @ ['] PulseUp IterRefVect
              ( neu-addr)
       THEN ( neu-addr)
       Neuron.State @ IF #ExcitedNeurons 1+! THEN ;
```

Words `PulseDn` and `PulseUp` propagate excitation through the synapse, passed to them as an address on stack, downward or upward respectively, using a common subroutine `PulseDn/Up` :

```
: PulseDn ( SynAddr--) \ Propagate pulse downward through the synapse
      DUP Synapse.NeuUp 2@ ( SynAddr,NeuDnAddr,NeuUpAddr) PulseDn/Up ;
: PulseUp ( SynAddr--) \ Propagate pulse upward through the synapse
      DUP Synapse.NeuUp 2@ ( SynAddr,NeuDnAddr,NeuUpAddr) SWAP PulseDn/Up ;
```

An RNN pulse propagation path is specified through non-recursive enumeration of RNN fields, each two adjacent elements in this list belonging to adjacent layers (in case of a two-layer

RNN this means alternating). The first element of this list is its entry – it may accept signals from the RNN environment in form of an input unitary image (IUI) which is a matrix of $m \times n$ binary values; each bit being mapped to a neuron in the entry field with the same matrix indices. If this neuron is unexcited, then it accepts the respective binary signal which may result in a change of the neuron potential. If the neuron potential accumulated from all synapses incoming to it reaches the value U_{min}, it becomes excited for a period of time equal to some Δt_{excite} (during that period the neuron accepts no other signals) and all excited neurons of the input field pass their excitation to other neurons and the process reiterates.

A number of paths should be specified in a configuration file to determine the routes for excitation to propagate among the RNN layers. This constitutes a distinguishing feature of the AIM. As An XML-like technique is used for that which employs a pair of words `<Path>` and `</Path>` to frame a particular path, while words `<F>` and `</F>` frame coordinates of each particular field in this path the respective order. Field coordinates consist of three integers: the layer number (0 or more), the row number (0..M – 1) in and the column number (0..N – 1) this layer; e.g.:

```
0 <Path> \ Specify a path number 0
<F> 0 0 0 </F> <F> 1 0 0 </F> <F> 0 0 1 </F> <F> 1 0 1 </F> <F> 0 0 2 </F> <F> 1 0 2 </F>
<F> 0 0 3 </F> <F> 1 0 3 </F> <F> 0 0 4 </F> <F> 1 0 4 </F> <F> 0 0 5 </F> <F> 1 0 5 </F>
<F> 0 1 5 </F> <F> 1 1 5 </F> <F> 0 1 4 </F> <F> 1 1 4 </F> <F> 0 1 3 </F> <F> 1 1 3 </F>
<F> 0 0 2 </F> <F> 1 1 2 </F> <F> 0 1 1 </F> <F> 1 1 1 </F> <F> 0 1 0 </F> <F> 1 1 0 </F>
<F> 0 2 0 </F> <F> 1 2 0 </F> <F> 0 2 1 </F> <F> 1 2 1 </F> <F> 0 2 2 </F> <F> 1 2 2 </F>
<F> 0 2 3 </F> <F> 1 2 3 </F> <F> 0 2 4 </F> <F> 1 2 4 </F> <F> 0 2 5 </F> <F> 1 2 5 </F>
<F> 0 3 5 </F> <F> 1 3 5 </F> <F> 0 3 4 </F> <F> 1 3 4 </F> <F> 0 3 3 </F> <F> 1 3 3 </F>
<F> 0 3 2 </F> <F> 1 3 2 </F> <F> 0 3 1 </F> <F> 1 3 1 </F> <F> 0 3 0 </F> <F> 1 3 0 </F>
<F> 0 4 0 </F> <F> 1 4 0 </F> <F> 0 4 1 </F> <F> 1 4 1 </F> <F> 0 4 2 </F> <F> 1 4 2 </F>
<F> 0 4 3 </F> <F> 1 4 3 </F> <F> 0 4 4 </F> <F> 1 4 4 </F> <F> 0 4 5 </F> <F> 1 4 5 </F>
</Path>
```

This path alternatively enumerates all fields of the RNN in Fig. 2, starting from the field $F_0[0,0]$ (the input field) and terminating with the field $F_1[4,5]$ (the output field). When excitation reaches the output field, the ultimate potentials of its neurons are converted in OUI (output unitary images), similar to IUI, and transmitted to the RNN environment as the result of RNN functioning. As mentioned before, adjacent fields in a path are the closest: the distance between their two neurons with the same coordinates is minimal irrespectively of the actual distance between them in a 3D space. Thus, different paths impact the RNN behavior differently.

Similarly, a scenario of input signals is specified with the pair of words `<Images>` and `</Images>` which frame the scenario, while words `<I>` and `</I>` frame each separate input matrix within it. The word `<I>` is preceded by two integers: the moment of the system time when this image enters the RNN and the path number. Binary representations of the rows of the given image reside between `<I>` and `</I>` (leading zeros may be omitted). The number of such elements should be equal to m. E.g., the following scenario specifies that 10 images:

$$\begin{bmatrix}0001100\\0010110\\0100011\\1111111\\1000011\\1000011\end{bmatrix},\begin{bmatrix}1111000\\1100100\\1111100\\1100110\\1100011\\1111110\end{bmatrix},\begin{bmatrix}1111111\\1001100\\0001100\\0001100\\0001100\\0001100\end{bmatrix},\begin{bmatrix}0111110\\1100011\\1100011\\1100011\\1100011\\0111110\end{bmatrix},\begin{bmatrix}1100011\\1010111\\1001011\\1001011\\1000011\\1000011\end{bmatrix},\begin{bmatrix}0001100\\0010110\\0100011\\1111111\\1000011\\1000011\end{bmatrix},\begin{bmatrix}1111111\\1001100\\0001100\\0001100\\0001100\\0001100\end{bmatrix},\begin{bmatrix}1111110\\1100011\\1100011\\1111110\\1100000\\1100000\end{bmatrix},\begin{bmatrix}1100011\\1100011\\1100011\\1111111\\1100011\\1100011\end{bmatrix},\begin{bmatrix}0111110\\1100011\\1100000\\1100000\\1100011\\0111110\end{bmatrix}$$

enter the input field $F_0[0,0]$ of the path number 0 which starts in the upmost left corner of the layer L_0 at the time moments 1, 72, 148, 232, 321, 414, 515, 625, 745, and 868 in this order.

```
<Images>
1   0 <I> 0001100 0010110 0100011 1111111 1000011 1000011 </I>
72  0 <I> 1111000 1100100 1111100 1100110 1100011 1111110 </I>
148 0 <I> 1111111 1001100 0001100 0001100 0001100 0001100 </I>
232 0 <I> 0111110 1100011 1100011 1100011 1100011 0111110 </I>
321 0 <I> 1100011 0010110 1001011 1001011 1000011 1000011 </I>
414 0 <I> 0001100 0010110 0100011 1111111 1000011 1000011 </I>
```

