Timestamp Peripherals for Precise Real-Time Programming

John Hui
j-hui@cs.columbia.edu
Columbia University
New York, New York, USA

Kyle J. Edwards
edwards.kyle.j@gmail.com
Columbia University
New York, New York, USA

Stephen A. Edwards
sedwards@cs.columbia.edu
Columbia University
New York, New York, USA

ABSTRACT

On microcontrollers, timer devices provide high-precision timing, but that precision is lost when using high-level languages without suitable abstractions for temporal behavior. So, for timing-sensitive applications, programmers resort to low-level languages like C which lack expressiveness and safety guarantees. Other programmers use specialized precision-timing hardware which is expensive and difficult to obtain.

In this work, we achieve sub-microsecond precision from a high-level real-time programming language on the RP2040, a cheap, widely available microcontroller. Our work takes advantage of the RP2040's Programmable I/O (PIO) devices, which are cycle-accurate coprocessors designed for implementing hardware protocols over the RP2040's GPIO pins.

We use the PIO devices to implement timestamp peripherals, which are input capture and output compare devices. We use timestamp peripherals to mediate I/O from programs written in Sslang, a real-time programming language with deterministic concurrency. We show that timestamp peripherals help Sslang programs achieve the precise timing behavior prescribed by Sslang's Sparse Synchronous Programming model.

KEYWORDS

real time systems, concurrency control, computer languages, timing

ACM Reference Format:
John Hui, Kyle J. Edwards, and Stephen A. Edwards. 2023. Timestamp Peripherals for Precise Real-Time Programming. In *21st ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '23), September 21–22, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3610579.3611084

1 INTRODUCTION

Systems often have a mix of real-time requirements, ranging from picosecond-level precision to best-effort. This paper proposes timestamp peripherals—general-purpose peripherals that timestamp input events and emit output events according to timestamps—as an interface between the hardest real-time layer and the first software layer (Figure 1). While a handful of existing peripherals timestamp events, most are specialized.

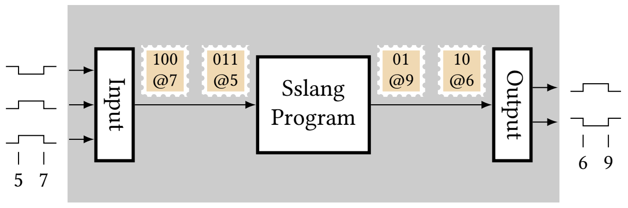


Figure 1: Our approach: a peripheral interprets changes on input pins as timestamped events, which are passed to a real-time discrete-event simulator (the Sslang program), which sends timestamped output events to another peripheral that generates precisely timed output waveforms.

These hardware-managed timestamps make it much easier to develop and analyze real-time software. Lohstroh et al. [13] argue that real-time programming models should provide software with some notion of *logical time*, an engineering fiction that is easier to reason about than physical models of time. Within a real-time system, the timestamp peripherals we propose here form the boundary between the logical software and the physical external environment.

Timestamping in software, such as with an interrupt service routine that records a system timer value, is imprecise because of interrupt response time uncertainty. Another approach would be to implement such timestamping hardware in an FPGA with a processor core, but such chips are substantially more expensive than commodity microcontrollers.

To demonstrate timestamp peripherals, we implement them on the inexpensive (US$0.70), widely available RP2040 microcontroller, using its programmable input/output (PIO) blocks and interface them with the Sparse Synchronous Model (SSM) runtime. The resulting peripherals sample input pins at 16 MHz and allow output changes to be scheduled with the same precision, far more accurately than is possible using only the RP2040's 1 MHz timer. Overall, our system[†] gives users the ability to write high-level programs that can measure and produce output signals with 62.5 ns precision.

In this paper, we describe and evaluate the performance of our real-time software environment with timestamp peripherals. We based our environment on Edwards and Hui's [5] Sparse Synchronous Model and propose a real-time language called Sslang (Sparse Synchronous Language), described in Section 2. Sslang relies on timestamp peripherals, which we implemented on the RP2040 microcontroller and its PIO blocks, described in Section 3 and Section 4. To determine the performance limits of our approach, we ran experiments and describe our findings in Section 5. Section 6 summarizes related work; we conclude in Section 7.

This work was supported by the NIH under grant RF1MH120034-01.

[†]Source code available at https://github.com/ssm-lang/pico-ssm

2 THE SPARSE SYNCHRONOUS MODEL

The Sparse Synchronous Model [5, 7] is a discrete-event model of computation for specifying real-time behavior. Like traditional discrete-event systems, it is built around scheduled variable updates managed by an event queue that executes them in temporal order; its main novelty is a deterministic mechanism for resolving logically simultaneous events, inspired by the synchronous languages [4].

Sslang is an imperative-functional language built on SSM that provides scheduled variable updates, blocking waits on variables, and parallel computation. Like Python, Sslang uses indentation to signify grouping; like Haskell and OCaml, Sslang features strong and static typing with type inference. Here is the "hello world" of the embedded world in Sslang, which blinks an LED at 10 Hz:

```
blink led =  // blink takes one parameter: led
  loop        // Repeat the following lines
    // 50 ms from now, toggle the value of led
    after ms 50, led <- not (deref led)
    wait led           // Wait for led to be updated
```

Here, `blink led =` defines the function *blink* with a single argument *led*, an otherwise ordinary mutable variable that has been connected to an output peripheral. **`loop`** starts an infinite loop that begins by scheduling an update to toggles the *led* variable 50 ms in the future. In that **`after`** statement, `ms` is a function applied to 50 that computes the number of system ticks in a 50 ms duration, and **`deref`** reads the current value of *led*, a mutable variable. The subsequent **`wait`** statement suspends this function's execution until the next write to *led*; when execution resumes, the loop restarts.

While this example resembles a "blink" program in C or Python, Sslang manipulates an event's time with the same care as its value. In particular, logical time does not advance except at *wait* statements, meaning each loop iteration logically takes *exactly* 50 ms, regardless of how long the processor physically takes to execute the machine instructions in the loop. Using a timestamp output peripheral with 62.5 ns precision, our implementation of this program on the RP2040 generates a 10 Hz square wave that is as accurate and precise as the microcontroller's crystal oscillator.

Figure 2 showcases more of Sslang with a larger program that transforms presses of a bouncy pushbutton switch into clean 200 ms pulses. The oscilloscope traces in Figure 3 illustrates its behavior.

The first two functions illustrate how we build abstractions from Sslang primitives. The *sleep* function provides the familiar "suspend execution for a period of time" functionality. It creates a local variable *timer* that conveys pure events (written "`()`" in Sslang), schedules a future event on this variable, then waits for that update. The *waitfor* function blocks until a given variable takes a given value, returning instantly if the variable already has that value. Constructs such as *sleep* and *waitfor* are standard library functions.

