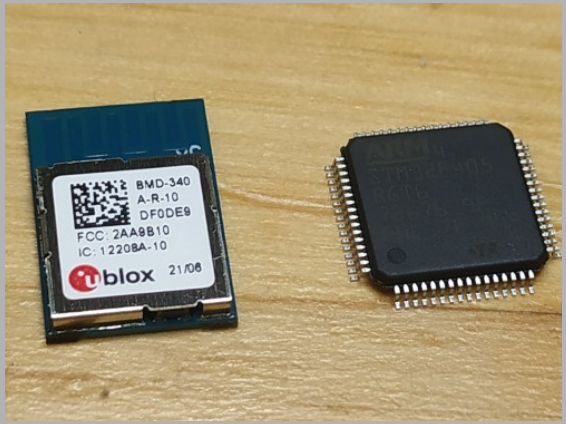
Mind the gap!


GTX480

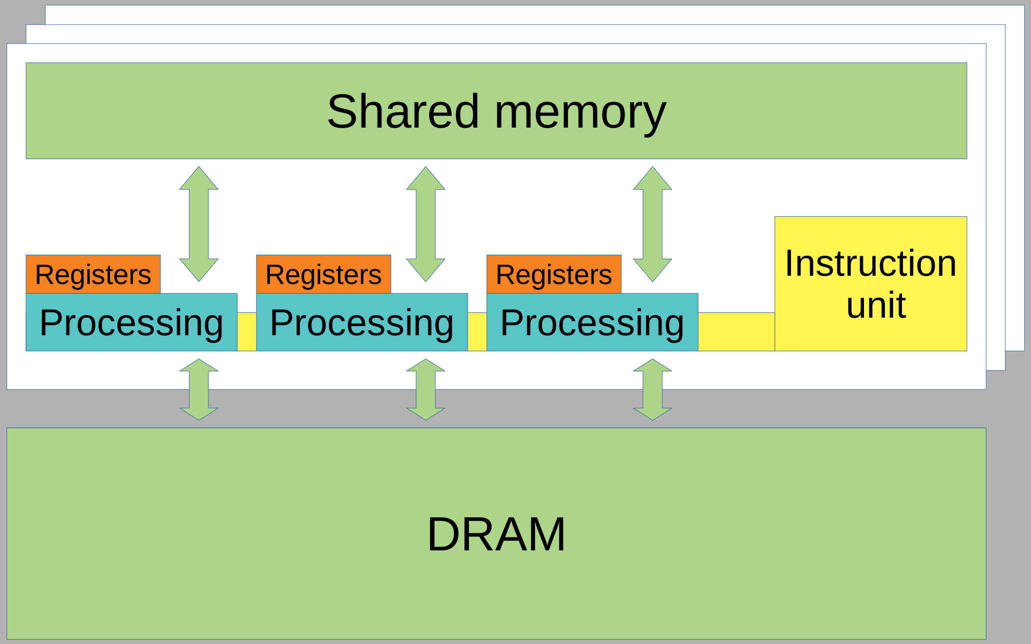

BMD-340
A-R-10
DF0DE9
FCC: 2AA9B10
IC: 1220BA-10
ublox
21/06


XILINX
ARTIX-7
100T
CSG324
IC8

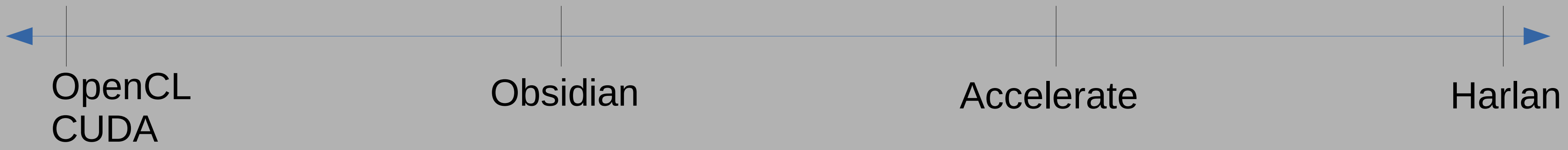
The gap!

- Interesting hardware on one side
  - Fast
  - Special purpose
  - Energy efficient
  - Resource constrained
- Languages with nice properties on the other
  - Terse/elegant/powerful
  - Safe
  - Secure
  - Pure

GPU Programming



GPU Programming



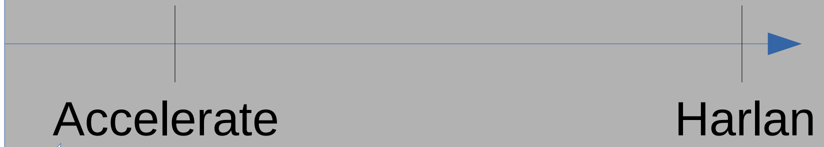
```
(define (cast-view-rays width height fov eye)
  (let* ((aspect (/ (int->float width) (int->float height)))
        (fovX (* (int->float fov) aspect))
        (fovY (int->float fov)))
    (kernel* ((x (iota width))
              (y (iota height)))
      (let ((x (point-of-index width x))
            (y (point-of-index height y)))
        (unit-length (point-diff (point3f (* x fovX)
                                          (* (- 0 y) fovY)
                                          0)
                                 eye))))))
```

OpenCL
CUDA

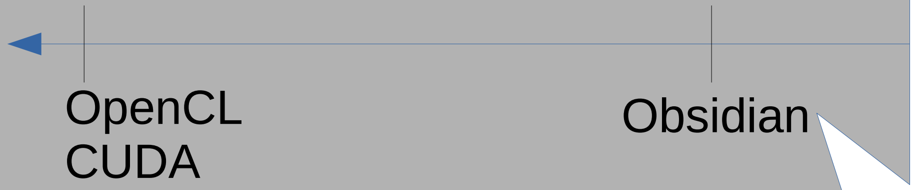
Harlan

mming

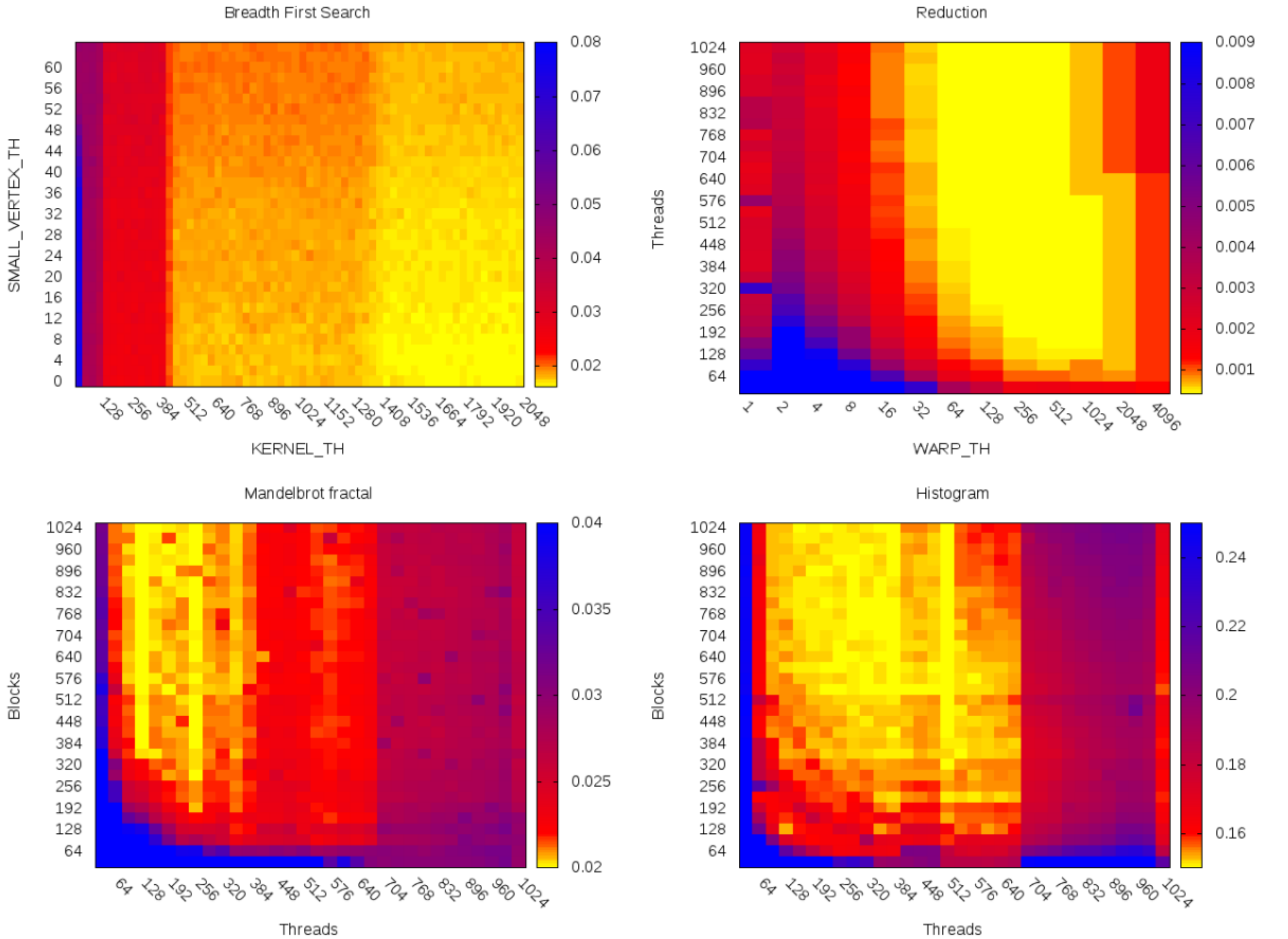
```
castViewRays
    :: Int                                -- width of the display
    -> Int                                -- height
    -> Int                                -- field of view
    -> Exp Position                       -- eye position
    -> Acc (Array DIM2 Direction)         -- all rays originating from the eye position
castViewRays sizeX sizeY fov eyePos
  = let
        sizeX'          = P.fromIntegral sizeX
        sizeY'          = P.fromIntegral sizeY
        aspect          = sizeX' / sizeY'
        fov'            = P.fromIntegral fov
        fovX            = fov' * aspect
        fovY            = fov'
    in
    A.generate (constant (Z :. sizeY :. sizeX))
               (\ix -> let (x, y) = xyOfPoint $ pointOfIndex sizeX sizeY ix
                       in  normalize $ lift (V3 (x * fovX) (y * fovY) 0) - eyePos)
```