```
515 0 <I> 1111111 1001100 0001100 0001100 0001100 0001100 </I>
625 0 <I> 1111110 1100011 1100011 1111110 1100000 1100000 </I>
745 0 <I> 1100011 1100011 1100011 1111111 1100011 1100011 </I>
868 0 <I> 0111110 1100011 1100000 1100000 1100011 0111110 </I>
</Images>
```

Running an experiment with the specified RNN parameters and scenario is initiated by the command `Simulate`.

Output Data. AIM produces three outputs: the log, an output file with OUIs, and an auxiliary file with additional data used for debugging and further analysis of the AIM behavior. However, other outputs may be easily added. All these outputs are plain texts and may be further processed by other tools; e.g., MS Excel or others. Fig. 3 visualizes how the number of excited neurons changes with time. The diagram was built in Excel directly from the log data.

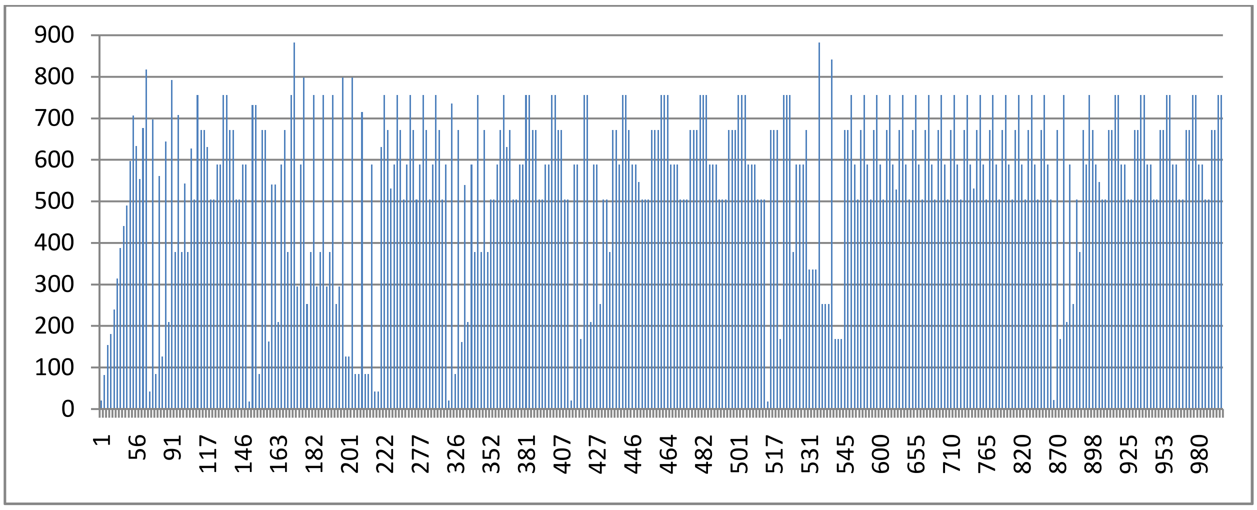


Fig. 3. The number of excited neurons vs. time in a sample RNN

Specialized tools and libraries allow for dynamic animation of the RNN functioning. A surface of a 3D manifold represents current potentials of neurons and the neuron is state rendered with the color – blue for unexcited and red for excited ones. Fig. 4 presents such an image for a frame #8 which corresponds to the *time*=41 of the above mentioned example. The left-hand chart represents the upper layer #0 and the right-hand one represents the bottom layer #1.

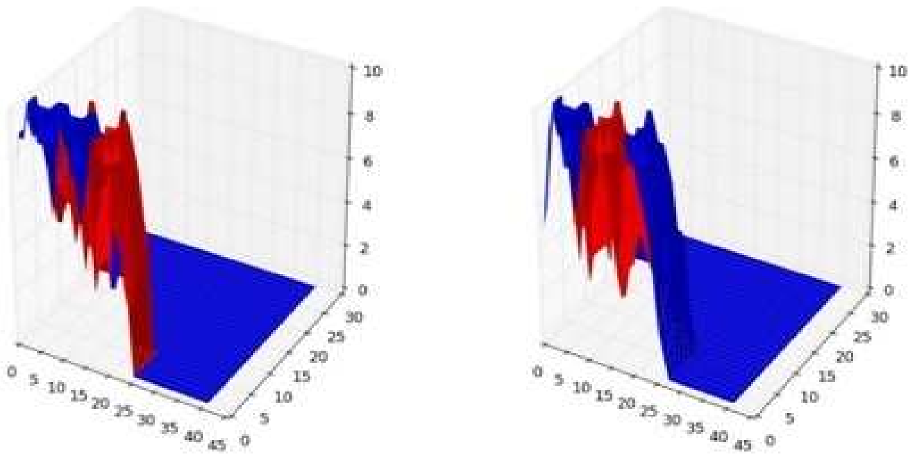


Fig. 4. A snapshot of the RNN in progress at a particular moment

The respective mapping of the output data was obtained from data generated by the AIM simulator in a plain text format with the NumPy package [21] and the Matplotlib library [22] (by the courtesy of my colleague Dr.Sergey Podkorytov). These packages allow to form a video file from a series of such images in the mp4 format as well.

The Simulator. The AIM simulator reuses the RTMT architecture [18] with a different set of events. Its overall workflow is presented in Fig. 5.

Four types of events are considered in this model: 1/2) Receive/Send an image from/to the external environment; 3) propagate excitement from excited neurons of the given field to all unexcited neurons connected with them through synapses, and excite these neurons if the accumulated potential is high enough; and 4) unexcite all excited neurons of the given field.

As already mentioned above, the simulation process is controlled by a list of events `EventList` ordered w.r.t. their time stamps: 0, t_1, t_2, t_3,... and assembled into same-time event groups. The main simulation loop consists in advancing the system time counter to the nearest time stamp of the events in this list and processing the events of this group one after another, which may produce new events with the same or later time stamp.

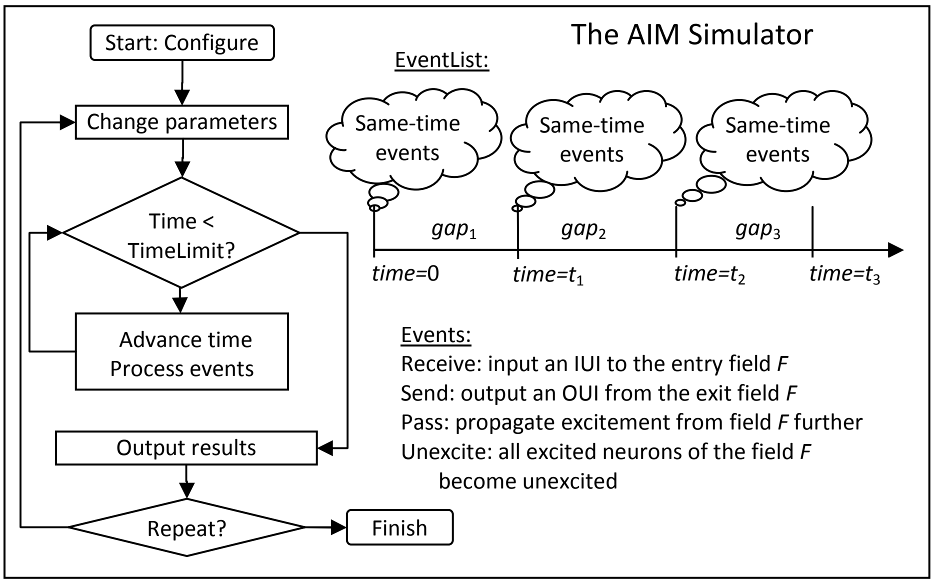


Fig. 5. The overall workflow of the AIM simulator

The main loop reiterates until the list of events becomes exhausted or the overall time limit for system time is reached, or a fatal error was encountered.

Conclusions. The described simulator is a relatively simple but powerful tool for studying various RNN structures and various combinations of pulse propagation paths and RNN parameters under various circumstances. It allows to easily specify the respective user interfaces and interoperate with other powerful tools for elaborated representing and visualization of experimental results.

The described programming solution based on the event list structure turned out to be both effective and efficient, so it's worth for reuse in other applications or subject domains. The described simple system log allows for relatively easy detecting violations and errors in the simulation process and helps in debugging of the simulator and its input data.

The simulator demonstrated acceptable performance on a regular laptop with relatively small RNNs of up to one million of synapses. Its performance can be even further improved with the assembler option offered by most Forth systems, which allows for direct programming of performance critical words in assembler, thus ensuring the most efficient realization of such critical data structures and respective processing means.

The application area of the AIM program is R&D of associative memory mechanisms for development of the respective hardware with improved characteristics and reliability. Future work will be focused on developing a variety of interfaces and typical solutions, as well as accumulating and analyzing the results of experiments with various RNN structures and data.

References.

1. Haykin, S.S., et al. Neural Networks and Learning Machines, vol. 3., Upper Saddle River: Pearson Education, (2009)
2. Brette, R., et al. Simulation of Networks of Spiking Neurons: a Review of Tools and Strategies. Journal of Computational Neuroscience, 23.3, pp. 349-398 (2007)
3. Zell, A., et al. SNNS (Stuttgart Neural Network Simulator). Neural Network Simulation Environments, Springer US, pp. 165-18 (1994)
4. Demuth, H., Beale, M. Neural Network Toolbox for Use with MATLAB, The MathWorks, Natick, MA (1998)
5. Elman J.L. Learning and Development in Neural Networks: the Importance of Starting Small. Cognition, 48, pp. 71-99, (1993)
6. Goodman D., Brette R. Brian: a simulator for spiking neural networks in Python. Frontiers in Neuroinformatic, vol. 2, article 5, web: http://www.frontiersin.org (2008)
7. Davison A. et al. PyNN: a common interface for neuronal network simulators. Frontiers in Neuroinformatic, vol. 2, art. 11, web: http://www.frontiersin.org (2009)
8. Osipov V.Yu. Associative Intellectual Machine with Three Signaling Systems. Informatsionno-upravliaiushchie sistemy (Information and Control Systems), vol. 5, pp.12-17, web: http://i-us.ru/en/article888 (in Russian) (2014)
9. Osipov V.Yu. Space-Time Structures of Recurrent Neural Networks with Controlled Synapses. Advances in Neural Networks – ISNN 2016. – Springer International Publishing, 2016, LNCS 9719, pp.177-184, web: http://link.springer.com/chapter/10.1007/978-3-319-40663-3_21 (2016)
10. Jo, Sung Hyun, et al. Nanoscale Memristor Device as Synapse in Neuromorphic Systems. Nano Letters 10.4 (2010): 1297-1301, web: http://web.eecs.umich.edu/~mazum/PAPERS-MAZUM/92_MemristorSynapse.pdf (2010)
11. Frenger P. A Forth-Based Hybrid Neuron for Neural Nets. Proc. of the Second and Third Annual Workshops on Forth. – ACM, 1991. – pp.99-102, web: http://dl.acm.org/citation.cfm?id=260009 (1991)
12. Dress W.B. Alternative Knowledge Acquisition: Developing A Pulse-Coded Neural Network. – Journal of Forth Application and Research, 1989, volume 5, number 3, pp.397-406, web: http://soton.mpeforth.com/flag/jfar/vol5/no3/article7.pdf (1989)
13. Hendrix M. Forth: Neural Net Programs, web: http://home.iae.nl/users/mhx/programs.html (1993)
14. Baranov S.N. A Practical Simulator of Associative Intellectual Machine. Advances in Neural Networks – ISNN 2016. – Springer International Publishing, 2016, LNCS 9719, pp.185-195, web: http://link.springer.com/chapter/10.1007/978-3-319-40663-3_22 (2016)
15. Forth 200x, web: http://www.forth200x.org/forth200x.html (2016)
16. VFX Forth for Windows. User manual. Manual revision 4.70, 19 August 2014. – Southampton: MPE Ltd, 2014. – 429 p., web: http://www.mpeforth.com/ (2014)
17. gForth. Free Software Foundation, Inc., web: https://www.gnu.org/software/gforth/ (2016)
18. Baranov S.N. A Forth-Simulator of Real-Time Multi-Task Applications. 31th EuroForth Conference, October 2-4, 2015, Pratts Hotel, Bath, England, pp.33-40, web: www.complang.tuwien.ac.at/anton/euroforth/ef15/papers/proceedings.pdf (2015)
19. Osipov V. Yu. Erase Outdated Information in Associative Intelligent Systems. Mekhatronika, avtomatizatsiya, upravleniye (Mechatronics, Automation, Control), vol. 3, pp.16-20, web: http://novtex.ru/mech/mech2012/annot03.html (in Russian) (2012)
20. Request for Discussion: Quotations, web: http://www.forth200x.org/quotations.txt (2016)
21. NumPy – Package for Scientific Computing with Python, web: http://www.numpy.org/ (2016)
22. Matplotib – Python 2D Plotting Library, web: http://matplotlib.org/ (2016)

EuroForth 2016

Tunnel Vision

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

An industrial control system written in Forth is described, using multiple techniques introduced in previous Euroforth presentations.

1. Introduction

Commercial laundering is possibly the largest industry to be almost invisible. Have any of you ever considered how clean sheets miraculously appear on your hotel bed every morning? There are a lot of hotel beds, even in a small place like Reichenau. To wash all those sheets, one needs a serious washing machine. The machine in your back kitchen perhaps takes a 5kg load, every 90 minutes. A *real* washing machine takes a 100kg load, every 90 seconds. If you'd like to buy a new one, you won't get much change from €1m.

2. Economics

Buying a new washing machine is a major investment. To make a return on that investment, you need to keep the machine in continuous operation over a period of many years. The machines are therefore made of high quality materials, and arranged so that parts that wear out, such as bearings, can be easily replaced. The machines use large quantities of energy, water and chemicals, which means that a sophisticated control system is needed to minimise costs and maximise profitability.

The mechanical life of a high quality machine can be 20 years or more. But the rapid improvements in automation technology in the past ten years, means that control systems of older machines have become obsolete, and spare parts are very hard to source. This has created a lively market for automation system upgrades.
Option a) Buy a new machine for up to €1,000,000
Option b) Keep your old machine and buy an automation upgrade for €30,000
This is a compelling economic argument.

3. Machine description

A typical CBW - this is a 50kg 14 compartment model, capable of washing up to 35 tonnes per day

Most modern high-throughput washing machines are Continuous Batch Washers (CBWs) - commonly known as "Tunnel" washers. They essentially consist of a long cylindrical tube containing a mechanism similar in action to an Archimidean screw. The screw oscillates for a period, to give the washing action, then performs a complete rotation to transfer the wash load from one section to another. Various valves enter the cylinder in places, to supply and drain water, introduce chemicals, and inject steam for heating.

An array of pumps move water between various sections, and a high proportion of water is recycled. There may be water level sensors, temperature sensors and possibly pH sensors.

Soiled washing is delivered to the machine from an overhead rail conveyor system. Clean washing is passed to a press or centrifuge, then to an array a large tumble dryers.

The normal operation of the entire installation is completely automated.

4. Traditional automation solution

All the actuators and sensors are connected to a programmable logic controller (PLC). All machine control is carried out by the PLC, which is programmed in one of the IEC 61131-3 family of languages. A Human-Machine Interface (HMI) display is used to show information to the operator, and to enable wash program parameters to be entered. The HMI is typically programmed in a vendor-specific WYSIWYG environment in which display elements are directly linked to PLC registers.

5. Limitations of the traditional solution

All washing machines are different, according to original manufacturer, capacity, water flow diagram, sensor requirements, and inlet / outlet interfaces. Therefore, a certain amount of new software is required for every machine, and software productivity is therefore an essential consideration for an economical automation solution. A PLC is clearly necessary for the physical connection of sensors and actuators - but it has been demonstrated that programming in IEC 61131-3 is extremely inefficient when compared to a highly flexible language such as Forth. The HMI programming method is highly restrictive in both style and content.

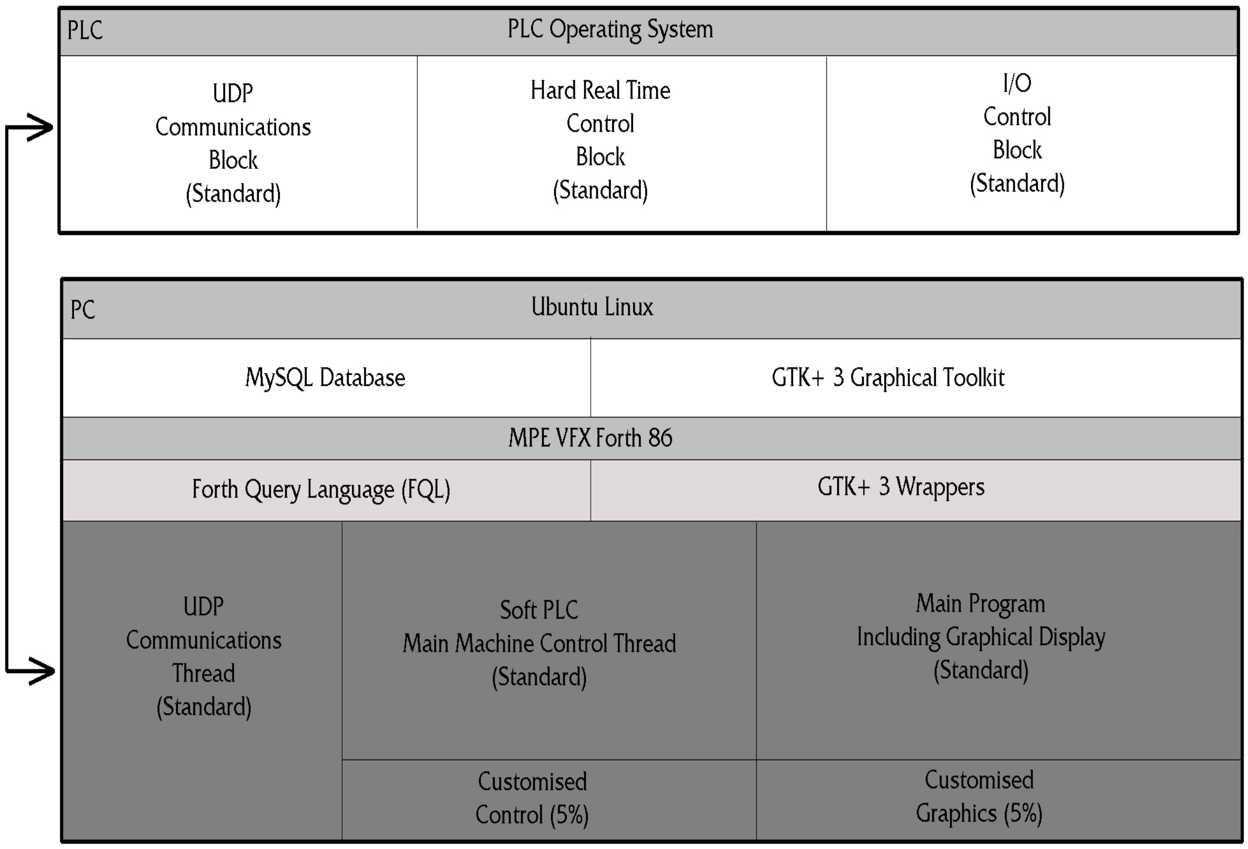
6. A better automation solution

From a software point of view, the only advantage of programming in the PLC, is that it is "hard" real time. But for an application like a tunnel washing machine, only a very small number of functions require hard real time, and these can be standardised. For all other functions, "soft" real time is adequate. This includes all the most complex functions such as selection and calculation of chemical proportions, and all functions that might need different treatment for different machines. All these functions can be handled much more easily in Forth.

Therefore, instead of using a vendor-specific HMI display unit, we opted for a functionally standard PC. Although the PCs we use are fairly standard from a programmer's perspective, they are in fact specialised industrial PCs, which are rugged, fanless, and have multiple connectivity options. A standard touchscreen is also used.

The actual machine control (except for the few critical hard real time functions) all takes place within a high priority thread within the same applications program as the main display. This technique has been proven over many years in a Windows environment. However, in the past few years, it has become increasingly difficult to structure reliable automation programs in Windows. Recent favourable experience with Linux (but for display only programs) prompted us to make a bold decision, to use Linux for the first time in a mission-critical automation environment.

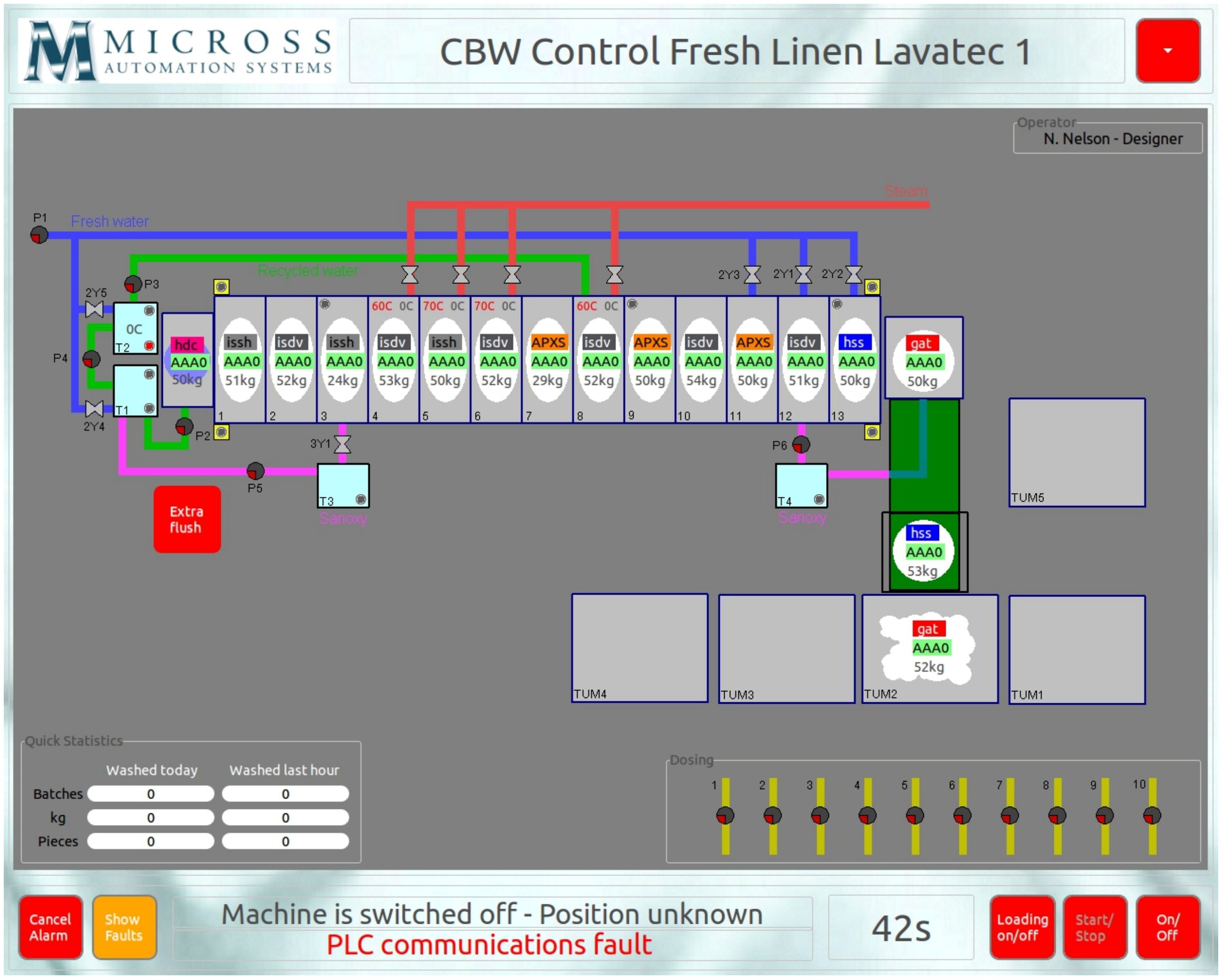
7. Program structure


PLC
PLC Operating System
UDP Communications Block (Standard)
Hard Real Time Control Block (Standard)
I/O Control Block (Standard)
PC
Ubuntu Linux
MySQL Database
GTK+ 3 Graphical Toolkit
MPE VFX Forth 86
Forth Query Language (FQL)
GTK+ 3 Wrappers
UDP Communications Thread (Standard)
Soft PLC Main Machine Control Thread (Standard)
Main Program Including Graphical Display (Standard)
Customised Control (5%)
Customised Graphics (5%)

8. How to program a CBW (it's simple!)

