# Green Computing for the Internet of Things

Mart Lubbers and Pieter Koopman

Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands
firstname@cs.ru.nl

## 1 Exercises

### 1.1 Hello world!

The first program one writes when trying a new programming is *Hello world!* that simply prints the text to the screen to verify that the compilation and execution process works. In case of microcontrollers, the builtin LED (a monochrome 1-pixel screen) is often blinked to signify that everything is set up correctly. Therefore, this first assignment is just to verify your mTask installation is working and you are able to compiler and execute Clean/mTask programs.

```
 1 module blink
 2 import StdEnv, iTasks                  // Imports
 3 import mTask.Interpret
 4 import mTask.Interpret.Device.TCP
 5
 6 Start w = doTasks main w               // Start the engine
 7
 8 main :: Task Bool                      // Main task
 9 main =          enterDeviceInfo
10     ≫? \spec→withDevice spec (\dev→liftmTask blink dev)
11 where
12     enterDeviceInfo :: Task TCPSettings   //Ask the user for the device settings
13     enterDeviceInfo = enterInformation [] <<@ Label "Device information"
14
15 blink :: Main (MTask v Bool) | mtask v    //mTask task
16 blink = declarePin D4 PMOutput \d4→         // Builtin LED on the D1 Mini
17     fun \blinkfun=(\x→                 // Recursive function to blink
18              delay (ms 500)            // Wait for 500ms
19         ≫|. writeD d4 x               // Set the LED to the current state
20         ≫|. blinkfun (Not x))         // Recursively call with inverse state
21     In {main=blinkfun true}            // Main program
```

All Clean programs start with a module declaration, in this case the module name is `blink`. To work with the mTask library, some imports are required (see Line 2). As mTask is embedded in iTask, iTask's `doTasks` is called on Line 6 as the `Start` rule of the program. This function starts the iTask engine and execute the `main` task that is given as the second argument. The `main` task is defined at Line 8 and first asks the use to enter the device information (see Line 12). With this information, mTask's `withDevice` function is called that allows the program to interact safely with a connected microcontroller. Using `liftmTask`, an mTask task is lifted to iTask, i.e., it is compiled to bytecode, sent to the microcontroller and the result is observable in iTask. The `blink` task is an mTask task defined at Line 15 with the type `Main (MTask v Bool) | mtask v` that can be read as: *An mTask task of the type* Bool *parametric in the view* `v` *as long as this* `v` *implements the classes defined by the* `mtask` *class collection.*

Every mTask program is wrapped in a `main` record to assure that functions and sensors are only defined at the top level. First on Line 16 the GPIO pin D4, connected to the builtin LED is set to output mode. Then on Line 17 the recursive function `blink` is defined that has one boolean argument, the state. This function first waits for 500 milliseconds (Line 18), then writes the state to the pin to either turn on or turn off the LED (Line 19) and finally it calls itself recursively with the inverse of the state (Line 20). When calling this function at Line 21 with some initial state, it results in a blinking LED when executing.

To measure power, connect the devices according to the diagram shown in Figure 1 if you have the version with the Wemos D1 mini pro (green) or Figure 2 if you have the regular D1 mini (blue).

**Connect only the leftmost microprocessor, the one on the breadboard, with an USB cable to your PC!** This microprocessor is preprogrammed to run a power-monitor program. The second microprocessor, fitted on the triple base, is executing mTask programs.

