Timestamp Peripherals for Precise Real-Time Programming

John Hui        Kyle J. Edwards        Stephen A. Edwards


COMPUTER SCIENCE AT COLUMBIA UNIVERSITY
RP2040

PIO

CPU w/ language runtime

Slang Program

PIO

Input Capture

Output Compare

100@7  011@5  01@9  10@6

100@7  011@5  01@9  10@6
What is the right programming model?

Programs should compose, but we have limited hardware timers.

Polling input (in software) is wasteful: events are bursty.

Timing prescriptions should not depend on device clock rate.
Slang at a glance

**Procedural** language with ML-like (**functional**) features:

- polymorphism, static type inference, first-class functions, automatic memory management

waitfor (var: &a) (val: a) =  // type: ∀a, &a -> a -> ()
while deref var != val  // current value is not val
    wait var  // suspend until next update

Extended with **synchronous** primitives: **after**, **wait**, **par** (more on this later)

All computation other than **wait** takes zero logical time
Slang at a glance

**Variables** (&) are mutable references

```
new : a -> &a  // construct variable from value
deref : &a -> a  // read current value of variable
_ <- _ : &a -> a -> ()  // update value of variable
```

Variables convey buffered **events** (value + timestamp), both internal and external
sleep (d: Time) =
    let timer = new ()  // construct event variable
    after d, timer <- () // schedule a wake-up event
    wait timer          // suspend until then
blink (press: &()) (led: &Led) =

loop
    wait press // block until button press
    led <- On // turn LED on immediately
    after ms 200, led <- Off // schedule LED off after 200ms
    wait led // block until LED turns off
debounce (button: &PushButton) (press: &()) =

loop

  waitfor button Pressed  // active-low button pressed
  press <- ()            // send “press” event
  sleep (ms 10)          // debounce press
  waitfor button Released // button released
  sleep (ms 10)          // debounce release
blink (press: &()) (led: &Led) =

  loop
  wait press // block until button press
  led <- On // turn LED on immediately
  after ms 200, led <- Off // schedule LED off after 200ms
  wait led // block until LED turns off

debounce (button: &PushButton) (press: &()) =

  loop
  waitfor button Pressed // active-low button pressed
  press <- () // send “press” event
  sleep (ms 10) // debounce press
  waitfor button Released // button released
  sleep (ms 10) // debounce release

main (button: &PushButton) (led: &Led) =

  let press = new ()
  par debounce button press press // run debounce and blink
  blink press led // in parallel
Input Capture

- Input pin
- From input capture
- Input FIFO
- Polled by CPU
Output Compare

From CPU

0

@8

value
timestamp

Polled by output compare

Output pin

1@4

0@8
PIO: Programmable I/O

On each PIO device:

- 4 “state machines”
- 32 instruction memory
- 9 op codes
- 4 registers
- Single-cycle execution

**Limited programmability**

Clocked using system clock, derived from external crystal oscillator
Timestamp Peripherals

System clock @ 128MHz / 8-cycle counter = PIO sample rate @ 16MHz
Experimental Goals

What is the **overhead** of processing events through this system?

10-20 us

What level of **accuracy** and **precision** can we achieve with timestamp peripherals?

62.5 ns / 16 MHz
Input (button)

Output (LED)

~20us reaction time
C: 1.80us reaction time

Sslang: 13.8us reaction time
\Delta: 62.6\text{ns} \approx 1/16\text{MHz}
## Pulse Width Measurement

<table>
<thead>
<tr>
<th>Pulse Input</th>
<th>Expected</th>
<th>Observed</th>
<th>Jitter</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 ms</td>
<td>1 280 000</td>
<td>1 280 021</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>8 ms</td>
<td>128 000</td>
<td>128 002</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>800 µs</td>
<td>12 800</td>
<td>12 800</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>80 µs</td>
<td>1 280</td>
<td>1 280</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8 µs</td>
<td>128</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>800 ns</td>
<td>12.8</td>
<td>13</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>80 ns</td>
<td>1.28</td>
<td>2</td>
<td>1</td>
<td>0.72</td>
</tr>
<tr>
<td>40 ns</td>
<td>0.64</td>
<td>2</td>
<td>1</td>
<td>1.36</td>
</tr>
</tbody>
</table>
# Frequency Counter

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Expected Events</th>
<th>Observed Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 kHz</td>
<td>60000</td>
<td>60000</td>
</tr>
<tr>
<td>40 kHz</td>
<td>80000</td>
<td>74271</td>
</tr>
<tr>
<td>50 kHz</td>
<td>100000</td>
<td>72670</td>
</tr>
<tr>
<td>60 kHz</td>
<td>120000</td>
<td>71390</td>
</tr>
<tr>
<td>70 kHz</td>
<td>140000</td>
<td>70013</td>
</tr>
<tr>
<td>80 kHz</td>
<td>160000</td>
<td>68574</td>
</tr>
<tr>
<td>&gt;90 kHz</td>
<td>180000</td>
<td>unstable</td>
</tr>
</tbody>
</table>
Timestamp peripherals enable precise timing behavior from expressive synchronous languages.

https://github.com/ssm-lang/sslang  
https://github.com/ssm-lang/pico-ssm