```
: (MAIN-LOOP)
  LEVELTIMERS          \ Service water level timers
  MACHINE              \ Overall machine control
  PUMP1                \ Control pump 1 (fresh water in)
  PUMP2                \ Control pump 2 (flush water into compartment 1)
  PUMP3                \ Control pump 3 (tank 2 to compartment 8)
  PUMP4                \ Control pump 4 (tank 2 to tank 1)
  PUMP5                \ Control pump 5 (tank 3 to tank 1)
  PUMP6                \ Control pump 6 (tank 4 to compartment 12)
  LINTMOTOR            \ Control lint motor
  WEIRBOXES            \ Control weir boxes
  REFRESH12            \ Control fresh water to tanks 1 & 2
  RINSE12              \ Control compartment 12 rinse water
  RINSE13              \ Control compartment 13 rinse water
  RINSE11              \ Control compartment 11 rinse water
  DOSING               \ Control dosing
  ROTATION             \ Control rotation
  LOADING              \ Control loading
  PRESS                \ Control press
  STEAM                \ Control steam
  ALARMCANCEL          \ Control alarm cancel
  TRACKING             \ Track loads through shuttle and tumblers
  RAILCOMS             \ Monitor rail comunications
; ASSIGN (MAIN-LOOP) TO-DO MAIN-LOOP
```

9. Result and conclusion



CBW Control Fresh Linen Lavatec 1

Operator: N. Nelson - Designer

Quick Statistics

	Washed today	Washed last hour
Batches	0	0
kg	0	0
Pieces	0	0

Machine is switched off - Position unknown
PLC communications fault

a) Soft real time control is not only feasible in Linux, it is actually much more stable than it is in Windows.
b) Moving the control elements from the PLC to the PC greatly simplifies the communications, removing the barrier between control and visualisation.
c) It is much quicker and more efficient to program the machine control in Forth than it is in IEC 61131-3.
d) The look and feel of the visualisation received much more favourable comments from the customer and the operators, when compared with standard HMIs.

References:
1. Industrial control languages: Forth vs. IEC61131
   N.J. Nelson, EuroForth 2000
2. Experiments in real time control in Windows using Forth
   N.J. Nelson, EuroForth 2004
3. The Nearly Invisible Database or ForthQL
   N.J. Nelson, EuroForth 2006

NJN
August 2016

The Halting Problem in Forth

Bill Stoddart

September 1, 2016

Abstract

Forth can be used to formulate a simplified but fully general statement of the halting problem and to formulate a short and simple proof.

Keywords: Forth, halting problem, proof

1 Introduction

In his 1936 paper "On Computable Numbers", Alan Turing formulated the idea of a Turing machine and its tape as a way of describing "effective procedure" and showed that there were some limitations in such machines. A slight variation in the limitations that Turing demonstrated gave us the "Halting Problem". This phrase may hafirst have appeared in the 1958 textbook *Computability and Unsolvability* by Martin Davis, but the problem is generally attributed to Turing due to its closeness to the material in his paper. It shows that no Turing machine can exist such that, if supplied with the description of another arbitrary Turing machine S and its data (tape) D, would be able to predict whether S would eventually come to a halt if activated on D. Given that we recognise Turing machines as representing computations in a general sense, it tells us that no program can be written which can take another program S and data D as input and reliably tell us whether S will halt when executed on data D.

In this paper we first formulate a slight simplification of the halting problem in Forth. We then discuss what the mathematical implications would be if a solution to the halting problem, in the form of some program H, *did exist*, showing that this would provide a effective procedure for demonstrating the truth of mathematical propositions.

We then give a Forth based proof of the halting problem, and set a variation of the problem as an exercise.

2 Describing the halting problem in Forth

When a Forth program is executed from the keyboard it either comes back with an "ok" response, or exhibits some pathological behaviour such as reporting an error, not responding because it is in a n infinite loop, or crashing the whole system. We classify the "ok" response as what we mean by "halting".

We specify a putative Forth program H by its stack effect:

$xt \rightarrow f$, f will be true if and only if execution of xt from the current state would halt.

Note that we do not talk about the application of a program to its data, but it is implicit that there is a stack where any data required by the execution of `xt` may be found.

Were H to exist, we could use it as follows:

$4\ 2\ '/\ H\ .\ \hookleftarrow -1$ *ok*
$4\ 0\ '/\ H\ .\ \hookleftarrow 0$ *ok*

3 Implications of the existence of H

Fermat's last theorem states that for any integer $n > 2$ there are no integers a, b, c such that:

$$a^n + b^n = c^n$$

Fermat died leaving a note in the margin of his notebook saying he had found a truly marvellous proof of his theorem, but this proof was never found. All subsequent attempts proof failed until 1995, when Andrew Wiles produced a proof 150 pages long.

However, with the aid of our program `H` we could have investigated Fermat's last theorem by providing a Forth program *FERMAT* which searches exhaustively for a counter example and halts when it finds one. Then we could have proved the theorem by the execution:

′ FERMAT H . ↲ 0 *ok*

This tells us the program *FERMAT* does not halt, implying that the search for a counter example will continue forever, in other words that no counter example exists and the theorem is therefore true.

In the same way we could explore any mathematical conjecture by writing a program to search exhaustively over the variables of the conjecture until a counter example is found. Then use *H* to determine if the program fails to halt, in which case there is no counter example, and the conjecture is proved.

4 A proof of the halting problem in Forth

Traditional proofs of the halting problem and friends rely on a diagonalisation argument - see §8 of Turing's paper. We will permit ourselves a more direct approach.

We assume a program *H* exists with stack effect $xt \rightarrow f$ where f will be true if execution of *xt* halts, and false otherwise.

We specify a program *IH* ($xt \rightarrow$) which inverts the halting behaviour of *xt*, i.e. it halts if execution of *xt* would fail to halt, and it fails to halt if execution of *xt* would halt. We can define this program by:

: *IH* ($xt \rightarrow$) *DUP H IF BEGIN AGAIN THEN* ;

We then consider whether *′ IH IH* will halt.

When we execute *′ IH IH* the invocation of *H* within *IH* finds *′ IH ′ IH* on the stack, so it will report whether *′ IH IH* will halt.

If we assume *H* returns true, reporting that *′ IH IH* will halt, then *IH* will enter a non-terminating loop, so we must discard this assumption.

If we assume *H* returns false, reporting that *′ IH IH* will not halt, the invocation of *H* in *IH* must have yielded false, which would yield immediate termination. Again we must discard this assumption.

So we are forced to reject our assumption that the program H exists.

5 A similar non-existence proof for the zero test program - exercise

Turing considered a the analysis of a slightly different machine, one supposed to tell whether a Turing machine with a given tape will ever output a specific symbol, say a zero, to that tape.

We adapt this to Forth by investigating whether a program Z could exist with this specification:

$xt \rightarrow f$, f will be true if an only if execution of xt leaves a zero at the top of the stack.

Exercise: Prove that no such program Z can exist.

6 Conclusion

To demonstrate the halting problem in Forth we assume the existence of a program H $xt \rightarrow f$, f is true if and only if execution of xt halts. We then use H to define a program:

: *IH* (*xt* $\rightarrow$) *DUP H IF BEGIN AGAIN* ;

and we show *IH* cannot exist by a reductio ad absurdum obtained from considering execution of *′ IH IH*.

This approach is very uncluttered, due to the minimalism of Forth, but also differs from other approaches to the halting problem in that it does not require formulation in terms of a program acting on given data - with our approach, using a stack, the presence of any data required for our arguments can be left implicit.

Request for comments
(appendix to workshop introduction)

Andrew Read
andrew81244@outlook.com
July 2016

An Axiomatic Approach to Forth

1. Introduction

Forth has traditionally been implemented by writing a certain number of code words in assembly language, out of which the remainder of the Forth dictionary is built up. Forth virtual machines may implement code words in C or another high level language. Forth processors may offer instructions that correspond directly to code words.

Traditionally there have been few constraints on code words. ANSI Forth defines the behaviour of high level Forth words and leave implementation details to the system designer. This approach has its benefits, but also leads to certain practical problems.

Firstly, mature Forth systems that are ANSI compliant may actually behave differently, especially when programmed at a “technical” level.

Secondly, implementers of new FORTH systems, virtual machines or Forth processors have to “make it up from scratch” every time. Whether the resulting Forth systems are truly ANSI compliant cannot be tested until after completion.

Thirdly, there is no straightforward way of porting mature Forth implementations to new targets. Each time new, machine specific, code words must be written and somehow tested before the porting of Forth itself can begin.

2. A conceptual viewpoint

A conceptual objection may also be made: Forth has been long proven to “work” as a programming language, but because the code words upon which the implementation of every Forth system depends are arbitrary, there is not a certain foundation to the language.

On the other hand, there is an opportunity here: Forth has no syntax so the behaviour of Forth words can be completely defined in terms of their effects, irrespective of the context in which they occur. So it should be possible to determine a completely deductive chain of logic from the most fundamental underlying elements of Forth through to ANSI Forth words, via a well-defined set of code words.

3. Objectives of the project

This project aims to take a deductive approach to the definition of Forth from conceptual underpinnings. There are four stages

(A) Elemental structures (“structures”)

Identify and document the elemental structures of Forth at a conceptual level.

“Structures” in this context means something akin to “physical entity”, or perhaps “mechanical entity”, rather than just data structures in the traditional sense. Hopefully the intended meaning may become clearer through the following discussion.

Some structures are explicit in Forth (e.g. the parameter stack), while others are implicit (e.g. the program counter, system memory, or the locus of arithmetic logic). Yet others may require more careful thought. For example, is the return stack a single elemental structure or is it actually the mapping of multiple conceptual elements (a LIFO store accessed with >R and R> and a subroutine return program counter store) to a single implementation entity? What kind of structure is the Forth dictionary itself?

The objective of this stage will be to “find” all of the elemental structures that underlie what we commonly understand as Forth, to properly separate them, and to describe them concisely and rigorously.

(B) Elemental operators (“operators”)

Identify the elemental operators which act on the structures and document them by stating their effects.

Again some elemental operators make themselves very evident and in fact are cognates with Forth words. For example, “+” is an operator that acts on the parameter stack, the locus of arithmetic logic, and on the parameter stack again.

Other operators are less obvious, for example is there an operator “BNE”, that acts on the program counter conditionally depending on the value held at the top of the parameter stack?

The objective of this stage will be to find all of the elemental operators that we believe comprise Forth and document them in the form of a table that shows their impact on the elemental structures.

This stage is likely to be highly iterative with the identification of elemental structures. For example, when we consider “BNE”, if it is an operator, does it imply there is also a structure that is the locus of logical comparison?

Referring to the title of this RFC, the elemental structures and operators might loosely be considered a set of “axioms” for Forth.

(C) Code words

The next stage of the project is to bring together the structures and the operators into a suitable set of Forth words from which a complete Forth implementation can be developed.

The code words serve as the abstraction layer to provide “Forth-like” access to the elemental structures and operators.

Some code words may map directly onto individual operators (perhaps “+” for example). Others code words will be combinations of operators, acting serially or in parallel.

The code words will need to take account that there may be differences between the data width of the Forth system (e.g. 32 bits) and the data width of the underlying structures (e.g. 16 bits or 8 bits). This project does not intend to prescribe any expected data width at either the structure or the Forth system level.

From a practical perspective, code words may be implemented in a machine-dependant manner in the language of the underlying system, as has always been the case. (That language might be C for a Forth virtual machine, assembly language, or the primitives of a Forth processor.) However, the implementation of the code words will no longer be arbitrary because (i) the set of code words will be explicitly defined and (ii) the function of each code word will be completely specified in term of the fundamental structures and operators.

This stage of the project is likely to be rather judgmental. The optimally chosen set of code words is unlikely to be the minimal set (for example there is actually no need of “+”, provided we have “0” and “-”, but is this a sensible economy?).

A staging post of this phase in the project is likely to be the articulation of a set of policies or guidelines for deciding which words should be defined in terms of the fundamental structures and operators (the code words) and which in terms of other Forth words (the remainder of the dictionary).

(D) Forth implementation

Finally, the code words can be leveraged to develop an ANSI Forth implementation.

4. Working approach

(A) Relative weighting of effort

I anticipate that the first two stages, finding the elemental structures and operators likely represents 60% of the effort that would be required. Although a first draft can no doubt be drawn up quickly, consideration of subtle points and generally iterating and polishing the thinking will take much more time. The third stage, the code worlds is perhaps 25% of the effort, and much of that spent on consideration of words at the boundary between “the code” and the rest of the dictionary. The final ANSI Forth implementation, whilst probably the greatest number of written lines, may only be 15% of the effort if the Forth implementation is limited to the CORE wordset and a few others, and good advantage is taken of readily available prior work.

(B) Verification

It will be necessary to verify the results of each stage. A number of possible approaches exist and the actual verification approach adopted will depend on the preference of the project participants.

Firstly, there is the possibility of using some sort of “logical calculus” to prove results in a manner similar to pure mathematics. Although this approach has been adopted before, particularly in relation to verifying stack operations, experience suggests that such an approach is likely to prove unwieldy in practice and that the difficulty of developing the “calculus” in the first place will probably exceed its benefit.

Secondly, there is the use of informed debate to discuss critical decisions, not just in terms of functionality but also from the perspectives of desirable aesthetics and symmetry. We can call this the “philosophical” approach. *Hoc tam ars quam scientia est*. Examination from an aesthetic perspective will be invaluable to for an elegant result.

Thirdly, there is the mechanical approach. By explicitly simulating (perhaps with pencil and paper at first) the structures, operators and code words it should be possible to verify the effect of any sequence of operations. The mechanical approach needs to be alert to “corner cases”, and here again there is a role for informed debate as a source of suitable challenge.

Finally, and this is really an extension of the mechanical approach, a working Forth system built on these foundations will help to convince that the foundations are satisfactory.

5. Uses and benefits

It is intended that the four components that will be developed in this project (the structures, operator, code words and ANSI Forth implementation) may serve as an “axiomatic” reference model that enhances and clarifies the Forth language. It is not intended that they should be advocated as “standard”, or that they should proscribe other approaches. If the reference model is intellectually appealing and helpful in itself, that will be justification enough for the effort expended.

The root of my own interest in this project is my experience of developing an instruction set and Forth system for the N.I.G.E. Machine. In the last few years I have become interested in how Forth constructs can be visualized as structures and then taken from software into hardware. This approach has allowed exception handling and multitasking to be implemented as atomic machine language instructions in the N.I.G.E. Machine.

Ulli Hoffmann mentioned to me some time ago how a Forth meta-compiler could be used to “make seamless” the Forth held in RAM and that included from source files on SD-card. I now wish to extend the Forth system software and before doing so it would be expedient to migrate the N.I.G.E. Machine to a meta-compiled system. At the same time, I would like to re-examine and potentially reconfigure the N.I.G.E. Machine instruction set. Both of these aims will be better accomplished in the light of a conceptually rigorous approach to the fundamental structure of Forth. Hence my wish for a reference model with axiomatic foundations.

I believe the reference model could also be interesting to anyone working with Forth virtual machines, since there is really very little difference between a Forth processor in hardware and a Forth virtual machine in software.

The reference model might be helpful to anyone who wishes to use Forth on the multitude of new microprocessor-based development boards since consistent system behaviour will be assured. In addition, perennial practical difficulties such as efficient Forth file transfer can potentially be addressed at a low level by defining interfaces at the level of elemental Forth structures and building suitable operators for their handling deep into the language.

Planet Holonforth

Wolf Wejgaard
EuroForth 2016

I described the first version of Holon in the paper "Not Screens nor Files but Words" at EuroFORML 1989 in the village Forth near Nuremberg. Holon was warmly received, and I naively expected the concept to spread in the community and spawn an evolution of similar systems. That did not happen. Instead, Holon began its long, exciting non-standard voyage around the Forth galaxy.

Please join me for a visit to planet Holonforth and a look at the results of a little refactoring of Moore's classical Forth system.

The main change concerned the rôles of editor and compiler regarding the dictionary. In Holon, the editor creates the dictionary, whereas the compiler concentrates on compiling and optimizing and merely adds the code pointer.

In Holon86, a Smalltalk-inspired fully integrated Forth IDE, the source is handled in a structured browser, with immediate access to and change of every word in the target program, including its code. The target resides in a separate code space independent of the development system and is controlled by an umbilical connection.

As a particularly useful feature, the permanently available dictionary allows selective loading of only the code that is used in the target. Which, by the way, lets you use Holon86 as a Literate Programming system with intrinsic TANGLE.

HolonS is a multi-platform source code management system that uses a real database to implement the Holon dictionary structure and browser. The source is presented in a book structure of chapters, sections, and unit pages. The chapters are copied to external files, and the files are updated with every change in the browser. Thus you have the source stored in a database - with all the pleasant features that I want to illustrate - yet constantly available in source files for code production.

For a closer inspection of planet Holonforth, see www.holonforth.com.

Sections

M. Anton Ertl[*]
TU Wien

Abstract

A section is a contiguous region of memory, to which data or code can be appended (like the Forth dictionary). Assembly languages and linkers have supported multiple sections for a long time. This paper describes the benefits of supporting multiple sections in Forth, interfaces and implementation techniques.

1 Introduction

A section is a contiguous memory area, to which new data can be appended at the end; the Forth dictionary is a section. Assemblers and linkers have supported multiple sections or segments for many decades [Lev00]. In contrast, Forth traditionally has had only one section; some systems have had separated headers (another section), and cross-compilers have uninitialized memory for `buffer:`, but by and large, Forth systems have made do with just one section: the dictionary. With multiple sections, each section has it's own start, dictionary pointer (what `here` reads), and end (used in `unused`, but otherwise not used much).

This paper presents various uses of sections and why they are better than the current workarounds (Section 2), presents a programming interface (Section 3), and discusses various implementation approaches (Section 4).

2 Uses

2.1 Nested structures

You often build one structure A in memory, and in the middle of that, have to build some out-of-line part B, and afterwards continue building A. If you have two sections, that is easy: you put A in one section, and B in the other section. In Forth, you traditionally use one of the workarounds:

- You select a representation for A that does not require contiguity.
- You put B in `allocate`d memory. Unfortunately, that usually means that B does not survive a `savesystem`, and it's also cumbersome if B is a growable structure.

A particular instance of this problem is when A is a colon definition under construction, and B is the data for a string or floating-point literal. Forth compilers traditionally work around this by not requiring contiguity.

A typical solution is to call a word such as `(s")` or `flit`, and follow that with the inline data. These words get the return address from the return stack, use that to push the relevant data on the data/FP stack, then increment the return address to skip over the data, and then either return to the changed return address or jump to it. Both ways are very expensive on modern CPUs, because they cause mispredictions from the hardware return stack[1]: If the changed address is returned from, the return incurs a branch misprediction (about 20 cycles on a modern Intel or AMD CPU); if the changed address is jumped to, the jump has a chance to predict correctly, but all outer returns will mispredict once (at about 20 cycles per misprediction).

A faster approach is to jump across the data, and then let some code push the data on the data/FP stack. This does not cause significant mispredictions, but the code is bigger (jump plus inlined literal code).

Finally, if you put the data elsewhere (i.e., a different section), you get fewer mispredictions, and you save the space for the jump around the data.

As an example of the benefit of putting the data out-of-line, consider the following micro-benchmark:

