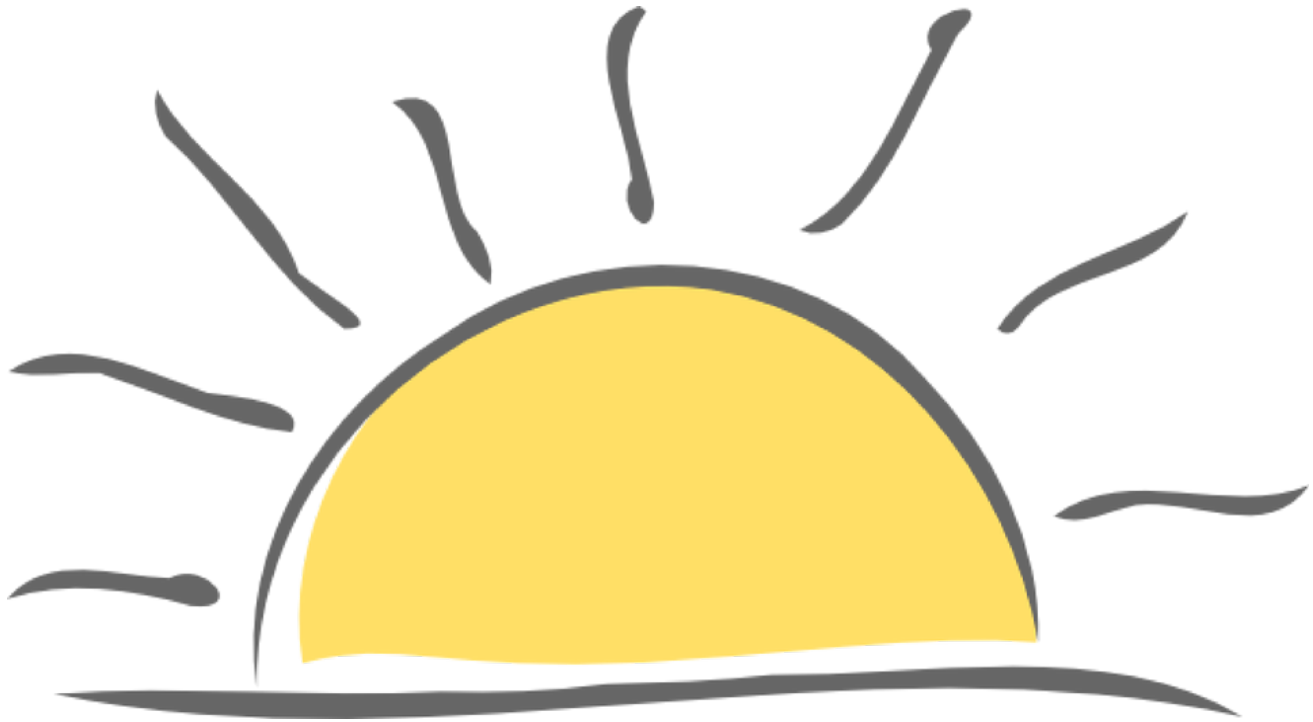
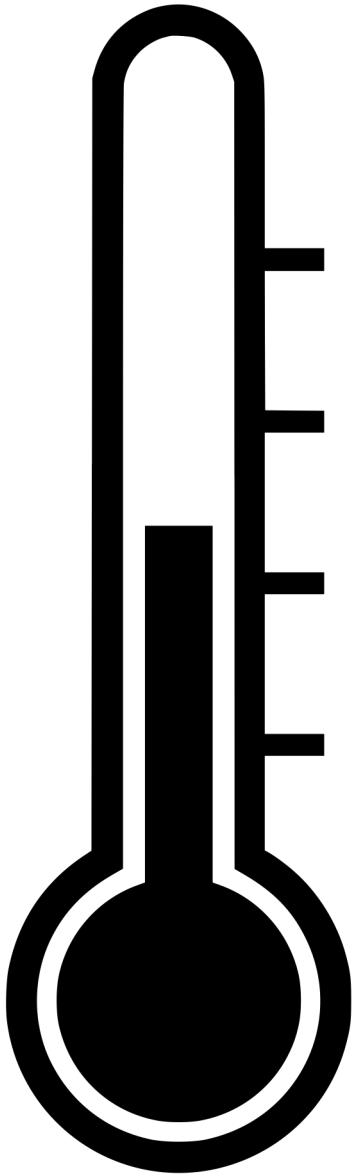
