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).

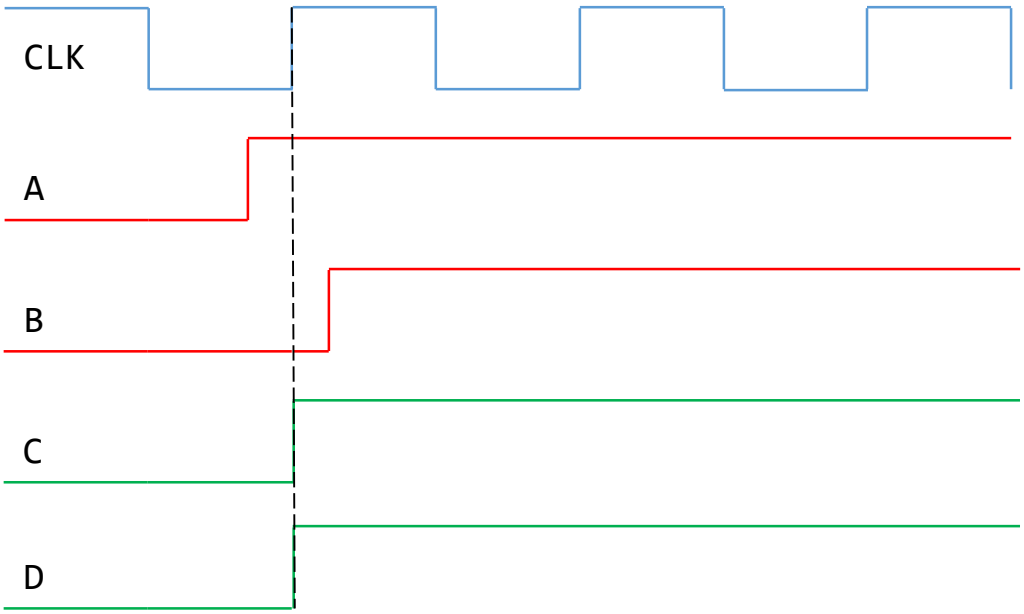

Figure 1: An example of asynchronous and synchronous signals. A and B update in their own time without reference to CLK. C and D update synchronously with each other and with the rising edge of CLK.

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

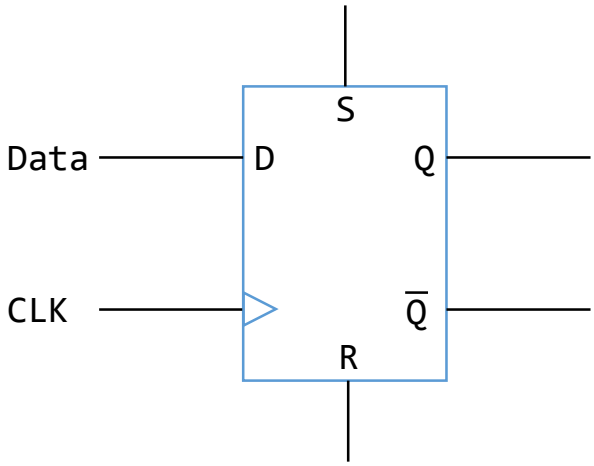

Figure 2: A schematic of a D-type flip-flop. The signal at input D is registered on the rising edge of CLK. The registered signal subsequently appears on outputs Q and inverted Q and hold steady until the next rising edge of CLK. Set (S) and reset (R) inputs are available to drive Q high and low respectively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

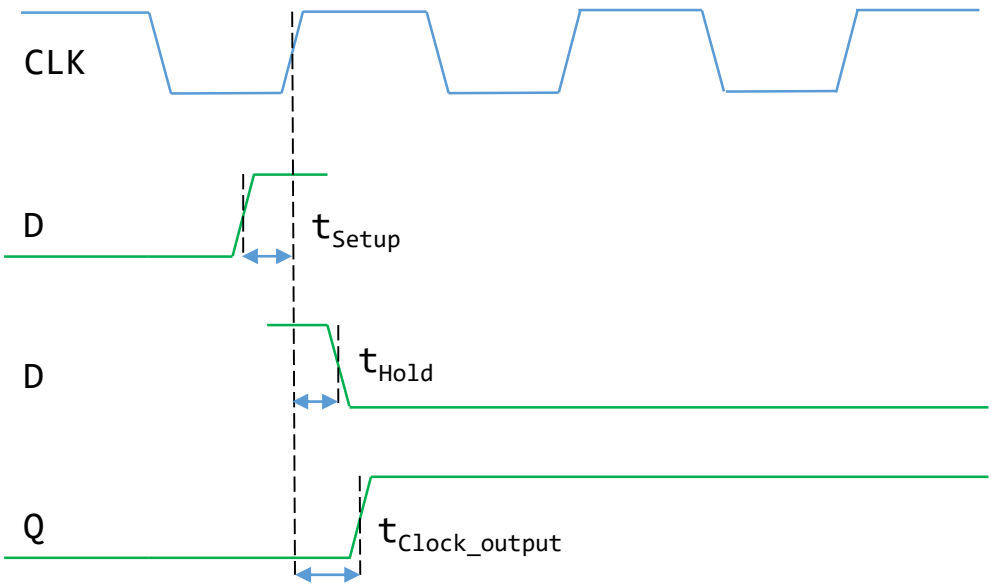

Figure 3: Timing characteristics of a D-type flip-flop. The data signal must be stable t_{Setup} before the leading edge of the clock. The data signal must remain stable t_{Hold} after the leading edge of the clock. The output signal is updates t_{Clock_Output} after the leading edge of the clock.