```
\ inline variant
: foo1 123e f+ ;

\ out-of-line simulation
123e fconstant x
: foo2 x f+ ;

defer foo
: bench 0e 100000000 0 do foo loop f. cr ;
```

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

[1]The hardware return stack is not the Forth return stack; it is a hardware branch predictor that predicts that returns will return right behind the corresponding calls).

With VFX[2] 4.71 on a Core i3-3227U (Ivy Bridge), the `foo1` version takes 48 cycles, 11 instructions and 1 branch misprediction per iteration, while the `foo2` version takes 6.5 cycles, 7 instructions, and 0 branch mispredictions per iteration. If VFX would put the floating-point number in `foo1` in a separate section instead of inline, it could achieve the same performance as `foo2`.

Quotations are another case of having to build something else in the middle of a colon definition; in this case the "something else" is a colon definition itself. Again, the usual implementation is to jump around it (used in, e.g., Gforth), and putting the quotation in a separate section can save that overhead. In this case, however, two sections are not sufficient, as quotations can be nested arbitrarily deeply, so you need a whole stack of sections.

Locals are another case where you have to build some additional stuff (in this case, headers) in the middle of a colon definition; the headers are no longer needed at the end of the colon definition and their space can be reclaimed, so the usual inline-and-skip approach is particularly inefficient here. Locals in Gforth were developed before sections, and the code for dealing with that problem is complicated; we foresee it becoming much simpler once we take advantage of sections, but we have not made these changes yet.

One way of implementing recognizers is to create a temporary word for each recognized string, then treat the temporary word like an ordinary word (i.e., `execute` or `compile,` it), and finally, delete the temporary if no longer needed [Ert16]. With sections, this is relatively straightforward to implement (especially the case when you cannot delete the "temporary" and have to make it permanent).

2.2 Separate code and data

Most Forth systems still put code and data in the dictionary in an interleaved way. Since the Pentium (1993) and its separate instruction and data caches, this interleaving has been a performance problem on Intel and AMD CPUs (e.g.,). Forth systems have tried to mitigate this problem by at least not putting code and data in the same cache line (by inserting appropriate padding); e.g., VFX aligns data to 32-byte boundaries, but apparently 64-byte alignment is necessary on recent Intel CPUs to achieve the desired effect. And in some cases an important padding is missing, resulting in 350–500 cycles per iteration in VFX and SwiftForth:

```
0 value x
: foo 10000000 0 do 1 x +! loop ;
here to x 0 ,
foo
```

[2]I use VFX for the performance results in this paper, because it is a high-performance system, where one would expect good performance also for the micro-benchmarks I present.

With sections, the data can just stay in the ordinary dictionary section, and the code can have a separate section (or a stack of them, for quotations), thus separating code and data for good. Moreover, systems can get rid of all the padding they insert at the moment to work around this problem.

2.3 Further uses

The uses above are not an exhaustive list. When I presented sections to other Gforth developers, they came up with uses I had not thought of during development (e.g., simplifying the locals implementation).

3 Progamming interface

In the following, "switching a section" means that the dictionary pointer (what `here` reports, and where `allot` allocates) is now the dictionary pointer of the switched-to section.

The words for working with sections are:

`.sections ( -- )`
: display all sections

`next-section ( -- )`
: switch the current section to the next section in the stack, creating it if necessary

`previous-section ( -- )`
: switch the current section to previous section (the next section still exists afterwards).

`extra-section ( size "name" -- )`
: create a named section stack *name*.
name execution: `( xt -- )`
switch the current section to the first section of *name* if there is no outer call to *name*, or the next section if there is; execute *xt*, and switch the current section back on leaving *name*.

For multi-tasking, the dictionary and the named section stacks should have per-task instances that are instantiated on-demand.

Currently the section implementation in Gforth only supports the dictionary as a section stack, named sections without stack, and no proper handling of per-task section stacks, yet.

4 Implementation

The implementation of sections for use within a session is pretty straightforward: Just a data structure with start, end, and section-dp per section, and ways to manage named sections and a stack of sections.

Things get more interesting when it comes to implementing `savesystem`. There are two basic options:

- Keep all the sections, and preserve them into the next session.
- Collapse all the sections into one (the dictionary), and start the next session with just the dictionary, and with empty named sections.

The current implementation in Gforth takes the collapsing approach. One advantage is that the loader (which does not know about sections) does not need to be changed.

Traditionally, Gforth creates a relocatable image by running Gforth twice and doing the same things, and finally saving one non-relocatable image per run; the non-relocatable images are for different addresses, and by comparing them, we can tell if a cell is an address (then they differ by the difference in image start addresses), or something else (then they do not differ); if they differ, but by a different amount (e.g., because the cell contains the address of an `allocate`d piece of memory), the process outputs a warning.

With sections, this process had to be enhanced as follows: When saving an image, first the dictionary is written, then the other sections, and sections meta-data last. The sections meta-data contains the length and the original start address of each section, and also allows us to determine where in the non-relocatable image the sections are. If two cells differ, the comparison program looks for each image, whether the value of the cell, interpreted as address points into one of the sections, and computes the offset into the collapsed image from that; if, for both images, this gives the same offset o, then the cell is a relocatable address, with value *image-start*+o, and that's what the comparator stores in the relocatable image. I.e., in the relocatable image, the original section structure is no longer present.

Of course, that is not the only option. E.g., a system without relocatable images could just save each section as ELF or COFF section with a fixed virtual start address. The OS loader would then load all the sections where they belong (untested).

5 Conclusion

Supporting multiple sections in a Forth system provides a number of benefits. Forth systems have used workarounds to deal with the absence of sections, but these workarounds have a cost both in complexity and sometimes also in performance that can be eliminated by adding sections.

The interface for working with sections is simple, consisting of just a few words.

The implementation is not that complex, either. Dealing with sections across `savesystem` does take some additional effort, however.

References

[Ert16] M. Anton Ertl. Recognizers: Arguments and design decisions. In *32nd EuroForth Conference*, 2016.

[Lev00] Levine. *Linkers and Loaders.* Morgan Kaufmann, San Francisco, 2000.

Recognizers: Arguments and Design Decisions

M. Anton Ertl[*]
TU Wien

Abstract

The Forth text interpreter processes words and numbers. Currently the set of words can be extended by programmers, but not the recognized numbers. User-defined recognizers allow to extend the number-recognizer part, too. This paper shows the benefits of recognizers and discusses counterarguments. It also discusses several design decisions: Whether to define temporary words, or a set of interpretation, compilation, and postponing actions; and whether to hook the recognizers inside `find` or in the text interpreter.

1 Introduction

A strength of Forth is its extensibility. You can define new words to build an application-specific language, and then program in that language (or at least that's a frequently-told tale). However, the text interpreter conists of two parts: dealing with dictionary words and dealing with numbers; and while the former is extensible, the latter is not (in standard programs).

A recognizer tries to recognize a class of strings (e.g., numbers), and, if successful, provides the necessary information for text-interpreting it in the recognized sense; e.g., push the value of the number during interpretation or (for compilation) at run-time.

In this paper, we first look at the benefits of introducing recognizers (Section 2), then discuss some counterarguments (Section 3. We also look at two design decisions: Whether to let the recognizers define temporary words or or a set of interpretation, compilation, and postponing actions (Section 4), and where recognizers should hook in (Section 5). Finally, we look at the history of recognizers (Section 7).

2 Benefits and Uses

This section describes some benefits of implementing recognizers, in particular some uses.

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

2.1 Factoring of numbers

Gforth's (integer) number parsing has been a horrible mess. A long time ago I tried to refactor it to be less horrible, but the result was not much better; I particularly dislike the words with variable stack effects due to the words handling both single-cell and double-cell numbers. Other Forth systems have similar horrors in this area. This failure is probably due to my shying away from refactoring the text interpreter itself.

Recognizers provide an easy way to solve the variable stack-effect problem: Have one recognizer for single-cell numbers and one for double-cell numbers. Of course this kind of factoring is also possible without support for user-defined recognizers, but the implementation difference to also supporting user-defined recognizers would be small.

That being said, the current implementation in Gforth (by Bernd Paysan) uses one number recognizer that calls the not-much-better-factored words, but now that is easy to change.

2.2 Floating-point numbers

Standard floating-point numbers require a recognizer for the FP numbers. Most Forth systems initially just support the (integer) number recognizer, and add the FP recognizer at a later point in time (sometimes only after user intervention); they typically use a hook for this that is used for this particular purpose. User-defined recognizers are a generalization of this principle.

2.3 Other literals

SwiftForth supports various non-standard ways to write doubles, such as 2016-09-07, 9/7/2016, 7.9.2016, which is supposedly good for writing dates or telephone numbers, but, as you can see from these examples, where we get three different doubles for the same date, the technique has significant limitations. A full-blown recognizer for dates (or three, one for each date syntax) will interpret the correct fields as year, month, and day, depening on the separator character, and also perform the conversion into an appropriate format.

2.4 Parsing words

In the absence of user-defined recognizers, people have written parsing words as a workaround, e.g.

```
char a   \ made unnecessary by 'a'
[char] a \ made unnecessary by 'a'
' word
['] word
s" some string"
```

Some people have even claimed that the parsing words are the Forth way to do such things, but the fact that Forth originally had a (non-extensible) number recognizer and no `s"` is counterevidence to this claim.

`s"` has the problem of surprising behaviour in some corner cases in many systems (e.g., due to `state`-smartness [Ert98] or because the replacement of `state`-smartness in some implementations does not cover all corner cases, either).

`'`/`[']` does not have corner-case problems, but it has the problem that the frequent case of cutting and pasting code between interpreted code and compiled code requires changing the code.

By adding a recognizer for `"some string"`, we get rid of the corner cases. One unusual thing here is that the recognized string can extend across a white space; while recognizers get a `parse-name`-parsed string as parameter, they can do their own parsing. However, that parsing is always done at text-interpret time, avoiding the problems of `state`-smartness etc.

Likewise, we can add a recognizer that recognizes `'word` and produces the xt of `word`, and it will work like `' word` interpretively, and like `['] word` while compiling.

2.5 `to`

Parsing words are not just used for literals; `to` is also a parsing word that has the same problems as `s"` (except that the corner cases are ambiguous conditions in the standard), and recognizers can also be used to replace `to`. Gforth has a recogizer that recognizes `->`x, and that is equivalent to `to` x (for locals, values etc.) and `is` x (for deferred words).

2.6 Dot-Notation Parser

One problem that Forth has had is the naming of structure and object fields. Frequent field names are `next`, `count`, `val`, `left`, `right`, but you normally don't want to define them more than once (and, in the case of `count`, the name is already taken).

One workaround has been to include the structure name in the name, e.g. `list-next`, but with inheritance of fields in object-oriented programming, this does not work so well: e.g., you would have `intlist-val` for the val field, but `intlist-next` would not be defined (instead, `list-next`). Some object-oriented systems (in particular, `objects.fs` [Ert97]) work around this problem by putting the fields in class-specific wordlists and changing the search order appropriately, but that restricts field access to only the current object (or at least the current class).

Therefore, a desired and missing feature in Forth has been to change the search order for one word only, in order to use the right field word (among a number of such words with the same name) without too much ado. One example of this desire is the Prelude concept [Mah98]; another is the dot-notation parser of ClassVfx OOP [MPE16, Section 29.11]. In the dot notation, if you have a type `Point` with field `x` and an instance `MyPoint` of type `Point`, you can access the field `x` either with `MyPoint.x` or with `MyPoint Point.x`.

A dot-notation parser can be implemented as a recognizer.

2.7 `Postpone`, `'`, and `[']`

In standard code, when you want to postpone a literal, you cannot do it directly, but have to find a workaround. E.g., write `5 postpone literal` instead of `postpone 5`; that's also true for literals produced by parsing words: instead of `postpone s" bla"`, you have to write `s" bla" postpone sliteral`.

And it's also true for other things you do with parsing words: when you want to postpone `to this`, you cannot do it directly, but have to define

```
: to-this to this ;
```

and then `postpone to-this`.[1]

Recognizers (as proposed in the RfD) support `postpone`ing recognized strings. One benefit is that this feature allows writing smaller and easier-to-read code, but the main benefit is that it closes the hole that made the workarounds necessary.

One recognizer approach (Section 4.1) also supports ticking recognized strings, so one could write `'15` instead of having to define

```
15 constant fifteen
```

first and then writing `'fifteen`.

3 Counterarguments

There have been quite a number of negative reactions to the proposal for user-defined recognizers. They are generally not technical, but nevertheless, let's examine some of the arguments.

[1] If you think that this is a contrived problem, you are wrong. This problem and this solution occur in `objects.fs` [Ert97].

3.1 Recognizers are not needed

As Section 2 shows, there is a need, e.g., adding a floating point recognizer or a dot-notation parser. Currently systems add these things through system-specific hooks; standardized recognizers would make it possible to do such things portably, and define and use them in portable libraries.

You may not see a need for all the features mentioned in Section 2, but if there is just one you need and that your vendor does not provide, the recognizers have paid off for you and for the vendor (who does not have to develop and maintain the feature himself).

3.2 People could misuse recognizers

People can already misuse a lot of things in Forth (e.g., `: 0 1 ;`), but Forth is not a nanny language. Forth design centers around responsible programmers, so while we will see some cases that most will consider misuses, it is much more important whether we will see some good uses. While not everyone will see all the uses mentioned above as good uses, as long as there are some that are considered good uses, it's a good reason to standardize recognizers. After a period of experimentation, there will be a rough consensus on what are good uses of recognizers and what aren't.

3.3 Recognizers are an attempt to make Forth more like C

C does not have a way to extend literals in a user-defined way, so, in a way, recognizers make Forth less like C. One could use recognizers to recognize some C lexical or small syntactic elements, and one can see the dot-notation recognizer as going in that direction. But note that people have been doing that even without standardized recognizers (in a non-standard way); also note that various people, including Chuck Moore, Julian Noble, and Andrew Haley have implemented infix notation or infix programming languages in Forth (something that recognizers do not facilitate), so if a Forth programmer is determined to go there, leaving recognizers away won't stop him.

3.4 Use parsing words! It's more Forth-like

Technically, parsing words cause problems: Either when trying to cut-and-paste between interpretation and compilation, such as `'`/`[']`, or in corner cases, such as `s"`. Recognizers avoid these problems and are therefore preferable. Indeed, one of the big advantages of recognizers is that they provide a long-term perspective for eliminating these problems.

As for *Forth-like*, recognizers for integers (singles and doubles) were part of Forth from the start. And simple ways to allow inputting dates and telephone numbers were part of the number recognizers of Forth, Inc. The only thing that was missing was the possibility to add user-defined recognizers; the use of parsing words, such as `s"`, is a workaround for this shortcoming, not a virtue.

4 Implementing a recognizer

This section looks at different implementation approaches for defining a recognizer. The outside interface of these implementation approaches can be made compatible, so these implementation techniques can both be used in the same system if desired.

4.1 Temporary words

A recognized string should behave like a word, so one way to implement a recognizer is to actually let it define a word when recognizing a string; e.g., for a string `123`, there is a temporary definition (not in any wordlist, name not important):