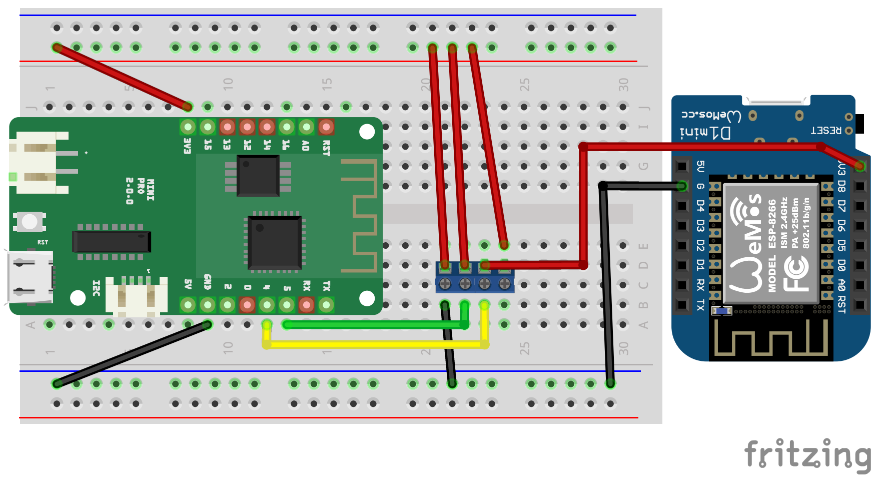

**Figure 1.** Wiring instructions for the powermonitor using the Wemos D1 mini pro.

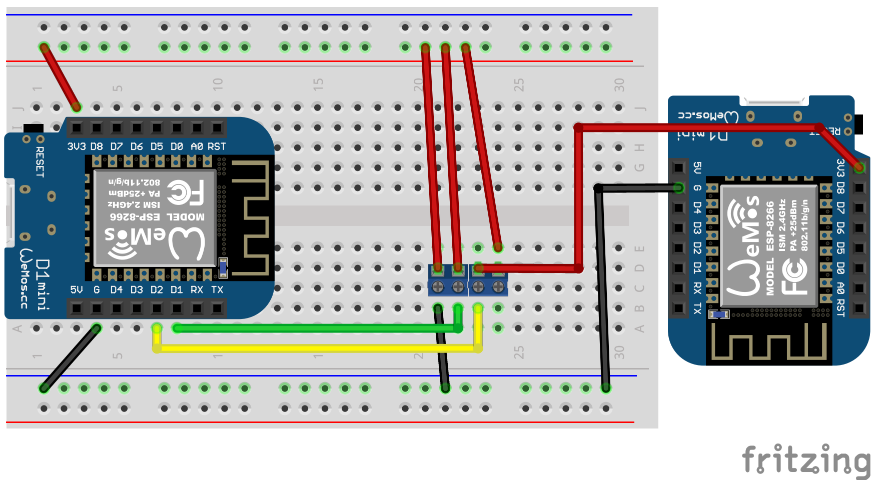

**Figure 2.** Wiring instructions for the powermonitor using the Wemos D1 mini.

**Exercise 1:** Hello world! (`blink.icl`)

Compile and run the `blink` module by running (see the readme for instructions).

```
nitrile build --only=blink
./blink
```

When you have connected the device properly, it should connect to either one of the networks and show its address. Enter this address in the `enterDeviceInfo` task together with the default mTask port number 8123. If you then press *Continue* the light on top of the microcomputer should turn on and off according to the given frequency (500 milliseconds).

mTask programs are constructed and compiled to byte code at runtime so it is possible to tailor make the program according to the needs of the current state. For example, it may is possible to ask a time between state changes from the user and inline that in the blink program.

**Exercise 2:** Tailor-made blinking (`blinkparam.icl`)

Change the `blink` function so that it gets a parameter, i.e. the time between state changes as follows:

```
blink :: Int → Main (MTask v Bool) | mtask v
blink wait = declarePin D4 PMOutput \d4→
```

This does require you to provide this extra argument as well. By using the `-&&-` combinator, the `enterDeviceInfo` can be combined with another task that asks the user for a time in milliseconds. The result of this line will then be a tuple that can be pattern matched and passed on to the `blink` function as follows:

```
    >>? \(spec, wait)→withDevice spec (\dev→liftmTask (blink wait) dev)
```

Finally, adapt the `delay` task in the mTask task so that it uses `wait` instead of a fixed number of milliseconds.
**Hint:** `wait` *is of type* `Int` *and not of type* `v Int` *so you have to lift it to the mTask domain first.*

Tasks in mTask can share information through SDSs with iTask. This allows for very dynamic behaviour, such as setting the blinking frequency during the run time of the mTask task.

