Green Computing for the Internet of Things

Mart Lubbers(✉) and Pieter Koopman

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

Abstract. The Internet of Things (IoT), contains a huge number of nodes and this amount is rapidly increasing. This makes it worthwhile to invest in energy saving for the edge nodes even when they are not battery powered. Using cheap and energy efficient ocessor based hardware for the edge nodes yields a significant contribution to green computing. However, this hardware has major restriction on the processing power and memory available. These restrictions impose substantial constraints on the software executable on such nodes.

Task-oriented programming (TOP) offers a convenient way to program entire IoT applications from a single source at a high level of abstraction. A tailor-made domain-specific dialect of TOP combines the advantages of this paradigm with the constraints of ocessors.

In this paper we show that we can reduce the energy consumption of the edge nodes significantly in TOP. The system switches automatically to a low-power sleep mode when the current state of all tasks in the system allow a nap. In addition, we introduce a high-level way to deal with interrupts. This can replace the power hungry polling of sensors by much greener waiting for an event in a low-power mode of the system.

Keywords: internet of things · green computing · task-oriented programming · functional programming · Clean · mTask

This work received financial support through the Erasmus+ Strategic Partnership for Higher Education *SusTrainable—Promoting Sustainability as a Fundamental Driver in Software Development Training and Education* (project number 2020-1-PT01-KA203-078646), funded by the European Union and coordinated by the University of Coimbra, Portugal.
The information and views set out in this publication are those of the authors and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
C. Grelck et al. (eds.), *Promoting Sustainability as a Fundamental Driver in Software Development Training and Education*, LNCS 15670, pp. 1–28, 2026.
https://doi.org/10.1007/978-3-032-22278-7_1

1 Introduction

The Internet of Things (IoT), is the collection on interconnected smart devices that senses and controls objects in the world. It consists of a wide variety of nodes including smart thermostats, lightbulbs, doorbells, human presence sensors and electricity meters. The number of IoT devices is huge. It is estimated at 14.4 billion at the end of 2022 and is expected to grow to 27 billion connected devices at the end of 2027.[1] As a reference the world population is nearly 8 billion persons in 2022 and is expected to grow with a few percent over this period.[2]

The number of nodes in the IoT is not only large, most of these nodes are also running 24/7. By the nature of the IoT there is a significant amount of communication between the devices. This makes it challenging to run the IoT in a sustainable way. Fortunately there are some opportunities to limit the energy consumption of the IoT. First, many tasks of the IoT nodes are rather simple. This implies that we can use a small and energy-efficient microcontroller based node instead of a solution that is based on a single-board computer, like a Raspberry Pi. Second, we can use edge computing to limit the amount of communication. For instance, a sensor reports only significant changes in its readings to a server instead of submitting each value. Third, many microcontrollers have facilities to reduce the power consumption. For instance they can switch off the WiFi-radio when there is temporarily no need for communication. When there is no need for any activity during some time the microcontroller can even switch to a low-power sleep mode. Finally, some sensors have the ability to wake up the microcontroller whenever required from a sleep state using hardware interrupts. This prevents regular waking up of the microcontroller to check whether something interesting is happening.

The IoT not only consumes a lot of energy, its applications help to save power by optimizing climate control systems in buildings, lighting based on occupancy, predictive maintenance to name just a few application areas. There is obviously a balance in the power it takes to operate the IoT and the energy savings and increase of comfort that is provided by the IoT. This makes it worthwhile to ensure that IoT systems themselves are sustainable.

To give an impression of the power consumption of IoT nodes we list some figures. A single-board ocessor based computer like the Raspberry Pi 4 consumes about 4W when it is idle and 6W when its four cores are active. This power consumption excludes the power used by peripherals and the WiFi communication. This is not bad compared to the 60 W consumed by the average 15.6 in laptop.[3] The popular ESP8266 microcontroller consumes a maximum of 0.5 W during active WiFi communication. When the WiFi is switched off, modem sleep, this is reduced to 0.05 W. When the microcontroller takes a nap, light sleep,

[1] https://iot-analytics.com/number-connected-iot-devicesaccessed 11th November, 2022.
[2] https://ourworldindata.org/world-population-growthaccessed 11th November, 2022.
[3] https://www.netbooknews.com/tips/how-many-watts-laptop-use/accessed 14th November, 2022.

it consumes only 0.001 W.[4] Moreover, a single-board computer is typical about ten times more expensive than a simple microcontroller board. There are microcontrollers that consume considerable less energy, but they are less common and usually more expensive.