```
123 constant #123
```

and the xt of this word is `execute`d in interpretation state, or `compile,`d in compile state, like a regular word. It can also be `postpone`d or ticked.

However, one problem is that, in some of these uses, the word must be preserved and cannot be just temporary. After `execute`ing the word, we no longer need it and can reclaim the memory it uses; for `compile,`, it depends on how that is implemented. The classic threaded-code implementation (just `,`) would require that the word is preserved; however, many modern compilers have an intelligent `compile,` that compiles constants to literals, without reference to the compiled word, and therefore there is no need to preserve the word in that case. For `postpone` and ticking, the word must generally be preserved.

These words can be created in a separate section [Ert16], with the space reclaimed if the word does not need to be preserved.

There remains the problem of knowing whether the word needs to be preserved when the word is `compile,`d. The defining word could leave a flag in the defined word (or maybe a global flag) that indicates whether the word leaves a reference to itself when it is `compile,`d.

As a simple example, the core of a recognizer for unsigned single numbers looks as follows:

```
: (single-rec) ( c-addr u -- nt )
  0. 2swap >number 0= if \ it is a number
    2drop noname constant lastxt exit then
  2drop drop r:fail ;
```

`>Number` is used to check and convert the string, and if successful (0 unconverted characters), an unnamed constant is created and it's name token (nt) is returned; if unsuccessful, it returns a failure indicator (`r:fail`).

There is also a need to switch sections and perform other management tasks; these are always the same, so they are factored into a word `rec2-wrapper ( c-addr u xt -- nt )`, and the full recognizer is:

```
: single-recognizer ( c-addr u -- nt|0 )
    ['] (single-rec) rec2-wrapper ;
```

and this recognizer is added to the recognizer stack in the second position with:

```
get-recognizers
' single-recognizer -rot 1+ set-recognizers
```

4.2 RfD approach

The temporary word approach is relatively easy to understand and write, but it puts quite a number of demands on the system: The system needs to support another section[2], it must be able to create words in the middle of another word, possibly nameless or with their name coming from the stack, and ideally it should inline the code for that word when `compile,`ing it.

While most modern systems have these features or can add them without too much trouble, for standardization we may prefer an approach that puts fewer demands on systems and that does not require standardizing all the features that the temporary-word approach requires. The Recognizer RfD [Tru15] proposes such an approach.

Let's look at our example of a recognizer for unsigned single numbers again.

```
: comp-lit postpone literal ;

' noop     \ interpretation
' comp-lit \ compilation
' comp-lit \ part of postponing
recognizer: r:single

: single-rec ( c-addr u -- u2 rec | r:fail)
    0. 2swap >number 0= if \ it is a number
        2drop r:single exit then
    2drop drop r:fail ;
```

[2]If only interpretation and compilation of recognizers is supported, and compilation of the word created by the recognizer does not leave a reference to that word, then a buffer for one word instead of a full section is sufficient.

```
get-recognizers
' single-rec -rot 1+ set-recognizers
```

The recognizer `single-rec`[3] looks very similar to `(single-rec)` in the temporary word approach, but instead of putting the number in a newly-created word and returning that, the number is left on the stack and in addition `r:single` is pushed.

`R:single` is a word defined with `recognizer:` as a handle for the three actions, and the text interpreter (and `postpone`) access the actions they need through `r:single`:

- When interpreting, just leave u2 on the stack by `execute`ing `noop`.
- When compiling, compile u2 as a literal by `execute`ing `comp-lit`.
- When postponing, the final compile also happens with the compilation action, but one level later, so the compilation action `comp-lit` is `compile,`d. That requires that the data is transfered from the time when the recognizer runs to the time when the compilation action runs; to achieve that, the postponing part `comp-lit` is `execute`d before `compile,`ing the compilation action.

For literals, the usual pattern of the actions is `noop` for interpretation and the appropriate `literal` variant for both compilation and postponing.

The option for deviating from this pattern is useful for other applications of recognizers, such as replacing `to`.

The RfD does not specify the representation of `r:single`. In the current version of Gforth, this is implemented in a way that is compatible with the temporary-word approach: `r:single` returns the nt of a word, and you can get the interpretation action with `name>interpret`; you get the compilation action with `name>compile`; and there is also a field `>vtlit,` for the postpone-part action.

5 Recognizers where?

Recognizers can recognize words as well as numbers, so where should they hook in? There are at least three answers:

5.1 In `find`

(or its modern replacement, e.g., `find-name`). The benefits of this approach are:

[3]Matthias Trute would call it `rec:single`, but I find the presence of both "r:..." and "rec:..." confusing.

- It's a very natural fit for the temporary-word approach: `Find` returns a word, and so do temporary-word-creating recognizers.
- `Find` already has a way to add or remove things to be recognized: the search order. So recognizers could be added or removed from the search order, like wordlists, avoiding an additional mechanism. However existing code dealing with the search order may not be designed to deal with various recognizers on the search order.

The disadvantages are:

- Existing users of `find` (e.g., cross-compilers) would be surprised by `find` recognizing numbers, and the user's own number handling would be shadowed. That could be worked around by changing the search order appropriately when calling such users.
- For the RfD approach, the fit is not so great. In particular, we would now have a `find` that can generate additional values in addition to the xt (or nt for `find-name`) that it should produce. In many cases that is probably not a problem, but in some cases, it would be.
- Also, depending on the way words like `r:single` are implemented, and the actual `find` replacement that we want to hook in, there may be a mismatch; viewed differently, the implementation options for `recognizer:` would be restricted (but that is not necessarily a disadvantage).

5.2 In the text interpreter

The classical text interpreter first tries `find` and then tries numbers. In the current Gforth implementation, and in the text interpreter example given in the RfD, the `find` and number-handling parts are replaced by a recognizer-handling part. The search order search is performed by a word recognizer in the recognizer stack.

The advantages and disadvantages are the converse of those for the `find`-hooking approach:

Advantages: `find` users are unaffected, and the implementation of recognizers has fewer restrictions.

Disadvantages: We need the recognizer stack in addition to the search order (but don't need to worry about existing programs doing bad things to it).

5.3 As text interpreter hook

Instead of replacing the text interpreter the recognizer handling is added as a hook to the existing text interpreter. This would make the text interpreter more complex and reduce the options available to the programmer, so it offers only disadvantages, except that some consider it advantageous to reduce the options available to the programmer.

6 Other design decisions

There are some other design decisions where the right decision is not obvious, in particular: How to deal with recognizer stacks; whether to use `r:fail`, 0, or an exception as a failure indication. These and other design decisions are discussed at length in the RfD, which is recommended reading [Tru15].

7 History

System-specific hooks in the text interpreter have existed for a long time.

In 2003, Josh Fuller used *recognizer* in the sense used here, and proposed doing things like recognizing dates and (something like) dot notation by adding new recognizers[4]; the ensuing discussion points out that many systems have mechanisms for adding new recognizers. In that discussion, Jonah Thomas considered ways to deal with multiple recognizers, but not how to deal with interpretation, compilation, etc. in that context.

In 2007, in a discussion about number parsing hooks, I sketched some ideas about recognizers `news:<2007Aug4.093801@mips.complang.tuwien.ac.at>` `news:<2007Aug4.161609@mips.complang.tuwien.ac.at>`. I did not pursue this idea further at the time, but Matthias Trute picked it up and proposed using it for a dot-parser `<0dgjs6-h4e.ln1@wolf.stein.zeit>`, and implemented them in amForth [Tru11]. Subsequently, they were also implemented in Win32Forth, Bernd Paysan implemented them in Gforth [Pay12a, Pay12b], and Matthias Trute made a Forth 200x RfD [Tru15] proposing standardization.

8 Conclusion

User-defined recognizers generalize the Forth number recognizer and various system-specific hooks in the text interpreter. They allow to replace parsing words and their problems (e.g., `state`-smartness), write a dot-notation parser, and have other benefits. While a number of people have argued against user-defined recognizers, I have not seen a technical argument against them yet.

For implementing a recognizer, we look at two options: Creating a temporary word is a little easier

[4]`http://compgroups.net/comp.lang.forth/additional-recognizers/73467`

to understand, and you get correct `postpone` behaviour for free, but it requires more infrastructure from the system, in particular support for a section for these temporary words and knowledge about whether `compile,` produces a reference to the temporary word. The other option, defining interpretation, compilation, and postponing behaviour is a little harder to understand, but not much longer, and it requires less infrastructure from the system. The latter approach has been proposed for standardization and is preferable for this purpose.

Another design decision is whether to hook into `find` or into the text interpreter. While hooking into `find` has some advantages, the advantages of hooking into the text interpreter, in particular with respect to backwards compatibility, outweigh them.

References

[Ert97] M. Anton Ertl. Yet another Forth objects package. *Forth Dimensions*, 19(2):37–43, 1997. 2.6, 1

[Ert98] M. Anton Ertl. `State`-smartness — why it is evil and how to exorcise it. In *EuroForth'98 Conference Proceedings*, Schloß Dagstuhl, 1998. 2.4

[Ert16] M. Anton Ertl. Sections. In *32nd EuroForth Conference*, pages 55–56, 2016. 4.1

[Mah98] Manfred Mahlow. Prelude and finale: Implicit context switching based on pre- and post-executed words. In *14th EuroForth*, 1998. 2.6

[MPE16] MPE. *VFX Forth for x86/x86 64 Linux*, 4.72 edition, 2016. 2.6

[Pay12a] Bernd Paysan. Recognizer. *Vierte Dimension*, 28(2):37–38, 2012. 7

[Pay12b] Bernd Paysan. Recognizers. In *28th EuroForth Conference*, pages 108–110, 2012. 7

[Tru11] Matthias Trute. Recognizer — interpreter dynamisch verändern. *Vierte Dimension*, 27(2):14–16, 2011. 7

[Tru15] Matthias Trute. Forth recognizer — request for discussion. 3rd RfD, Forth200x, 2015. 4.2, 6, 7

The Sockpuppet Forth to C interface

Stephen Pelc
MicroProcessor Engineering
133 Hill Lane
Southampton SO15 5AF
England
t: +44 (0)23 8063 1441
e: sfp@mpeforth.com
w: www.mpeforth.com

Abstract

As processors become ever more complex and the software we are asked to write becomes more complex, it takes ever longer to write the basic drivers for an embedded system. A full digital audio chain is vastly more complex than pumping DAC output into an audio amplifier. Silicon vendors provide C libraries to make using their chips easier. Rather than convert these libraries to Forth, MPE now provides a mechanism to call the C library from Forth.

Introduction

With the ever increasing complexity of microcomputers such as the Cortex cores and systems, manufacturers are providing development hardware and software systems based on C libraries to make using their chips easier. Such libraries reduce the requirement for chip documentation at the expense of software documentation. This tendency has increased to the level that C header files include registers undocumented in the chip user manual.

The conventional approach to providing support for development boards in Forth has been to manually port the C library sources to Forth. The SockPuppet system takes a different approach by providing an interface solution between Forth and C; the Forth system calls the underlying C libraries. In turn, this allows the details of the hardware to be abstracted away by the C libraries, whilst allowing the Forth system to provide a powerful, uniform and interactive user interface.

The MPE ARM/Cortex Forth cross-compiler supports calling functions in C or any language that can provide functions that use the AAPCS calling convention. This is an ARM convention documented in IHI0042F aapcs.pdf. Calls with a variable number of parameters (varargs) are not supported.

The example code in both Forth and C is available for the Professional versions of the ARM Cortex cross compiler. The example code provides a simple GUI for an STM32F429I Discovery board using sample C code provided by ST and others. The interface is defined for Cortex-M CPUs only.

This work is directly inspired by Robert Sexton's Sockpuppet interface:
`https://github.com/rbsexton/sockpuppet`
His contribution and permission are gratefully acknowledged.

How the Forth to C interface works

Both Flash and RAM memory are partitioned, one pair for C and the other for Forth. Because of the arcane and undocumented nature of start-up for C compiler target code, the initial boot of the system is performed by the C code in order to make sure that the initialisation is correct.

Every function that is exported from the C world to the Forth world appears as one of a number of types of call. These words are called "externs". You can handcraft these words in assembler, but the MPE cross compiler compiler includes code generators for several techniques. The call format and return values match the AAPCS standard used by ARM C compilers.

Each calling technique has its own pros and cons. They are discussed in following sections.

- SVC calls. You just need to know the SVC numbers. SVC calls provide the greatest isolation between sections of code written in other languages. The functions foreign to Forth are accessed by SVC calls and/or jump tables. The example solution uses SVC calls for most foreign functions. Regardless of the primary call technique used, all techniques rely on a small number of SVC calls.
- Jump table. The base address of the table can be set at run time, e.g. by making a specific SVC call. The calling words fetch the run-time address from the table, given an index.
- Double indirect call. A primary jump table is at a fixed address and contains the addresses of secondary tables, which hold the actual routine addresses. The fixed address and both indices must be known at compile time. This technique is used by TI's Stellaris parts and some NXP parts to access driver code in ROM.
- Direct calls to the address of the routine. You need to know the address at compile time.

There is a practical limit of four arguments if you use SVC calls for the insulation between Forth and C because Cortex CPUs automatically stack four registers for an interrupt. The other interface methods do not suffer from this limit. It is a matter of convention between the Forth and C code as to parameter passing order. It can be changed by either side. MPE convention is for the left-most Forth parameter to be passed in R0. This matches the AAPCS code used by the hosted Forth compilers such as VFX Forth for ARM Linux.

SVC calls

The examples use the MPE calling convention and are illustrated in assembler as well as by using the code generator. The code generator interface is much to be preferred and preserves far more of the information in the C prototype. The decision to use the C prototype is deliberate and follows long-established practice in MPE's hosted systems.

```
SVC( 67 ) void BSP_LCD_DrawCircle( int x, int y, int r );
\ SVC 67: draw a circle of radius r at position (x,y).
```

The code generator parses the extern definition above and generates the extern as a function with three parameters implemented as SVC call 67. If you really want to demonstrate your assembler prowess, the code below performs the same operation.

```
CODE BSP_LCD_DrawCircle \ x y r --
\ SVC 67 draw a circle of radius r at position (x,y).
  mov r2, tos                     \ r
  ldr r1, [ psp ], # 4            \ y
  ldr r0, [ psp ], # 4            \ x
  svc # __SAPI_BSP_LCD_DrawCircle
  ldr tos, [ psp ], # 4           \ restore TOS
  next,
END-CODE
```

When the SVC call occurs, the Cortex CPU stacks registers R0-R3, R12, LR, PC, xPSR on the calling R13 stack with R0 at the lowest address. The SVC handler places the address of this frame in R0/R4, extracts the SVC call number, reloads the AAPCS parameters from the frame and jumps to the appropriate C function. In this case

```
void BSP_LCD_DrawCircle(
  uint16_t Xpos, uint16_t Ypos, uint16_t Radius
);
```

SVC calls provide the highest insulation between Forth and C, but suffer from several issues.

- The SVC call mechanism is part of the Cortex interrupt and exception system. The assembler and/or C side of this uses code written in assembler to allow the C routines called from a jump table to to be AAPCS compliant.
- The SVC mechanism is inefficient compared to a direct AAPCS handler.
- Because SVC calls are part of the CPU interrupt mechanism, you have to care how long a call takes. Playing games with the Cortex interrupt mechanism can fix this, but is complex.

Jump table

In order to avoid the penalties of the SVC call mechnism, you can make an array of function pointers in C or assembler and call functions using an index into the table.