GPU Programming

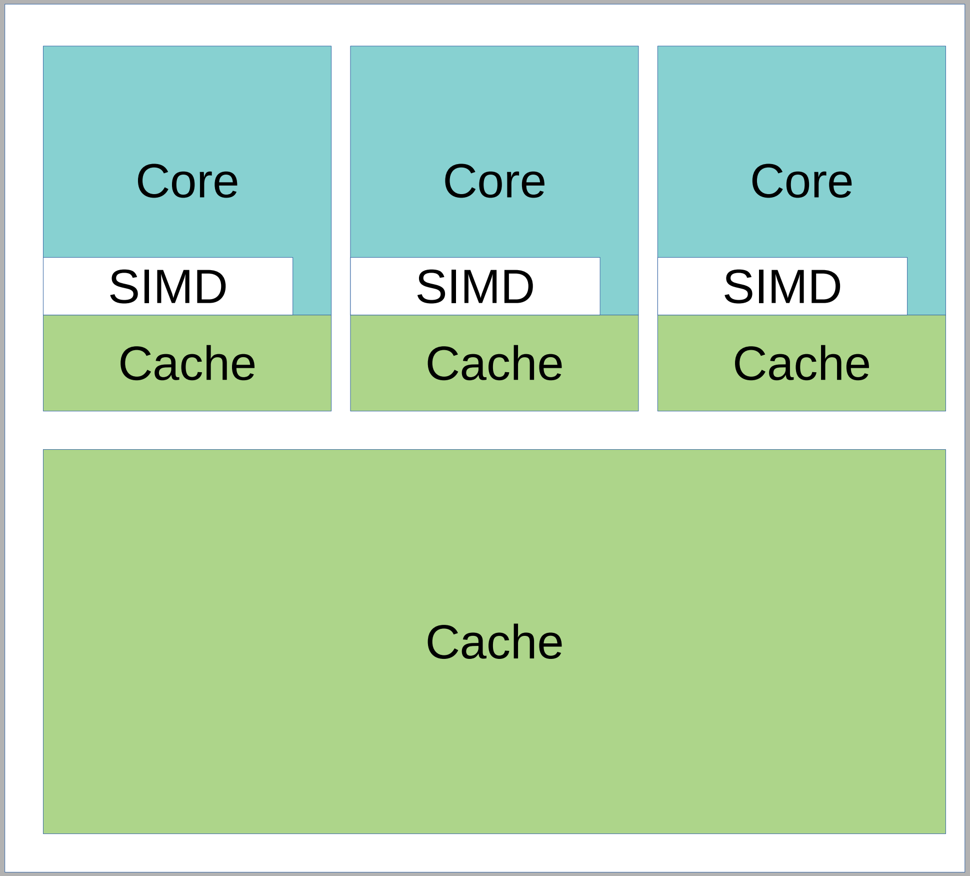



```
reduceKernel :: (Compute t, Data a)
                => (a -> a -> a)
                -> Pull Word32 a
                -> Program t (SPush t a)
reduceKernel f arr
   | len arr == 1 = return $ push arr
   | otherwise =
     do let (a1,a2) = halve arr
          arr' <- compute $ zipWith f a1 a2
          reduceKernel f arr'
```



Breadth First Search
SMALL_VERTEX_TH
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
KERNEL_TH
128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
0.08 0.07 0.06 0.05 0.04 0.03 0.02
Reduction
Threads
64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
WARP_TH
1 2 4 8 16 32 64 128 256 512 1024 2048 4096
0.009 0.008 0.007 0.006 0.005 0.004 0.003 0.002 0.001
Mandelbrot fractal
Blocks
64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Threads
64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
0.04 0.035 0.03 0.025 0.02
Histogram
Blocks
64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Threads
64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
0.24 0.22 0.2 0.18 0.16

Multicore CPU with Wide SIMD



Multicore CPU with Wide SIMD

```
__m512i vresult1 = _mm512_maddubs_epi16(v1_int8, v2_int8);
__m512i vresult2 = _mm512_madd_epi16(vresult1, v4_int16);
vresult = _mm512_add_epi32(vresult2, v3_int);
_mm512_storeu_si512((void *) result, vresult);
```

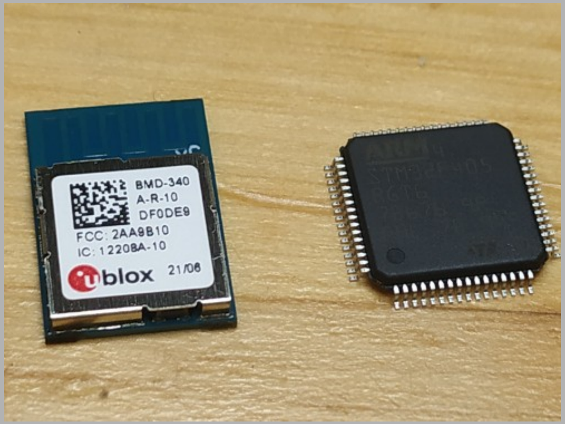
Or, bridge the gap

Intel ArBB and our EmbArBB

```
data Op =
   -- elementwise and scalar
     Add | Sub | Mul | Div | Max | Min
   | Sin | Cos | Exp
   ...

   -- operations on vectors
   | Gather | Scatter | Shuffle | Unshuffle
   | RepeatRow | RepeatCol | RepeatPage
   | Rotate | Reverse | Length | Sort
   | AddReduce | AddScan | AddMerge
   ...
```

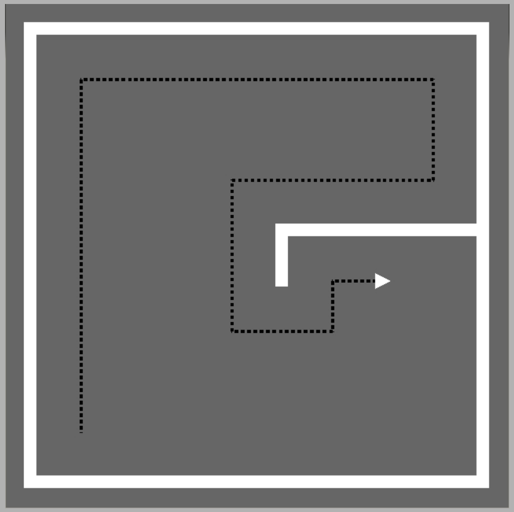
```
matVec :: Exp (DVector Dim2 Float)
       -> Exp (DVector Dim1 Float)
       -> Exp (DVector Dim1 Float)
matVec m v = addReduce rows
           $ m * (repeatRow (getNRows m) v)
```