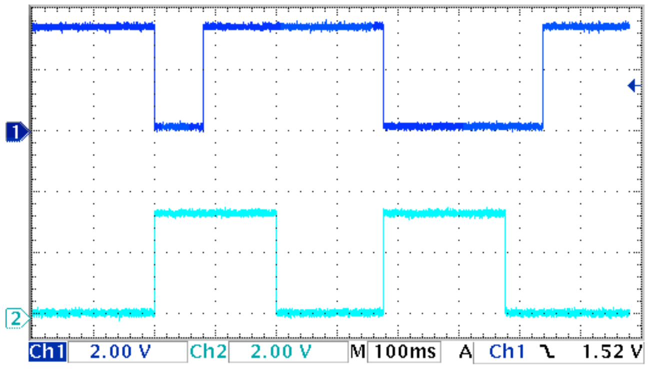
The *debounce* and *pulse* functions implement the two halves of our pulse generator. The *debounce* function is an infinite loop that generates a pure event when it sees an active-low button pressed, waits some time for any bouncing to subside, then waits again for the button to be released. The *pulse* function waits for a button-press event; after one arrives, *pulse* immediately sets the output high while scheduling it to become low again in the future. The last *wait* makes *pulse* ignore any event before the end of the pulse.

```
sleep delay =
  let timer = new ()     // Allocate a pure event variable
  after delay, timer <- ()  // Schedule a wake-up
  wait timer                // Suspend until then

waitfor var value =
  while deref var != value  // Current value is not value
    wait var                // Wait for update to var

debounce delay input press =
  loop
    waitfor input 0     // Active-low button pressed
    press <- ()         // Send "press" event
    sleep delay         // Debounce
    waitfor input 1     // Button released
    sleep delay         // Debounce

pulse period press output =
  loop
    wait press          // Wait for the "press" event
    output <- 1         // Pulse high immediately
    after period, output <- 0  // Schedule low
    wait output         // Wait for low

buttonpulse button led =
  let press = new ()  // Debounced button press signal
  par debounce (ms 10)  button press
      pulse    (ms 200) press  led
```

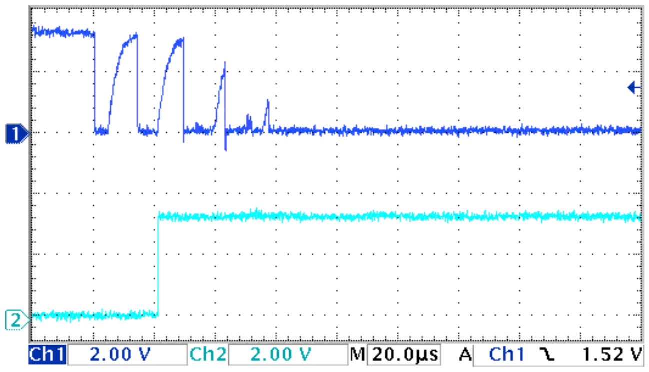
Figure 2: A debounced pulse generator in Sslang. The *sleep* function pauses execution; *waitfor* pauses until a variable takes the specified value; *debounce* filters a bouncy pushbutton input into clean press events; *pulse* emits a pulse at each press event; *buttonpulse* runs *debounce* and *pulse* in parallel.

The *buttonpulse* function, the main entry point to our program, runs *debounce* and *pulse* together. It creates a pure-event variable *press* to convey clean button-press events between *debounce* and *pulse*, which are run in parallel using the **`par`** statement. **`par`** runs earlier operands at a higher priority than later operands, ensuring that an event generated by *debounce* is seen instantly by *pulse*.

Following the techniques of Hui and Edwards [7], our Sslang compiler generates C code that links against the SSM runtime [5], a discrete-event simulator that provides a *tick* function to execute the system for an instant, updating the event queue (a priority heap) as needed. The SSM runtime library is platform-agnostic and requires a timing-aware platform runtime to call *tick* at the right time. The platform runtime is also responsible for managing variables mapped to external I/O, scheduling external inputs as delayed assignments to input variables, and fowarding output variables updates to the environment. Our RP2040 platform runtime does so using timestamp peripherals, which we describe below in detail.



(a)



(b)

Figure 3: Behavior of the debounced pulse generator program. The top trace (blue) is the active-low pushbutton input; the bottom (cyan) is the LED output. (a) The program emits two 200 ms pulses in response to two button presses. (b) Zoomed in, button bounces and the 20 µs reaction time become visible.

3 THE RP2040 AND ITS PIO BLOCKS

The RP2040 microcontroller, produced by the Raspberry Pi Foundation [19], features dual ARM Cortex-M0+ cores, 264 kB of on-chip SRAM, a 64-bit counter/timer with 1 µs precision, a QSPI interface for off-chip flash memory backed by an execute-in-place cache, and a direct memory access (DMA) controller. We run the core processors at 128 MHz, clocked by a 12 MHz crystal-driven PLL.

In addition to traditional peripherals such as GPIO and UARTs, the RP2040 includes Programmable I/O (PIO) devices that execute tiny PIO-specific assembly language programs designed to act as conduits between the RP2040's 30 GPIO pins and its ARM cores. Typical applications include "soft UARTs" and drivers for the unusual serial protocol used by WS2812 color LEDs. The RP2040 provides two such PIO blocks, each comprised of four independent state machines (SMs) that each have their own program counter, two 32-bit shift registers, and two 32-bit scratch registers. While these 8 SMs provide ample parallelism, program memory is limited to 32 instructions per PIO block shared among four SMs, they do not have direct access to any memory, and the only arithmetic operations they support are decrement and equality comparison.

The SMs provide precise timing by guaranteeing each non-blocking instruction executes in a single cycle, followed by a fixed number of stall cycles prescribed in the instruction itself. Blocking instructions wait on events such as an inter-SM interrupt signal or data arriving from the ARM cores. As such, programs on two SMs execute in lockstep if neither block and their instruction counts align (accounting for stall cycles). PIO devices send and receive data from the ARM cores via two 4x32-bit FIFOs (one in each direction) and may raise interrupts that can run interrupt service routines (ISRs) on either of the two ARM cores.

Although the PIO was not designed for implementing timestamp peripherals, it is fast, predictable, and powerful enough to do so. To timestamp inputs, we implement a precisely timed loop that maintains a counter and emits a timestamped input update event when it sees a change in input levels. The output system also implements a counter with a precisely timed loop, but checks the counter against an alarm time to emit a new output when the two timestamps match.