**Exercise 3:** Dynamic blinking behaviour (`blinkshare.icl`)

To create an SDS that is available globally, the `sharedStore` function is used:

```
delayShareI :: SimpleSDSLens Int
delayShareI = sharedStore "delay" 500
```

This SDS can be lifted to an mTask SDS using the `liftsds` construct as follows:

```
blink :: Main (MTask v Bool) | mtask, liftsds v
blink = declarePin D4 PMOutput \d4→
    liftsds \delayShareM=delayShareI
    In fun \blinkfun=(\x→
   ...
```

Adapt the `blinkfun` function so that it first reads the value of `delayShareM` and uses it as the time to wait.
**Hint:** `getSds` *yields an unstable value so you have to use a sequential combinator that steps on an unstable value.*

## 1.2 Temperature monitor

***Exercise 4:*** *Initial temperature monitor (*`tempmon.icl`*)*

The appointed device contains a SHT30x digital humidity and temperature sensor (DHT) that communicates with the microprocessor using I$^2$C. While task values can be observed directly using the appropriate combinators (`>&>`, `>&*`), writing the value to an SDS is an easier option.
Create a globally available SDS to store the temperature.
**Hint:** *In mTask, temperature is measured in °Celcius stored in a* `Real`*.*
*To view this SDS during operation, create* `viewTemperature` *task and run that in parallel with the* `liftmTask` *task:*

```
    viewTemperature :: Task Real
    viewTemperature = viewSharedInformation [] tempShareI
        <<@ Label "Current temperature (C)"
```

*Finally, implement the temperature monitoring task.*

```
tempmon :: Main (MTask v Real) | mtask, dht, liftsds v
tempmon = DHT (DHT_SHT (i2c 0x45)) \dht→
    liftsds \tempShareM=tempShareI
    In fun \tempfun=(\()→temperature dht
        >>~. \t→setSds tempShareM t
        >>|. tempfun ()
    ) In {main=tempfun ()}
```

As you can see, this version of the temperature monitoring application generates a lot of traffic since the temperature is bound to be different each time it is measured.

**Exercise 5:** Temperature monitor, second iteration (`tempmon2.icl`)

The temperature is not bound to change every millisecond so it's not required to measure it that often.
Adapt the program so that there is a 5 second delay in between the measurements. This is done by adding a `delay` somewhere.

The second iteration reduced the number of reading greatly but still even the most minute temperature changes are reported.

***Exercise 6:*** *Temperature monitor, third iteration (*`tempmon3.icl`*)*

Adapt the program so that only changes bigger than half a degree Celcius are reported. Using the step combinator (`>>*.`) you can observe the value of a task and step when the predicate holds.
If the predicate does not match, the left hand side of the step combinator is scheduled for execution some other time, depending on the characteristics of the task. In case of the temperature sensor, the next execution will be within two seconds.
**Hint:** *Use functions to do the heavy lifting. For example, to take the absolute value over real numbers you can define:*

```
In fun \abs=(\x→If (x >. lit 0.0) x (lit (-1.0) *. x))
```

*Furthermore, remember that multiparameter functions are always written with tuple notation. So a function that determines if two real numbers differ more than* 0.5 *is defined as:*

```
In fun \differsenough=(\(old, new)→abs (old -. new) >. lit 0.5)
```

## 1.3 Motion detection

If a person enters the room, the temperature is bound to change faster. Using the passive infrared (PIR) sensor, motion can be detected. If the PIR detects motion, the GPIO pin it connects to will be high for a couple of seconds. Polling this pin continuosly to check whether there is motion consumes a lot of energy. Luckily, using an *high* interrupt handler we get notified when the pin is high, i.e. when there is motion.

**Exercise 7:** Temperature monitor, Motion detection (`tempmon4.icl`)

Adapt the temperature monitor so that the temperature is only measured every minute. In addition, if motion is detected, record the motion and take an extra temperature measurement.