Programming Microcontrollers

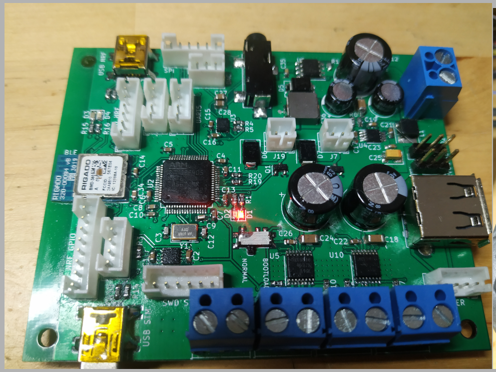


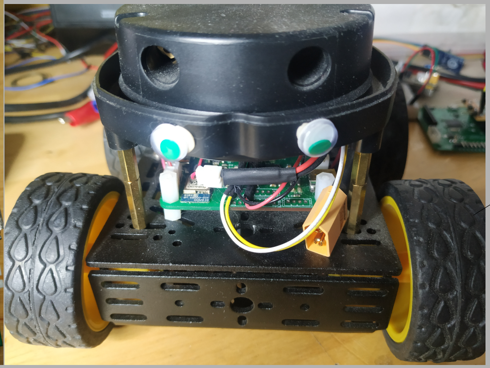
Let's program a robot

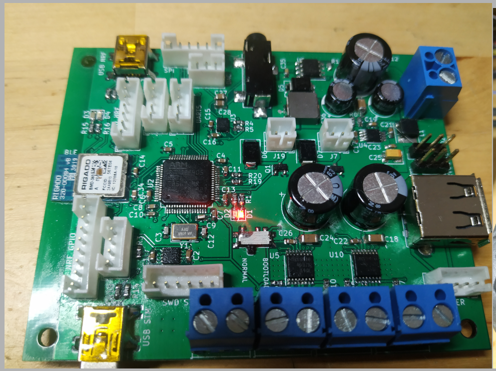
```
sMove :: Program ()
sMove = cond sensor turnRight
move

followWall :: Program ()
followWall =
  while (return true) $
    cond checkLeft sMove $
      do turnLeft
           move

checkLeft :: Program BoolE
checkLeft = do
  turnLeft
  s <- sensor
  turnRight
  return s
```

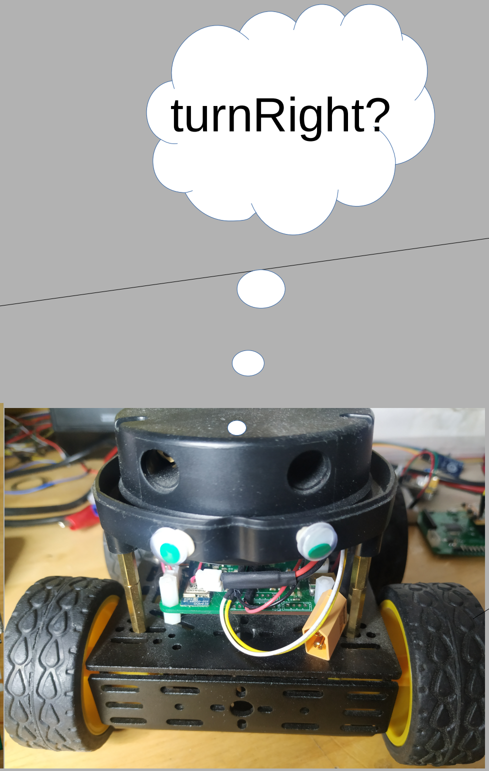


```
sMove :: Program ()
sMove = cond sensor turnRight move

followWall :: Program ()
followWall =
  while (return true) $
    cond checkLeft sMove $
      do turnLeft
         move

checkLeft :: Program BoolE
checkLeft = do
  turnLeft
  s <- sensor
  turnRight
  return s
```



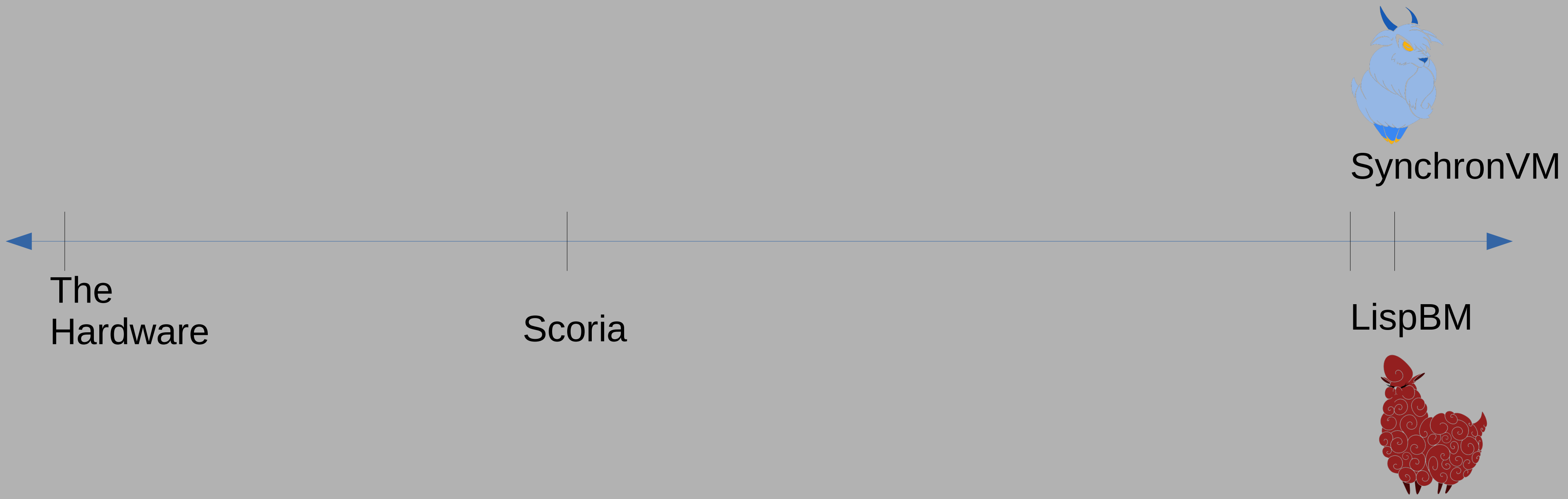







```
sMove :: Program ()
sMove = cond sensor turnRight move

followWall :: Program ()
followWall =
  while (return true) $
    cond checkLeft sMove $
      do turnLeft
           move

checkLeft :: Program BoolE
checkLeft = do
  turnLeft
  s <- sensor
  turnRight
  return s
```

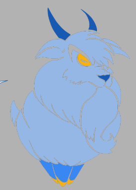
Current lines of work



https://abhiroop.github.io/pubs/sensevm_mplr.pdf

Current lines of work

Mary Sheeran
Abhiroop Sarkar



SynchronVM

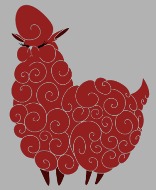
The Hardware

Scoria

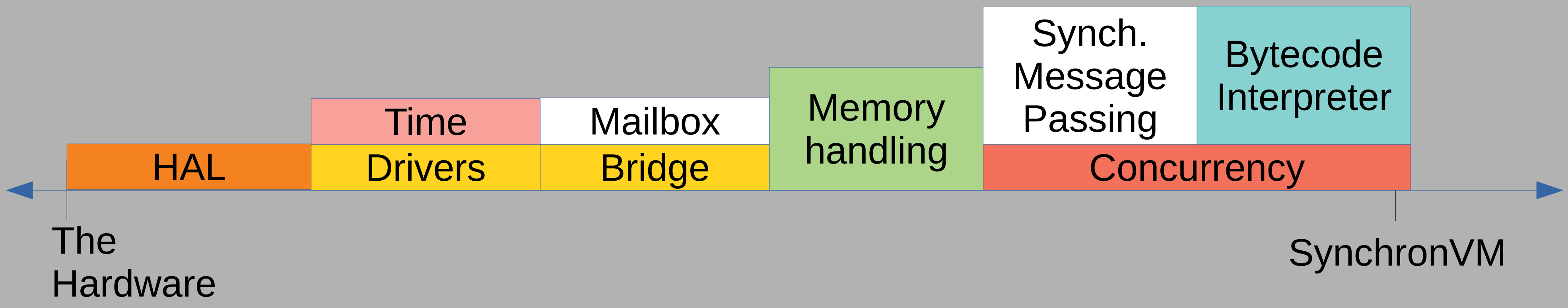
Robert krook
Koen Claessen
Stephen Edwards
John Hui