```
jumptable:
dd func0 ; address of function 0
dd func1 ; address of function 1
..
```

We still need to know the address of the jump table. This is found using an SVC call (15) and stored in a variable. The jump table address could be hard-coded, but given the horrors of perverting link map files and the like, the overhead of a single SVC call is preferable.

```
SVC( 15 ) void * GetDirFnTable( void );
\ Returns the address of the jump table.
variable JT \ -- addr
\ Holds the address of the jump table.
JT holdsJumpTable
\ Tell cross compiler where jump table address is held.
: initJTI \ -- ; initialise jump table calls
  GetDirFnTable JT ! ;
JTI( n ) int open(
  const char * pathname, int flags, mode_t mode
);
```

If constructed in assembler, the SVC despatch table and the main jump table can be the same table; it's just a question of what you put in the table.

Double indirect call tables

Some vendors, particularly TI, use a table of tables approach. The sub-tables provide the API for a particular peripheral, e.g. UARTs. Before use, you have to declare the base address of the primary ROM table used for calling ROM functions. For Luminary/TI CPUs, this will probably be:

```
$0100:0010 setPriTable
```

Now you can define a set of ROM calls, for example, again for a TI CPU.

```
DIC( 4, 0 ) void ROM_GPIOPinWrite(
  uint32 ui32Port, uint8 ui8Pins, uint8 ui8Val
);
```

where:

- ROM_APITABLE is an array of pointers located at 0x0100.0010.
- ROM_GPIOTABLE is an array of pointers located at ROM_APITABLE[4].
- ROM_GPIOPinWrite is a function pointer located at ROM_GPIOTABLE[0].

Parameters:

- ui32Port is the base address of the GPIO port.
- ui8Pins is the bit-packed representation of the pin(s).
- ui8Val is the value to write to the pin(s).

To call this function, use the Forth form:

```
  port pins val ROM_GPIOPinWrite
```

Direct calls

Where the address of the routine is known at the Forth compile time, you can use a direct call.

```
  DIR( addr ) int foo( int a, char *b, char c );
```

The Forth word marshalls the parameters and calls the subroutine at target address addr.

Extracting information from C

It is convenient to have a certain amount of information available from the C portion of the code. This is supported by a few SVC calls that exist in all versions of the Sockpuppet API.

```
svc( 0 ) int SAPI-Version( void );
SVC 00: Return the version of the API in use.
svc( 1 ) int GetSharedVars( void );
SVC 01: Get the address of the shared variable list.
svc( 15 ) int GetSvcFnTable( void );
SVC 15: Get the address of the SVC function table.
```

In order to support data sharing between C and Forth, the C can export named objects which can appear as Forth words.

C Linkage structure

```
#define DYNLINKNAMEMLEN 22
typedef struct {
// This union is a bit crazy, but it's the simplest way of
// getting the compiler to shut up.
union {
void (*fp) (void);
int* ip;
unsigned int ui;
unsigned int* uip;
unsigned long* ulp;
} p;            //< Pointer to the object of interest (4)
int16_t size;   //< Size in bytes (6)
int16_t count;  //< How many (8)
int8_t kind;    //< Is this a variable or a constant? (9)
uint8_t strlen; //< Length of the string (10)
const char name[DYNLINKNAMEMLEN]; //<Null-Term C string.
} runtimelink_t;
```

When the Forth system powers up it runs the Forth word **`dy-populate`** which uses SVC call 01 to get the address of the **`dynamiclinks[]`** table, and walks through the table creating Forth named variables whose addresses match those in the C system. A Forth word **`dy-show`** is provided to list the entries in the table.

Forth Linkage structure

```
interpreter
: hword 2 field ;
: byte 1 field ;
target
struct /runtimelink \ -- len
\ Forth equivalent of the C structure above.
  int fdy.val         \ usually a pointer 0, 4
  hword fdy.size      \ size in bytes 4, 2
  hword fdy.count     \ how many 6, 2
  byte fdy.type       \ variable or constant 8, 1
  byte fdy.nlen       \ name length 9, 1
  22 field fdy.zname  \ zero terminated name 10, 22
end-struct
```

The accessor words just read the fields defined above. They are defined as compiler macros. For interaction on the target, use the field names above.

```
compiler
: dy.val fdy.val @ ;        \ addr -- n
: dy.size fdy.size w@ ;     \ addr -- w
: dy.count fdy.count w@ ;   \ addr -- w
: dy.type fdy.type c@ ;     \ addr -- c
: dy.name fdy.nlen ;        \ addr -- addr'
target
```

A set of support words allow us to run down the table and create Forth **`VALUE`**s and **`CONSTANT`**s.

Demonstration code

In order to evaluate the Sockpuppet technique and to provide a demonstration environment we decided to port the MPE PowerView GUI code to an STM32F429I Discovery board, which includes a small QVGA colour panel.

We made a decision to standardise on the gcc compiler maintained by ARM:
`https://launchpad.net/gcc-arm-embedded`
This seems to be a clean compiler, but it has a few deficiencies:

- It does not include a make utility,
- Every silicon vendor ships a different version of Eclipse with different make tools,
- They are all incompatible.

The alternative is just to take the silicon vendor's "free" tools, accepting that we will need a huge amount of disc space (almost free these days) and a degree of pain in learning the tool-chain. The days when you could just download and go are long past. Whatever you do, there will be pain.

Conclusions

Mixed language programming for embedded systems is entirely feasible and productive.

Do not assume that the C libraries provided by the silicon vendors are bug-free.

You can use the Forth to debug the C.

Once you have set it up, it all works surprisingly well, but compared to Forth cross-compilation, the C compilation chain is baroque.

Using the C libraries for hardware access saves a huge amount of time reading chip documentation. As the use of silicon vendor C libraries increases, silicon vendor are placing less importance on correct documentation. We have already found devices whose C libraries depend on undocumented registers.

Acknowledgements

Robert Sexton is responsible for the ideas and implementation of Sockpuppet. His dedication has made it a production-grade environment.

Elizabeth Rather convinced me long ago of the necessity of good notations. This convinced us to port the extern interface from hosted systems to the cross-compiler.

Implementing the Forth Inner Interpreter in High Level Forth

Ulrich Hoffmann `<uh@fh-wedel.de>`

Abstract

This document defines a Forth threaded code (inner) interpreter written entirely in high level standard Forth. For this it defines a specific threaded code structure of colon definitions, a compiler from high level Forth to this threaded code and a corresponding inner interpreter to execute it. This inner interpreter can run in a stepwise way and so gives the surrounding environment control of its execution behavior. A real time environment thus might slice the execution of threaded code in small pieces and provide an interactive command shell while still meeting its real time requirements.

1 Introduction

Forth 94 and Forth 2012 define the semantics of many Forth words. As this opens the space for various implementations and optimizations they do not however specify much of the dictionary structure: words are identified by their so called execution token (*xt*) and later executed; given a word name the *xt* can be identified (`FIND`) along with its immediacy status.

For words defined via `CREATE`, the execution token can be transformed into the memory address of its parameters by means of `>BODY`. Standard programs must not assume a specific header structure or the structure of colon definitions, nor must they rely on a specific way the system uses the return stack. Certain programming techniques that are based on such assumptions cannot be expressed in standard programs. That's ok.

Traditionally Forth is implemented as threaded code [2]. The body of colon definitions contains a list of addresses of the words it invokes. There are *primitives* expressed in machine code and *high level definitions* that are defined in threaded code.

This document defines a threaded code interpreter written entirely in standard Forth. It defines a specific threaded code structure of colon definitions. This allows to also define an inner interpreter (traditionally known as `NEXT`) for this threaded code in high level Forth. The inner interpreter defined below can run in a stepwise way so the execution of threaded code can be sliced in small pieces in a real time environment.

The primitives of this threaded code interpreter are all the words that the underlying Forth system defines (be they machine code or

```
 body
 of sq
|-----------+---------+-------------|
| xt of dup | xt of * | xt of ~exit |
|-----------+---------+-------------|
    ^
    |
    |
 |----|
 | IP |   Interpreter pointer
 |----|
```

Figure 1: Threaded code of the definition `~: sq ( x -- x ) dup * ~;`

```
body
of test
|------------+---+------------+---+---------+-------------|
| xt of ~lit | 3 | xt of ~lit | 4 | xt of + | xt of ~EXIT |
|------------+---+------------+---+---------+-------------|
```

Figure 2: Threaded code for literals `~: test ( -- u ) 3 4 + ~;`

colon words there). Threaded code high level words are defined by a special version of colon (*thread-colon* `~:`). Its definition is given and explained in section 3.

Also, an outer compiler that compiles *to* threaded code as well as an outer interpreter defined *in* threaded code is defined below giving an interactive shell to real time systems.

The current implementation does not define dictionary/header structures but leaves these as unspecified as the standards do. We still get an interactive system with a well defined threaded code structure. If more specific knowledge about a system is required, it would of course be possible to also specify the exact structure of headers and the dictionary layout and to define appropriate operations on these structures.

2 Threaded Code

This section describes the chosen threaded code structure of colon definitions. We will look at simple words with invocation of primitives, at string and number literals, and at control structures.

Let's assume the definition:

```
~: sq ( x -- x ) dup * ~;
```

then the threaded code for `sq` looks as shown in figure 1 on the preceding page. For each word, that is referenced by `sq` a corresponding execution token is stored. A thread-colon definition ends with the execution token of `~exit` compiled by `~;` (*thread-semicolon*). `sq` invokes only primitives, but the threaded code structure would be identical if thread-colon words were invoked: They also have execution tokens and these would be stored in the body of the newly defined word.

An interpreter pointer `IP` references the current point of execution. The threaded code interpreter will modify `IP` while executing the code.

2.1 Number literals

When there is a number literal in the source code, it is later processed by means of `~lit`:

```
~: test ( -- u )  3 4 + ~;
```

as can be seen in figure 2 on the top of this page.

2.2 Printing string literals

Printing string literals (used by `~."`) is handled by `(~."`, see figure 3 below.

```
~: test3 ( -- )   ~." it works" ~;
```

```
body
of test3                                         aligned
|------------+---+-----+-----+-----+-----+-----+---+-------------|
| xt of (~." | 8 | 'i' | 't' | ' ' | ... | 's' |   | xt of ~exit |
|------------+---+-----+-----+-----+-----+-----+---+-------------|
```

Figure 3: Threaded code for string literals `~: test3 ( -- ) ~." it works" ~;`

```
  |------------+---+------------+----|
A | xt of ~lit | 0 | xt of ~lit | 10 |
  |------------+---+------------+----|

  |----------+-----------+----------------+--------------|
B | xt of 1- | xt of dup | xt of ~?branch | address of D |
  |----------+-----------+----------------+--------------|

  |---------+------------+---------+------------+---------------+--------------|
C | xt swap | xt of over | xt of + | xt of swap | xt of ~branch | address of B |
  |---------+------------+---------+------------+---------------+--------------|

  |---------+-------------|
D | xt drop | xt of ~exit |
  |---------+-------------|
```

Figure 4: Threaded code for a `BEGIN WHILE REPEAT` loop

A counted string with string length and the string characters is placed inline in the code . In order to place the following token on an aligned address the bytes after the inline string are padded.

Our threaded code is (arbitrarily) restricted to just printing strings. A more general approach could easily define threaded string literal words corresponding to `S"`.

2.3 Control structures

Control structures compile unconditional and conditional absolute branches:

For illustration let's define the word `looptest` with a `BEGIN WHILE REPEAT` loop:

```
: looptest ( -- )
   0 10
   ~BEGIN
     1-  dup
   ~WHILE
     swap over + swap
   ~REPEAT drop  ~;
```

Figure 4 shows the corresponding threaded code. There are 4 basic blocks labeled *A* through *D* with appropriate branches at the end of basic block *B* (conditional) and *C* (unconditional). The threaded code primitives `~?branch` and `~branch` modify the interpreter pointer `IP` appropriately.

2.4 Limitations

The current threaded code defines just as much structure so that a simple interactive Forth outer interpreter can be defined on top.

It does not define a code for a complete standard system. Specifically the current threaded code does not contain

- (user) variables or constants. However because definitions of the underlying system become primitives of threaded code, it inherits variables and constants.
- `DO LOOP`s. Adding this would be similar to the branching words already available.
- Neither defining words nor `DOES>` as they are not required to program an interactive outer interpreter.
- a primitive for pushing address and length of string literals on the stack. It could easily be defined similar to the inline string printing word.

3 Implementation

This section explains how standard Forth definitions will allow to interpret and construct the threaded code structure explained above. As a general naming convention, names that start

```
\ Perform a single interpretation step
: step ( i*x -- j*x )
   IP @ dup cell+ IP !
   @ catch
   ?dup IF cr ." Error " . reset THEN ;

\ Loop steps
: run ( i*x -- j*x )
   BEGIN IP @ WHILE  step  REPEAT ;
```

Figure 5: The threaded code inner interpreter

with the tilde-character ~ denote threaded code words. They often have corresponding words in the underlying system with similar functionality.

The state of the threaded code interpreter has the following components:

- the already mentioned Interpreter Pointer `IP`:

  ```
  Variable IP  0 IP !
  ```

- a return stack `~RP` with its corresponding return stack pointer `RP`.

  ```
  Create ~R0  20 cells allot
  Variable RP  ~R0 RP !
  ```

  Having a return stack of our own, allows us to explicitly define the return stack behavior when nesting. Using that knowledge return stack tricks can work (we make no use of them here, though).

- a data stack shared with the underlying system, and

- memory (code and data) also shared with the underlying system.

Also headers, wordlists and the dictionary structure is shared with the underlying system.

3.1 Inner Interpreter

We now define the threaded code inner interpreter. It is defined in Figure 5. It works very similar to the `NEXT` code in classical Forth implementations:

`step` first gets the address of the next execution token and increments `IP`. It then fetches the execution token and executes it, which modifies the interpreter state as desired. In case of an error the word `reset` is invoked, which re-initializes the interpreter.

In traditional Forth implementations every invoked word ends in a jump to the `NEXT` code (or inlines it, if short enough) which results in continuous threaded code interpretation. In our case, as `step` invokes words via `catch`, they return to `step` and no continuous interpretation takes place. This has the benefit, that we can stepwise interpret threaded code (thus the name `step`) and gain control after each step. Continuous interpretation is handled by `run` in simple cases which calls `step` in a loop. Setting `IP` to `0` in one of the invoked words would stop `run`. Note, that `IP` had been initialized to `0` so that a `~exit` from the top level word will set `IP` to `0` as well and thus also stop threaded code interpretation. A real time environment would not call `run` but would do single interpretation `step`s when appropriate.

3.2 Return stack operations

The return stack we defined above grows towards increasing addresses. It should operate

```
: ~>r ( x -- )
  \ push a cell to the return stack
  RP @ !   1 cells RP +! ;

: ~r> ( -- x )
  \ pop a cell from the return stack
  -1 cells RP +!   RP @ @ ;
```

Figure 6: Return stack operations

using post increment and pre decrement operations. `RP` is supposed to point to the next available cell. Figure 6 shows the appropriate definitions for `~>r` and `~r>`.

3.3 Inline number literals

Inline number literals are prefixed with the `~lit` instruction as shown in figure 7. Its definition is

```
: ~lit ( -- n )
  \ extract inline number literal
  IP @ @  1 cells IP +! ;
```

Before execution, the interpreter pointer points to the code cell that contains `val` (solid line). Execution extracts the value *val* and puts it on the stack. After execution the interpreter pointer points past the literal (dotted line). This threaded code structure is generated by the threaded code outer interpeter, that we will define in section 3.6.

3.4 Printing inline string literals

Inline string literals are handled similar to number literals by `(~."` as shown in figure 8 on the next page. Note, that we are only interested in printing inline string literals here:

```
: (~." ( -- )
   \ extract inline string
   \ literal and print it
   IP @  count  2dup + aligned IP !
   type ;
```

Moving the interpreter pointer past the inline string requires alignement as specified for our threaded code structure.

The threaded code of figure 8 is generated by the compiling word `~."` that is defined like this:

```
: ~." ( <ccc>" -- )
  \ Compile inline string to be
  \ printed later when executed.
  \ Like ." but for threaded code
  ['] (~." ,
  [char] " word count
  here over 1+ chars allot place align
; immediate
```

It first compiles `(~."`, then the counted string and also takes care of the required alignment.

3.5 Control structures

Up to now threaded code execution is sequential. `step` moves the interpreter pointer suc-

```
|----|
| IP |-------------+......
|----|             |     .
                   |     .
                   V     V
|-----+------------+-----+-----|
| ... | xt of ~lit | val | ... |
|-----+------------+-----+-----|
```

Figure 7: The execution of `~lit`

```
|----|
| IP |-------------+........................
|----|             |                       .
                   |                       .
                   V                       V
|-----+------------+-----+----+----+-----+----+-----|
| ... | xt of (~." | len | c1 | c2 | ... | cn | ... |
|-----+------------+-----+----+----+-----+----+-----|
```

Figure 8: The execution of `(~."`

cessively over threaded code cell by cell. No branches take place.

Defining branches and control structures is simple. As in traditional Forth implementations we define unconditional (`~branch`) and conditional (`~?branch`) branches. For the sake of simplicity they branch to absolute addresses. Other branching regimes or additional branching instructions such as `DO-LOOP` primitives would easy to add.