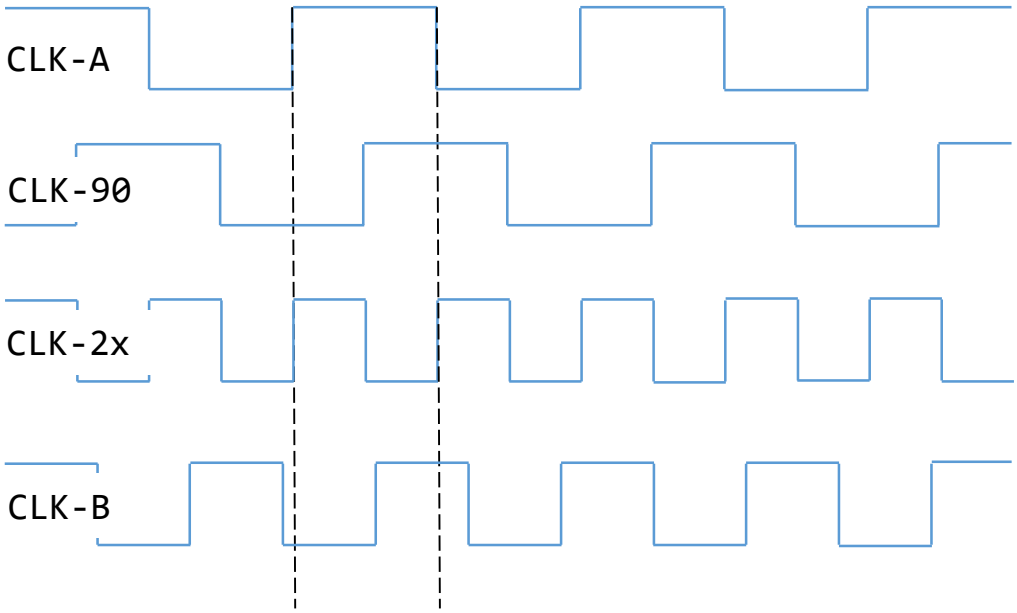

Figure 4: An illustration of multiple clock domains. Clock-A is the reference clock. Clock-90 has the same period as Clock-A but is retarded 90 degrees in phase. Clock-2x has twice the period of Clock-A but is in phase. Clock-B has a period somewhere between Clock-A and Clock-2x and does not have a fixed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

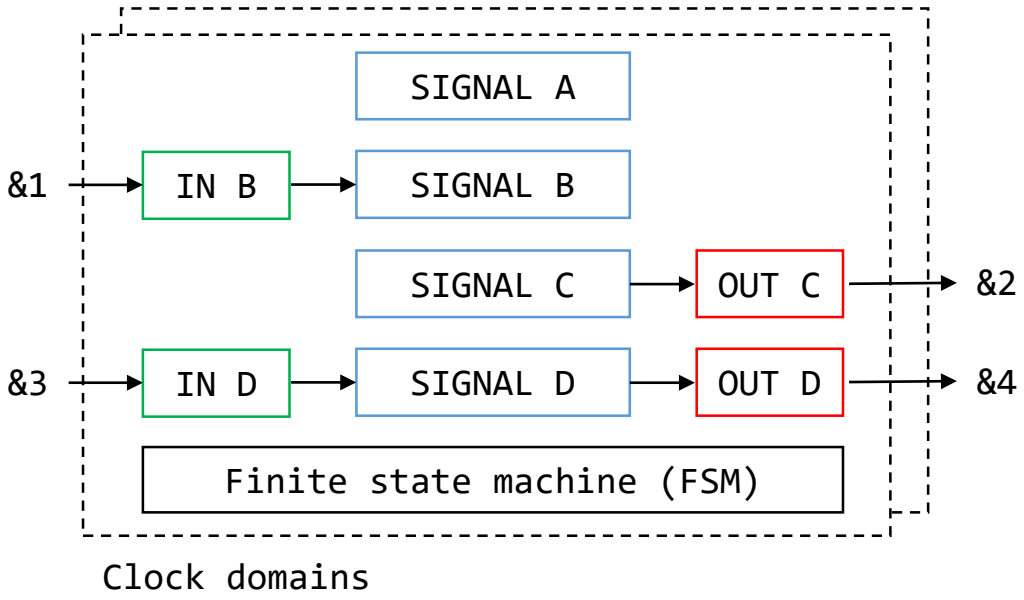

Figure 5: Schematic overview of the FORTH framework. Entities are organized within CLOCK domains that have a defined frequency and phase. Within each CLOCK domain the primary construct is a SIGNAL which