LispBM

Benjamin Vedder

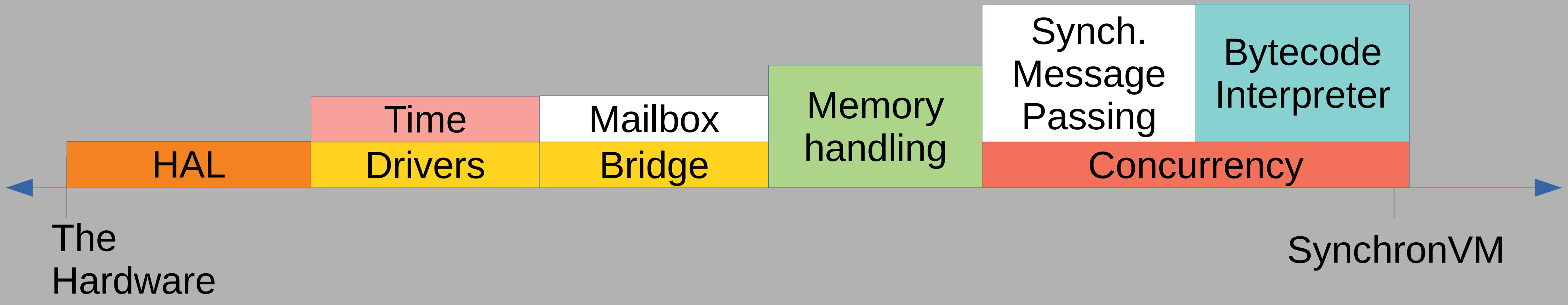


SynchronVM

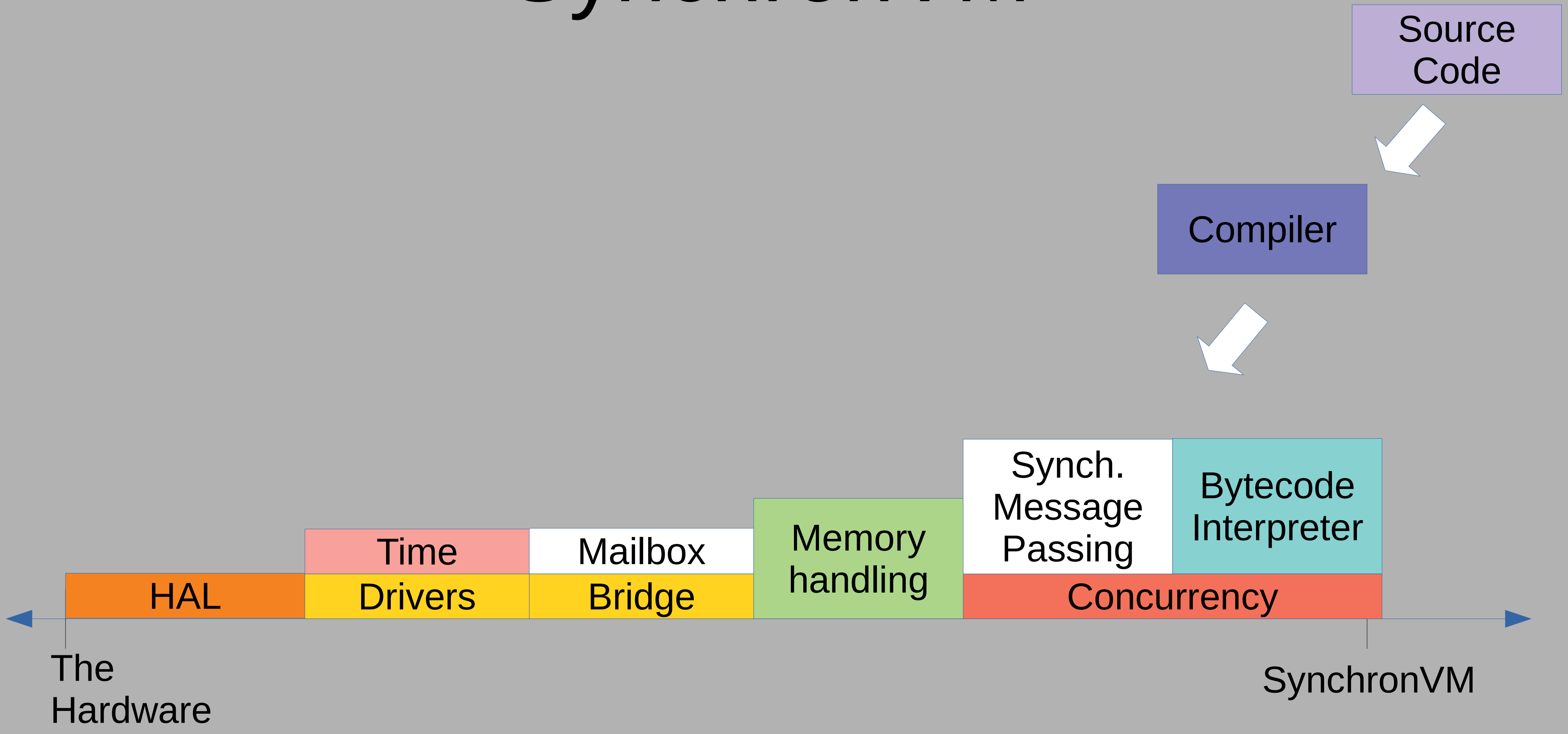


Synch.
Message
Passing
Bytecode
Interpreter
Time
Mailbox
Memory
handling
HAL
Drivers
Bridge
Concurrency
The
Hardware
SynchronVM

SynchronVM

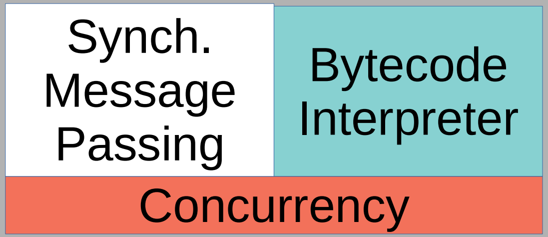

Synch.
Message
Passing
Bytecode
Interpreter
Time
Mailbox
Memory
handling
HAL
Drivers
Bridge
Concurrency
The
Hardware
SynchronVM

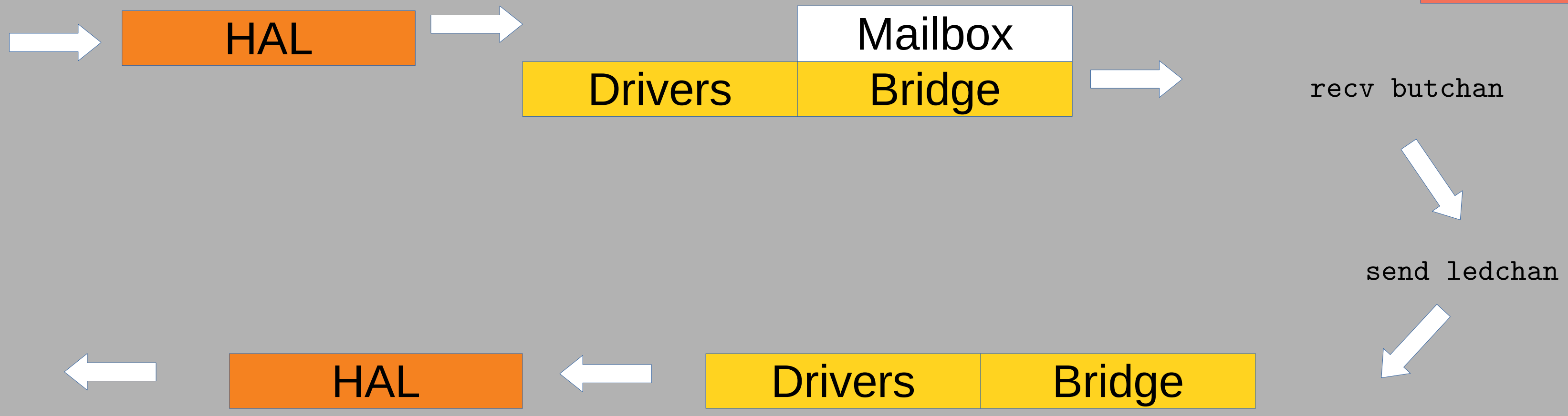
SynchronVM


Source Code
Compiler
Synch. Message Passing
Bytecode Interpreter
Time
Mailbox
Memory handling
HAL
Drivers
Bridge
Concurrency
The Hardware
SynchronVM