1. Adapt the temperature function so that it measures only once every minute (this is done using `temperature``).
2. Add the declaration of the PIR sensor to the top level of the mTask task using the `PIR` functions:

   ```
   tempmon = PIR D3 \pir→
   ```

3. Add an extra SDS to record the number of movements. E.g.

   ```
   movementShareI :: SimpleSDSLens Int
   ```

4. Extend the main task using a parallel combinator so that it not only calls `tempfun` but also `motionfun`.
5. Extend `motionfun` so that it also measures the temperature when detecting motion.

## A Solutions

### **Solution 1:** Hello world! (`blink.icl`)

### **Solution 2:** Tailor-made blinking (`blinkparam.icl`)

```
import StdEnv, iTasks
import mTask.Interpret
import mTask.Interpret.Device.TCP

Start w = doTasks main w

main :: Task Bool
main =                 enterDeviceInfo -&&- enterDelayTime
    >>? \(spec, wait)→withDevice spec (\dev→liftmTask (blink wait) dev)
where
    enterDeviceInfo :: Task TCPSettings
    enterDeviceInfo = enterInformation [] <<@ Label "Device information"

    enterDelayTime :: Task Int
    enterDelayTime = enterInformation [] <<@ Label "Time between state change (ms)"

blink :: Int → Main (MTask v Bool) | mtask v
blink wait = declarePin D4 PMOutput \d4→
    fun \blinkfun=(\x→
            delay (lit wait)
        >>|. writeD d4 x
        >>|. blinkfun (Not x))
    In {main=blinkfun true}
```

### **Solution 3:** Dynamic blinking behaviour (`blinkshare.icl`)

```
import StdEnv, iTasks
import mTask.Interpret
import mTask.Interpret.Device.TCP

Start w = doTasks main w

delayShareI :: SimpleSDSLens Int
delayShareI = sharedStore "delay" 500

main :: Task Bool
main =         enterDeviceInfo
    >>? \spec→withDevice spec (\dev→
            liftmTask blink dev
        -|| updateSharedInformation [] delayShareI
    )
where
    enterDeviceInfo :: Task TCPSettings
    enterDeviceInfo = enterInformation [] <<@ Label "Device information"

blink :: Main (MTask v Bool) | mtask, liftsds v
blink = declarePin D4 PMOutput \d4→
    liftsds \delayShareM=delayShareI
    In fun \blinkfun=(\x→
            getSds delayShareM
        >>~. \wait→delay wait
        >>|. writeD d4 x
        >>|. blinkfun (Not x))
    In {main=blinkfun true}
```

## **Solution 4:** Initial temperature monitor (`tempmon.icl`)

```
import StdEnv, iTasks
import mTask.Interpret
import mTask.Interpret.Device.TCP

Start w = doTasks main w

tempShareI :: SimpleSDSLens Real
tempShareI = sharedStore "temp" 0.0

main :: Task Real
main =          enterDeviceInfo
    >>? \spec→withDevice spec (\dev→
            liftmTask tempmon dev
        -|| viewTemperature
    )
where
    enterDeviceInfo :: Task TCPSettings
    enterDeviceInfo = enterInformation [] <<@ Label "Device information"

    viewTemperature :: Task Real
    viewTemperature = viewSharedInformation [] tempShareI
        <<@ Label "Current temperature (C)"

tempmon :: Main (MTask v Real) | mtask, dht, liftsds v
tempmon = DHT (DHT_SHT (i2c 0x45)) \dht→
    liftsds \tempShareM=tempShareI
    In fun \tempfun=(\()→temperature dht
        >>~. \t→setSds tempShareM t
        >>|. tempfun ()
    ) In {main=tempfun ()}
```

## **Solution 5:** Temperature monitor, second iteration (`tempmon2.icl`)

```
import StdEnv, iTasks
import mTask.Interpret
import mTask.Interpret.Device.TCP

Start w = doTasks main w

tempShareI :: SimpleSDSLens Real
tempShareI = sharedStore "temp" 0.0

main :: Task Real
main =          enterDeviceInfo
    >>? \spec→withDevice spec (\dev→
            liftmTask tempmon dev
        -|| viewTemperature
    )
where
    enterDeviceInfo :: Task TCPSettings
    enterDeviceInfo = enterInformation [] <<@ Label "Device information"

    viewTemperature :: Task Real
    viewTemperature = viewSharedInformation [] tempShareI
        <<@ Label "Current temperature (C)"

tempmon :: Main (MTask v Real) | mtask, dht, liftsds v
tempmon = DHT (DHT_SHT (i2c 0x45)) \dht→
    liftsds \tempShareM=tempShareI
    In fun \temp=(\()→temperature dht
        >>~. \t→setSds tempShareM t
        >>|. delay (ms 5000)
        >>|. temp ()
    ) In {main=temp ()}
```