is essentially a variable with clock-synchronous update. IN's link memory-mapped addresses to signals 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
```

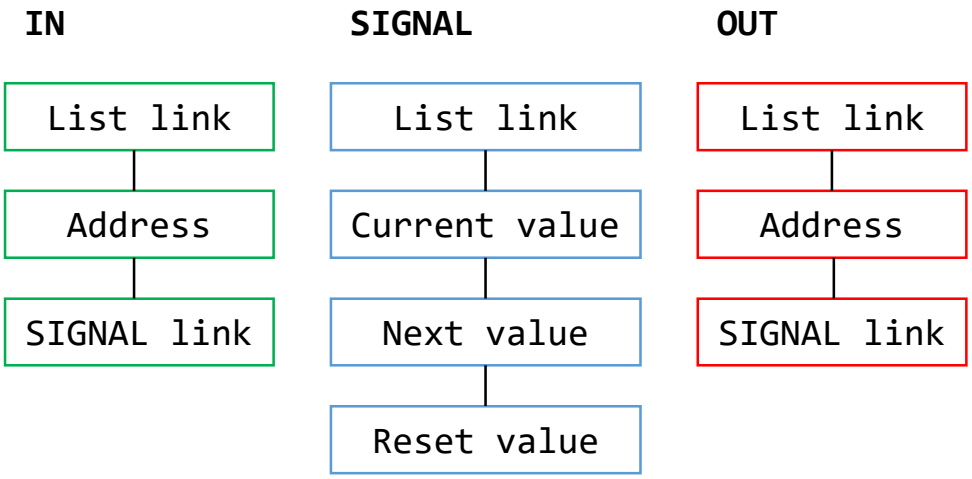

Figure 6: Structure of the SIGNAL, IN and OUT entities. Each entity type is organized within a linked list

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

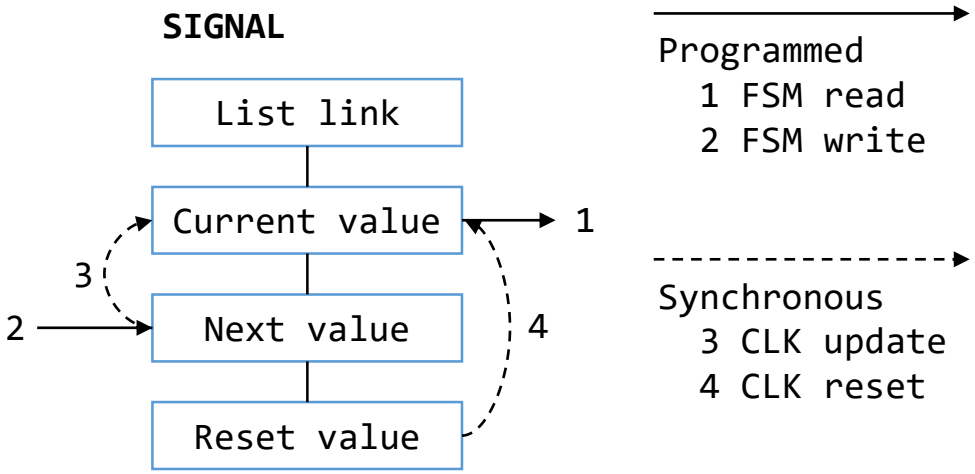

Figure 7: Schematic of the programmed and synchronous update of SIGNAL's

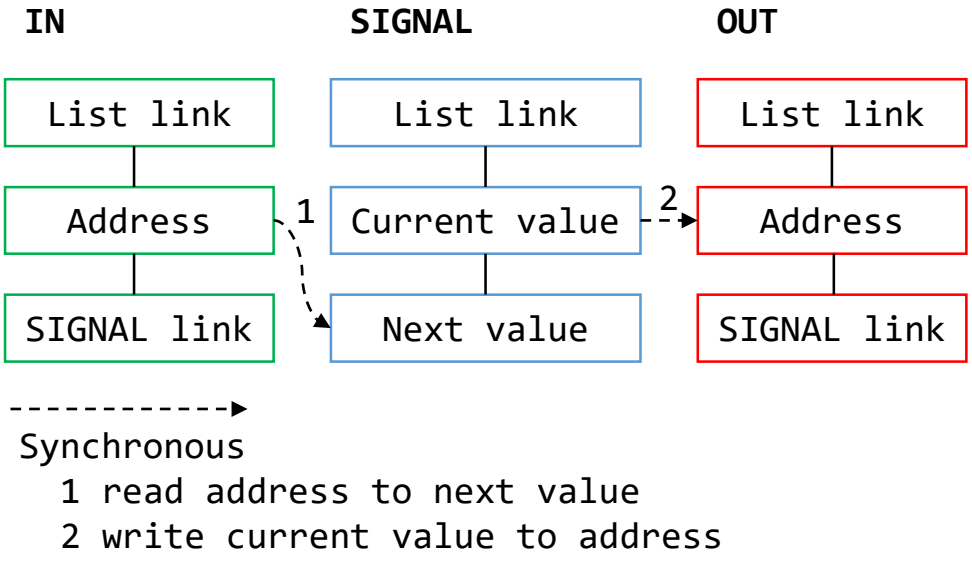

Figure 8: Schematic of the synchronous updates of IN's and OUT's

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).

