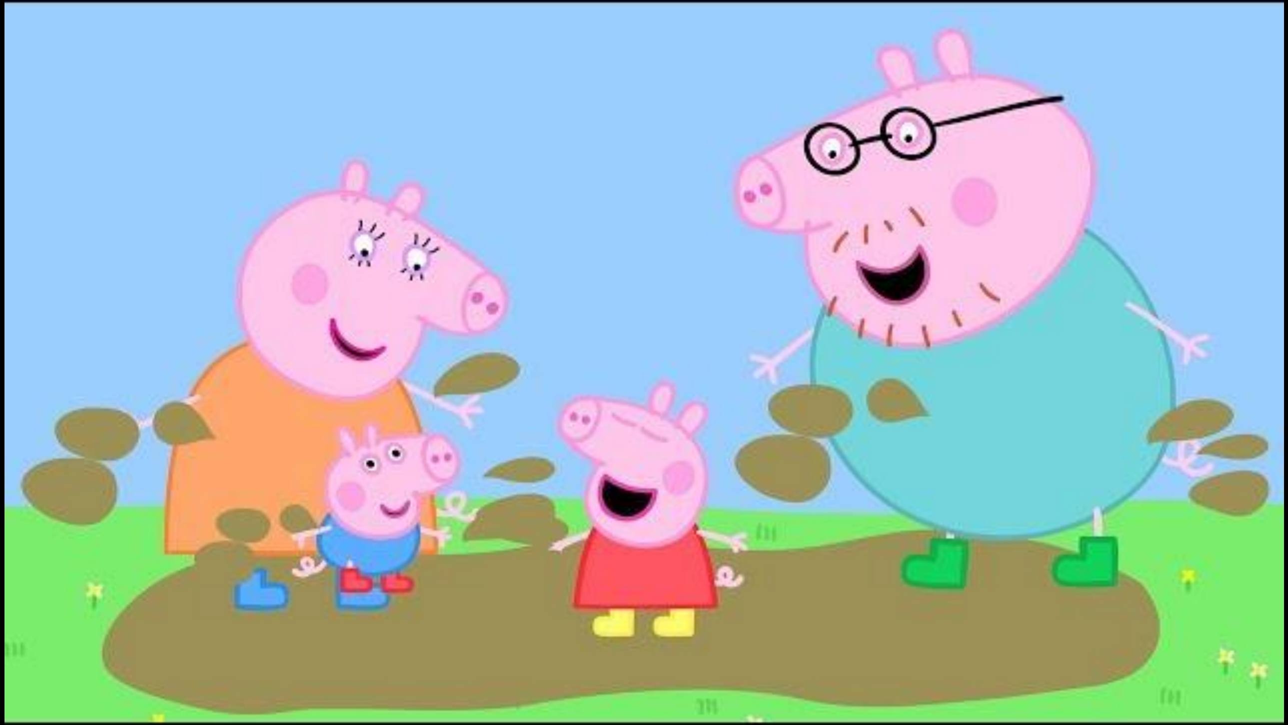
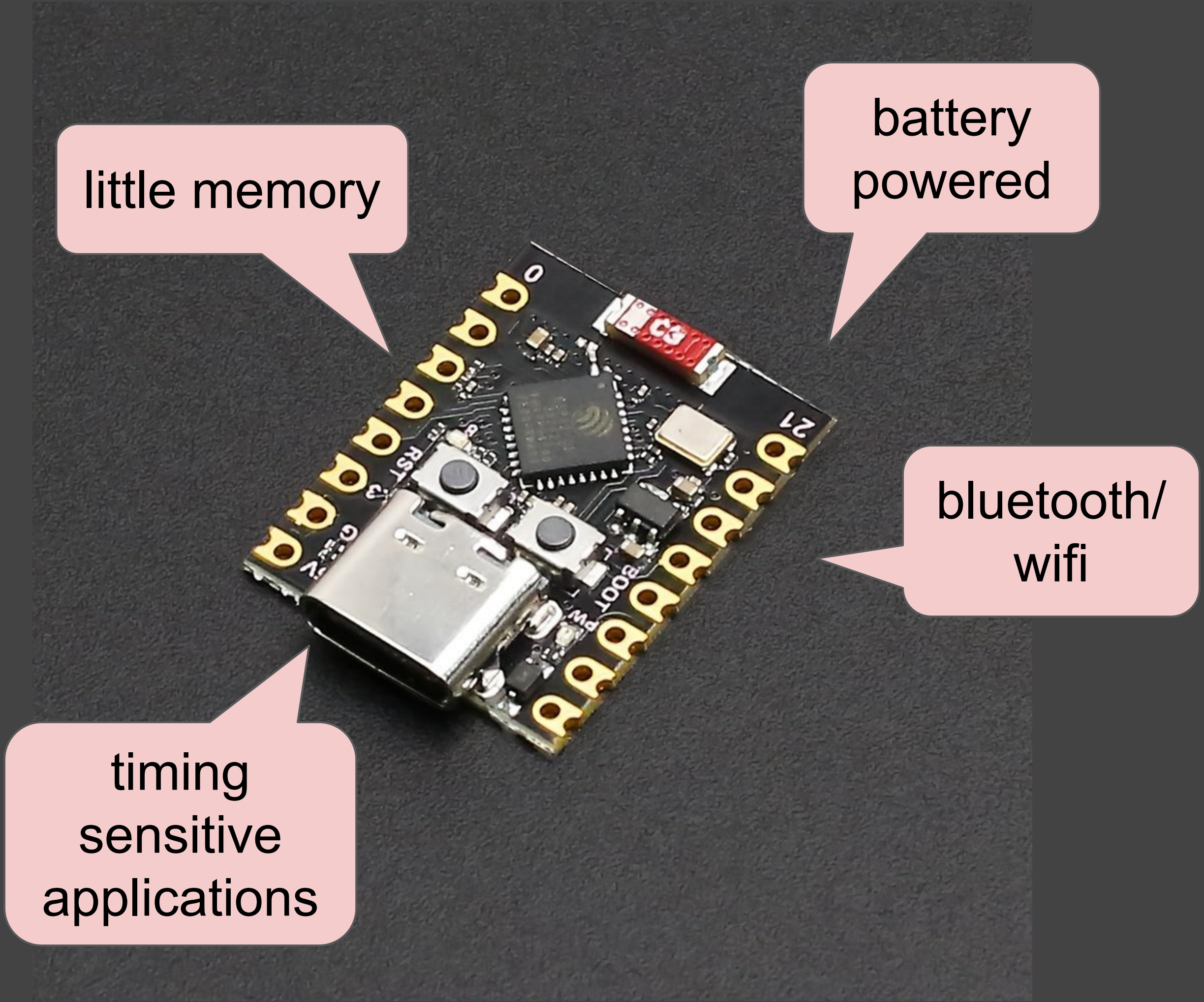