4 THE RP2040 PLATFORM RUNTIME

To run Sslang programs in real-time, our RP2040 platform runtime uses Edwards & Hui's platform-agnostic SSM runtime library to schedule internal events and processes. The RP2040 platform runtime coordinates execution with the hardware timer and relays external inputs and outputs to the SSM runtime.

The tick loop procedure for our RP2040 platform runtime, shown in Figure 4, keeps up with physical time while calling the SSM runtime's *tick* function to execute the Sslang program for an instant and advancing logical time. We based this on Hui & Edwards [7]. At each iteration, the platform runtime checks for available input events that may have preempted internally-scheduled events, and forwards these as delayed assignments to the corresponding scheduled variables. If there is nothing to be done, the tick loop sleeps until it is time to execute the next SSM instant, or when some external input wakes up the system: it blocks on a semaphore until it is unblocked by an interrupt service routine (ISR).

The block diagram in Figure 5 illustrates how the tick loop communicates with the rest of the system. Notably, our platform runtime uses the RP2040's cycle-accurate PIO hardware to predictably manage external input and output events, isolated from main processor delays. The input system uses a single SM that timestamps input events; the output system uses two SMs that emit output events at target timestamps specified by the running program. In the rest of this section, we describe our PIO input and output implementation, and how we integrate them with the rest of the system.

4.1 Timestamps and Clock Synchronization

Our RP2040 platform runtime uses 64-bit timestamps (`time_t`) that count at 16 MHz (62.5 ns), which we chose due to our PIO programs running 8-cycle loops at 128 MHz and the RP2040 system timer running at 1 MHz. At this speed, 32-bit timestamps would wrap around in under 5 min; 64-bit timestamps give us 36,533 years.

We use the RP2040's 1 MHz system timer as the master clock, which measures time since it was started. We plan to eventually synchronize this clock to, say, a GPS reference.

```
procedure tick_loop(invar, outvar):                  // The main tick loop, with PIO input and output variables
    init_ssm_runtime()                                // Initialize the SSM runtime
    tick()                                            // Run the program for time zero
    forever
        rt ← timer_read()                             // Read the real time from the system timer
        nt ← next_time()                              // Get the time of the next scheduled event
        if input_queued() && input_peek().time < nt   // Is there a pending input event before any other event?
            schedule(invar, input_dequeue())          // ... yes: move it from the PIO queue to the SSM runtime queue
        elseif nt ≤ rt                                // Has the model fallen behind physical time?
            tick()                                    // ... yes: run the program for an instant; update next time
            if outvar.next_time ≠ ∞                   // Is there a scheduled PIO output?
                pio_output(outvar.next_time,          // ... yes: send it to the PIO
                           outvar.next_value)
        elseif nt ≠ ∞                                 // Is there an event scheduled for the future?
            set_alarm(nt)                             // ... yes: schedule an alarm to wake up then
            wait(semaphore)                           // Wait for the alarm or an input event
            cancel_alarm()                            // If an input event awakened us, cancel the alarm
            release(semaphore)                        // Release the semaphore if an alarm came just after an input event
        else
            wait(semaphore)                           // Wait for an input event
```

Figure 4: The RP2040 platform runtime tick loop, which calls *tick*() to advance model time, then sleeps until the next scheduled event or external input. Based on the tick loop from Hui & Edwards [7].

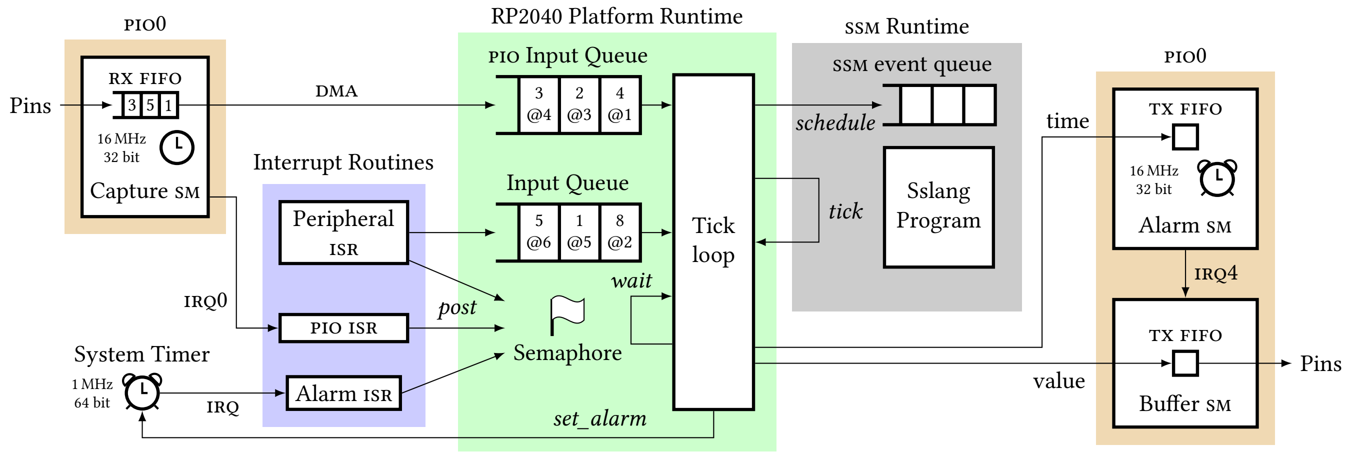


Figure 5: System block diagram. The Capture SM (in PIO0) timestamps input pin events; the DMA controller enqueues them. The tick loop (Figure 4) gathers the next event from the input queues, schedules it in the SSM event queue, calls *tick* to run the Sslang program for an instant, feeds updated time/value to Alarm and Buffer SMs (also in PIO0), sets an alarm, and sleeps.

Because the PIO programs cannot directly read the system timer, we maintain two additional real-time clocks in the PIO programs that need access to the current time. Fortunately, all three timers are driven by clocks derived from the external 12 MHz crystal, so we set them to run at precisely the same rate. They will remain synchronized provided we start them in phase. We initialize the PIO counters with the code in Figure 6, which reads the system timer, sends the initial count value to the counting SMs, and starts all the three SMs simultaneously.

The initialization routine compensates for its own latency, which we measured to be roughly 3 µs. We add this offset to the initial PIO counter to ensure it runs slightly *ahead* of the system clock. This offset is critical for the correctness of the tick loop, which assumes that if the PIO input queue is empty, future queued events will have a greater timestamp than the current system clock time. If the PIO counters were run *behind* the system clock, PIO timestamps could be smaller, violating this assumption. We verify our clocks are synchronized using the loopback test described in Section 5.4.

```
void start_pio_counters(void) {
  // Read the 1 MHz system timer
  uint32_t tmr = timer_hw->timerawl;
  // Convert to 16 MHz countdown value
  uint32_t ctr = ~((tmr + 3) << 4);

  // Send initial count to input and output SMs
  pio_sm_put(pio0, capture_sm);
  pio_sm_put(pio0, alarm_sm);

  pio_set_sm_mask_enabled(pio0, // Start SMs in sync
      CAPTURE_SM | ALARM_SM | BUFFER_SM, true);
}
```

Figure 6: Initializing PIO counters using the system timer.