Energy efficiency is obviously very important for battery operated IoT nodes. However, due to the huge number of devices it is important for the entire IoT. Operating 14.4 billion IoT nodes based on Raspberry Pi's for a year requires about 600TWh. As a reference, the annual productions of the single Dutch nuclear power plant is 4TWh.[5] This indicates that operating the IoT takes a serious amount of energy and reducing the power demand of individual nodes makes the system more sustainable.

These numbers make it very attractive to build IoT nodes with microcontrollers rather than with single-board computers such as the Raspberry Pi. However, there is a significant price to be paid. First of all the microcontrollers are much slower and have far less memory than the single-board computers. As a consequence these IoT nodes run typically without a full operating system (OS). Often there is just the user program. Sometime the user program is constructed with the help of a special purpose OS like FreeRTOS. This makes programming IoT systems and updates of these programs very complicated. Especially when several subtasks have to run on the IoT node. Without OS there is no support for multithreading. This implies that the user program has to interleave the subtasks. Modem sleep and light sleep have to be controlled by the user program containing those subtasks. The system can only go to a low-power state when this is possible in all subtasks.

Adding such a node to the IoT is also challenging. Due to the limitations of the microcontroller it is typically programmed in a stripped-down dialect of C++ or Python. For the communication it uses a protocol like MQTT on top of the normal internet protocols. The server that has to cooperate with the IoT node is typically written in high-level language like Java. Since the IoT node has a very limited amount of memory, the sensor reading is be stored in a database on the server. The user interface of the program either runs in a browser, using HTML and for instance JavaScript or WebAssembly, or in a smartphone app for Android and iOS. The program on the IoT node also has to take care about its own updates since there is no OS and physical access to the node for an update is usually too demanding. This implies that many programming languages, protocols and systems are involved in a single task of the IoT. These differences in data types and interaction are known as *semantic friction*. In general, this makes the development of IoT applications cumbersome and its maintenance even more challenging.

[4] https://www.espressif.com/sites/default/files/documentation/9b-esp8266-low_power_solutions__en.pdf accessed 11th November, 2022.

[5] https://www.rijksoverheid.nl/onderwerpen/duurzame-energie/opwekking-kernenergie accessed 15th November, 2022.

1.1 Tierless Program Stacks

These problems can be prevented by generating all required software for the entire system from a single source. Such tierless systems are known for web stacks, e.g. Links [4], Hop [15], and iTask [12]. These systems focus on webpages and the associated servers.

In this paper we introduce a tierless system focussed on task-oriented programming (TOP). As the name suggests this declarative formalism focusses on tasks. These pieces of work are executed by humans or machines, including the edge nodes of the IoT. Tasks are here either primitive tasks like reading a sensor, operating an actuator, and filling out a web form or subtasks composed by combinators to more complex tasks. For instance, we can compose tasks in parallel as an alternative for each other or subtasks that are both required without fixing their order. Subtasks can also be composed sequentially. In contrast to ordinary computations, tasks can inspect the current value of subtasks. They can decide to move on based on the current value of a subtask, even when that subtask is not finished yet.

The iTask system is the oldest implementation of TOP [12,14]. It creates a web server, the associated storage and tailor-made web pages for all users of the system. This web interface guides the users through their current tasks. To ensure good performance, a part of the task is executed in the browser rather than on the server. The entire system is built as a shallowly embedded domain-specific language (DSL), in the pure functional programming language Clean [1,13]. This implies that the iTask system is a set of functions and associated data types that are fully integrated with Clean. The tasks can use all data types and computations of Clean. Each iTask program is a single piece of software in Clean. This implies that the compiler checks all types and definitions as usual. The entire system with dynamic web pages, storage control, the web server and the required networking is generated from this single tierless source.

The TOP approach is also very suited to control the IoT edge devices. They run tasks reading sensors, controlling peripherals and communicating with the servers. However, the microcontrollers powering these devices are not powerful enough to run the iTask programs. Since these edge devices do not require to have an integrated web server nor support heavy computations, it is also not necessary to run a full iTask server there. The mTask system is created to solve this gap. It is an embedded DSL in Clean that is fully integrated with iTask. Just like tasks in iTask, tasks in mTask can be dynamically created in the hosting programming language. The task-oriented programs written in the mTask language are compiled at runtime to domain-specific bytecode that fits on the edge devices with very limited power. The device is only programmed once with a domain-specific tailor-made tiny OS. This bytecode is shipped to the edge device and executed by the OS. Since everything is in the same source we obtain the best of both worlds. The tierless approach ensures consistency of the entire software system, while the separate mTask DSL facilitates executions of distinguished parts of this system on cheap and energy-efficient hardware.