I 🧡

Lustre

FRP + Lustre = Fruste

Koen Claessen


little memory
battery powered
bluetooth/ wifi
timing sensitive applications

"Scoria"

MEMOCODE 2022

Creating a Language for Writing Real-Time Applications for the Internet of Things

Robert Krook*, John Hui†, Bo Joel Svensson*, Stephen A. Edwards†, and Koen Claessen*
*Chalmers University of Technology, Gothenburg, Sweden
†Columbia University, New York, USA
Email: krookr@chalmers.se, j-hui@cs.columbia.edu, joels@chalmers.se, sedwards@cs.columbia.edu, and koen@chalmers.se

imperative language

Sparse Synchronous Model

***Abstract*—… language Sco… real-time lar… (SSM), desig… C code that … not yet in a s… we carefully profile the timing behaviour and identify bottlenecks that can improve performance. The language and compiler are implemented as an Embedded Domain-Specific Language (EDSL) on top of Haskell.**

***Index Terms*—Real-time, IoT, Compilers, Embedded Domain-Specific Languages**

I. Introduction

Devices for the Internet of Things (IoT) typically contain hardware for sensors, actuators, and wireless communication, and often need to run on batteries whose life expectancy

```
… SSM ()
…f ?out0))
…> SSM ()
    else hperiod <~ max  (deref hperiod /. 2) 1)

entry :: (?ble :: BLE, ?out0 :: Ref GPIO) => SSM ()
entry = routine $ do
  hperiod <- var (time2ns (secs 1))
  fork [sigGen hperiod, remoteControl hperiod]
```

Scoria

```
summer :: Ref Int32 -> Ref Int32 -> SSM ()
summer diff sum =
  while true $ do
    wait diff
    sum <~ deref sum + deref diff
```

```
ticker :: Int32 -> Ref Int32 -> SSM ()
ticker n x = routine $ do
  after (s 1) x n
  wait x
  ticker (n+1) x
```

Sparse Synchronous Model

program is in rest by default

important to limit work during "reaction"