4.2 The Input System

The input system uses a single PIO SM to sample a group of input pins at 16 MHz and send a sequence of timestamped changes to the platform runtime. This Capture SM is conceptually simple: it reads an initial counter value from the CPU to synchronize with the system timer, then enters a loop that increments the counter and polls the input pins. If any input pin state has changed, the Capture SM emits the new pin values and current counter value into a FIFO, and interrupts the CPU to notify it of the input event.

The actual PIO code for this (Figure 7) is complicated because the PIO instruction set is highly idiosyncratic. For example, only the two scratch registers X and Y can be compared, and decrement can only be done as part of a conditional jump. To compensate for this limitation, we complement PIO counter values when we convert them to and from the system timestamps that SSM uses (Figure 8).

We have tuned our PIO code so that the counter decrements every eight cycles regardless of any input change, keeping the counter synchronized with the system timer. We run the PIO at 128 MHz, so our code samples and timestamps inputs at 16 MHz. This frequency is a power-of-two multiple of the 1 MHz system clock frequency, which lets us efficiently convert between the time bases with bit-shifting (Figure 9).

While our implementation samples inputs at 16 MHz, it cannot resolve *consecutive* events occurring faster than 8 MHz: when the Capture SM detects an input event, it takes extra instructions to send the captured event to the CPU. We pad these instructions to eight cycles to keep the counter decrementing at a constant rate.

To allow our system to handle longer input event bursts, we program a channel of the RP2040's DMA controller to empty the 4-word hardware RX FIFO from the Capture SM into a 64-word ring buffer in main memory. We leverage the controller's built-in support for power-of-two-sized ring buffers, and use a trick to make the transfer continue indefinitely: a second channel, configured to start the moment the first channel completes, restarts the first channel.

4.3 The Output System

The output system allows the Sslang program to schedule a single new value to be placed on the output pins at a specific 16 MHz timestamp in the future. It does so with two PIO SMs: the Alarm SM acts as a real-time alarm that triggers the Buffer SM to emit a new value on the pins at the scheduled time. This split arose because a Sslang program can "change its mind" about when and which outputs need to be emitted. SSM semantics allow only one pending event per variable, but allows that pending event to be overwritten, which is useful, say, when handling timeout behavior. As such, we needed the output system to be able to reschedule an alarm and the value to be written at that time, and the PIO's compulsory per-SM FIFOs were getting in the way.

Figure 10 shows the code for the Alarm SM: after reading an initial counter value to synchronize with the system timer, it enters an 8-cycle loop, which we padded to operate at the same frequency as the input Capture loop. Each loop iteration, the Alarm SM checks for an updated alarm target before decrementing the counter and

```
.program input_capture
; X:    The previous GPIO pins' values
; Y:    Counter/timestamp value (decreasing)
; ISR:  For reading GPIO pins
; OSR: Scratch
  pull                      ; Read initial counter value
  mov  y, osr
  in   pins, IN_PINS        ; Read initial GPIO pin state
  in   null, 32-IN_PINS    ; Pad unused pin bits
  mov  x, isr [13]          ; Save initial GPIO pin state

.wrap_target
cmp:
  jmp  y--, decr            ; Decrement counter
decr:
  mov  osr, y               ; Back up counter value
  in   pins, IN_PINS        ; Read GPIO pin state
  in   pins, 32-IN_PINS    ; Pad unused pin bits
  mov  y, isr
  jmp  x!=y, event          ; Jump if state changed
  mov  y, osr               ; Restore counter value
  jmp  cmp                  ; Restart the loop

event:                      ; Input value change: send event
  mov  x, isr               ; Remember new value
  push noblock              ; ...and enqueue it
  mov  isr, osr             ; Enqueue the current counter
  push noblock
  irq  0                    ; Notify CPU of the event
  mov  y, osr               ; Restore counter
  jmp  y--, cmp [3]         ; Decrement counter, stall 3 cycles
.wrap                       ; Restart the loop

void pio_irq0_isr(void) {
  sem_post(&sem); // Post to semaphore; awaken tick loop
}
```

Figure 7: Input Capture PIO program and the ISR triggered by `irq 0`. Every 8 cycles, this checks the input pins and, if they have changed, pushes the new value and the current time to the RX FIFO.

```
uint32_t time_to_pio(time_t t) {
  return ~(uint32_t) t;  // PIO counter decrements
}

time_t pio_to_time(uint32_t ctr) {
  time_t himask = ~(((time_t) 1 << 32) - 1);
  time_t rt = read_timer();   // System timer
  time_t hi = rt & himask;    // Get top 32 bits
  time_t lo = (time_t) ~ctr;  // Counter decrements
  if (lo & (1 << 31) && !(rt & (1 << 31))) {
    hi -= (time_t) 1 << 32;   // Correct near epoch
  }
  return hi | lo;
}
```

Figure 8: Conversion between 64-bit SSM time and the PIO's 32-bit 16 MHz decrementing counters. We take the top 32 bits from the system timer, being cautious around 32-bit epochs.

```
time_t read_timer(void) {
  uint32_t lo = timer_hw->timelr;  // Latches timehr
  uint32_t hi = timer_hw->timehr;
  uint64_t us = ((uint64_t) hi << 32) | lo;
  return us << 4;   // Convert 1 MHz timer to 16 MHz
}

void set_alarm(time_t t) {  // Convert to 1 MHz timer
  timer_hw->alarm[ALARM_NUM] = t >> 4;
}
```

Figure 9: Translation between SSM time and the RP2040's 64-bit 1 MHz timer. Reading the lower 32 bits from the *timelr* register latches the upper 32 bits, to avoid a wraparound race.

comparing it against the current alarm target, sending an interrupt to the Buffer SM when the alarm target matches the counter. The Buffer SM (Figure 11) program blocks on IRQ4 before sending data to the output pins (IRQ4 is only visible within the PIO block where the Output and Buffer SMs reside, and used for synchronizing SMs).

The main processor changes the Alarm target and the Buffer data by writing to their respective FIFOs, as shown in Figure 12. Because the Buffer SM does not poll its FIFO like the Alarm SM, the main processor injects a *pull* instruction to force the Buffer SM to read the new data from its FIFO. If there is not enough time to set up an Alarm-triggered output, it injects an *out* instruction to directly emit the output at the expense of precise timing. This happens when a Sslang program makes an instantaneous assignment to the output variable, or when it schedules a delayed assignment for too soon.