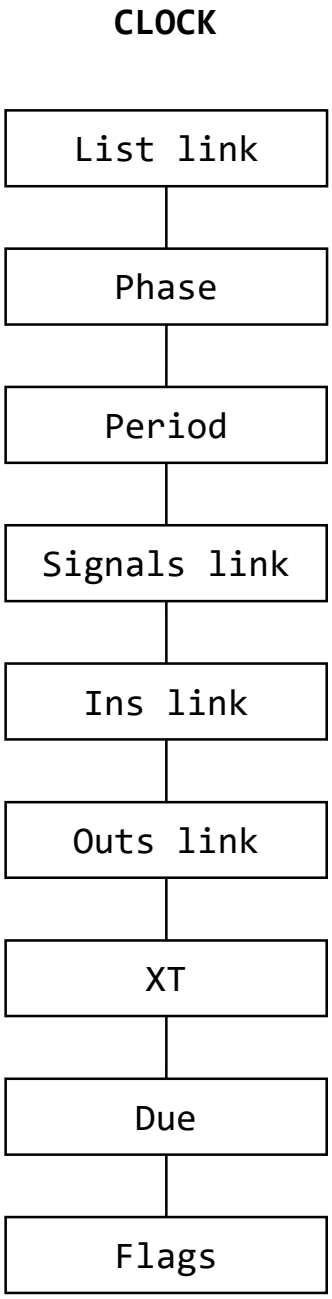

Figure 9: Structure of the CLOCK domain entity

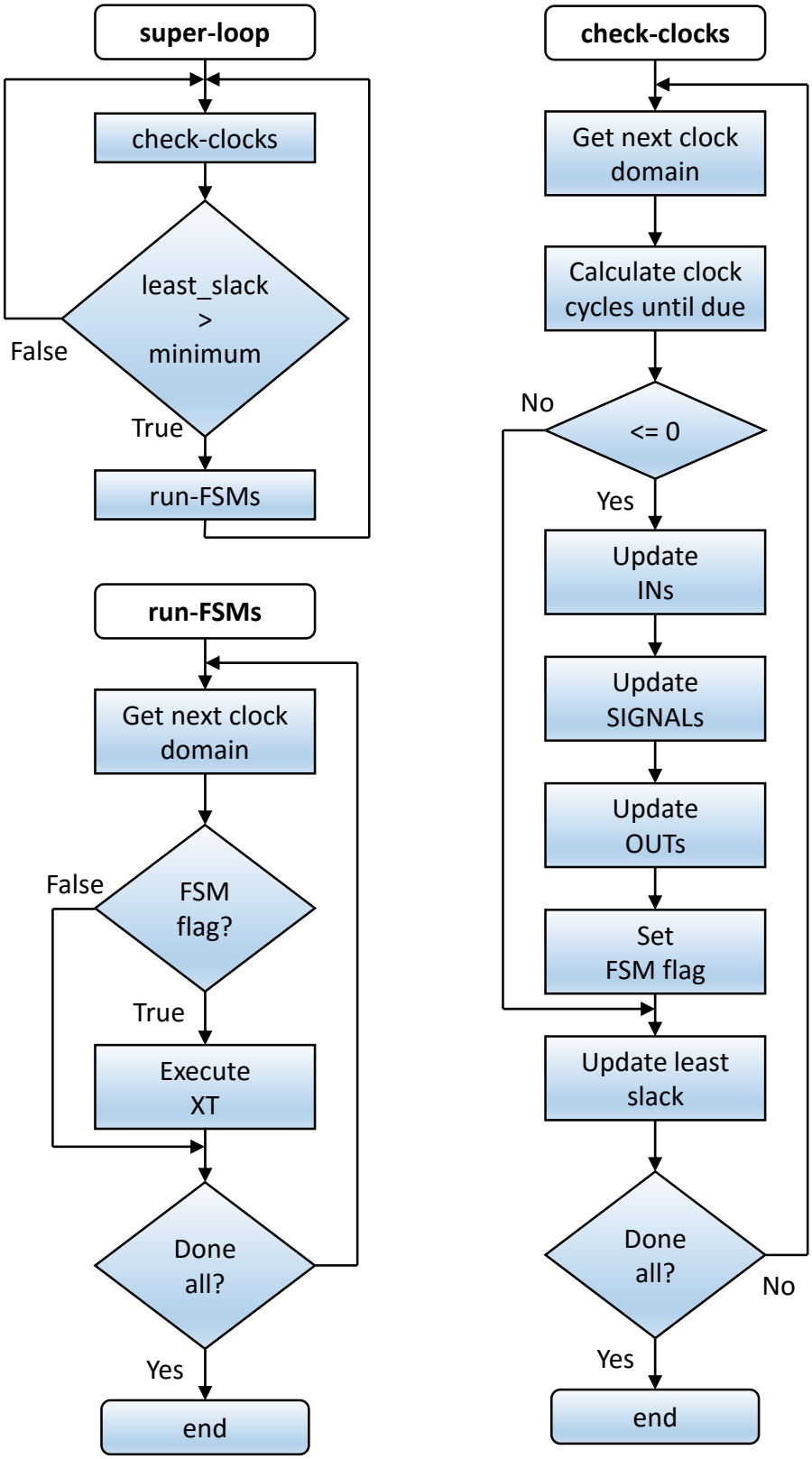

Figure 10: Flowchart of the FORTH software for the framework.