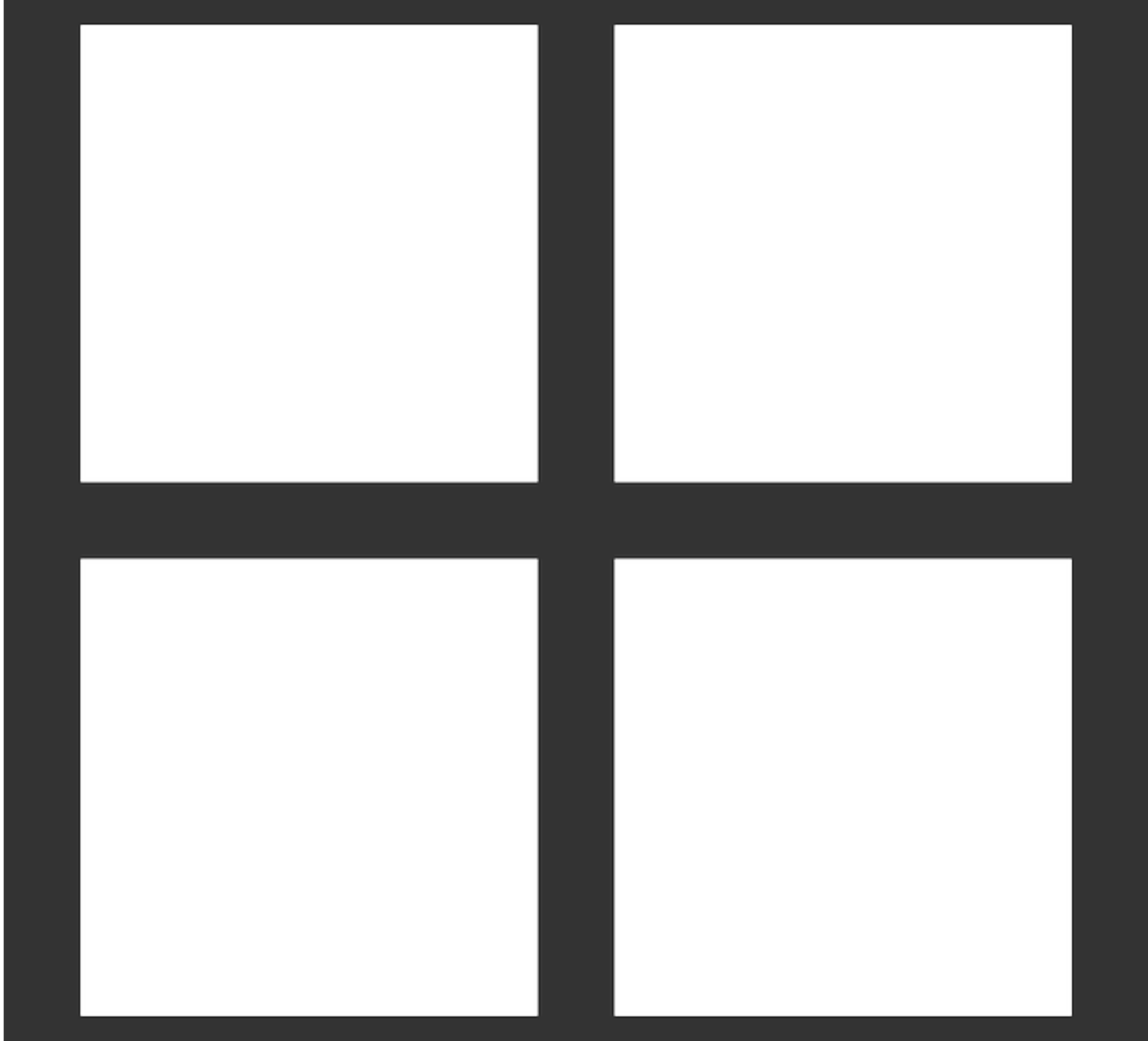
2019