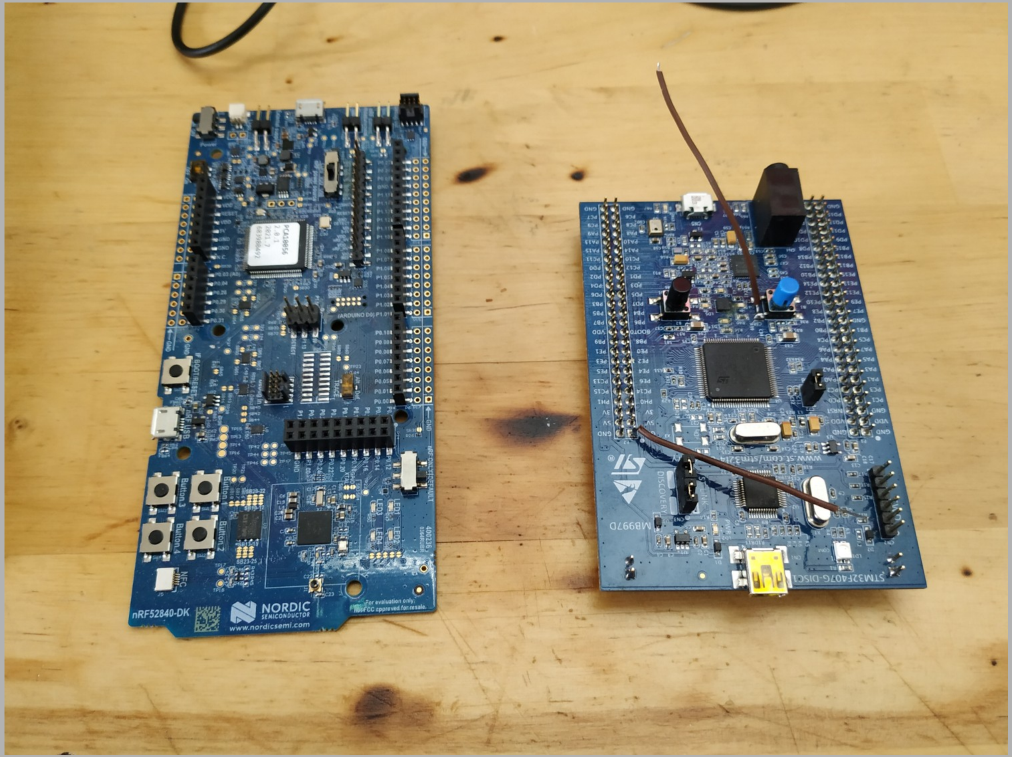
SynchronVM

```
glowled i = sync (send ledchan i)

f : ()
f = let _ = sync (wrap (recv butchan) glowled) in f
```