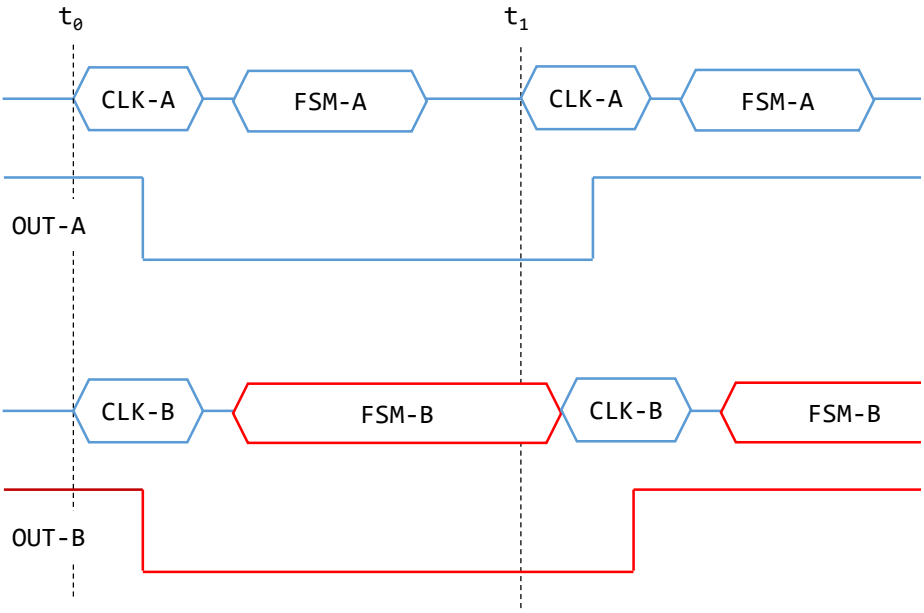

Figure 11: Timeline diagram of the timing framework in operation. A and B are separate cases and are presented on the same diagram only for comparison purposes. Case A illustrates typical operation of the framework with a single clock domain. Case B illustrates that there is a practical lower limit that 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

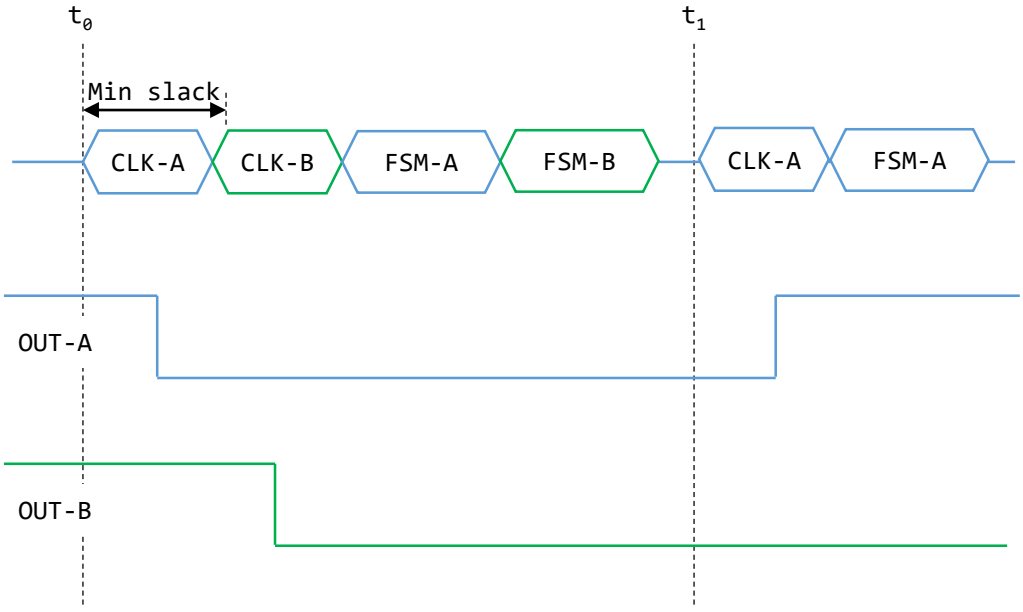

Figure 12: Multiclock timeline. In this example it is assumed that clock domain A has exactly half the period of clock domain B and that the two clock domains are (approximately) in phase.

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_{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_{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_{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.

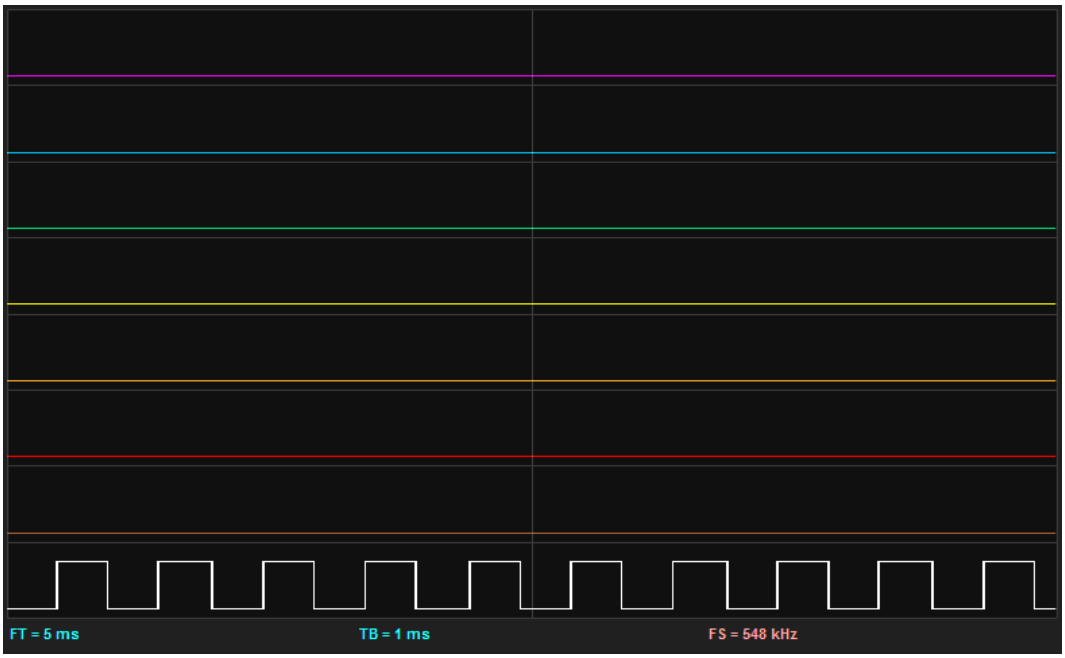

Figure 13: Measurement of jitter. A single square wave is being generated. The duration of each cycle from rising-edge to rising-edge was measured using the on-screen cursor of a PC oscilloscope. In all 30 cycles were measured.