```
: ~branch ( -- )
   \ absolute unconditional jump
   IP @ @ IP ! ;

: ~?branch ( f -- )
   \ absolute conditional jump
   IF  1 cells IP +!
   ELSE ~branch THEN ;
```

The interpreter pointer is adjusted appropriately. After execution it points to the branch target or (in case of a non taken conditional branch) to the instruction following `~?branch` (skipping the branch address).

In order to compile branches in a structured way we define the zoo of Forth control structure as depicted in figure 9.

These correspond to the standard control structures but compile threaded code branches with embedded absolute threaded code addresses.

3.6 Compiler *to* threaded code

In section 3.3 we saw the handling of inline number literals. It is still open, how the appropriate threaded code structure (figure 7 on the previous page) is generated. As in a traditional implementation this is done by the Forth text interpreter.

In figure 10 on the following page we define a variant of the classical outer compiler that looks for words in the dictionary, executes them when they are immediate or compiles them when not.

If a word is not found in the dictionary, the compiler tries to see if it is a number and then compiles it as number literal if so, or else raises an error. Note, that this compiler does not handle double numbers or base prefixes.

```
: ~IF ( -- x )  ['] ~?branch , here 0 , ; immediate
: ~AHEAD ( -- x )  ['] ~branch here 0 , ; immediate
: ~ELSE ( x -- x' )  ['] ~branch , here 0 , swap here swap ! ; immediate
: ~THEN ( x -- )  here swap ! ; immediate

: ~BEGIN ( -- x ) here ; immediate
: ~WHILE ( x1 -- x2 x1 )  ['] ~?branch ,  here 0 , swap ; immediate
: ~AGAIN ( x -- )  ['] ~branch , , ; immediate
: ~UNTIL ( x -- )  ['] ~?branch , , ; immediate
: ~REPEAT ( x2 x1 -- )  postpone ~AGAIN  postpone ~THEN ; immediate
```

Figure 9: Threaded code control structures

```
variable ~state   ~state off  \ threaded code outer interpreter state

-13 Constant #notfound

: ~] ( -- )  ~state on
   BEGIN ( )
      BEGIN ( )
        bl word dup c@   \ scan next token
      WHILE ( c-addr )   \ another token found
        find  ?dup       \ look up in dictionary
        IF   -1 = IF ,  ELSE execute  THEN  ~state @ 0= IF EXIT THEN  \ found
        ELSE   0 0 rot count
          over c@ [CHAR] - = dup >r IF  1- swap char+ swap THEN \ word not found
          >number IF  #notfound throw THEN
          drop drop r> IF negate THEN
          ['] ~lit , ,  \ compile threaded code literal
        THEN
      REPEAT ( c-addr )  \ no more tokens in input stream
      DROP
      SOURCE-ID 0= IF CR ." ] " THEN
      REFILL 0=    \ read more from input stream
   UNTIL ;  \ input stream exhausted

: ~[ ( -- )  ~state off ; immediate   \ stop threaded code compiler

: ~: ( <name> -- )
   \ push IP to return stack and set IP to start of threaded code.
   Create ~]  Does> IP @ ~>r  IP ! ;

: ~EXIT ( -- )
   \ Pop IP from return stack
   ~r> IP ! ;

: ~; ( -- )
   \ Compile end of definition and leave threaded code outer compiler
   ['] ~EXIT ,   ~state off   ; immediate
```

Figure 10: Compiler to threaded code

Threaded code words are defined with

```
~: name ....   ~;
```

`~:` invokes `~]` that compiles the following source code until `~state` becomes `false` (by executing `~;` or `~]`) or the input stream is exhausted. Inside the definition we have to use corresponding threaded code words (e. g. control structurres) to compile the right code.

We can then interactively execute the threaded code definition with

```
 name run
```

The (normal) Forth word `name` sets the interpreter pointer to the beginning of its threaded code. `run` then interprets this code. If the interpreter executes `name`'s `~exit` `IP` will become zero (being initialized to zero and pushed on the return stack) and the `run` loop terminates. We return to the underlying system.

As dictionary structure and headers are shared with the underlying system, the headers for threaded code are just defined in the base Forth system.

Note also that execution of a threaded code word is split into two parts. On execution of the word in the underlying system interpretation

```
~: ~interpret ( -- )
    ~BEGIN ( )
      bl word dup c@     \ scan next token
    ~WHILE ( c-addr )    \ another token found
      find               \ lookup in dictionary
      dup  1 = ~IF drop execute  ~ELSE   \ immediate
      dup -1 = ~IF drop state @ ~IF  compile,  ~ELSE  execute ~THEN ~ELSE
      \ word not found, number?
      drop   0 0 rot count   over c@ 45 = dup ~>r ~IF  1- swap char+ swap ~THEN
           >number ~IF  #notfound throw ~THEN drop drop    \ maybe number
           ~r> ~IF  negate ~THEN
           state @ ~IF  postpone LITERAL  ~THEN  \ compile literal
      ~THEN ~THEN
    ~REPEAT ( c-addr )
   drop ~;

   ~: ~quit ( -- ) clear-stack   ~R0 RP ! ~state off  interpret-mode
     ~BEGIN cr state @ ~IF ~." ] " ~THEN  ~query ~interpret ~." ~ok"  ~AGAIN ~;
```

Figure 11: Interpreter in threaded code

does *not* start immediately but only the interpreter pointer is adjusted appropriately (saving its old content to the return stack). By this we can explicitly control execution by `step` and `run`.

3.7 Interpreter *in* threaded code

Ultimately we want to have a Forth outer interpreter that is defined *in* threaded code so that we can have an interactive shell which can be executed via `step` and `run`. Up to now we just have a compiler *to* threaded code, but this is defined in the underlying Forth system and we cannot control its execution.

So — here we go. We define a Forth outer interpreter in threaded code. Figure 11 shows a FIG forth style interpreter loop that combines interpretation and compilation state. It compiles to code of the base system (using `compile,` and `Literal`). So — it is similar in function to the base system outer interpreter but is itself compiled to threaded code using the threaded code compiler `~[` defined above. And so, we can control its execution via `step`.

`~quit` is the corresponding `quit` loop that successively expects user input and interprets it. (`~query` is an appropriate `query` implementation in threaded code, not shown).

4 Conclusion

In this document we defined a threaded code structure for Forth colon definitions and an inner interpreter, `step`, in high level Forth. Stepwise execution of threaded code can be controlled by periodicaly invoking `step`.

We constructed a compiler to generate this threaded code and also an interactive outer interpreter *in* threaded code. As `step` can control the execution of this outer interpreter, its execution time can be distributed according to the requirements of a real time system, such as the synchronous Forth framework in [1].

References

[1] *A synchronous FORTH framework for hard real-time control*, U. Hoffmann and A. Read, euroForth 2016

[2] *Threaded Interpretive Languages: Their Design and Implementation*, R. G. Loeliger, McGraw Hill, 1981

f
A package manager for (the)Forth(.net)

Gerald Wodni

EuroForth 2016

Abstract

Let us join Forthes and share great code with each other – in an orderly fashion.

1 Introduction

The Forth Net[1] has matured into a simple meta-repository system, which provides a mindbogglingly new idea to the Forth programmer: Build your projects upon existing code from others instead of reinventing the wheel all over again.

Starting with code snippets up to full blown libraries authors can upload their code (behold, some even ship documentation!) to The Forth Net. Each package contains a Forth-parseable description-file which was discussed in the last Forth200x-Meeting in Bath.

Packages can easily be searched and downloaded by using *f*, the Forth package manager. Typing `fget mrot 1.0.0` in Gforth will install mrot in version 1.0.0 directly onto your box.

2 A small Package

To proof how simple the creation of a package is, let us share a non standardized – yet very useful – word with the forth community:

Listing 1: mrot.4th

```
: -rot ( x1 x2 x3 -- x3 x1 x2 ) rot rot ;
```

As stated before we shall also offer a minimalistic documentation which happens to be machine readable and easily parseable in Forth. The package.4th-file is an obligatory file which contains key-value definitions. `key-value <name> <value>` parses a (key-)name and uses the remaining line as value. Mandatory keys are:

name name of the package [a-z]+[-a-z0-9]*

version semantic version number consisting of 3 decimal numbers separated by '.' [2]

license name of the license you publish the package with, i.e. "GPL", "MIT", . . .

Further information about this file (key-list, tags, dependencies, . . .) can be found in the package guidelines [3].

Our example does not need any dependencies, but we add a short description, specify the main include-file and the license.

Listing 2: package.4th

```
forth-package
    key-value name mrot
    key-value version 1.0.0
    key-value description save TOS in 3rd
    key-value license public domain
    key-value main mrot.4th
end-forth-package
```

3 f - a package manager

Us Forth programmers are not fond of leaving our beloved system. This is why *f* was coded into existence, a simple package manager which lets us explore and download packages without leaving the realms of our system. This is its API:

fall list all packages

fsearch <needle>

finfo <name> show a package's ReadMe

fget <name> <version> download a package

finclude <name> <version>

4 We need you!

Packetize your code now! By the date of this writing, we have 15 amazing packets ready to be used in your next project, and 321140 to go to become the biggest package repository in the world! ;)

References

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

[2] SemVer 2.0. `http://semver.org/`.

[3] Package guidelines. `http://theforth.net/guidelines`.

net: Using net2o

reinventing the internet

Bernd Paysan

EuroForth 2016, Konstanz/Reichenau

Outline

3 years after Snowden

What happend to change the world:

Politics	Manhatten project to find “the golden key”?
Users	don’t want their dick picks be watched and use DuckDuckGo and encrypted chat
Software	NSA backdoors have been refitted by attackers (Juniper)
Solutions	net2o starts to be increasingly usable

net2o in a nutshell

net2o consists of the following 6 layers (implemented bottom up):

2. Path switched packets with 2^n size writing into shared memory buffers
3. Ephemeral key exchange and signatures with Ed25519, symmetric authenticated encryption+hash+prng with Keccak, symmetric block encryption with Threefish
   onion routing camouflage probably with AES
4. Timing driven delay minimizing flow control
5. Stack–oriented tokenized command language
6. Distributed data (files) and distributed metadata (prefix hash trie)
7. Apps in a sandboxed environment for displaying content

Objectives

net2o’s design objectives are

- lightweight, fast, scalable
- easy to implement
- secure
- media capable
- works as overlay on current networks (UDP/IP), but can replace the entire stack

Basic Frameworks

PKI	Create, import, and exchange keys
Named file copy	For testing only
Vault	A container for encrypted data without metadata exposure
DHT	Query key/value pairs (keys are pubkeys or hash keys)
Chat	Instant messaging 1:1 or in chat groups
Version control system	For larger/structured content
Sync	to synchronize your computers (RSN)
Audio/Video Chat	Real time data streaming (RSN)

Get it: Debian and Android

Debian — Use the Debian package, and enter as root:

```
cat >/etc/apt/sources.list.d/net2o.list <<EOF
deb [arch=amd64,all] http://net2o.de/debian
testing main
EOF
wget -O -
https://net2o.de/bernd@net2o.de.gpg.asc | \
apt-key add -
aptitude update; aptitude install net2o
```

Android — Get Gforth from play store or
https://net2o.de/Gforth.apk
Open/close (back button) Gforth if you like; then open net2o.

Get it: Windows and macOS

Windows — Get the two current setup.exes for Gforth and net2o, and install them:
http://www.complang.tuwien.ac.at/forth/gforth/Snapshots/current/gforth64.exe
https://net2o.de/windows/net2o64.exe
You will be asked for accepting the unsigned exes, as neither Gforth nor net2o are signed now

macOS — Once I got around creating a brew tap, it will be easy to install under Mac OS X (or whatever it is called now), too.

Get it from Source

From Source for Linux, Mac OS X, Windows (cygwin) you need:

```
git automake autoconf make gcc libtool
libltdl7 fossil
```

you run: `mkdir net2o; cd net2o`

```
wget
https://fossil.net2o.de/net2o/doc/trunk/do
chmod +x do; ./do
```

This will install some stuff and take some time (I will try to improve that).

Try it — Generate a Key

Linux you run:

```
n2o cmd
keygen <nick>
```

Enter your passphrase twice.

Android Tap on the little nettie to start the app, it will autodetect that you don't have a key generated. Enter nick and passphrase twice.

Try it — get a key and chat

- To get my key, search for it (32 bit is sufficient now, but easy to attack)
  `keysearch kQusJ`
- Send me an invitation
  `invite @bernd`
- Try to chat with me
  `chat euroforth@bernd`
- Aquire more keys by observing a group chat. List your keys with
  `n2o keylist`
  from within the chat.
- Change networks with your Android and watch that the chat still works.
- Leave the chat with `/bye` or Ctrl+D (back on Android)

Try it — Vault en/decryption

- Take a file and encrypt it
  `enc test.txt`
- Show it's content
  `cat test.txt.v2o`
- Sign a file with a detached signature
  `sign test.txt`
- Verify the signature
  `verify test.txt`

Try it — Use the DVCS

- Create a directory and add a few files into it, keep a net2o instance running inside that directory with
  `n2o cmd`
- Initialize the directory
  `init`
- Add the files in the directory
  `add *`
  `ci -m "My checkin message"`
  and check them in
- Change a file and see what has changed
  `diff`
- Check in the changed file
  `ci -m "Second checkin"`
- Show the commit messages
  `log`

For Further Reading I

Bernd Paysan
net2o source repository and wiki
http://fossil.net2o.de/net2o

Security

M. Anton Ertl, TU Wien

Problem

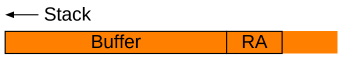
- Attacks on computer systems
- "Hackers"
- Nation-level attackers

Problem for Forth?

- We do embedded systems
  Input limited, therefore not attackable
- Example: TV Set with 10 Buttons
  Therac-25
  Remote control
  Teletext
  DVB-C/S/T
  Smart TV (Internet/WLAN)
- StuxNet
- Internet of Things

Possible Attacks

- Arbitrary Code Execution
- usually enabled by a **buffer overflow** vulnerability

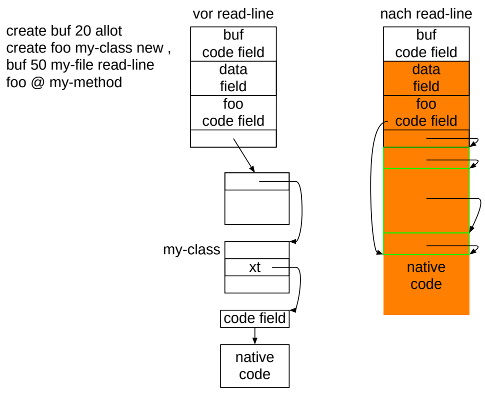


- Dangling pointer
- Other attacks
  read buffer overflow (Heartbleed)
  SQL injection
  …

Buffer overflows in Forth?

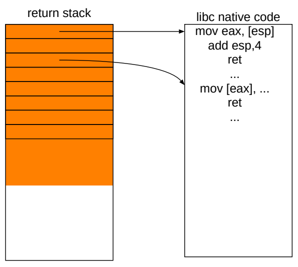
- Length of access explicit
  `move ( c-addr1 c-addr2 u -- )`
  but: two lengths involved
  Mistakes possible
- Memory accesses with `! c!` etc.
- `xc!+?` instead of `xc!+`

Separate return stack: does it protect?



Does non-executable data help?

- NX DEP W^X …
- Return-Oriented Programming



Now what?

- Code audit
- Secure programming language
  No access beyond buffer boundaries

And in Forth?

- Divide the program
- A part in full Forth
  Needs audit against Buffer overflows
- A part in a secure Dialect
  More cumbersome to program
  simpler audit
  Not secure against malicious programmer
  for more you would need more type checking

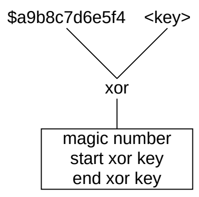
```
... move ! ...
secure
... move* !* ...
insecure
...  ! !* ...
```

Secure Dialect

- Buffer descriptors: start, end
- Pass and check against buffer descriptor on every write access
- Have checking variants of all writing words
- `move* ( from to count buf -- )`
  `!* ( x addr buf -- )`
  `read-file* ( c-addr u file-id buf -- u2 ior )`
- Variables?
  Use `value`
  `variable* !! @@`

Protect against mistakes

- Stack arrangement mistakes
  Magic number in descriptor
  “encrypt” descriptor

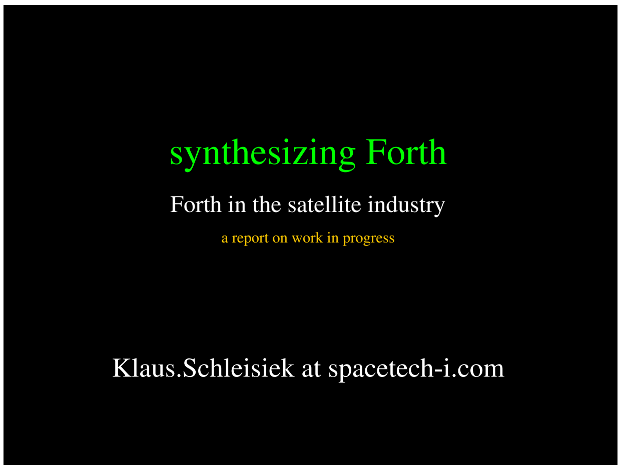


Dangling Pointers

- `buf @ free-buf 5 a 24 + buf @ !*`
- garbage collection instead of `free`
- or overwrite descriptor on `free`

Conclusion

- Buffer overflows ⇒ arbitrary code execution
- Secure Forth dialect for defense
  each writing word takes buffer descriptor
- No typechecking,
  but magic number and/or descriptor “encryption”
  to protect against mistakes

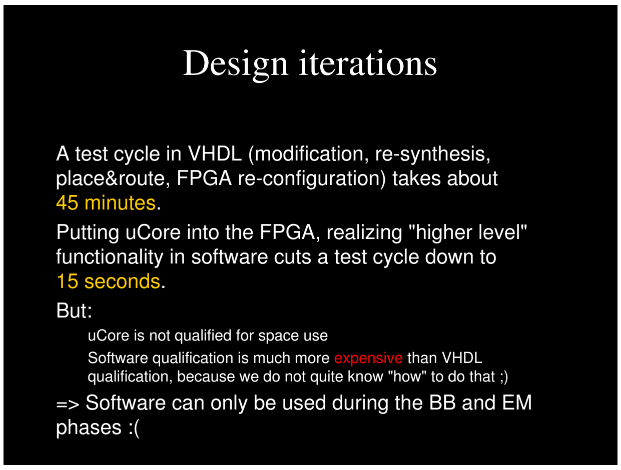


synthesizing Forth

Forth in the satellite industry

a report on work in progress

Klaus.Schleisiek at spacetech-i.com



Design iterations

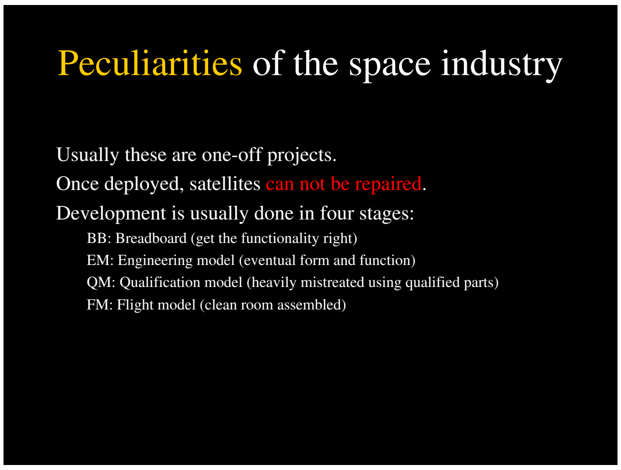
A test cycle in VHDL (modification, re-synthesis, place&route, FPGA re-configuration) takes about 45 minutes.

Putting uCore into the FPGA, realizing "higher level" functionality in software cuts a test cycle down to 15 seconds.

But:

- uCore is not qualified for space use
- Software qualification is much more expensive than VHDL qualification, because we do not quite know "how" to do that ;)

=> Software can only be used during the BB and EM phases :(



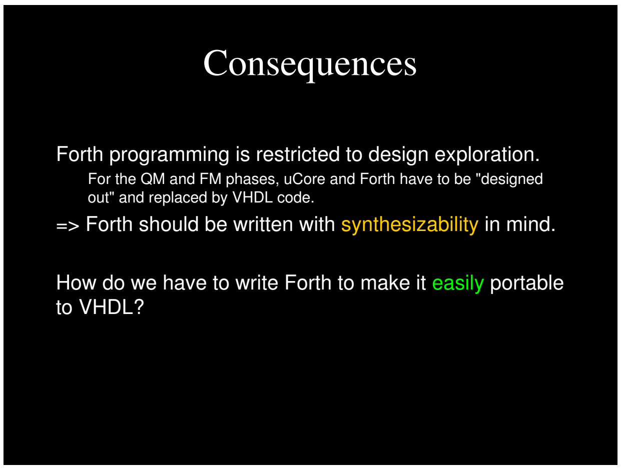
Peculiarities of the space industry

Usually these are one-off projects.

Once deployed, satellites can not be repaired.

Development is usually done in four stages:

- BB: Breadboard (get the functionality right)
- EM: Engineering model (eventual form and function)
- QM: Qualification model (heavily mistreated using qualified parts)
- FM: Flight model (clean room assembled)



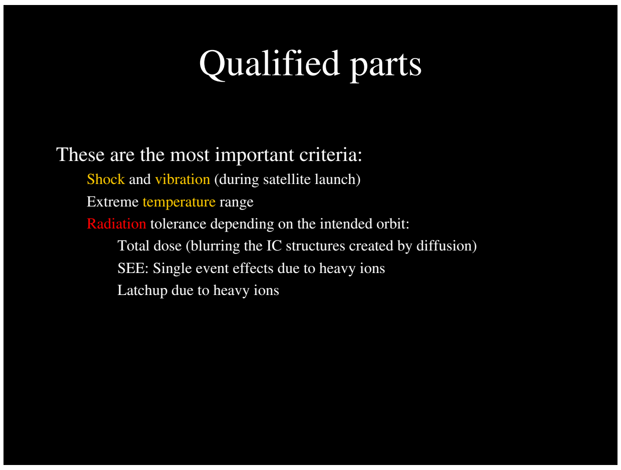
Consequences

Forth programming is restricted to design exploration.

- For the QM and FM phases, uCore and Forth have to be "designed out" and replaced by VHDL code.

=> Forth should be written with synthesizability in mind.

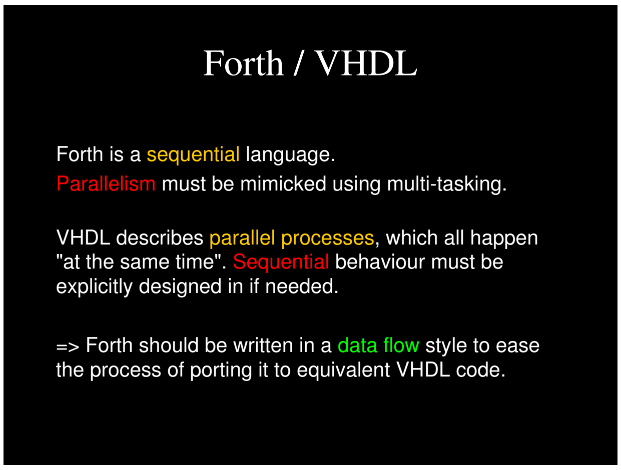
How do we have to write Forth to make it easily portable to VHDL?



Qualified parts

These are the most important criteria:

- Shock and vibration (during satellite launch)
- Extreme temperature range
- Radiation tolerance depending on the intended orbit:
  - Total dose (blurring the IC structures created by diffusion)
  - SEE: Single event effects due to heavy ions
  - Latchup due to heavy ions



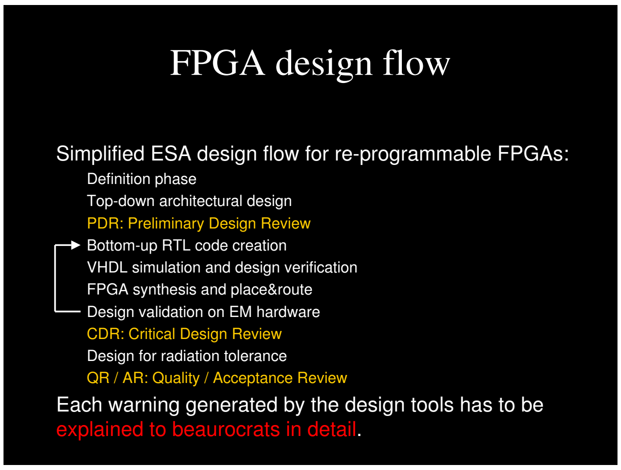
Forth / VHDL

Forth is a sequential language.

Parallelism must be mimicked using multi-tasking.

VHDL describes parallel processes, which all happen "at the same time". Sequential behaviour must be explicitly designed in if needed.

=> Forth should be written in a data flow style to ease the process of porting it to equivalent VHDL code.

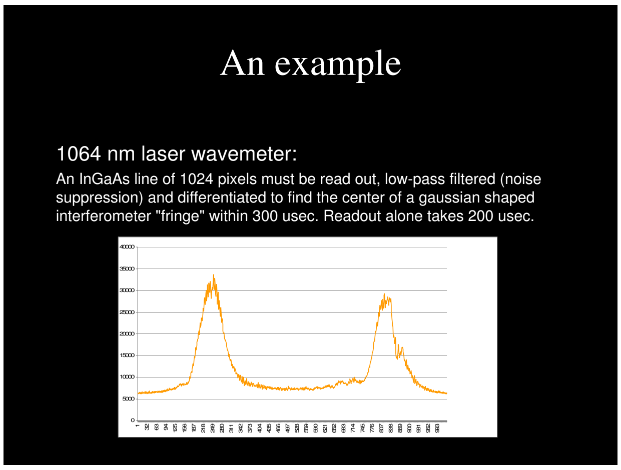


FPGA design flow

Simplified ESA design flow for re-programmable FPGAs:

- Definition phase
- Top-down architectural design
- PDR: Preliminary Design Review
- Bottom-up RTL code creation
- VHDL simulation and design verification
- FPGA synthesis and place&route
- Design validation on EM hardware
- CDR: Critical Design Review
- Design for radiation tolerance
- QR / AR: Quality / Acceptance Review

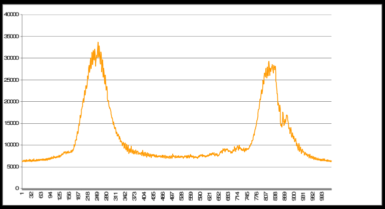
Each warning generated by the design tools has to be explained to beaurocrats in detail.

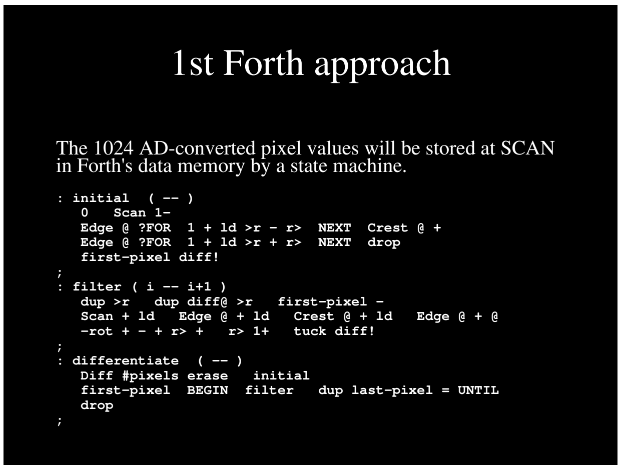


An example

1064 nm laser wavemeter:

An InGaAs line of 1024 pixels must be read out, low-pass filtered (noise suppression) and differentiated to find the center of a gaussian shaped interferometer "fringe" within 300 usec. Readout alone takes 200 usec.

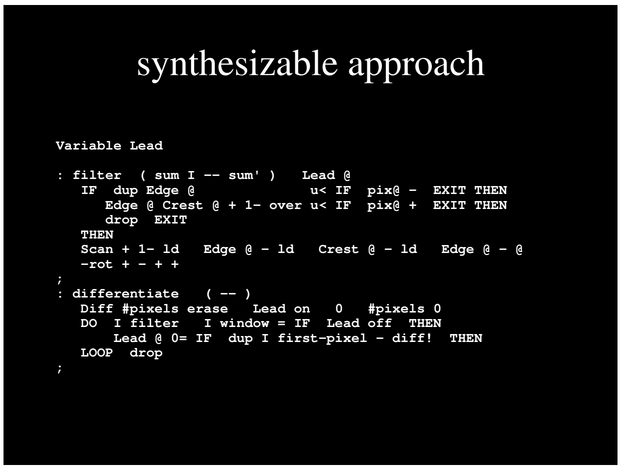




1st Forth approach

The 1024 AD-converted pixel values will be stored at SCAN in Forth's data memory by a state machine.

```
: initial  ( -- )
   0   Scan 1-
   Edge @ ?FOR  1 + ld >r - r>  NEXT  Crest @ +
   Edge @ ?FOR  1 + ld >r + r>  NEXT  drop
   first-pixel diff!
;
: filter ( i -- i+1 )
   dup >r   dup diff@ >r   first-pixel -
   Scan + ld   Edge @ + ld   Crest @ + ld   Edge @ + @
   -rot + - + r> +   r> 1+   tuck diff!
;
: differentiate  ( -- )
   Diff #pixels erase   initial
   first-pixel  BEGIN  filter   dup last-pixel = UNTIL
   drop
;
```



synthesizable approach

```
Variable Lead

: filter  ( sum I -- sum' )   Lead @
   IF  dup Edge @              u< IF  pix@ -  EXIT THEN
      Edge @ Crest @ + 1- over u< IF  pix@ +  EXIT THEN
      drop  EXIT
   THEN
   Scan + 1- ld   Edge @ - ld   Crest @ - ld   Edge @ - @
   -rot + - + +
;
: differentiate   ( -- )
   Diff #pixels erase   Lead on   0   #pixels 0
   DO  I filter   I window = IF  Lead off  THEN
       Lead @ 0= IF  dup I first-pixel - diff!  THEN
   LOOP  drop
;
```