**Solution 6:** Temperature monitor, third iteration (`tempmon3.icl`)

```
import StdEnv, iTasks
import mTask.Interpret
import mTask.Interpret.Device.TCP

Start w = doTasks main w

tempShareI :: SimpleSDSLens Real
tempShareI = sharedStore "temp" 0.0

main :: Task Real
main =        enterDeviceInfo
    >>? \spec→withDevice spec (\dev→
            liftmTask tempmon dev
        -|| viewTemperature
    )
where
    enterDeviceInfo :: Task TCPSettings
    enterDeviceInfo = enterInformation [] <<@ Label "Device information"

    viewTemperature :: Task Real
    viewTemperature = viewSharedInformation [] tempShareI
        <<@ Label "Current temperature (C)"

tempmon :: Main (MTask v Real) | mtask, dht, liftsds v & fun (v Real, v Real) v
tempmon = DHT (DHT_SHT (i2c 0x45)) \dht→
    liftsds \tempShareM=tempShareI
    In fun \abs=(\x→If (x >. lit 0.0) x (lit (-1.0) *. x))
    In fun \differsenough=(\(old, new)→abs (old -. new) >. lit 0.5)
    In fun \tempfun=(\oldtemp→temperature dht
        >>*. [IfValue (\newtemp→differsenough (oldtemp, newtemp))
                \newtemp→setSds tempShareM newtemp]
        >>=. \newtemp→tempfun newtemp
    ) In {main=tempfun (lit 0.0)}
```

**Solution 7:** Temperature monitor, Motion detection (`tempmon4.icl`)

```
import StdEnv, iTasks
import mTask.Interpret
import mTask.Interpret.Device.TCP

Start w = doTasks main w

movementShareI :: SimpleSDSLens Int
movementShareI = sharedStore "movement" 0

tempShareI :: SimpleSDSLens Real
tempShareI = sharedStore "temp" 0.0

main :: Task Int
main =         enterDeviceInfo
    ≫? \spec→withDevice spec (\dev→
           liftmTask tempmon dev
        -|| viewMovement
    )
where
    enterDeviceInfo :: Task TCPSettings
    enterDeviceInfo = enterInformation [] <<@ Label "Device information"

    viewMovement :: Task Int
    viewMovement = (viewSharedInformation [] movementShareI <<@ Label "No. of detected movements")
        -|| (viewSharedInformation [] tempShareI <<@ Label "Temperature")

tempmon :: Main (MTask v Int) | mtask, PIR, dht, liftsds v
tempmon = PIR D3 \pir→
    DHT (DHT_SHT (i2c 0x45)) \dht→
    liftsds \movementShareM=movementShareI
    In liftsds \tempShareM=tempShareI
    In fun \motionfun=(\()→
            interrupt rising pir
        ≫|. updSds movementShareM ((+.)(lit 1))
        ≫|. temperature dht
        ≫~. \newtemp→setSds tempShareM newtemp
        ≫|. motionfun ())
    In fun \abs=(\x→If (x >. lit 0.0) x (lit (-1.0) *. x))
    In fun \differsenough=(\(old, new)→abs (old -. new) >. lit 0.5)
    In fun \tempfun=(\oldtemp→temperature` (BeforeSec (lit 60)) dht
        ≫*. [IfValue (\newtemp→differsenough (oldtemp, newtemp))
                \newtemp→setSds tempShareM newtemp]
        ≫=. \newtemp→tempfun newtemp)
    In {main= motionfun () .||. tempfun (lit 0.0)}
```