The tiny OS on the devices is tailor made to execute mTask programs. It can use this domain-specific knowledge for smart scheduling. The subtasks are only scheduled after a complete rewrite step in the mTask DSL. This implies that the subtasks are never interrupted during communication with the server or peripherals, nor while they are accessing shared memory. This makes it easy to interleave subtasks efficiently. Moreover, the OS knows what the tasks are doing and uses this knowledge to save energy. For instance, the system can switch to a light sleep mode when all subtasks are waiting in a delay. When the system detects that some tasks is checking a temperature sensor in a high frequency, these sensor reading can be delayed since it is unlikely that the temperature changes rapidly. This delay can enable more execution time for more useful tasks or switching to sleep mode when there are no urgent tasks to be done. The automatic delay of subtasks can be fine tuned within the mTask program whenever that is desired. In a traditional tiered system it is much harder to achieve similar energy savings. The programmer needs detailed knowledge of all subtask that are present dynamically and schedule the sleep explicitly. This is difficult and error prone, especially when new versions of the software are required.

The iTask/mTask system enables sustainability in various different interpretations. First, the tierless approach ensures programs that are much smaller and easier to maintain than traditional tiered programs [10]. Second, the mTask addition to this TOP system enable us to use energy efficient and cheap ocessor based hardware. Finally, the optimized scheduling of mTask tasks saves energy by switching to sleep mode automatically.

1.2 Structure of the Paper

Section 3 explains the mTask DSL and its implementation in the amount of detail required for this paper. For a more thorough overview of this language the reader is referred to [5,6,8,9]. In Sect. 4 we explain how the interpreter can save energy by inspecting the current tasks. When these subtasks can be delayed, the entire system goes to a sleep mode to save energy. We can save even more energy when we replace polling of sensors by interrupts. The system can be sleeping until the interrupt wakes it up. In Sect. 6 we draw conclusions.

Appendix A explains how the software used in this paper can be installed. This software is required to compile and run the exercises provided in this paper. Appendix B shows a way to measure the actual poser consumption of a ocessors with an appropriate sensor and another microcontroller. This can be used to verify that techniques discussed here actually decrease the amount of energy used. The solutions to the exercises are provided in Appendix C.

2 Introducing Edge Computing by an Example

Traditionally, the first program that one writes when trying a new language is the so-called *Hello World!* program. This program has the single task of printing

the text *Hello World!* to the screen and exiting again. It helps the programmer to become familiarised with the syntax of the language and to verify that the toolchain and runtime environment are working. Microcontrollers usually do not come with screens in the traditional sense. Nevertheless, almost always there is a built-in 1 pixel screen with a 1bit color depth, namely the on-board LED. The *Hello World!* equivalent for microcontrollers blinks this LED.

2.1 Traditional Edge Computing

Using Arduino's C++ dialect to create the blink program results in the code seen in Listing 1.1. Arduino programs are implemented as cyclic executives and hence, each program defines a `setup` and a `loop` function. The `setup` function is executed only once on boot, the `loop` function is continuously called afterwards and contains the event loop. In the blink example, the `setup` function only contains code for setting the General Purpose Input/Output (GPIO), pin to the correct mode. The `loop` function alternates the state of the pin representing the LED between `HIGH` and `LOW`, turning the LED off and on respectively. In between, it waits 500ms so that the blinking is actually visible for the human eye.

```
void setup() {
    pinMode(D2, OUTPUT);
}

void loop() {
    digitalWrite(D2, HIGH);
    delay(500);
    digitalWrite(D2, LOW);
    delay(500);
}
```

Listing 1.1. Blinking an LED in Arduino C++.

To execute such a program on a ocessor this C++ code is first compiled to machine code for that processor. Next the compiled code is stored in the persistent flash memory of the ocessor with help of a small bootloader available on the microcontroller. After a soft restart the `setup` function is executed once and the `loop` function is repeated as long as the processor runs. The flash memory has a specified life of 100 000 write cycles. So, you may need to be careful about how often you write to it. Many programs can be uploaded and executed in such a memory unit. As soon as you start updating this type of memory during program execution, it might wear out rather quickly.

2.2 Task-Oriented Edge Computing

The mTask/iTask variant of the *Hello World!* program looks very much like the Arduino variant and is as useful for testing the toolchain. Therefore, this first