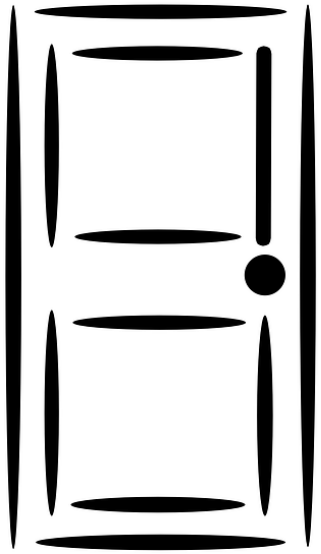
32°



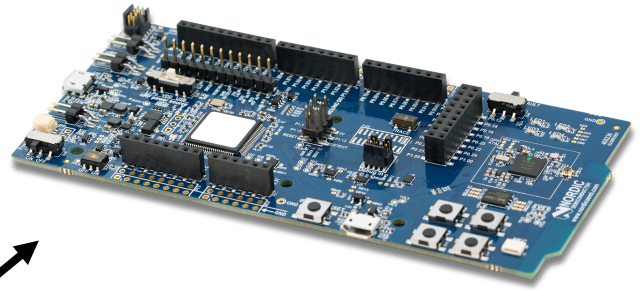
Open?



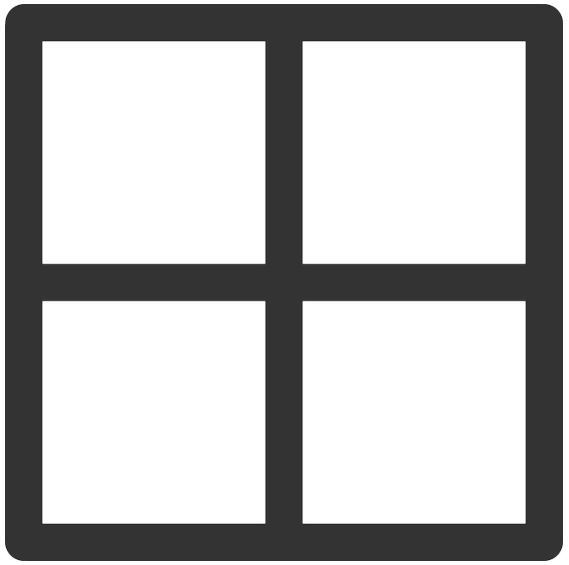
In



Halexa



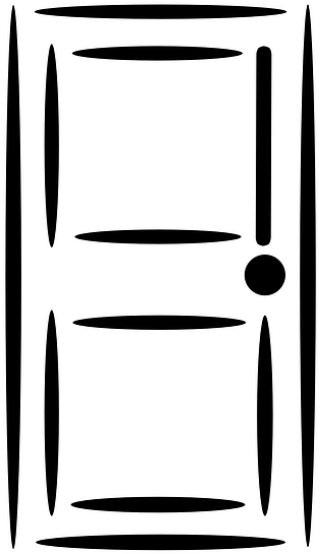
Open



> 30°

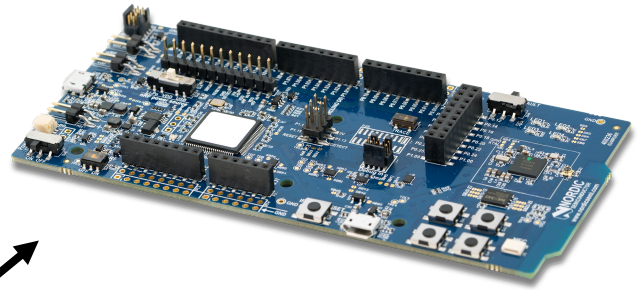
(64 MHz, 256 KB RAM, BLE)