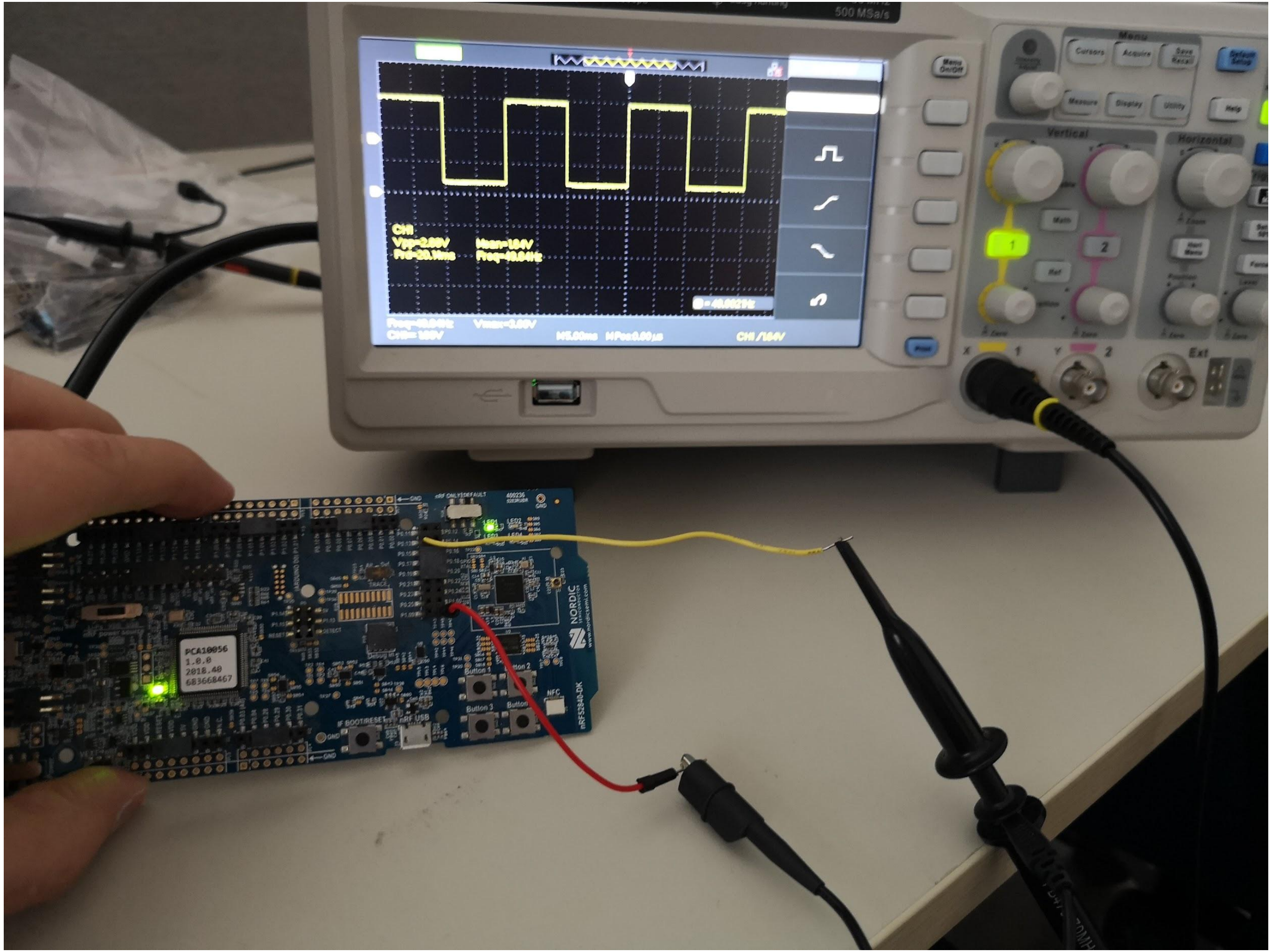
program reacts to external triggers

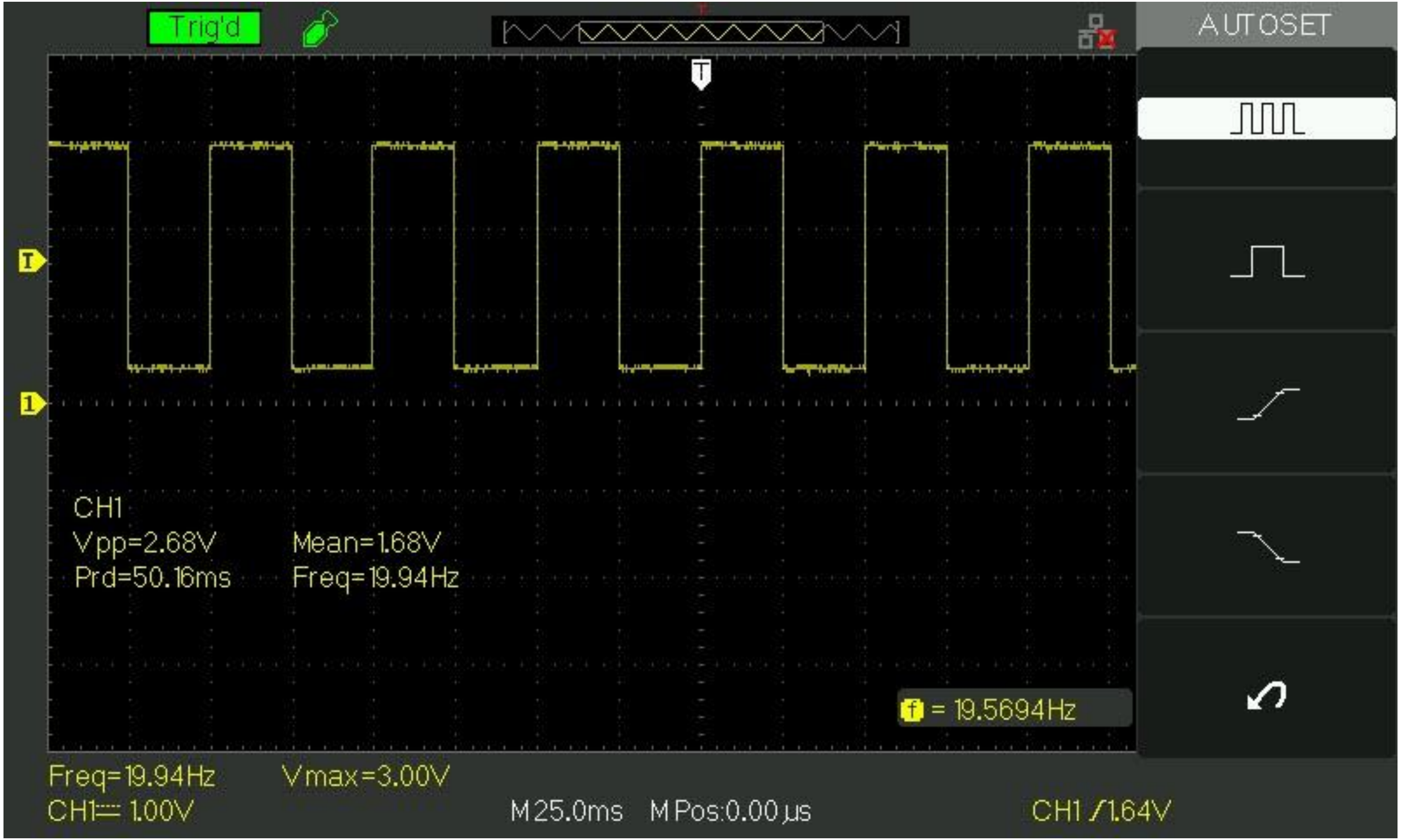
time does not pass while executing code

program can schedule events in the future

```c
#include <zephyr.h>
#include <drivers/gpio.h>

/* 1000 msec = 1 sec */
#define SLEEP_TIME_MS          50

static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(DT_ALIAS(led0), gpios);

void main(void)
{
    int ret;

    if (!device_is_ready(led.port)) {
        return;
    }

    ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_ACTIVE);
    if (ret < 0) {
        return;
    }

    while (1) {
        ret = gpio_pin_toggle_dt(&led);
        if (ret < 0) {
            return;
        }
        k_msleep(SLEEP_TIME_MS);
    }
}
```