exercise is just to verify that your mTask installation is working and that you are able to compile 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 task 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 which device to use
13     enterDeviceInfo
14      = enterInformation [] <<@ Label "Device information"
15
16 blink :: Main (MTask v Bool) | mtask v     // mTask task
17 blink = declarePin D4 PMOutput \ledPin->  // Builtin LED of D1 Mini
18     fun \blinkfun=(\state->                // Recursive function to blink
19              delay (ms 500)                // Wait for 500ms
20         >>|. writeD ledPin state           // Switch LED to state
21         >>|. blinkfun (Not state))         // Call with inverse state
22     In {main=blinkfun true}                // Main program
```

Listing 1.2. Blinking an LED in mTask and iTask.

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 user 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 16 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.* This has to do with the tagless-final embedding technique that is used to create the mTask language [2]. Do not worry when you do not understand all technical details.

The start expression of 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 17 the GPIO pin D4, connected to the builtin LED is set to output mode. As mTask is hosted in a functional programming language, instead of using a loop,

a recursive function is used. Line 18 shows this function (`blinkfun`), that has one boolean argument, the state. This function first waits for 500ms (Line 19), then writes the state to the pin to either turn on or turn off the LED (Line 20) and finally it calls itself recursively with the inverse of the state (Line 21). When calling this function at Line 22 with some initial state, it results in a blinking LED when executing.

Installation of all required software and suitable ocessors to execute the examples and exercises of this chapter is outlined in Sect. A. To measure the power consumption of the ocessor executing our programs one can use the machinery discussed in Appendix B.

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 (500ms).

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

Change the `blink` function so that it gets a parameter, an argument, 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 at the call site. By using the `-&&-` combinator, the `enterDeviceInfo` can be combined with another task that asks the user for a time in ms. 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 ms.

Hint: `wait` *is of type* `Int` *and not of type* `v Int` *so you have to lift it to the mTask domain first by* `lit`*.*

---

3 The mTask DSL

Every mTask definition consists of a `main` record defining the initial task. In Listing 1.2 this is on Line 22. The task definition is constructed from basic

tasks like `writeD` and `delay`. Declarations like `declarePin` and `fun` add user-defined elements to the task. Constructors like `>>|.` are used to compose tasks sequentially to compound tasks. This particular combinator denotes sequential composition. There are also combinators to combine subtasks in parallel.

Tasks produce a value after each rewrite step. Such a result can be `NoValue`, for instance when the task is waiting for input. Values can be either stable or unstable. As a rule of thumb; a result is unstable when it can be different in a next evaluation, like reading a sensor.

Tasks can also communicate via a shared data source (SDS). As the name suggests this is a container holding values that can be inspected and updated by various tasks. Every SDS is typed, it always hold a value of the assigned type. The semantics of TOP prevent race conditions for updating SDSs.

A full overview of the language is not needed here. Language elements are introduced as they are required.

3.1 Implementation of the mTask system

The main interpretation, or view/backend of the mTask language, is the bytecode compiler. With it, and a handful of integration functions and tasks, mTask tasks can be executed on microcontrollers and integrated in iTask as if they were regular iTask tasks. Furthermore, with a special language construct, SDSs can be shared between mTask and iTask programs as well.

When using the bytecode compiler interpretation in conjunction with the iTask integration, mTask is a heterogeneous DSL [16]. Some components—for example the RTS on the microcontroller—is unaware of the other components in the system, and it is executed on a completely different architecture. The mTask language is an enriched simply-typed λ-calculus with support for some basic types, arithmetic operations, function definitions, and a task language.

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

The generated bytecode is shipped dynamically to the corresponding interpreter running on a ocessor. Since we can ship multiple tasks to the interpreter, it is actually a light-weight OS tailor-made for mTask programs. The mTask interpreter is stored in flash memory of the ocessor. It is available after a power interruption. To prevent wear of the ocessor memory, the tasks to be executed and their current state are stored in volatile RAM. The interpreter removes the state of the tasks and all associated code once the task is no longer needed from the iTask host program. The user needs only to install the appropriate software, dynamic compilation, shipping and removing tasks is done automatically.

3.2 The mTask embedding

To leverage the type checker of the host language, types in the mTask language are expressed as types in the host language, to make the language type safe.

However, not all types in the host language are suitable for microcontrollers that may only have 2KiB of RAM so class constraints are therefore added to the DSL functions. The most used class constraint is the `type` class collection containing functions for serialization, printing, iTask constraints, etc. Many of these functions can be derived using generic programming. An even stronger restriction on types is defined for types that have a stack representation. This `basicType` class has instances for many Clean basic types such as `Int`, `Real` and `Bool`. The class constraints for values in mTask are omnipresent in all functions and therefore often omitted throughout the chapters for brevity and clarity (Table 1).

Table 1. Translation from Clean/mTask data types to C++ data types.

Clean/mTask	C++ type	№bits
`Bool`	`bool`	16
`Char`	**char**	16
`Int`	`int16_t`	16
`Real`	**float**	32
`Long`	`int32_t`	32
`T = A | B | C`	**enum**	16

Listing 1.3 contains the definitions for the auxiliary types and type constraints (such as `type` and `basicType`) that are used to construct mTask expressions. The mTask language interface consists of a core collection of type classes bundled in the type class `class mtask`. Every interpretation implements the type classes in the `mtask` class There are also mTask extensions that not every interpretation or backend implements such as peripherals and iTask integration. Those are specified separately as they are not contained in the `mtask` class.