In

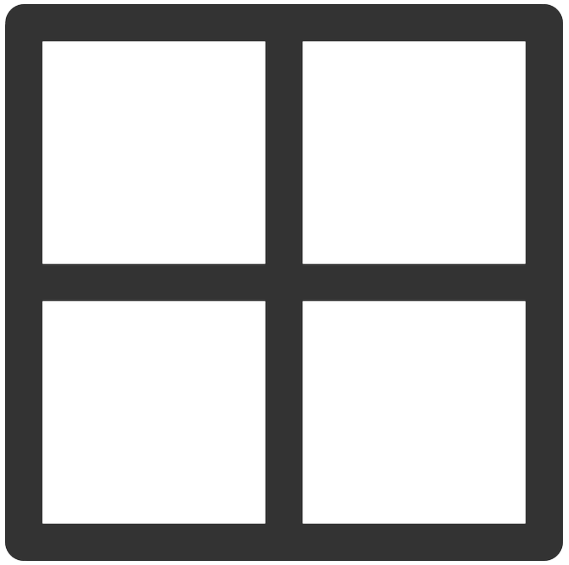


≤ 30°

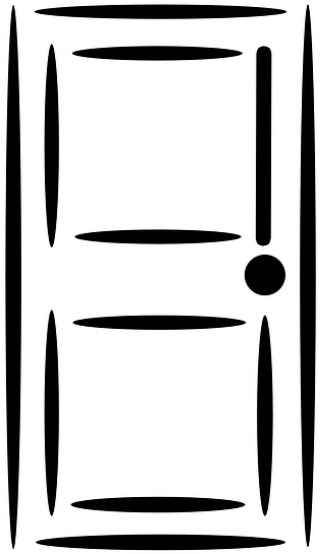
Halexa



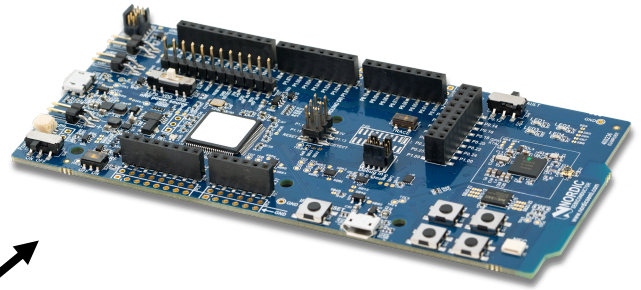
Close



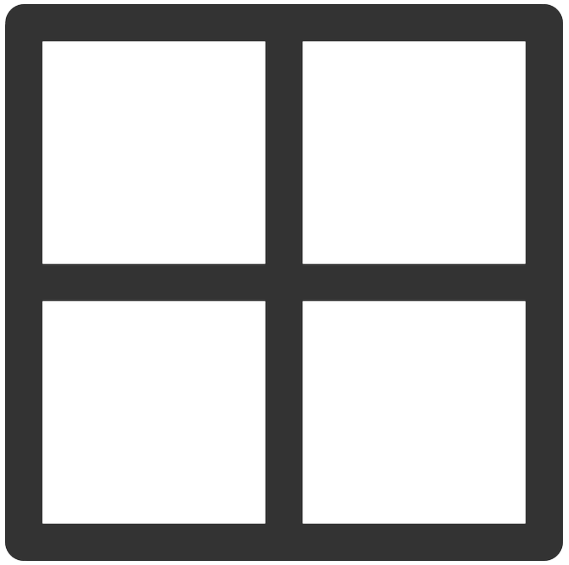
Out



Halexa

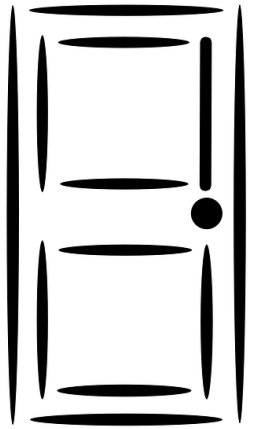


Close



Challenge: Modeling Inputs

Changes need not be *periodic* or *synchronized* with other inputs



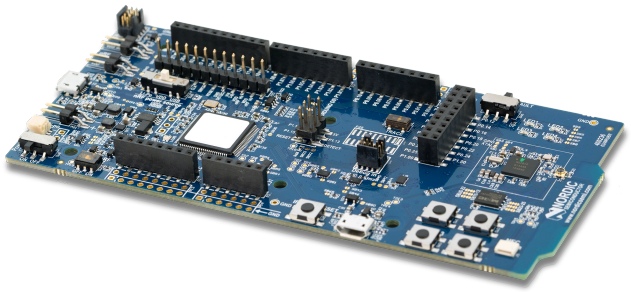
In, __ , __ , __ , Out , __ , In ,...