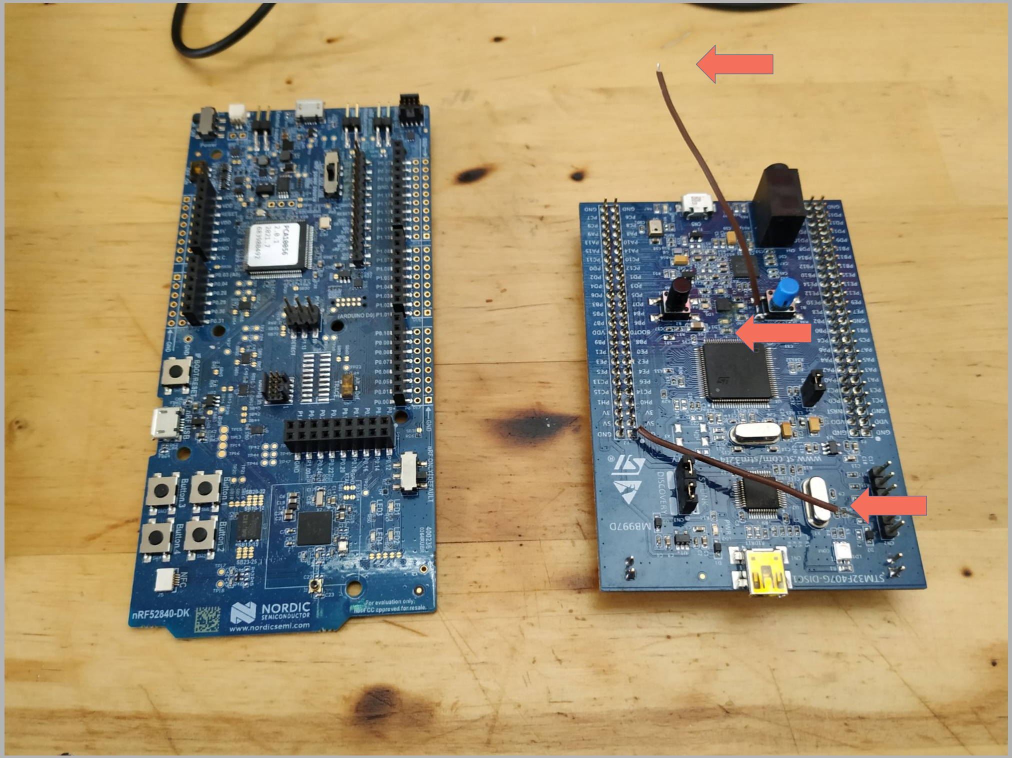


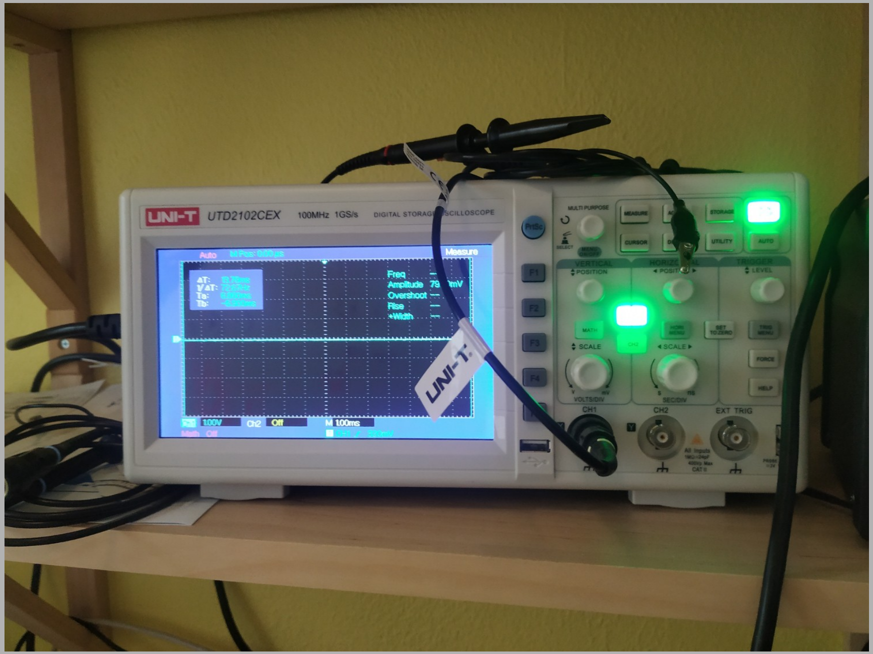


Performance testing

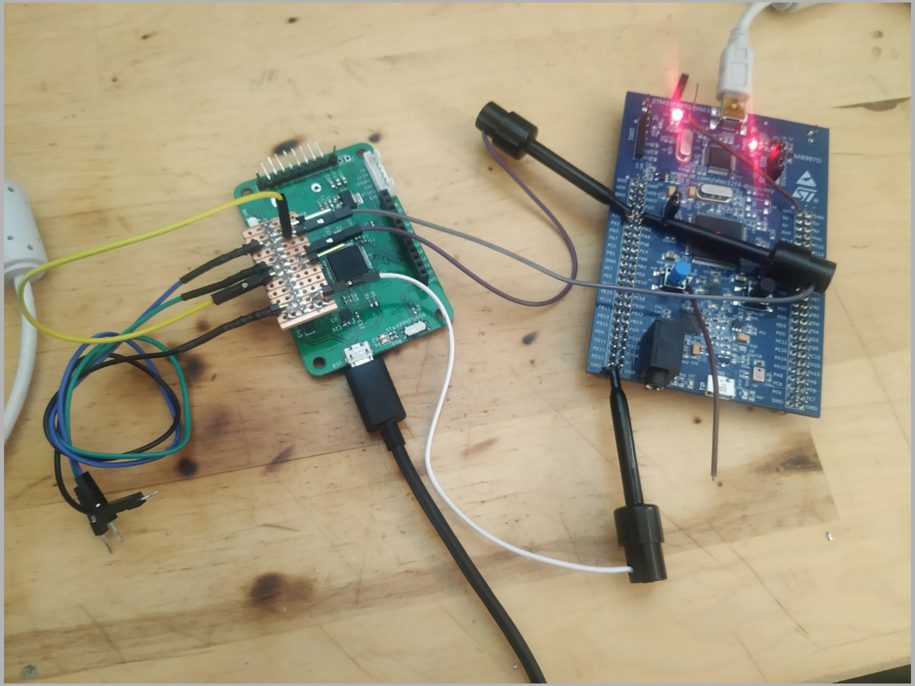


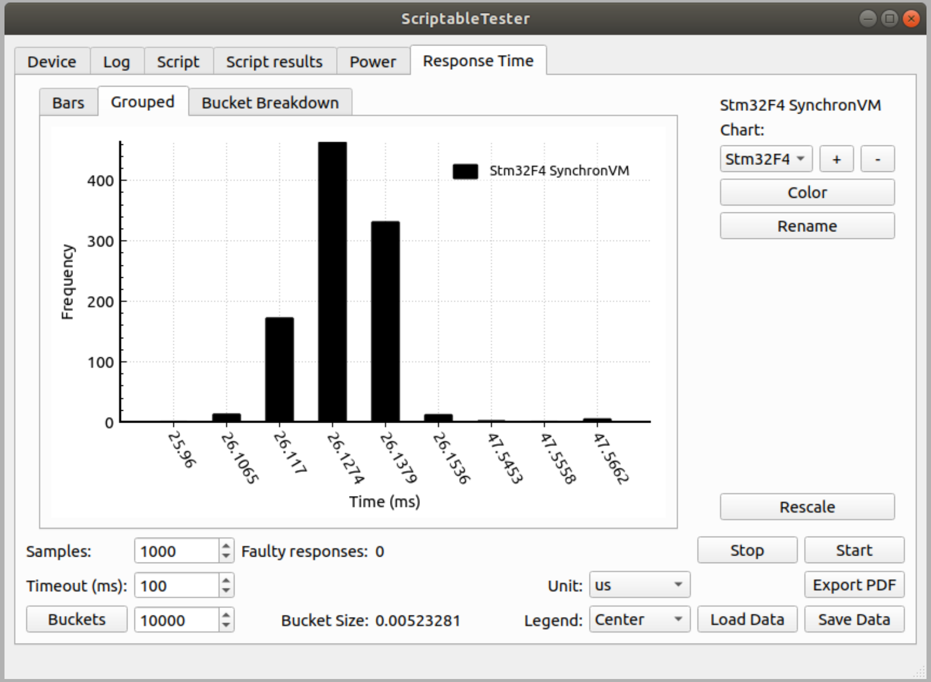
Performance testing – v0





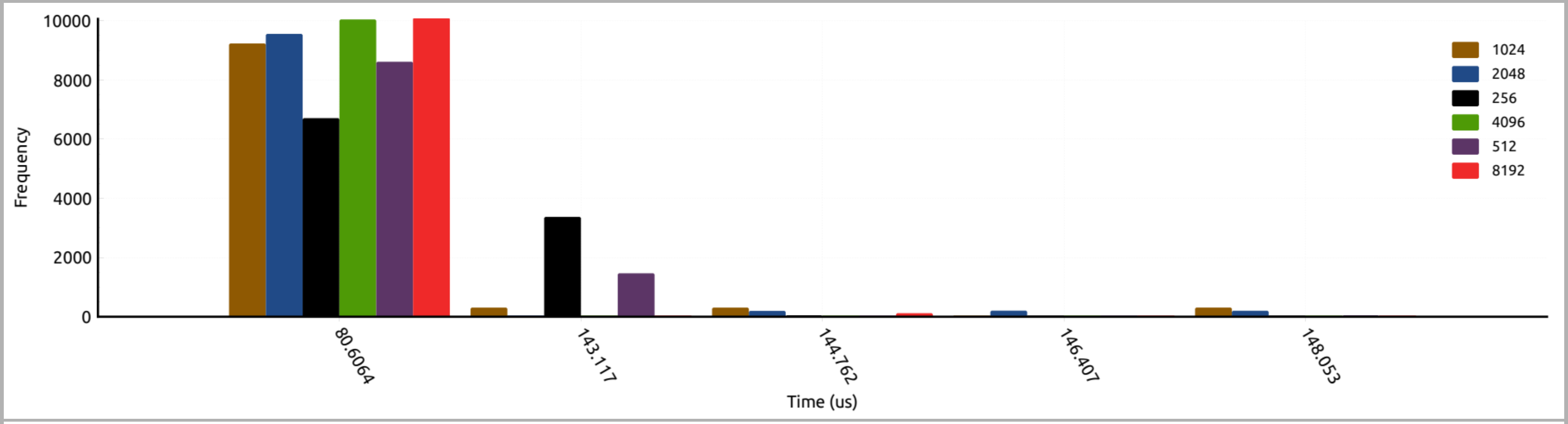
Scriptabletester



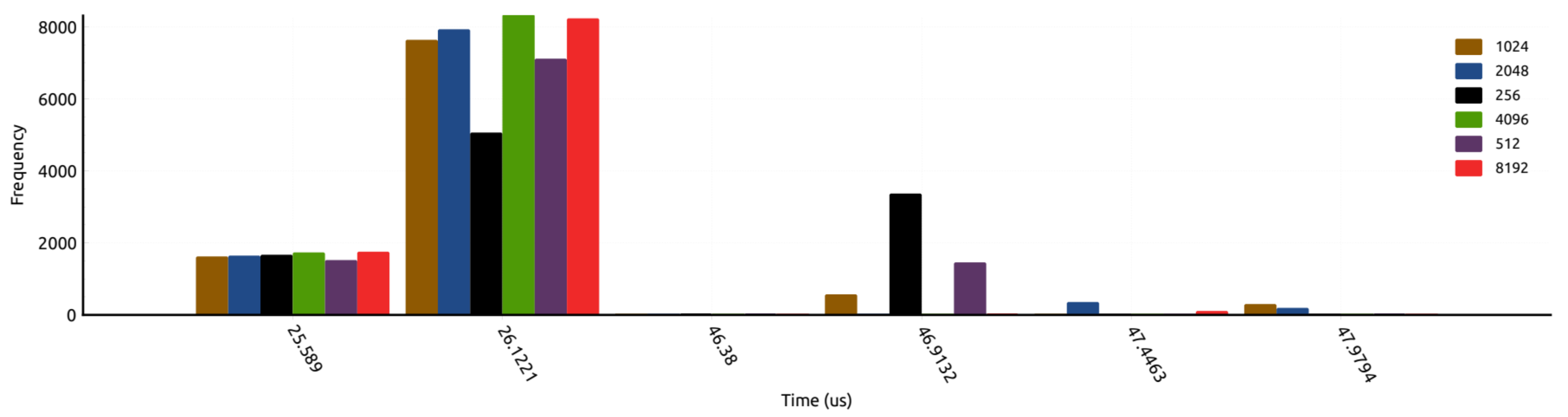


SynchronVM performance testing