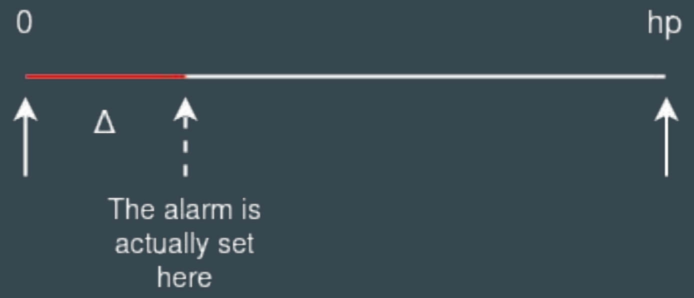
(Robert Krook)



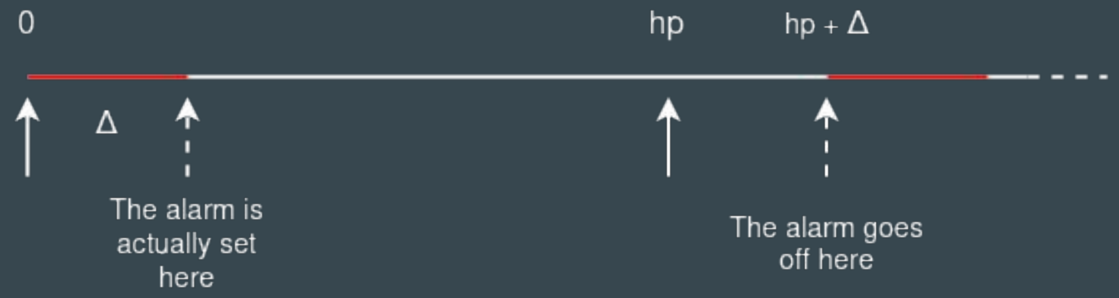


Trig'd
AUTOSET
CH1
Vpp=2.68V
Mean=1.68V
Prd=50.16ms
Freq=19.94Hz
f = 19.5694Hz
Freq=19.94Hz
Vmax=3.00V
CH1 1.00V
M 25.0ms
M Pos:0.00µs
CH1 1.64V