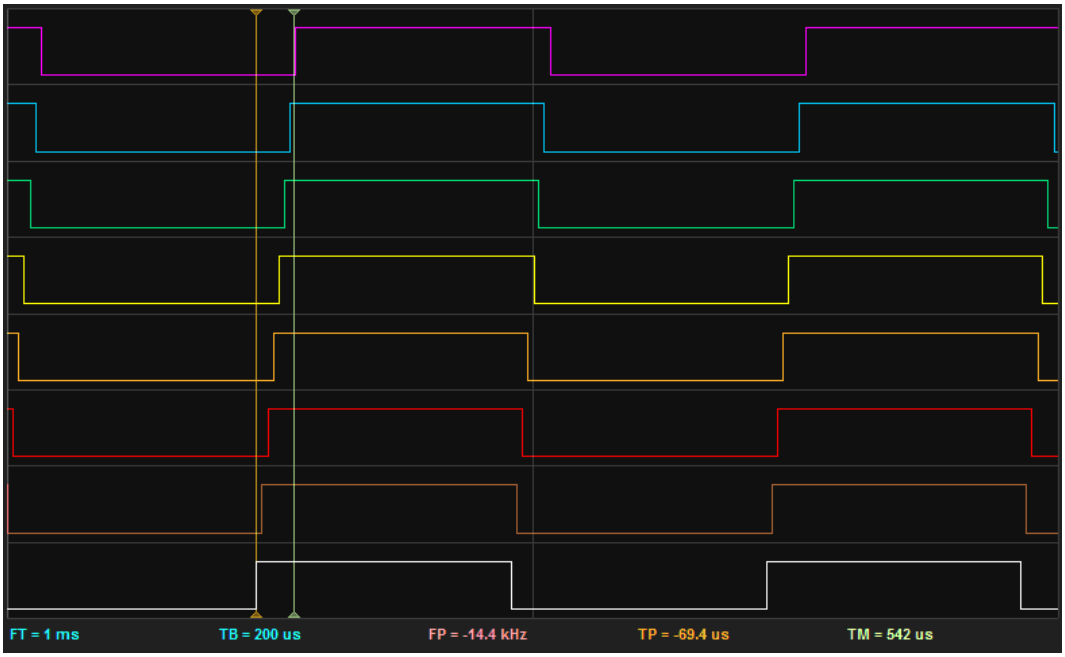

Figure 14: Measurement of t_{Clock_Output}. A series of signals have been generated. The lowest signal is taken as the leading edge of the clock and the delay in the output of the other signals was measured.

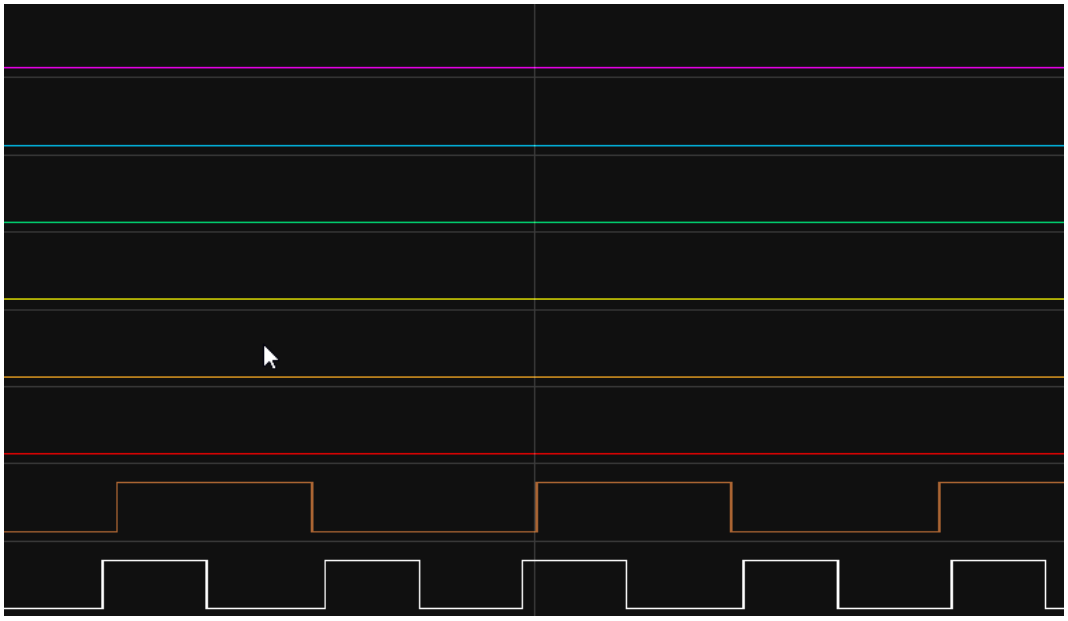

Figure 15: Two synchronous clock domains. The lower signal is a 1 kHz square wave (generated in a 2 kHz clock domain that inverts the output each cycle). The upper signal is a 0.5 kHz square wave specified with no phase offset to the lower signal. The transition edges of both signals should theoretically 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.