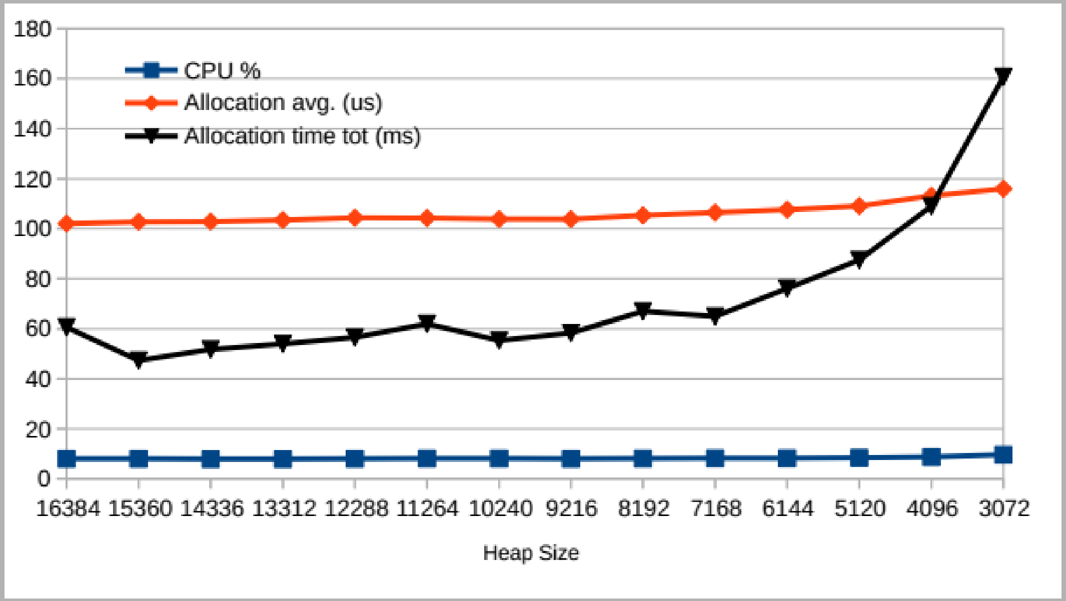
NRF52



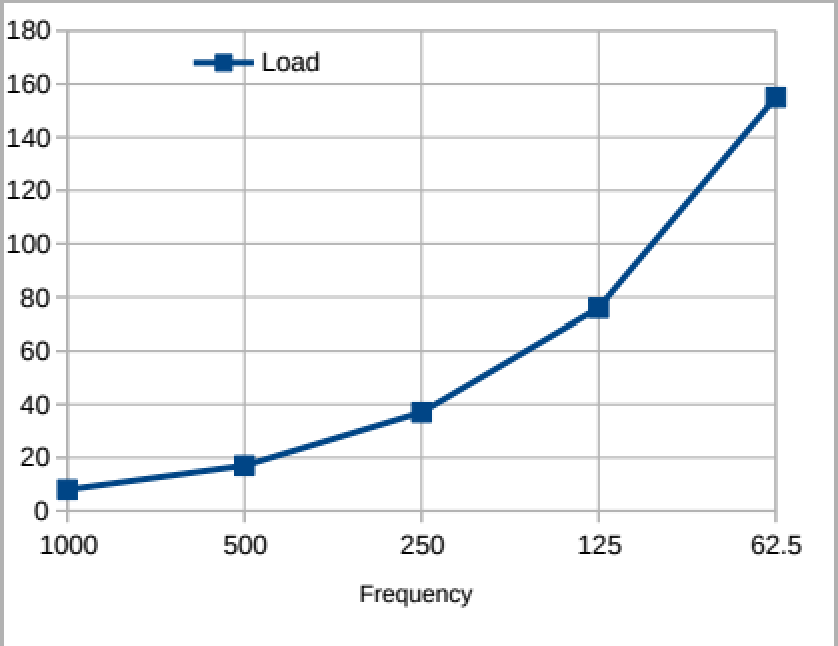
STM32F4



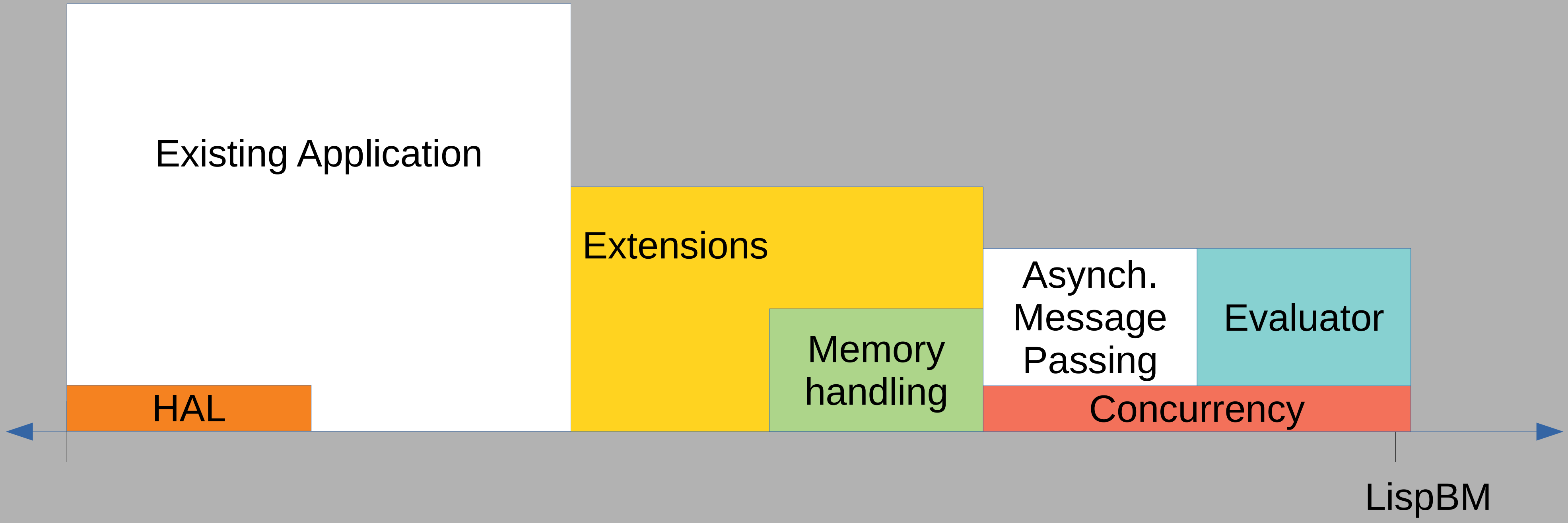
SynchronVM performance testing



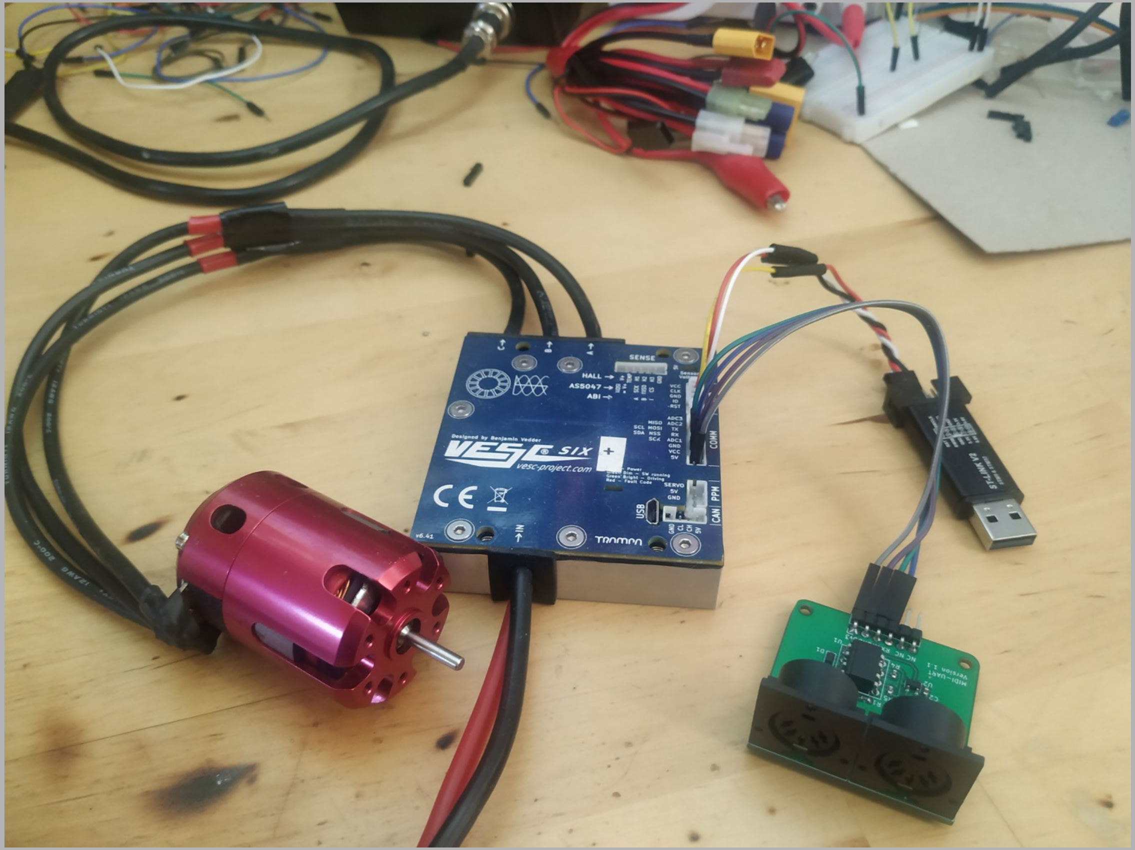
SynchronVM performance testing



LispBM


Existing Application
HAL
Extensions
Memory handling
Asynch. Message Passing
Evaluator
Concurrency
LispBM