0
hp

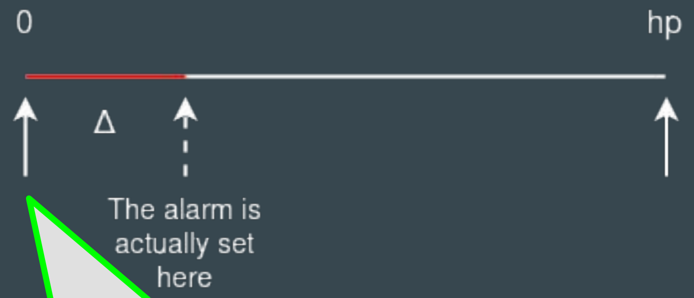

0
hp
Δ
The alarm is actually set here


0
hp
hp + Δ
Δ
The alarm is actually set here
The alarm goes off here

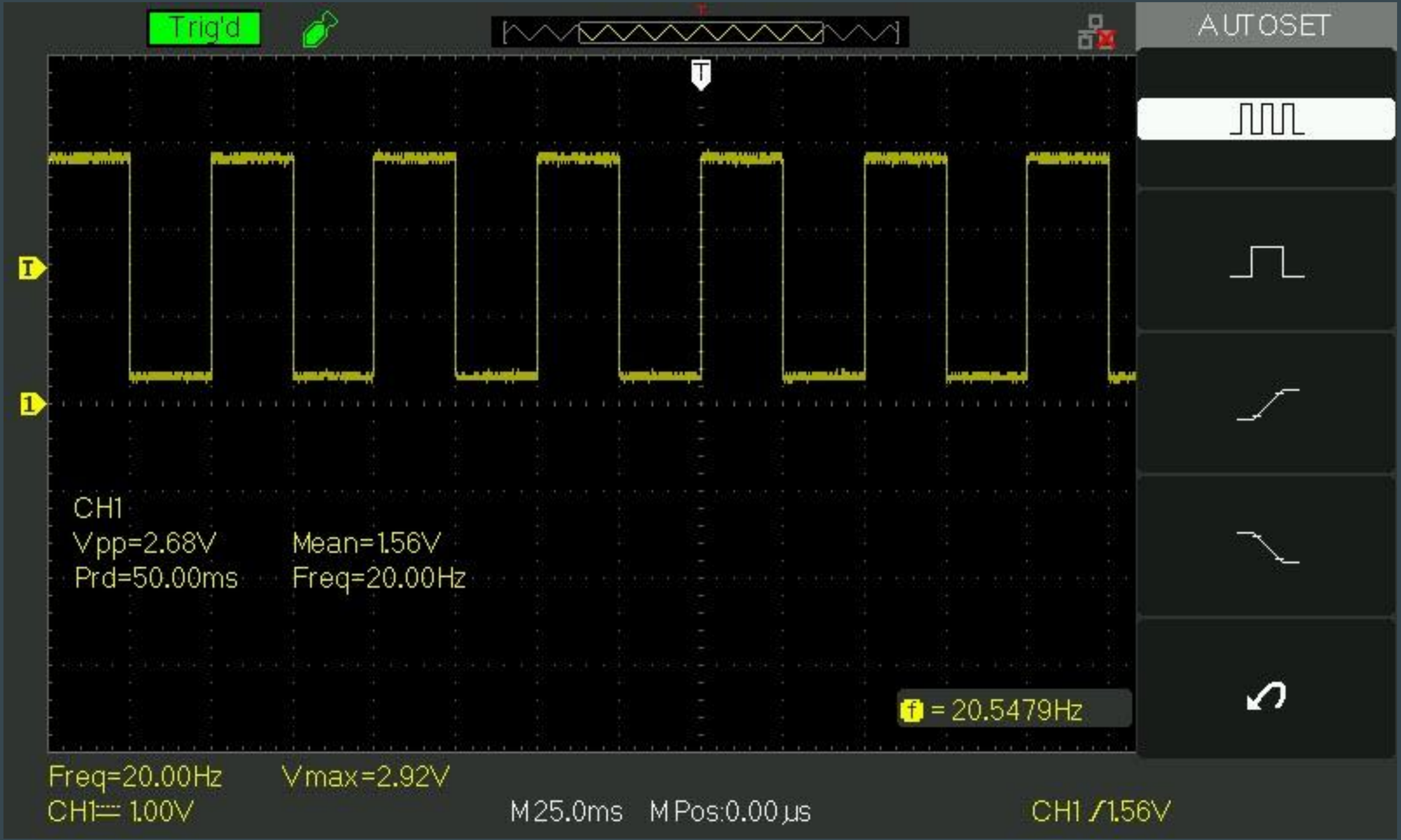


0
hp
hp + Δ
Δ
The alarm is actually set here
The alarm goes off here

keep track of *logical clock*



set alarm from where you were *supposed to* wake up

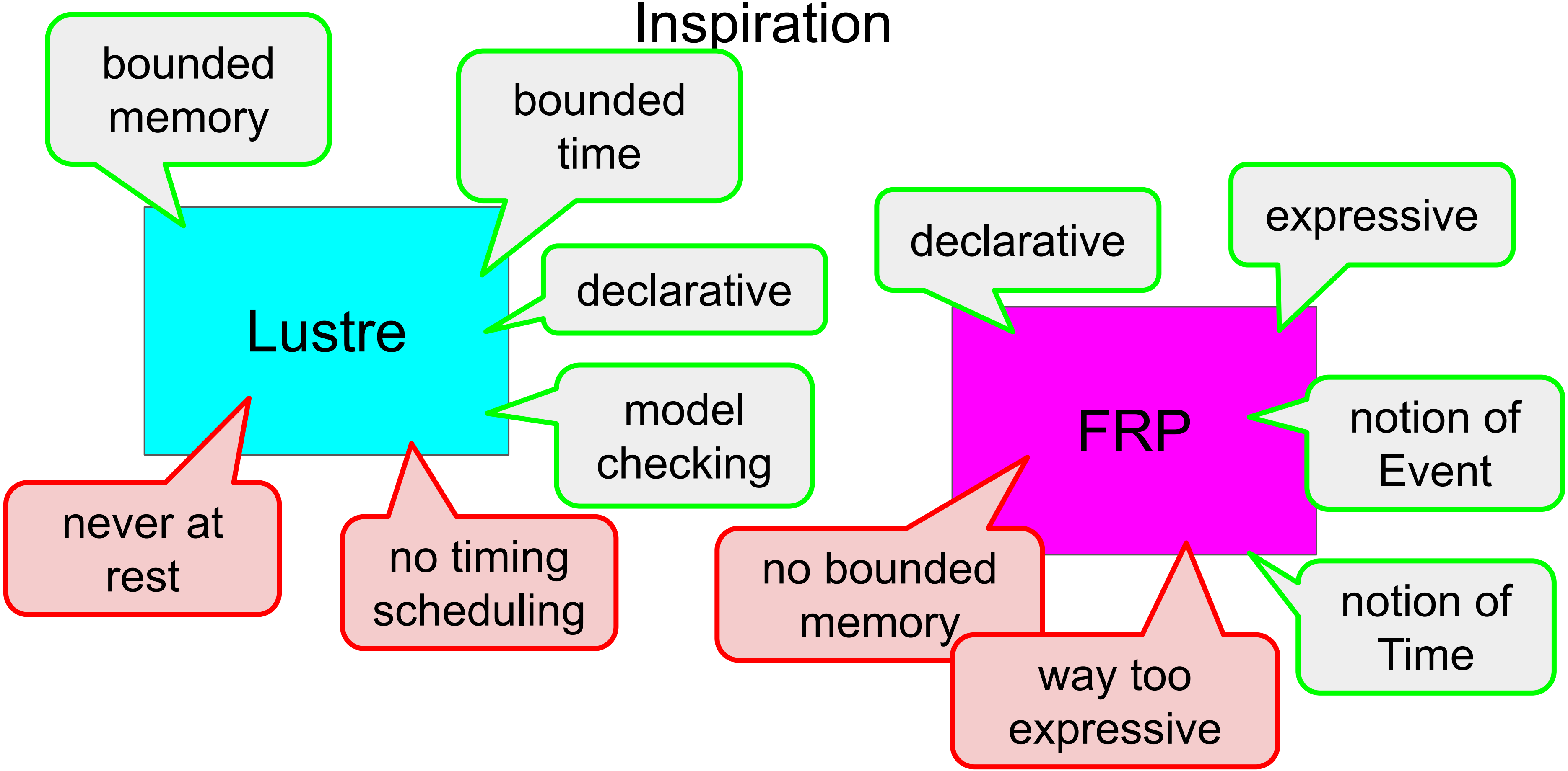

Trig'd
AUTOSET
CH1
Vpp=2.68V
Mean=1.56V
Prd=50.00ms
Freq=20.00Hz
f = 20.5479Hz
Freq=20.00Hz
Vmax=2.92V
CH1 1.00V
M 25.0ms
M Pos:0.00µs
CH1 1.56V

Why not Scoria?

What would a declarative counterpart to Scoria look like?