5 EXPERIMENTAL RESULTS

We run Sslang programs to evaluate our RP2040 platform runtime's input and output performance. We find that our timestamp peripherals provide 62.5 ns timing precision on both input and outputs, but that the reaction time (input-to-output latency) of the RP2040 platform runtime is at least 13 µs.

```
.program output_alarm
; X:   The alarm counter value
; Y:   Counter/timestamp value (decreasing)
; OSR: Reads the TX FIFO for a new alarm value
  pull noblock           ; Read initial counter value
  mov  y, osr

continue:
  nop                    ; Stall for 1 cycle

.wrap_target
  jmp  y--, decr [3]     ; Decrement counter, stall 3 cycles
decr:
  pull noblock           ; Read the new alarm value, if any
  mov  x, osr            ; (OSR reads X on TX FIFO empty)
  jmp  x!=y, continue    ; Loop again if alarm not reached
  irq  4                 ; Interrupt output buffer
.wrap                    ; Loop again
```

Figure 10: Output Alarm PIO program. Every 8 cycles, this decrements a counter, reads a new alarm value if the CPU has written one, and sends an interrupt to the Output Buffer PIO program if the counter matches the alarm.

```
.program output_buffer
; ISR:  Reads the initial GPIO state
; OSR: Holds output to GPIO
  in pins, 32            ; Read current GPIO state
  mov osr, isr           ; as the default output value
.wrap_target
  wait 1 irq 4           ; Wait for IRQ from Alarm SM
  out pins, 32           ; Write OSR to GPIO pins
.wrap                    ; Loop again
```

Figure 11: Output Buffer PIO program.

5.1 Signal Generator a.k.a. Blink

Our platform runtime is able to "tick" in as little as 13 µs, with the following variant of the *blink* program from Section 2:

```
loop  // The highest frequency Sslang blink program on RP2040
  after us 13, led <- 1 - deref led
  wait led
```

This generates a 38.46 kHz square wave. Its speed is limited by interrupt latency, the time it takes for the interrupt service routine to post to the semaphore, the time for the main tick loop to acquire the semaphore, check the input queues, tick for an instant, and schedule a future update with the Alarm and Buffer SMs. The input system goes unused here.

5.2 Pulse Timer

To test the precision of input timestamps and our system's response to high input loads, we use the program in Figure 13, which measures and reports the width of input pulses. Two parallel processes measure pulse widths and samples the results once a second.

```
void sched_pio_out(time_t t, uint32_t v) {
  // Enqueue new buffer output value
  pio_sm_put(pio0, buffer_sm, v);

  // Make the SM read this output value: inject a pull instruction
  pio_sm_exec(pio0, buffer_sm,
              pio_encode_pull(0, 1));

  if (t < read_timer() + OUTPUT_MARGIN)
    // Deadline too close: immediately send the output value
    // by injecting an out instruction
    pio_sm_exec(pio0, buffer_sm,
                pio_encode_out(pio_pins, 32));
  else {
    // Set Alarm time
    uint32_t tgt = time_to_pio(t);
    pio_sm_put(pio0, alarm_sm, tgt);
  }
}
```

Figure 12: Function that schedules a new value/time event on the output buffer and alarm programs

```
pulsewidth input =
  let result = new 0
  par loop
        wait input          // Wait for rising edge
        let b = now ()
        wait input          // Wait for falling edge
        let a = now ()
        result <- a - b     // Compute pulse width
      loop
        sleep (ms 1000)     // Pause between logging
        log_pwm (deref result)
```

Figure 13: A Sslang program to measure pulse width. Note that the *now* calls return model time, not wall-clock time.

Table 1: Pulse widths (in clock cycles) reported by the pulsewidth program

Pulse Input	Expected	Observed	Jitter	Error
80 ms	1 280 000	1 280 021	1	22
8 ms	128 000	128 002	1	3
800 µs	12 800	12 800	1	1
80 µs	1 280	1 280	1	1
8 µs	128	128	1	1
800 ns	12.8	13	1	0.2
80 ns	1.28	2		0.72
40 ns	0.64	2		1.36

Experimental data for the Sslang pulse timer program in Figure 13. Units are 16 MHz clock cycles (62.5 ns). We attribute the 17 ppm error in the 80 ms measurement to the crystal oscillator.

```
freqcount input =
  let count = new 0
      gate  = new ()

  after ms 1000, gate <- ()
  loop
    if updated gate       // Was gate assigned just now?
      log_count (deref count)

      if updated input    // Was input assigned just now?
          count <- 1      // Yes: reinitialize count to 1
      else
          count <- 0      // No: reinitialize count to 0

      after ms 1000, gate <- ()
      wait gate           // Pause before counting again

      after ms 1000, gate <- ()
    else
      count <- deref count + 1
    wait gate || input    // Block until either is assigned
```

Figure 14: A frequency counter that reports the number of events an input variable each second, after Krook et al. [12]. The *updated* function returns true when the variable was assigned in the current instant.

We test this program with 10 kHz pulses of varying widths and record the difference in timestamps between pulse edges. Table 1 shows the results. We observe a single least-significant bit of jitter in all cases, likely an artifact of sampling. We attribute the roughly 20 ppm errors in the long-period measurements to the expected precision of the crystal oscillator.

While we do not expect correct results for pulses shorter than the 62.5 ns sampling period, we were pleased that the resulting behavior was not absurd. The input SM was still able to observe certain pulses and conclude that they were short.

When short pulses are applied above 200 kHz, we begin to observe sporadic but drastic measurement errors. For instance, with a 320 kHz pulse signal with a 500 ns pulse, the program occasionally reports 808 or 809 ticks instead of the expected 8 or 9. These errors are due to incoming input events accumulating faster than the program can process them, overflowing the input ring buffer. It takes 32 events—16 cycles of the pulse signal—to overflow a 256 B ring buffer; at 320 kHz, 16 cycles is 50 µs, accounting for the extra 800 ticks we observe.

5.3 Frequency Counter

To further assess our RP2040 runtime's ability under high input load, we implement the frequency counter from Krook et al. [12] in Sslang, shown in Figure 14. This program measures the frequency of a signal by counting the number of events that appear on an input variable every second.

Table 2: Events observed by the frequency counter

Frequency	Expected Events	Observed Events
30 kHz	60000	60000
40 kHz	80000	74271
50 kHz	100000	72670
60 kHz	120000	71390
70 kHz	140000	70013
80 kHz	160000	68574
>90 kHz	180000	unstable

Experimental data for the frequency counter program in Figure 14.