LispBM – Scriptable Motorcontroller case study



```lisp
; Balance robot controller written in lisp

(defun #abs (x) (if (> x 0) x (- x)))

(defun #pos-x ()
    (* 0.5 (+
        (progn (select-motor 1) (get-dist))
        (progn (select-motor 2) (get-dist))
)))

(defun #set-output (left right)
    (progn
        (select-motor 1)
        (set-current-rel right)
        (select-motor 2)
        (set-current-rel left)
        (timeout-reset)
))

(defun #speed-x ()
    (* 0.5 (+
        (progn (select-motor 1) (get-speed))
        (progn (select-motor 2) (get-speed))
)))
```

```
(define #yaw-set (rad2deg (ix (get-imu-rpy) 2)))
(define #pos-set (#pos-x))

(define #pitch-set 0)
(define #was-running 0)

(define #kp 0.014)
(define #kd 0.0016)

(define #p-kp 50.0)
(define #p-kd -33.0)

(define #y-kp 0.003)
(define #y-kd 0.0003)

(define #enable-pos 1)
(define #enable-yaw 1)
```

```
; This is received from the QML-program which acts as a remote control
; for the robot
(defun proc-data (data)
    (progn
        (define #enable-pos (bufget-u8 data 4))
        (define #enable-yaw (bufget-u8 data 5))

        (if (= #enable-pos 1)
            (progn
                (define #pos-set (+ #pos-set (* (bufget-u8 data 0) 0.002)))
                (define #pos-set (- #pos-set (* (bufget-u8 data 1) 0.002)))
        ) nil)

        (if (= #enable-yaw 1)
            (progn
                (define #yaw-set (- #yaw-set (* (bufget-u8 data 2) 0.5)))
                (define #yaw-set (+ #yaw-set (* (bufget-u8 data 3) 0.5)))
        ) nil)

        (if (> #yaw-set 360) (define #yaw-set (- #yaw-set 360)) nil)
        (if (< #yaw-set 0) (define #yaw-set (+ #yaw-set 360)) nil)
))
```

```
(defun event-handler ()
    (progn
        (recv ((enable-data-rx . (? data)) (proc-data data))
              (_ nil))
        (event-handler)
))

(event-register-handler (spawn event-handler))
(event-enable 'event-data-rx)
```

```
(event-register-handler (spawn event-handler))
(event-enable 'event-data-rx)

(define #t-last (systime))
(define #it-rate 0)
(define #it-rate-filter 0)

(defun #filter (val sample)
    (- val (* 0.01 (- val sample)))
)

; Sleep after boot to wait for IMU to settle
(if (< (secs-since 0) 5) (sleep 5) nil)
```

```
(loopwhile t
    (progn
        (define #pitch (rad2deg (ix (get-imu-rpy) 1)))
        (define #yaw (rad2deg (ix (get-imu-rpy) 2)))
        (define #pitch-rate (ix (get-imu-gyro) 1))
        (define #yaw-rate (ix (get-imu-gyro) 2))
        (define #pos (+ (#pos-x) (* #pitch 0.00122))) ; Includes pitch
                                                      ; compensation
        (define #speed (#speed-x))

        ; Loop rate measurement
        (define #it-rate (/ 1.0 (secs-since #t-last)))
        (define #t-last (systime))
        (define #it-rate-filter (#filter #it-rate-filter #it-rate))
```

```
(if (< (#abs #pitch) (if (= #was-running 1) 45 10))
```

```
(progn
    (define #was-running 1)

    (if (= #enable-pos 0) (define #pos-set #pos) nil)
    (if (= #enable-yaw 0) (define #yaw-set #yaw) nil)

    (define #pos-err (- #pos-set #pos))
    (define #pitch-set (+ (* #pos-err #p-kp) (* #speed #p-kd)))

    (define #yaw-err (- #yaw-set #yaw))
    (if (> #yaw-err 180) (define #yaw-err (- #yaw-err 360)) nil)
    (if (< #yaw-err -180) (define #yaw-err (+ #yaw-err 360)) nil)

    (define #yaw-out (+ (* #yaw-err #y-kp) (* #yaw-rate #y-kd)))
    (define #ctrl-out (+ (* #kp (- #pitch #pitch-set))
                         (* #kd #pitch-rate)))

    (#set-output (+ #ctrl-out #yaw-out) (- #ctrl-out #yaw-out))
)
```

```
        (progn
            (define #was-running 0)
            (#set-output 0 0)
            (define #pos-set #pos)
            (define #yaw-set #yaw)
        )
    )
```

```
        (yield 1) ; Run as fast as possible
))
```

Concluding

We have seen approaches to bridging the gap between interesting hardware and nice languages.

1. Embedded domain specific languages.

    Sometimes multiple layers and JiT compilers involved.

2. Runtime systems.

Concluding

“Nice” has been mostly focused on terse, elegant and powerful but also touches on safe.

Lays a foundation for secure, perhaps?

Thoughts

Can we move nice languages even further across the divide?

There is interesting code on all levels that could potentially benefit from EDSL code generating approaches.

Thoughts

Performance and size of code becomes very important the closer to hardware we get.

Thoughts

1751 pages

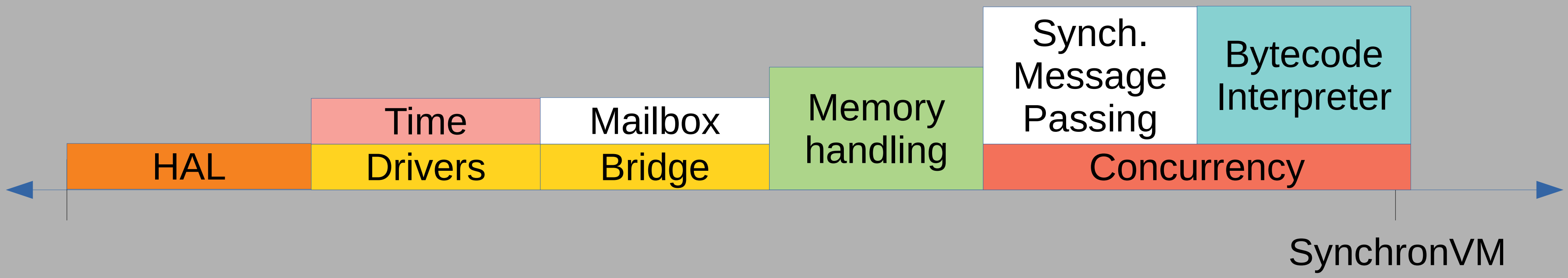
RM0090
Reference manual

STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm®-based 32-bit MCUs

Data sheet
BMI160
Small, low power inertial measurement unit

BME280
Combined humidity and pressure sensor

Optical Sensor
Product Data Sheet
LTR-303ALS-01



Credits

Mary Sheeran
Koen Claessen
Josef Svenningsson
Ryan Newton
Buddhika Chamith
Erik Holk
Mike Vollmer
Luke Dalessandro
Trevor McDonell
Prajith Ramakrishnan Geethakumari
Anders Thorsén
Kathrine Jahnberg
Eva Axelsson
Ayche Riza
Roger Johansson
Monica Månhammar
Marianne Pleen-Schreiber
Elisabeth Smedberg

Ioannis Sourdis
Pedro Trancoso
Vincenzo Gulisano
Javier San Martin
Ainhoa Cortés
Leticia Zamora-Cadenas
Benjamin Vedder
Jonny Vinter
Magnus Jonsson
Martin Dybdal
Martin Elsman
Robert Krook
Abhiroop Sarkar
Andrea Svensson
Rolf Snedsböl
Lars Svensson
Yinan Yu
Anneli Storberg

Thanks to all ex-colleagues at RISE and all current colleagues at Chalmers!

Links

https://github.com/AccelerateHS/accelerate
https://github.com/eholk/harlan
https://abhiroop.github.io/pubs/sensevm_mplr.pdf
http://svenssonjoel.github.io/writing/bb.pdf
http://svenssonjoel.github.io/writing/MetaAuto.pdf
http://svenssonjoel.github.io/writing/almost_free.pdf

Abstract

There is a divide that makes modern software development methodologies and tools inaccessible to programmers of many very interesting kinds of computer platforms. GPUs, for example, are very efficient for certain types of computations, but are programmed mainly in extensions to C with support for the quirky data-parallism where the GPU excels. Microcontrollers are limited in resources which makes it hard to support modern managed languages and runtime systems. GPUs and Microcontrollers are examples of two very fun, useful and ubiquitous computer platforms which are hard to program using high-level languages.

In this talk I outline my research history in programming of quirky hardware using functional languages and go more in depth on our current line of work in developing runtime systems that can support functional programming on microcontroller systems.