- Imperative
  - I like **declarative** languages
  - Declarative = less likely to make certain mistakes (?)
- Process model
  - process dynamically allocated
  - communication between processes is limited (and unintuitive)
  - **run out of time?**
- Memory
  - memory dynamically allocated
  - **run out of memory?**

Inspiration



Lustre

constantly ticking clock

never at rest

no timing scheduling

type Signal a ~ *[a] {- infinite -}*

```
(+)   :: Signal Int -> Signal Int -> Signal Int
n     :: Signal Int
pre   :: Signal a -> Signal a
(→)   :: Signal a -> Signal a -> Signal a
```

pre s = _|_ : s

(x:_) → (_:s) = x:s

```
summer :: Signal Int -> Signal Int
summer diff = sum
  where
    sum = diff + (0 → pre sum)
```

ICFP
1997

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is

- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the "what" of an interactive animation, one can hope to then automate the "how" of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serves comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/-presentation style. Moreover, we have found that non-strict

B = “behavior”

E = “event”

I… gramming (Fran-style)

Time is explicit

no bounded memory / time

type Time ~ *Nat*
type B a ~ *Time -> a* *-- Monad (Reader)*
type E a ~ *(Time,a)* *-- M*

change :: Eq a => B a -> B (E a)
change b = **do** x <- b; when (/=x) b

switch :: B a -> E (B a) -> B a
(+=>) :: E a -> (Time -> a -> b) -> E b
when :: (a->Bool) -> B a -> B (E a)

```
summer :: Int -> (Time -> E Int) -> Time -> B Int
summer n diff t =
  pure n `switch` (diff t +=> \t’ k -> summer (n+k) diff t’)
```

ICFP 2015

Practical Principled FRP

Forget the Past, Change the Future, *FRPNow*!

Atze van der Ploeg Koen Claessen

Chalmers University of Technology, Sweden

{atze,koen}@chalmers.se

Abstract

We present a new interface for practical Functional Reactive Programming (FRP) that (1) is close in spirit to the original FRP ideas, (2) does not have the original space-leak problems, without using arrows or advanced types, and (3) provides a simple and expressive way for performing I/O actions from FRP code. We also provide a denotational semantics for this new interface, and a technique (using Kripke logical relations) for reasoning about which FRP functions may "forget their past", i.e. which functions do not have an inherent space-leak. Finally, we show how we have implemented this interface as a Haskell library called *FRPNow*.

Categories and Subject Descriptors D.3.2 [*Applicative (functional) languages*]

Keywords Functional Reactive Programming, Space-leak, Purely

without compromising the original spirit behind FRP, and present an implementation of this interface in Haskell. Our contribution is thus a principled and practical way of programming reactive systems with FRP, without callbacks, nondeterminism or mutable state.

Let us delve a bit deeper into the two problems mentioned earlier.

Space Leaks The first problem, the space leak problem, can be analyzed as follows. A program in FRP can lead to space leaks in three ways:

1. The program using the FRP library can have a space leak.
2. The implementation of the FRP library can have a space leak.
3. The *interface* of the FRP library, i.e. the set of functions offered by the library, can be *inherently leaky*.

Each of these implies the previous: if we have an interface which

Functional Reactive Programming (FRPNow)

type Time ~ *Nat*
type B a ~ *Time -> a*
type E a ~ *(Time,a)*

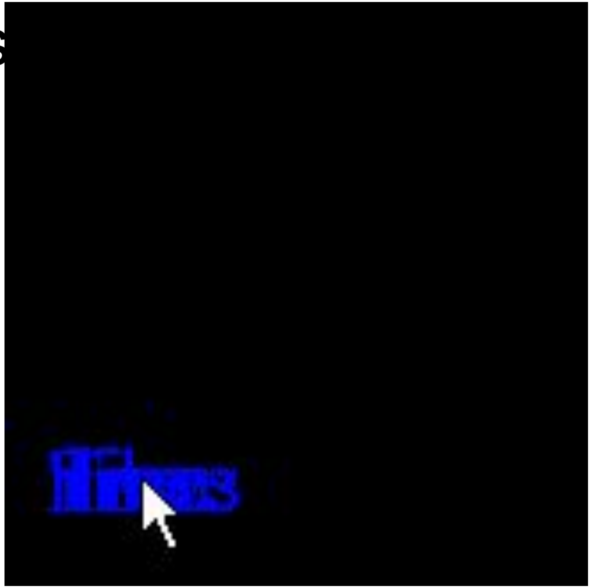
switch :: B a -> E (B a) -> B a

```
summer :: Int -> (Time -> E Int) -> Time -> B Int
```

```
summer :: Int ->     B (E Int)       ->      B (B Int)
summer n diff =
```

harder to write this

Functional Reactive Programming (Fran-s



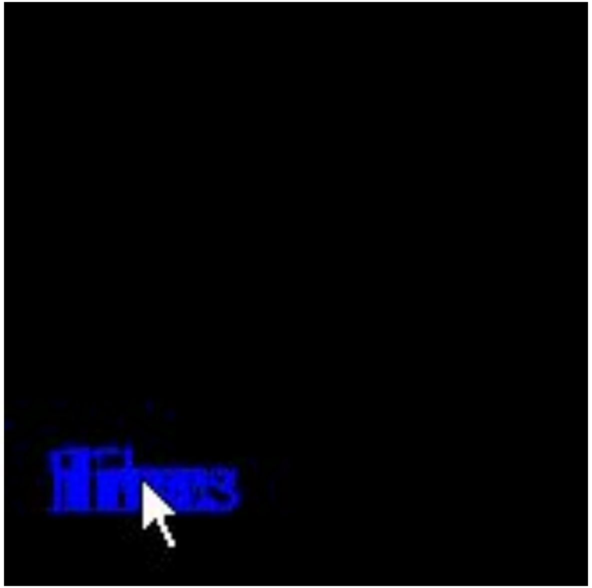
```
type Animation = B Image

text       :: String -> B Image
over       :: B Image -> B Image -> B Image
move       :: B (Int,Int) -> B Image -> B Image
mouseXY :: B (Int,Int)

later :: B Time -> B a -> B a
```

```
foldr1 over $
  zipWith later [0,1..]
    [ move mouseXY (text w)
    | w <- words “Time flows like a river” ]
```