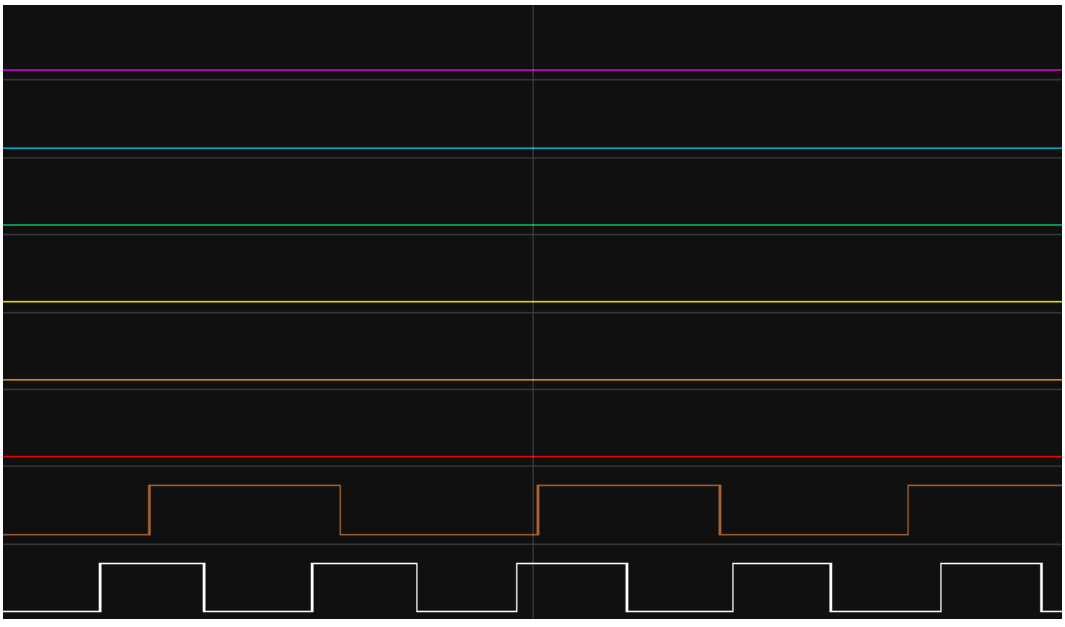

Figure 16: Two asynchronous clock domains. In this case the lower signal is a 1 kHz square wave and the upper signal is a 0.75 kHz square wave. The two clock domains are asynchronous in the sense that their transition edges can occur at any point relative to the phase of the other. Visual inspection 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

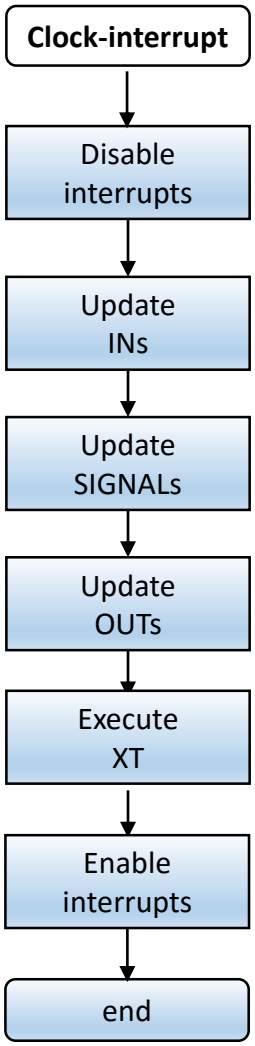

Figure 17: Alternative scheme for triggering each clock-domain with an interrupt

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)
```

```
68          ,                           ( default-value structure-base signal-link) \ link
69          !                           ( default-value) \ save this signal's location to signal-link
70          dup ,                       \ current
71          dup ,                       \ next
72          ,                           \ reset
73      Does> ( -- current value)
74          >current @                  \ return the current value of the signal
75  ;
76
77  : IN ( clock-domain addr <SIGNAL> --)
78  \ create a new IN port
79      swap here swap              ( addr structure-base clock-domain)
80      >in-link dup @              ( addr structure-base in-link last-in)
81      ,                           ( addr structure-base in-link)
82      !                           ( addr)
83      ,                           \ addr
84      ' >body ,                   \ signal
85  ;
86
87  : OUT ( clock-domain addr <SIGNAL> --)
88  \ create a new OUT port
89      swap here swap              ( addr structure-base clock-domain)
90      >out-link dup @             ( addr structure-base in-link last-in)
91      ,                           ( addr structure-base in-link)
92      !                           ( addr)
93      ,                           \ addr
94      ' >body ,                   \ signal
95  ;
96
97  : => ( n <name> -- )
98  \ store a value in the next field of a SIGNAL (better not to redefine TO?)