```
class type t | iTask,··· ,fromByteCode, toByteCode t
class basicType t | type t where···

class mtask v | expr,··· , int, real, long v
```

Listing 1.3. Classes and class collections for the mTask language.

Sensors, SDSs, functions, etc. may only be defined at the top level. The `Main` type is used to distinguish the top level from the main expression. Some top level definitions, such as functions, are defined using HOAS [3,11]. To make their syntax friendlier, the `In` type—an infix tuple—is used to combine these top level definitions as can be seen in `someTask` (Listing 1.4).

```
:: Main a = { main :: a }
:: In a b = (In) infix 0 a b

someTask :: MTask v Int | mtask v & liftsds v & sensor1 v &···
```

```
someTask =
    sensor1 config1 \sns1->
    sensor2 config2 \sns2->
       sds     \s1  = initialValue
    In liftsds \s2  = someiTaskSDS
    In fun     \fun1= (··· )
    In fun     \fun2= (··· )
    In { main = mainexpr }
```

Listing 1.4. Example task and auxiliary types in the mTask language.

3.3 Communication with iTask

As shown in Listing 1.2 the function `liftmTask` lifts an mTask task to an iTask task. The result of this task determines the value of the iTask and is automatically communicated. In Exercise 2 you learned how to communicate from iTask to mTask via parameters.

Communication via parameters works fine, but for dynamic communication tasks in mTask share information through SDSs with iTask. The same SDS is available on the edge device as well as on the server. In mTask we can read the value of an SDS by `getSds` and write a new value by `setSds`. In iTask we have `get` and `set` as well as web-based editors to update a SDS interactively. The machinery ensures that changes of the SDS in one world are automatically mirrored in the other world. 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: *Use* `getSds` *to read the value of a SDS. It yields an unstable value. So, you have to use a sequential combinator that steps on an unstable value. The operator* `>>~.` *does exactly this.*

---

4 Green Computing with mTask

MTask offers abstractions for edge layer-specific details such as the heterogeneity of architectures, platforms and frameworks; peripheral access; and multitasking but also for energy consumption and scheduling. In mTask, tasks are implemented as a rewrite system, where the work is automatically segmented in small atomic bits and stored as a task tree. During each cycle, a single rewrite step is performed on all task trees, during rewriting, tasks do a bit of their work and progress steadily, allowing interleaved and seemingly parallel operation. After a loop, the RTS knows which task is waiting on which triggers and is thus able to determine the next execution time for each task. Utilising this information, the RTS can determine when it is possible and safe to sleep and choose the optimal sleep mode according to the sleeping time. For example, the RTS never attempts to sleep during an I^2C communication because I/O is always contained *within* a rewrite step.

An mTask program is dynamically transformed to bytecode. This bytecode and the initial mTask expression are shipped to an mTask IoT node. The mTask rewrite engine rewrites the current expression just a single rewrite step at a time at the task level. Other computations are evaluated eagerly. When subtasks are composed in parallel, all subtasks are rewritten unless the result of the first rewrite step makes the result of the other tasks superfluous. This task design ensures that all time critical communication with peripherals is within a single rewrite step. Even functions that produce an infinite number of sequences, and thus rewrite steps, as in the `blink` example, are perfectly fine. The mTask system does tail-call elimination to facilitate this.

This rewriting of tasks is very convenient; the system can inspect the current state of all mTask expressions after a rewrite step. The system can detect if sleeping of the current task expression is useful and how long this is possible. When all current subtasks in the system allow a sufficiently long break, the ocessor can dynamically be switched to a low-energy sleeping mode. The system wakes up after the determined interval and continues rewriting of the tasks. The `delay` primitive is an obvious spot where sleeping is possible. In the next section we argue that there are many other spots where energy saving by inserting a short delay is possible without changing the behaviour of the system.

4.1 Temperature Monitor

Edge nodes in the IoT are very suited to read sensors. A typical example is a temperature sensor. Such a device communicates with the sensor via a dedicated protocol over the GPIO pins of the microcontroller. In this example the temperature is measured with the digital humidity and temperature (DHT) sensor that communicates via the I^2C protocol. We do not need to know any details about the sensor or protocol. The `DHT` primitive of mTask defines a tailor-made `dht` object. The task `temperature dht` reads the current value from this object.

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

The appointed device contains a SHT30x DHT sensor that communicates with the ocessor using I^2C. 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* °`C` *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 data traffic between the edge node and the server. Every update of the local SDS results in an automatic message to the server to keep both versions of the SDS in sync.

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

The temperature is not bound to change every 1ms 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.

4.2 Rewrite Intervals

Some mTask examples contain one or more explicit `delay` primitives, offering a natural place for the node executing it to pause. However, there are many mTask programs that just specify a repeated set of primitives. A typical example is the