Implementing FRP

type E a = *..some kind of blocking mechanism / callback..*

type B a = (a, E (B a))



Small insights

- Lustre has bounded memory because of **pre**
  - no recursion with accumulating parameters

- Let Lustres clock tick only when something happens? **- NO**
  - Haski

ICFP
2020

Towards Secure IoT Programming in Haskell

Nachiappan Valliappan
Chalmers University of Technology
Gothenburg, Sweden
nacval@chalmers.se

Robert Krook
Chalmers University of Technology
Gothenburg, Sweden
krookr@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

Koen Claessen
Chalmers University of Technology
Gothenburg, Sweden
koen@chalmers.se

Abstract

IoT applications are often developed in programming languages with low-level abstractions, where a seemingly innocent mistake might lead to severe security vulnerabilities. Current IoT development tools make it hard to identify these vulnerabilities as they do not provide end-to-end guarantees about how data flows *within and between* appliances. In this work we present Haski, an embedded domain specific language (eDSL) in Haskell for secure programming of IoT devices. Haski enables developers to write Haskell programs that generate C code without falling into many of C's pitfalls. Haski is designed after the synchronous programming language Lustre, and sports a backwards compatible

ACM Reference Format:

Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards Secure IoT Programming in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3406088.3409027

1 Introduction

The Internet of Things (IoT) conceives a future where "things" (embedded electronics) can be interconnected. While a compelling vision, recent events have demonstrated the *high vulnerability* of IoT (e.g., [Bertino and Islam 2017; Fernandes

Haski

```
type Temp = Float
data Status = Home | Away
data WindowOp = Open | Close | Skip
```

called when any input changes

```
halexa :: Signal Temp -> Signal Status -> Signal WindowOp
```

Haski

security labels

keeping track if halexa does something “bad”

```
halexa :: Signal Temp -> LSignal Status -> LSignal WindowOp
```

(maybe **Lustre clocks** are a solution)

Small insights

- Lustre has bounded memory because of **pre**
  - no recursion with accumulating parameters
- Let Lustres clock tick only when something happens? **- NO**
  - Haski
  - Non-modular behavior!

all parts of a system can observe when something happens

How to schedule timing events?

Fruste

type Flow a ~ *Time -> a*
type Stream a ~ *[(Time,a)]* *Functor*

zip :: Flow a -> Flo
n :: Flow Int
(~~>) :: a -> Stream a

pre :: Stream a -> S
sample :: Stream a -> F
merge :: (a->a->a) ->

```
summer :: Stream Int -> Stream Int
summer diff = sum
  where
    sum = diff !+ (0 ~~> pre sum)
```

```
(!+) :: Stream Int -> Flow Int -> Stream Int
s !+ a = fmap (uncurry (+)) (sample s a)
```

```
(><) :: Stream a -> Stream a -> Stream a
s1 >< s2 = merge (\a b ->a) s1 s2
```

```
at :: Flow a -> Stream b -> Stream a
a `at` s = fmap snd (sample s a)
```

generate C code

at rest

How to implement

wakes up at trigger

only run relevant code when trigger happens

How to schedule timing events

Implementing Fruste

type Stream a = *..some kind of blocking mechanism / callback..*

type Flow a = (a, Stream a)   -- *created by ~~>*

HARD

(But the right path)

(keep core calculus as small as possible)

Implementing Fruste (easy)

type Flow a = Signal a

type Stream a = (Signal Bool, Signal a)

reuse Lustre compiler

runs the whole program every time something happens

only has to make sense when trigger is True

Fruste implementation

```
(~~>) :: Val a => a -> Stream a -> Flow a
x ~~> (act, s) = y
 where
  y = ifThenElse act s (val x --> Lustre.pre y)
```

```
pre :: Stream a -> Stream a
pre (act, s) = (started' /\ act, s')
 where
  started' = false --> Lustre.pre (act \/ started')
  s'       =           Lustre.pre (ifThenElse act s s')
```

How to implement

add small extension to Lustre

How to schedule timing events

use it to implement Fruste timing combinator

Fruste timing

later :: Flow Int -> Stream a -> Stream a

needs unbounded memory :-(

later1 :: Flow Int -> Stream a -> Stream a

only schedules 1 event, ignores everything else

Fruste implementation

Lustre extension

```
timer :: Signal Bool -> Signal Int -> Signal Bool
```

```
later1 :: Flow Int -> Stream a -> Stream a
later1 t (set,inp) = (get,mem)
 where
  get     = timer set t
  ready  = get \/ (true --> Lustre.pre (nt set /\ ready))
  mem    = Lustre.pre (ifThenElse (set /\ ready) inp mem)
```

Fruste examples

```
(>+<) :: Stream Int -> Stream Int -> Stream Int
s1 >+< s2 = merge (+) s1 s2
```