30°, 31°, 32°, 33°, 31°, 28°, 26°,...

Input values may change over time

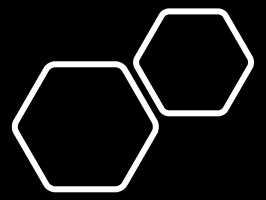
Challenge: SDK is low-level

C



- *manual* memory management
- *difficult* to protect sensitive data
- `Segmentation fault (core dumped)`

How should you program Halexa?



Towards Secure IoT Programming in Haskell

NACHIAPPAN VALLIAPPAN

ROBERT KROOK

ALEJANDRO RUSSO

KOEN CLAESSEN

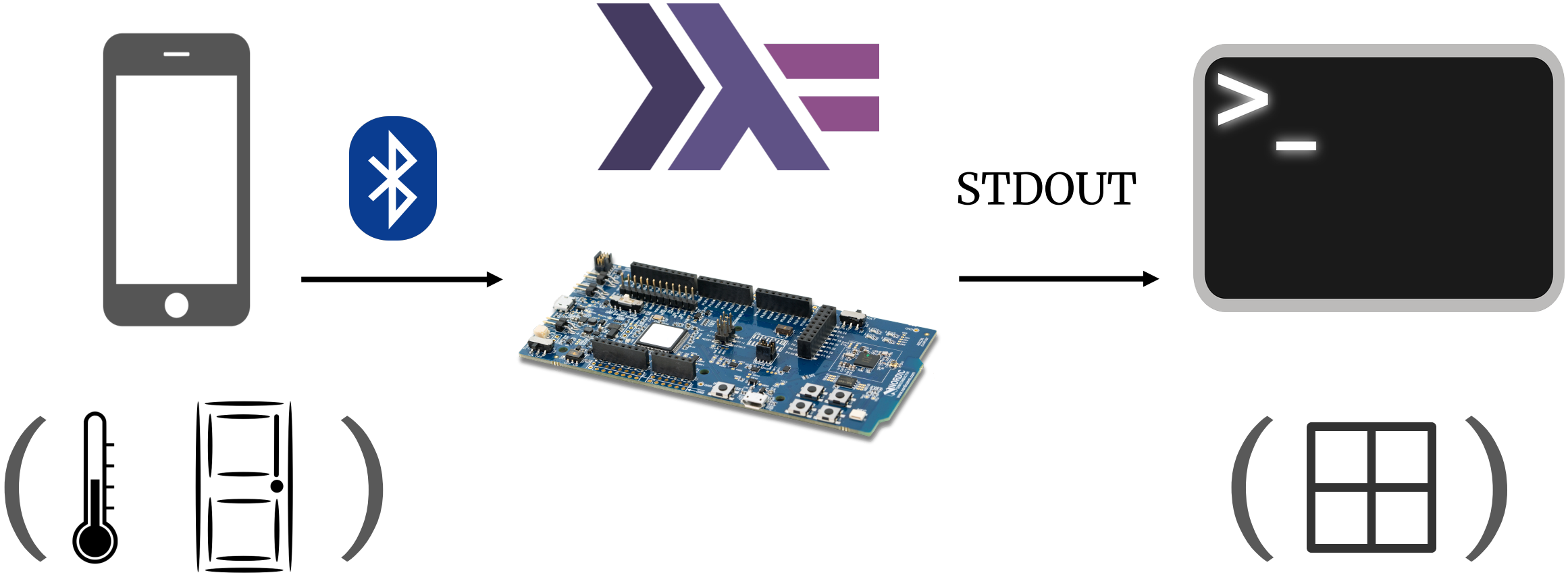
OCTOPI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Main Idea

Program Halexa and similar “reactive” IoT applications using a Haskell eDSL that generates C

Demo Setup



What is Haski?