99       ' >body >next  state @ IF postpone literal  postpone !  EXIT THEN ! ; immediate
100
101 : {update-signal} ( 'signal -- )
102 \ update a signal to its next value
103      dup >next @  swap >current ! ;
104
105 : {reset-signal} ( 'signal -- )
106 \ update a signal to its default value
107     dup >reset @  swap 2dup >current ! >next ! ;
108
109 : {update-in} ( 'in -- )
110 \ read an IN port and write to the >next field of its SIGNAL
111     dup >addr @ @ swap >signal @ >next ! ;
112
113 : {update-out} ( 'out -- )
114 \ write to an OUT port, the >current field of its SIGNAL
115     dup >signal @ >current @ swap >addr @ ! ;
116
117 : do-list ( xt link -- )
118     BEGIN                       ( xt link)
119         @ dup                   ( xt 'item)
120     WHILE                       ( xt 'item)
121         over over >r >r swap    ( 'item xt)
122         execute                 ( --)
123         r> r>                   ( -- xt 'item)
124     REPEAT
125     drop drop ;
126
127 : do-clocks ( xt --)
128 \ apply XT to all clocks in turn
129 \ XT must have signature (i*x 'clock -- j*x)
130     clock-link do-list ;
131
132 : do-signals ( i*x clock-domain xt -- j*x )
133 \ apply XT to each signal in turn in a given clock-domain
134     swap >signal-link do-list ;
135
136 : do-ins ( i*x clock-domain xt -- j*x )
137 \ apply XT to each IN in turn in a given clock-domain
138     swap >in-link do-list ;
139
140 : 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
;
```

```
214
215 : .outs ( clock-domain)
216 \ print all of the IN's in a clock domain
217     cr
218     ['] .out do-outs
219 ;
220
221
222 variable slack
223 \ slack is updated by check-clocks, it contains the number of CPU cycles until the next clock
        domain due time
224
225 variable min-slack  0 min-slack !
226 \ min-slack is fine-tuned by the designer. If slack < min-slack then super-loop will wait
227 \ for the next clock rather than proceed with FSM execution
228
229 : {initialize-clock} ( t0 'clock -- t0)
230 \ initialize a clock given the t0 value of CPU-time
231     >R dup R@ >period @ R@ >phase @ + + R@ >due !       \ set due time
232     0 R@ >flags !                                       \ clear flags
233     R> reset-signals                                    \ reset all signals
234 ;
235
236 : initialize-clocks ( --)
237 \ initialize all clock domains
238     CPU-time ['] {initialize-clock} do-clocks drop ;
239
240
241 : {check-clock} ( 'clock --)
242 \ check if this CLOCK is due and if so update the SIGNALS, set flags = alive, and schedule the
        next clock
243     dup >due @                  ( 'clock due)
244     CPU-time -                  ( 'clock cycles-until-due)
245     dup 0 <= IF                 ( 'clock cycles-until-due)
246         drop                        ( 'clock)
247         dup update-ins                  \ all IN's, SIGNALs and OUTs now update synchronously
248         dup update-signals
249         dup update-outs
250         dup >flags dup @            ( 'clock 'flags flags)
251         1 OR swap !                     \ set flags so to indicate that the FSM is due to run
252         dup >period @ dup >R        ( 'clock period R:period)
253         over >due @                 ( 'clock period last-due R:period)
254         +                           ( 'clock next-due R:period)
255         over >due !                     \ determine and save the next due time
256         R>                          ( 'clock period)
257     THEN                            ( 'clock cycles-until-due/period)
258     slack @ MIN slack ! drop    \ update the slack
259 ;
260
261 : check-clocks ( -- slack)
262 \ check all clocks, update SIGNALS and flags where clocks are due, and update slack
263     100000000 slack !               \ initial dummy value
264     ['] {check-clock} do-clocks
265     slack @
266 ;
267
268 : {run-FSM} ( 'clock --)
269 \ check if this clock is now active to run and if so, run the FSM
270     dup >flags @                    ( 'clock flags)
271     1 AND IF                        ( 'clock )
272         \ run the FSM logic
273         dup >r >xt @ execute r>
274         \ set flags so that this task will now sleep)
275         dup >flags dup @            ( 'clock 'flags flags)
276         254 AND swap !              ( 'clock)
277     THEN
278     drop
279 ;
280
281 : run-FSMs ( --)
282     ['] {run-FSM} do-clocks ;
283
284 : super-loop
```

```
285     BEGIN
286         BEGIN
287             check-clocks    ( slack)
288             min-slack @ >   ( flag)
289         UNTIL
290         run-FSMs
291 
292         key? drop           \ VFX FORTH needs this to facilitate Windows refresh
293     AGAIN
294 ;
295 
296 : main ( -- )
297         reset \ threaded code interpreter
298     initialize-clocks
299     super-loop
300 ;
```

Test listings

```
1  jitter.fs
2
3  erase-clocks
4
5  FCPU 2000 / 0 CLOCK 2kHz
6
7  2kHz 0 SIGNAL s0
8
9  : toggle
10     s0 not => s0
11 ;
12
13 ' toggle 2KHz FSM
14
15 2KHz LOGIC0 OUT s0
16
17 : test
18     main
19 ;
```

```
1  max_freq.fs
2
3  erase-clocks
4
5  10000 constant freq  \ frequency in Hz
6
7  FCPU freq / 0 CLOCK CLK
8  CLK 0 SIGNAL s0
9
10 0 variable t0
11
12 : toggle
13     s0 not => s0
14     100 0 DO i t0 ! LOOP
15 ;
16
17 ' toggle CLK FSM
18
19 CLK LOGIC0 OUT s0
20
21 : test
22     main
23 ;
```