```
thermostat :: Main (MTask v Bool) | mtask v
thermostat = DHT I2Caddr \dht->
    {main = rpeat (temperature dht >>~. \temp.
                   writeD builtInLED (goal <. temp))}
```

Listing 1.5. A basic thermostat task.

program that reads the temperature for a sensor and sets the system LED if the reading is below some given `goal`.

This program repeatedly reads the DHT sensor and sets the on-board LED based on the comparison with the `goal` as fast as possible on the mTask node. This is a perfect solution as long as we ignore the power consumption. The mTask machinery ensures that if there are other tasks running on the node that they are making progress. However, this solution is far from perfect when we take power consumption into account. In most applications, it is very unlikely that the temperature changes significantly within one minute, let alone within some ms. Hence, it is sufficient to repeat the measurement with an appropriate interval.

There are various ways to improve this program. The simplest solution is to add an explicit delay to the body of the repeat loop. A slightly more sophisticated option is to add a repetition period to the `rpeat` combinator. The combinator implementing this is called `rpeatEvery`. Both solutions rely on an explicit action of the programmer.

Fortunately, mTask also contains machinery to do this automatically. The key of this solution is to associate dynamically an evaluation interval with each task. The interval $\langle low, high \rangle$ indicates that the evaluation can be safely delayed by any number of ms within that range. Such an interval is just a hint for the RTS. It is not a guarantee that the evaluation takes place in the given interval. Other parts of the task expression can force an earlier evaluation of this part of the task. When the system is very busy with other work, the task might even be executed after the upper bound of the interval. The system calculates the rewrite rates from the current task expression. This has the advantage that the programmer does not have to deal with them and that they are available in each and every mTask program.

4.3 Basic Tasks

We start by assigning default rewrite rates to basic tasks. These rewrite rates reflect the expected change rates of sensors and other inputs. Basic tasks to one-shot set a value of a sensor or actuator usually have a rate of $\langle 0, 0 \rangle$, this is never delayed, e.g. writing to a GPIO pin. Basic tasks that continuously read a value or otherwise interact with a peripheral have default rewrite rates that fit standard usage of the sensor. Table 2 shows the default values for the basic tasks. Reading SDSs and fast sensors such as sound or light aim for a rewrite

every 100 ms, medium slow sensors such as gesture sensors every 1000 ms and slow sensors such as temperature or air quality every 2000 ms.

Table 2. Default rewrite rates of basic tasks.

task	default interval
reading an SDS	$\langle 0$ m sec, 2000 m sec$\rangle$
slow sensor	$\langle 0$ m sec, 2000 m sec$\rangle$
medium sensor	$\langle 0$ m sec, 1000 m sec$\rangle$
fast sensor	$\langle 0$ m sec, 100 m sec$\rangle$

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

Adapt the program so that only changes bigger than half a °`C` 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 is 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 0.0 -. x))
```

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

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

4.4 Tweaking Rewrite Rates

A tailor-made ADT (see Listing 1.6) determines the timing intervals for which the value is determined at runtime but the constructor is known at compile time. During compilation, the constructor of the ADT is checked and code is generated accordingly. If it is `Default`, no extra code is generated. In the other cases, code is generated to wrap the task tree node in a *tune rate* node. In the case that there is a lower bound, i.e. the task must not be executed before this lower bound, an extra *rate limit* task tree node is generated that performs a no-op rewrite if the lower bound has not passed but caches the task value.

```
:: TimingInterval v
    = Default
    | BeforeMs (v Int)           // yields ⟨0, x⟩
    | BeforeS  (v Int)           // yields ⟨0, x × 1000⟩
    | ExactMs  (v Int)           // yields ⟨x, x⟩
    | ExactS   (v Int)           // yields ⟨x × 1000, x × 1000⟩
    | RangeMs  (v Int) (v Int)   // yields ⟨x, y⟩
    | RangeS   (v Int) (v Int)   // yields ⟨x × 1000, y × 1000⟩
```

Listing 1.6. The ADT for timing intervals in mTask.

5 Interrupts

Most microcontrollers have built-in support for processor interrupts. These interrupts are hard-wired signals that can interrupt the normal flow of the program to execute a small piece of code, the interrupt service routine (ISR). While the ISRs look like regular functions, they do come with some limitations. For example, they must be very short, in order not to miss future interrupts; can only do very limited I/O; cannot reliably check the clock; and they operate in their own stack, and thus communication must happen via global variables. After the execution of the ISR, the normal program flow is resumed. Interrupts are heavily used internally in the RTS of the microcontrollers to perform timing critical operations such as WiFi, I^2C, or SPI communication; completed ADC conversions, software timers; exception handling; etc.