- Monadic eDSL that models input and output as streams

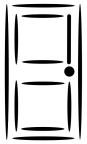
```
prg :: Stream Input -> Haski (Stream Output)
```

- Generates C code

```
Output prg_step(Input inp, prg_mem *self)
```

Programming in Haski, an Overview

```
halexa :: Stream Status
       -> Stream Int
       -> Haski (Stream WindowOp)
```



```
type Status   = Maybe Action
data Action   = In    | Out
data WindowOp = Open  | Close
```

Programming in Haski, an Overview

```
halexa :: Stream Status -> Stream Int -> Haski (Stream WindowOp)

halexa = node "halexa" $ \stat temp -> mdo
  isHot     <- letDef $ temp `gtE` 30
  recentAct <- stat `match` (maybe pastAct val)
  pastAct   <- Out `fby` recentAct
  dec       <- recentAct `match` \case
    In  -> ifte isHot (val Open) (val Close)
    Out -> val Close
  return dec
```

Mutually recursive definitions

```
{-# LANGUAGE RecursiveDo #-}

halexa                                    -> mdo
  ...
  recentAct <- stat `match` (maybe pastAct val)
  pastAct   <- Out `fby` recentAct
  ...

instance MonadFix Haski where
  mfix f = …
```

Pattern Matching

```
match :: … => Stream a -> (a -> Stream b)
      -> Haski (Stream b)
```

```
halexa
  ...
  dec <- recentAct `match` \case
    In  -> ifte isHot (val Open) (val Close)
    Out -> val Close
  ...
```

```
\x -> case x of
```

```
switch (recentAct) {
   case In:     dec = …
   case Out:   dec = …
}
```

Pattern Matching using “The Trick”

Require type ‘a’ to be *finitely enumerable*

```
match :: (HaskiType a, FinEnum a)
      => Stream a -> (a -> Stream b)
      -> Haski (Stream b)
```

Generate code for the branches by enumerating all values of ‘a’!
(a.k.a. “The Trick”)

Functions (or “nodes”)

```
halexa :: Stream Status -> Stream Int -> Haski (Stream WindowOp)
halexa = node "halexa" $ \stat temp -> mdo
```

```
node :: … => String -> (a -> b) -> (a -> b)
```

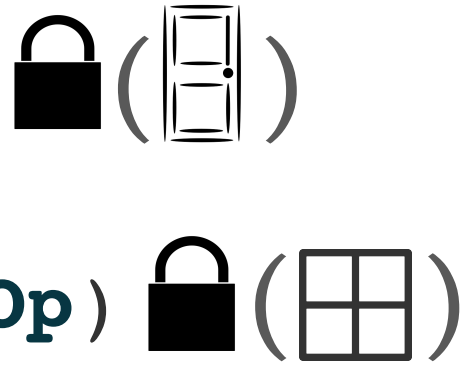
```
int hal_step(int stat, int temp, hal_mem *self){
    …
}

struct hal_mem { int prevAct; };
```

Your whereabouts must be private!

Haski supports an information-flow control extension

```
sec_halexa :: LStream Status
    -> Stream Int
    -> Haski (LStream WindowOp)
```



IFC Labeling primitives

```
label (High :: Label) (stat :: Stream Status)
      :: LStream Status

unlabel (secretStat :: LStream Status)
      :: Haski (Stream Status)

… (see paper)
```

Secure Halexa

```
sec_halexa :: LStream Status -> Stream Int
              -> Haski (LStream WindowOp)
sec_halexa = node "halexa" $ \secretStat temp -> mdo
    ...
    stat <- unlabel secretStat
    dec <- ...
    secretDec <- label High dec
    return secretDec
```