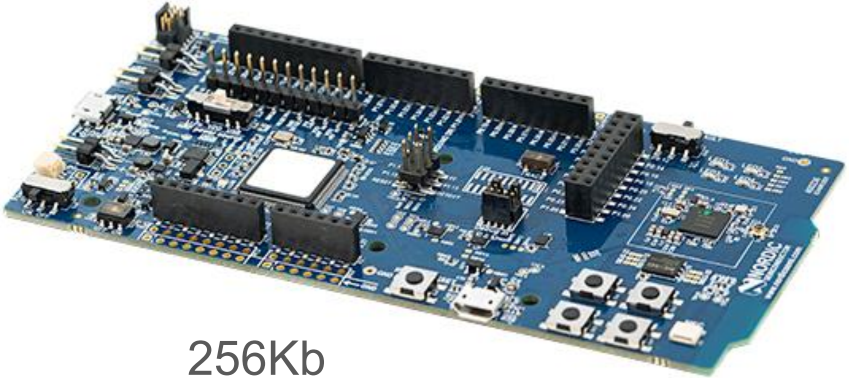
```
ticker :: Flow Int -> Stream ()
ticker t = s
  where
    s = start >< later1 t s
```

DEMO

```
counter :: Stream () -> Stream () -> Stream Int
counter down up = summer diff
 where
  diff =  ((-1) `at` down)
     >+< (1    `at` up)
     >+< (2    `at` ticking 1000)
```

Nordic Semiconductor NRF52840-DK

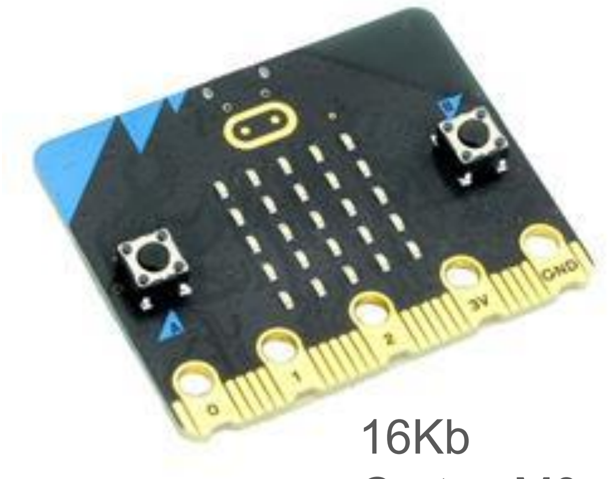
- Development board for IoT-like systems
- Has bluetooth
- Low power
- Fruste runs on Zephyr



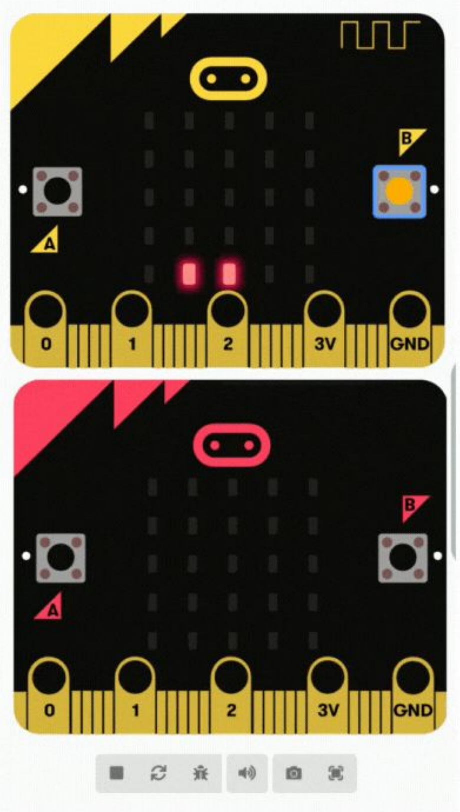
256Kb
Cortex M4 64MHz
BLE

Micro:bit

- Fruste runs on Micro:bit
- Has a 5x5 display
- Has bluetooth
- Low power
- Programming in scratch-like environment
  - or Python
  - or JavaScript
  - (or C)



16Kb
Cortex M0
BLE


B
A
0
1
2
3V
GND
B
A
0
1
2
3V
GND

```
data System
  = System
  { buttonA    :: Stream ()
  , buttonB    :: Stream ()
  , bltReceive :: Stream Msg
  , random     :: Flow Int
  …
  }
```

bltSend works by combining all streams from all parts of the program

```
arbiter :: System -> (Stream Msg, Stream ())
arbiter sys = (bltSend, victory)
 where
  bltSend = random sys `at` (start >< later1 3 retry)
  sent    = 0 ~~> bltSend

  recv    = bltSend ==> bltReceive sys
  victory = holds (recv !<  sent)
  retry   = holds (recv !== sent)
```

```
(==>) :: Stream a -> Stream b -> Stream b
```

only keeps b's right after a's

```
pingPong :: System -> (Stream Msg, Flow Display)
pingPong sys = (bltSend, display)
 where
  (bltSend1, beginMe)  = arbiter sys
  (display, endMe)        = gameLogic mbit (beginMe >< endYou)
  (bltSend2, endYou)     = waitLogic endMe

  bltSend = bltSend1 >#< bltSend2
```

merge with
uniqueness **assertion**

“when”

Create something **reactive** from something **non-reactive**

```
when :: (a->Bool) -> Flow a -> Stream a
```

OK if all inputs to the system are **reactive**

```
time :: Flow Time
```

```
temp :: Flow Temp
```

```
when (>30) temp
```

Implementing Flow/Stream

help!

- Only want to execute relevant code in every step
- IDEA 1:
  - Partial evaluation
  - Use same strategy as Lustre clocks
  - (Hard, because Flows always compute, even if their results are not used)
- IDEA 2:
  - Use the “HARD” implementation of Flow/Stream

Future / Ongoing work

- Haski-style security labels
- More serious model checking
  - Predicate abstraction very nice fit with timers
  - “Octogons”
  - Properties
    - Timing is appropriate
    - No double use of shared resources
    - …