Like Krook et al., we subject our frequency counter to a square wave signal that produces twice the number of events as the frequency of the signal (one each for the rising and falling edge). We are able to reliably measure frequencies below 37 kHz (74000 events), with only 1 Hz of error due to sampling artifacts. Beyond this "reliability ceiling," the program remains responsive, though it logs lower event counts than expected; for instance, at 50 kHz, the program consistently counts 71390 instead of 100000. Table 2 shows our observations.

Above 90 kHz, the frequency counter's event counts are no longer stable, though they continue to decrease as we push the input frequency ever higher. We find that the program continues to respond up to 740 kHz, albeit completely inaccurately. Above that, the program freezes, as the processor spends all of its time thrashing within the PIO ISR without any opportunity to make any progress with the user program.

These results show our platform runtime outperforms Scoria's, whose frequency counter could only handle frequencies below 14.5 kHz, and would freeze above that [12]. Our result comes despite the fact that our runtime only uses a 256 B ring buffer; Scoria used 4096 B. Part of Scoria's degraded performance is due to its use of Zephyr RTOS's ISR and device abstractions, which Krook et al. show produce considerable overhead. The RP2040's processor also runs twice as fast as the 64 MHz Cortex-M4 on the NRF52840-DK used by Krook et al. However, we believe the RP2040 runtime remains responsive for workloads far beyond its 37 kHz reliability ceiling chiefly because the responsibility to timestamp events is delegated to the PIO hardware, rather than in software. Our input ISR merely posts to the semaphore to wake up the main thread.

SSM was not designed for throughput; as its name suggests, it is best on sparse events, but it also performs well on input bursts. For example, we found that with a 256 B ring buffer, it could successfully handle 3 MHz bursts of 28 events with no loss of accuracy because the DMA controller could buffer them all before the software had to start processing them. A larger buffer would handle longer bursts.

5.4 Loopback

To compare the physical timing behavior of Sslang programs with their logical behavior, we test our system with a loopback connection running the program in Figure 15. This schedules a delayed assignment to the *output* pin, and awaits events on the *input* pin. Externally, we connect the *output* to the *input* pin, meaning we indirectly measure the timing of the delayed *output* assignment using the *input* variable. *loopback* should behave the same whether *input* and *output* refer to shorted external GPIO pins or the same internal scheduled variable.

```
loopback d input output =
  loop
    let b = now ()
    after d, output <- 1 - deref output
    wait input             // Should be updated after d
    let a = now ()
    log_latency (a - b)  // Measure actual latency
    sleep (ms 500)         // Pause between rounds
```

Figure 15: A loopback program, tested with the `input` and `output` pins connected externally.

```
// Triggered on rising edge of INPUT_PIN
void gpio_rise_isr(void) {
  gpio_put(OUTPUT_PIN, 1);
  busy_wait_us(100);
  gpio_put(OUTPUT_PIN, 0);
}
```

Figure 16: The reactive 100 µs pulse generator program in C. Additional code configures the GPIO to generate an interrupt that runs this code (not shown).

We find that the latency this program logs is exactly equal to the prescribed delay *d* when *d* is above 17 µs, comparable to the speed of the fastest *blink* program reported above.

In earlier runs of this experiment, we observed that the measured latency consistently lagged 1 tick behind the prescribed latency. This error arises when the input and output SMs' synchronous loops are not correctly phase-aligned, leading the input SM to sample the GPIO pin before the output SM writes the pin. We fixed this lag by starting the input SM a few instructions later to put it in phase.

The loopback test is also useful for detecting when the system clock is not correctly synchronized with the PIO counters, which we used to determine the 3 µs offset applied during the SM initialization procedure (see Section 4.1). Because the delayed assignment to *output* takes place in the same instant as the event on *input*, a poorly calibrated system clock—running ahead of the PIO timers—would lead the tick loop to execute the instant before waiting long enough for the *input* event to show up. When the tick loop tries to schedule the late-arriving input event in a later iteration, the SSM runtime complains that it cannot schedule a delayed assignment for an instant it has already executed, and throws a runtime error.

5.5 Reactive 100 µs Pulse Generator

To compare the performance of our approach with a more traditional C program, we wrote a reactive 100 µs pulse generator in C (Figure 16) and in Sslang (Figure 17). The program attempts to match a 100 µs input pulse by "immediately" setting the output high upon seeing input, then setting the output low after 100 µs.

Our measurements (Figure 18) show the C program reacts faster at first, but the Sslang falling edge is more accurate. The C program

```
pulse input output =
  loop
    wait input
    if deref input == 1
      output <- 1
      after us 100, output <- 0
```

Figure 17: The reactive 100 µs pulse generator in Sslang.

has a shorter reaction time (1.96 µs vs. 13.8 µs; see Figure 18b) because it eliminates most software overhead by performing all work in an input-triggered ISR. The Sslang program times the falling edge significantly better because of our PIO timestamp peripherals. The output of the C program is 1.43 µs–2.39 µs late (Figure 18c) because of the initial latency and imprecision in the busy wait loop, which polls the system timer. the Sslang program's falling output is 0 ns–62.4 ns late; that jitter in Figure 18d is purely due to phase differences between the PIO's 16 MHz sampling clock and the frequency generator's oscillator.

The Sslang output system uses two strategies to transmit output variable assignments to the environment: for sufficiently later assignments, the output system sets the Alarm SM's target counter to trigger the Buffer SM when the event is scheduled for; for instantaneous assignments, shorter delays, and when the system is running behind, the output system instructs the Buffer SM to immediately emit the event, rather than risk missing the output deadline while programming the Alarm SM. The result is high-precision output timing when possible, and best-effort timing otherwise.

6 RELATED WORK

6.1 Synchronous Software on Real Hardware

Our RP2040 runtime is not the first implementation of the Sparse Synchronous Model on real hardware; Krook et al. previously developed Zephyr bindings for Edwards & Hui's SSM runtime to run programs on an NRF52840-DK development board [7, 12]. In contrast to our work, Krook et al. implement the input and output timestamping in software: input event timestamps are captured during the GPIO interrupt service routine, and output event timing depends on when tick executes the output handler process. Though their approach does not require specialized hardware like the RP2040's PIO, their timestamps' accuracy is limited by the unpredictable latency of the interrupt handler. Our RP2040 platform runtime can capture and emit events far more reliably, as demonstrated by our pulse generator experiment in Section 5.5. Our approach supports Scoria-like non-timestamp peripherals alongside timestamp GPIO.

Other synchronous, discrete-event programming models have also been implemented on real hardware. Jellum et al. [11] implement an embedded target for Lingua Franca, a polyglot coordination language that supports event-driven execution like SSM [14]. Like Scoria, Jellum et al.'s embedded target is based on Zephyr RTOS and manages timestamps in software. Their square wave generator's 1 µs sleep-induced jitter appears consistent with that of our C implementation for the reactive pulse generator, and their 28 µs/35 µs input/output latency reflects the kind of error we eliminate using dedicated PIO hardware.