Interrupts offer two substantial benefits: fewer missed events and better energy usage. Sometimes an external event such as a button press only occurs for a very small duration, making it possible to miss it due to it happening right between two polls. Using interrupts is not a fool-proof way of never missing an event. Events may still be missed if they occur during the execution of an ISR or while the microcontroller is still in the process of waking up from a triggered interrupt. There are also some sensors, such as the CCS811 air quality sensor, with support for triggering interrupts when a value exceeds a critical limit.

Table 3 shows the different types of interrupts.

Table 3. Overview of GPIO interrupt types.

type	triggers
change	input changes
falling	input becomes low
rising	input becomes high
low	input is low
high	input is high

5.1 Arduino Platform

Listing 1.7 shows an exemplatory program utilising interrupts written in Arduino's C++ dialect. The example shows a debounced light switch for the built-in LED connected to GPIO pin 13. When the user presses the button connected to GPIO pin 11, the state of the LED changes. As buttons sometimes induce noise shortly after pressing, events within 30ms after pressing are ignored. In between the button presses, the device goes into deep sleep using the `LowPower` library.

Line 1 to Line 3 defines the pin and debounce constants. Line 5 defines the current state of the LED, it is declared **volatile** to exempt it from compiler optimisations because it is accessed in the interrupt handler. Line 6 flags whether the program is in debounce state, i.e. events should be ignored for a short period of time.

In the `setup` function (Line 8 to Line 13), the pinmode of the LED and interrupt pins are set. Furthermore, the microcontroller is instructed to wake up from sleep mode when a *rising* interrupt occurs on the interrupt pin and to call the ISR at Line 22 to Line 26. This ISR checks if the program is in cooldown state. If this is not the case, the state of the LED is toggled. In any case, the program goes into cooldown state afterwards.

In the `loop` function, the microcontroller goes to low-power sleep immediately and indefinitely. Only when an interrupt triggers, the program continues, writes the state to the LED, waits for the debounce time, and finally disables the `cooldown` state.

```
1  #define LEDPIN 13
2  #define INTERRUPTPIN 11
3  #define DEBOUNCE 30
4
5  volatile int state = LOW;
6  volatile bool cooldown = true;
7
8  void setup() {
9      pinMode(LEDPIN, OUTPUT);
10     pinMode(INTERRUPTPIN, INPUT);
11     LowPower.attachInterruptWakeup(
12         INTERRUPTPIN, buttonPressed, RISING);
13 }
14
15 void loop() {
16     LowPower.sleep();
17     digitalWrite(LEDPIN, state);
18     delay(DEBOUNCE);
19     cooldown = false;
20 }
21
22 void buttonPressed() {
23     if (!cooldown)
24         state = !state;
```

```
25     cooldown = true;
26 }
```

Listing 1.7. Light switch using interrupts in Arduino.

5.2 MTask language

Listing 1.8 shows the interrupt interface in mTask. The `interrupt` class contains a single function that, given an interrupt mode and a GPIO pin, produces a task that represents this interrupt. Lowercase variants of the various interrupt modes such as `change :== lit Change` are available as convenience macros.

```
class interrupt v where
    interrupt :: (v InterruptMode) (v p) -> MTask v Bool | pin p

:: InterruptMode = Change | Rising | Falling | Low | High
```

Listing 1.8. The interrupt interface in mTask.

When the mTask device executes this task, it installs an ISR and sets the rewrite rate of the task to infinity, $\langle\infty,\infty\rangle$. The interrupt handler is set up in such a way that the rewrite rate is changed to $\langle 0, 0\rangle$ once the interrupt triggers. As a consequence, the task is executed on the next execution cycle.

The `pirSwitch` function in Listing 1.9 creates, given an interval in ms, a task that reacts to motion detection by a PIR sensor (connected to GPIO pin 0) by lighting the LED connected to GPIO pin 13 for the given interval. The system turns on the LED again when there is still motion detected after this interval. By changing the interrupt mode in this program text from `High` to `Rising` the system lights the LED only one interval when it detects motion no matter how long this signal is present at the PIR pin.

```
pirSwitch :: Int -> Main (MTask v Bool) | mtask v
pirSwitch =
    declarePin D13 PMOutput \ledpin->
    declarePin D0 PMInput \pirpin->
    {main = rpeat (     interrupt high pirpin
                  >>|. writeD ledpin false
                  >>|. delay (lit interval)
                  >>|. writeD ledpin true) }
```

Listing 1.9. Example of a toggle light switch using interrupts.