Compile time IFC, *but not using types*

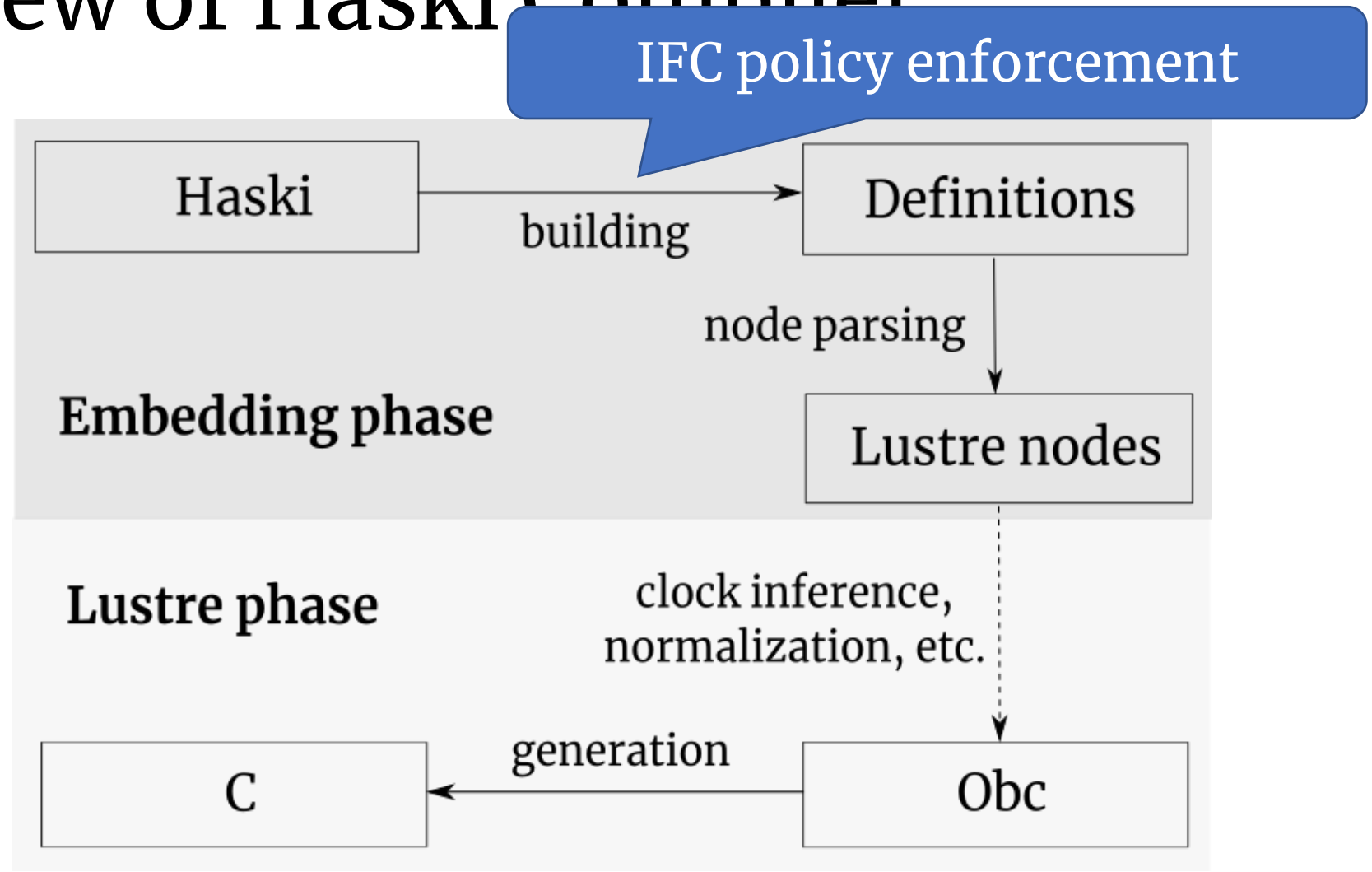
> Labels are *static values*, not types

i.e., `High :: Label`, not `High :: *`

> IFC primitives are “computed away” at compile time

i.e., `sec_halexa ≈ halexa`, if there’s no violation

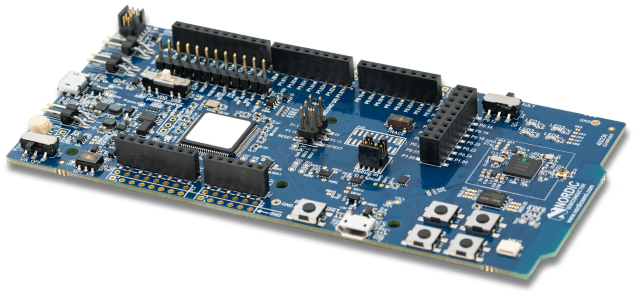
Overview of Haski Compiler



Halexa in Action

`Halexa`

Zephyr™



Haski + runtime for Zephyr

RTOS with a callback-based C API

nRF52840-DK Nordic Semiconductor

Stream functions and Callbacks