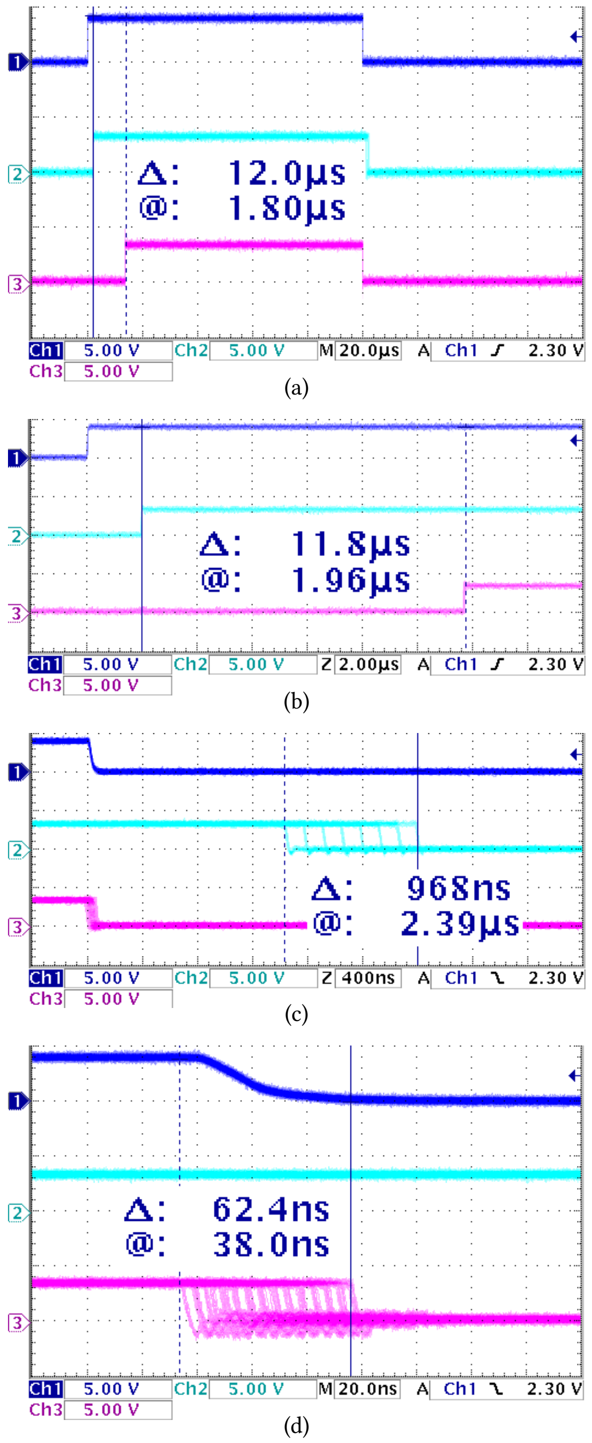


Figure 18: Behavior of the reactive 100 µs pulse generator. The top trace (blue) is the input; the middle (cyan) is the C program's output; the bottom (magenta) is from Sslang. (a) The C and Sslang programs try to match the 100 µs input pulse. (b) C responds faster (1.96 µs) than Sslang (13.8 µs). (c) The C program's falling edge is 1.43 µs–2.39 µs late, while (d) the Sslang program is at most 62.4 ns late.

Zou et al. [20–22] develop PtidyOS to execute programs implemented using the Ptides programming model, a precursor to Lingua Franca. PtidyOS features a preemptive EDF scheduler and runs on the Luminary microcontroller, though it uses the same software-based timestamping strategy as Scoria's runtime and does not appear to take advantage of the hardware's input capture features. To help developers account for this latency, Zou et al. show that that they can statically determine the schedulability of Ptides programs by annotating actors and input/output ports with worst-case latencies and simulating the execution, made possible by the fixed actor topology of Ptides programs. Sslang trades the ability to do such analysis for a more expressive programming model.

6.2 Timestamp Peripherals

Certain microcontrollers' peripherals are capable of a primitive form of timestamping termed input capture and output compare. For example, Atmel ATmega328P microcontrollers [1] include input capture units that sample a 16-bit timer on the rising or falling edge of a single input. The Microchip PIC32 family of microcontrollers [16] possess similar functionality with a 32-bit timer and also include an output compare device that can raise, lower, or toggle an output pin when the timer matches a target timestamp. These facilities are geared toward the measurement and generation of PWM signals, which are highly periodic and not bursty.

We chose to implement our SSM runtime on the RP2040 rather than on an ATmega or PIC32 device because we wanted to take advantage of the RP2040's 64-bit timer. Unlike the ATmega328P and PIC32, our timestamp peripherals are implemented using the RP2040's PIO device, and support reading from and writing to multiple consecutive GPIO pins at the same time.

Timestamping hardware devices also exist for specific applications. For instance, the IEEE's Time-Sensitive Networking protocols [8, 9] ensure deterministic networking between devices synchronized using the Precision Timing Protocol. These devices work by timestamping network packets; Austad and Mathisen [2] show that this capability is useful for minimizing network-induced jitter for distributed Lingua Franca programs.

Certain Nordic Semiconductor SoCs, such as the NRF52 series [17], include a "programmable peripheral interconnect" system that can configure a timer to timestamp and schedule events on arbitrary peripherals including single GPIO pins. This feature appears to enable timestamp peripherals, but we are unaware of any implementations.

6.3 Timing-Predictable Hardware

Rather than rely on peripherals for precise timing, Precision Timed (PRET) machine architectures ensure predictable for the main processor [6]. This approach typically sacrifices single-threaded performance in favor of highly parallel real-time workloads that benefit from numerous timing-predictable cores. Jellum et al. [10] propose InterPRET as a hardware architecture for running Lingua Franca programs. Their architecture is comparable to XMOS's XCore architecture [15], a commercial PRET machine.

Other approaches offload time-sensitive computation to timing-predictable co-processors. For instance, Vicuna [18] is a co-processor designed for massively parallel workloads. Meanwhile, the Beaglebone family of development boards [3] feature timing-predictable Programmable Real-time Unit (PRU) co-processors that execute alongside the Beaglebone's desktop-class processors.

Such PRET processors are currently much more expensive than the RP2040 we used in this work, and it is unclear how precise timing (as opposed to predictable) can be achieved on these machines.

The RP2040's PIO blocks are technically PRET machines (their parallel SMs even appear to be implemented with an interleaved pipeline), but their lack of memory access and most arithmetic operations make them far more limited than other PRET machines.