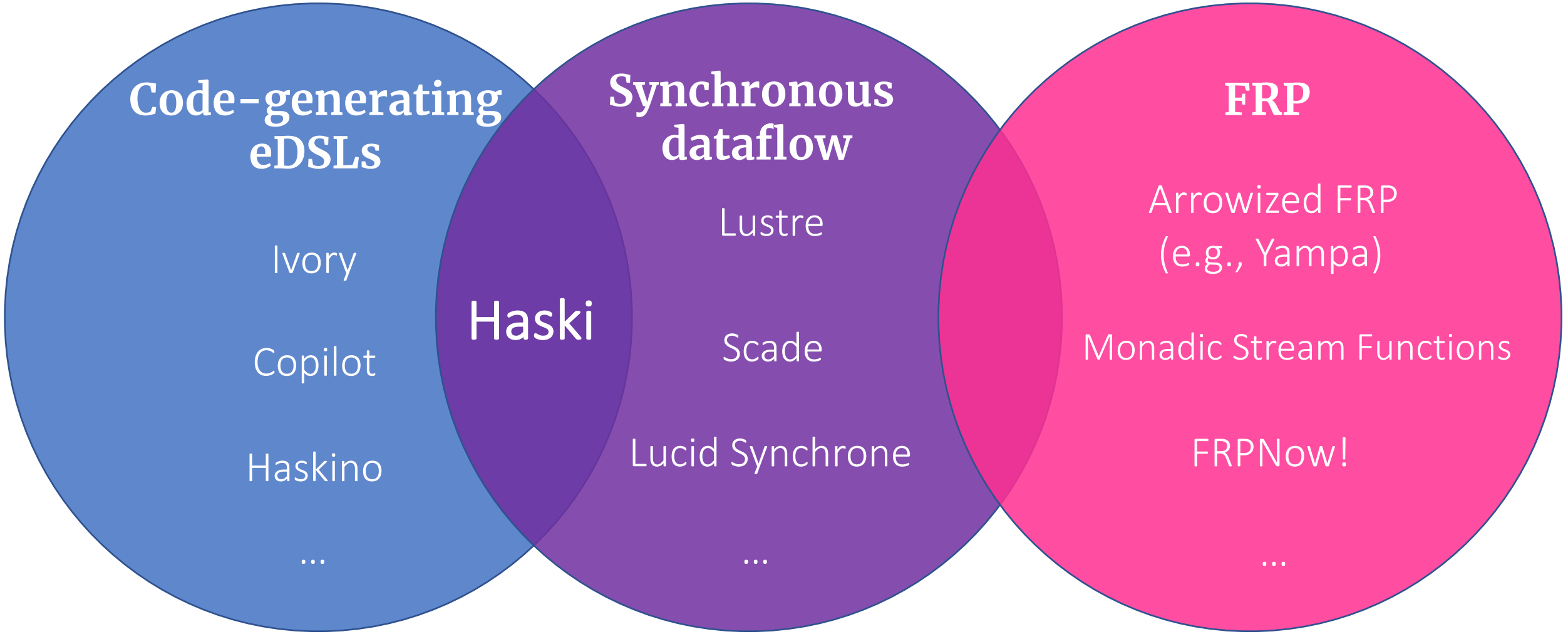
Halexa

```
int hal_step(int stat, int temp, hal_mem *self)
```

currently requires manual intervention


```
static u8_t notify_temperature(...,void* data){
    ...
    int *temperature = (int*)data;
    hal_step(*temperature, NOTHING, mem);
    ...
}
```

Landscape of reactive DSLs



In a nutshell

Haski presents a novel eDSL design, based on Lustre, for programming resource constrained IoT devices, and supports a lightweight IFC extension.

https://github.com/OctopiChalmers/haski/