7 CONCLUSIONS

This work shows how software can achieve high timing precision through access to peripherals that can timestamp input events and schedule timestamped output events. We demonstrated a system running on the RP2040, an inexpensive, commodity microcontroller, able to achieve 62.5 ns precision on both input and output, although minimum reaction time is in the 13 µs range. We implemented the input and output systems as precisely timed programs running on the RP2040's novel PIO system, but similar results could be achieved with peripherals implemented in an FPGA or directly on the processor chip.

Although the RP2040 has a 64-bit 1 MHz system timer designed to be a master time base, limitations of the PIO system forced us to implement separate clocks within the PIO devices, which provided higher timing precision (these clocks run at 16 MHz) as well as clock synchronization headaches. While the system clock and the PIO clocks run off the same crystal oscillator, it was very important to start them in sync and in phase so that the peripheral timestamps did not "time travel" and cause unexpected behavior. This confirmed to us that synchronized clocks are key to implementing the Sparse Synchronous Model.

An early plan for the output system had it consuming a sequence of time-value pairs from a FIFO, but this proved unworkable since SSM semantics allows a scheduled output event to be replaced with an earlier event. While the SSM runtime handles this with a heap that supports re-insertion, implementing such a data structure with a PIO is impractical. This led us to the simpler mechanism presented above: separate time and value "registers" that can be overwritten when preemption is needed. The disadvantage of this approach is that the software runtime needs to perform a separate action for each output event, even if the desired output sequence is known in advance and could be buffered. For future work, we plan to introduce non-preemptible events to Sslang for reducing software load, combined with a DMA-assisted output queue for more reliable and precise burst outputs.

While the Input SM clock and the RP2040's system timer are synchronized, there is a small but difficult-to-characterize latency between when an input event is observed (and timestamped) and when the DMA controller makes that event available to the main tick loop. The uncertainty arises from any DMA controller latency plus any interference from other bus traffic. While short, this latency raises the question of when the system can safely advance time past a certain point and be assured that no additional inputs will arrive before that point. Interestingly, this is exactly the problem that Zou et al. [22] considered for distributed systems, even though our system is not one that would traditionally be considered distributed.

REFERENCES

[1] Atmel 2015. *ATmega328P Datasheet*. Atmel, San Jose, California.

[2] Henrik Austad and Geir Mathisen. 2023. Bounding the End-to-End Execution Time in Distributed Real-Time Systems: Arguing the Case for Deterministic Networks in Lingua Franca. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023* (San Antonio, TX, USA) *(CPS-IoT Week '23)*. Association for Computing Machinery, New York, NY, USA, 343–348. https://doi.org/10.1145/3576914.3587499

[3] BeagleBoard Foundation 2023. *BeagleBone System Reference Manual*. BeagleBoard Foundation, Oakland Charter Township, Michigan.

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (Jan. 2003), 64–83.

[5] Stephen A. Edwards and John Hui. 2020. The Sparse Synchronous Model. In *Forum on Specification and Design Languages (FDL)*. Kiel, Germany. https://doi.org/10.1109/FDL50818.2020.9232938

[6] Stephen A. Edwards and Edward A. Lee. 2007. The Case for the Precision Timed (PRET) Machine. In *Proceedings of the 44th Design Automation Conference*. San Diego, California, 264–265.

[7] John Hui and Stephen A. Edwards. 2022. The Sparse Synchronous Model on Real Hardware. *ACM Transactions on Embedded Computing Systems* (Dec. 2022). https://doi.org/10.1145/3572920

[8] IEEE Computer Society 2019. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Computer Society, New York, New York. https://doi.org/10.1109/IEEESTD.2020.9120376

[9] IEEE Computer Society 2021. *IEEE Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems*. IEEE Computer Society, New York, New York. https://doi.org/10.1109/IEEESTD.2021.9653970

[10] Erling R. Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. 2023. InterPRET: A Time-Predictable Multicore Processor. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. San Antonio, TX, USA, 331–336. https://doi.org/10.1145/3576914.3587497

[11] Erling Rennemo Jellum, Shaokai Lin, Peter Donovan, Efsane Soyer, Fuzail Shakir, Torleiv Bryne, Milica Orlandic, Marten Lohstroh, and Edward A. Lee. 2023. Beyond the Threaded Programming Model on Real-Time Operating Systems. In *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023) (Open Access Series in Informatics (OASIcs), Vol. 108)*, Federico Terraneo and Daniele Cattaneo (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:13. https://doi.org/10.4230/OASIcs.NG-RES.2023.3

[12] Robert Krook, John Hui, Bo Joel Svensson, Stephen A. Edwards, and Koen Claessen. 2022. Creating a Language for Writing Real-Time Applications for the Internet of Things. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Shanghai, China.

[13] Marten Lohstroh, Edward A. Lee, Stephen A. Edwards, and David Broman. 2023. Logical Time for Reactive Software. In *Workshop on Time-Centric Reactive Software (TCRS)*. San Antonio, TX, USA, 313––318. https://doi.org/10.1145/3576914.3587494

[14] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems* 20, 4 (July 2021), 1–27. https://doi.org/10.1145/3448128

[15] David May. 2012. The XMOS Architecture and XS1 Chips. *IEEE Micro* 32, 6 (Nov. 2012), 28–37. https://doi.org/10.1109/MM.2012.87

[16] Microchip 2010. *PIC32 Family Reference Manual*. Microchip, Chandler, Arizona.

[17] Nordic Semiconductor ASA 2021. *NRF52840 Product Specification*. Nordic Semiconductor ASA. https://infocenter.nordicsemi.com/ v1.7.

[18] Michael Platzer and Peter Puschner. 2021. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 196)*, Björn B. Brandenburg (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:18. https://doi.org/10.4230/LIPIcs.ECRTS.2021.1

[19] Raspberry Pi Ltd 2023. *RP2040 Datasheet*. Raspberry Pi Ltd, Cambridge, England.

[20] Yang Zhao, Jie Liu, and Edward A. Lee. 2007. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *Proceedings of Real-Time Technology and Applications Symposium (RTAS)*. 259–268. https://doi.org/10.1109/RTAS.2007.5

[21] Jia Zou, Slobodan Matic, and Edward A. Lee. 2012. PtidyOS: A Lightweight Microkernel for Ptidesa Real-Time Systems. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012*. 209–218. http://chess.eecs.berkeley.edu/pubs/858.html

[22] Jia Zou, Slobodan Matic, Edward A. Lee, Thomas Huining Feng, and Patricia Derler. 2009. Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems. In *Proceedings of Real-Time Technology and Applications Symposium (RTAS)*. San Francisco, California, 77–86. https://doi.org/10.1109/RTAS.2009.39