5.3 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 is 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 8 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.

---

6 Conclusion

This paper shows how we enable green computing for the IoT. A first important step is to replace the single-board computers, like a Raspberry Pi, driving the edge nodes by an energy-efficient microcontroller, like the ESP8266. These microcontrollers consume an order of magnitude less energy and are an order of magnitude less expensive than the full-fledged single-board computers. For battery powered nodes the advantages are obvious. Given the enormous and growing amount of devices, it is also worthwhile to use microcontrollers for nodes that have a permanent power supply.

Microcontrollers have very limited amounts of memory and processing power. This implies that they typically have no OS and cannot be programmed like normal computers. Typically there is only a single program executing at the microcontroller. When several independent subtasks must be executed, this program must explicitly schedule these tasks. The mTask system discussed here enables

the execution of high-level TOP programs on these tiny devices. This system is fully integrated within the iTask DSL that enables multi-user web-based TOP. This implies that the host language ensures type safety and can generate the boilerplate code for storing values and communication between the edge nodes and the server. The task to be executed on an edge node is dynamically compiled to bytecode, shipped to the edge node, and interpreted there by a tailor-made feather-light TOP OS.

To ensure progress of all subtasks shipped to such an edge node, it implements small step rewriting of these tasks. After each and every rewrite step the state is stored to enable the rewriting of all other subtasks on the edge node. This implies that the mTask system is able to inspect the state of all subtasks. When this inspection shows that all subtasks can be delayed sufficiently, the entire system is switched to a low-power sleep state. To increase the possibilities to pause processing mTask implements heuristics. For instance a temperature sensor is by default used only once a second since we assume that its value does not change significantly during this interval. The user can tweak this reading delay at a high level of abstraction to adjust the balance between responsiveness and energy uses for a specific application.

Hardware supported interrupts enable the system to wait in a low-energy state until an event occurs. This is way more power friendly than the usual polling for the event. Moreover, it greatly reduces the chance of missing the event. The mTask system also offers these interrupts at a convenient high abstraction level that is much easier to use than the traditional interrupt service routines.

These high-level energy savings achieve a reduction of the power consumption of another order of magnitude with no or very limited effort from the programmer. Hence, we consider these extensions of the mTask system a very valuable contribution to green computing of the IoT.

A Installing the Software

To execute the iTask programs listed in this chapter one needs the functional programming language Clean and the iTask library. These systems use the `nitrile` package manager and their installation instructions can be found at https://clean-lang.org/. The Clean/`nitrile` ecosystem are under constant change. For 64-bit linux machines, an out-of-the-box working setup can be found here including desktop application to simulate an mTask microcontroller [7].

To prepare a microcontroller for the use with mTask, the domain-specific OS must be installed. A list of appropriate devices and the required software is available at https://gitlab.com/mtask/client, the examples from these lecture notes are tested on client version `v0.1.1`.

B Measuring Power Consumption

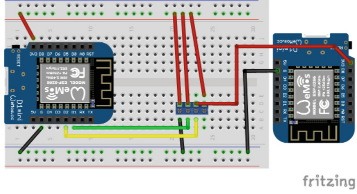
For the power measurements we recommend a INA260 or INA226 current sensor connected to a separate Wemos D1 mini controller. A program for the Wemos D1

mini measuring the energy consumed can be found in Listing 1.10. This program sends the actual energy consumption to the serial output and supports both current sensors. Two different versions of the Wemos D1 mini microcontroller can be used, Fig. 1 shows wiring instructions.

The Arduino IDE's serial plotter is used to graph of the energy consumption. Connect only the leftmost microcontroller, the one on the breadboard, with an USB cable to your PC! This microcontroller should run a power monitor program. The second ocessor is executing mTask programs. Follow the instructions from Appendix A to install the mTask system on this ocessor. To connect it to the SHT30x DHT temperature shield used in the exercises it might be convenient to stick the rightmost ocessor with the temperature sensor on a Wemos triple base. This assumes that you use the Wemos temperature shield for the sensor. The code should work for any SHT30x DHT sensor with I^2C communication.

```
#include <Wire.h>

//Comment if you use the INA 226 breakout board
#define INA260

#ifdef INA260
#include <Adafruit_INA260.h>
Adafruit_INA260 INA = Adafruit_INA260();
#define READPOWER readPower
#else

#include <INA226_WE.h>
#define I2C_ADDRESS 0x40
INA226_WE INA = INA226_WE(I2C_ADDRESS);
#define READPOWER getBusPower
#endif

void setup() {
    Serial.begin(115200);
    Wire.begin();
#ifdef INA260
    if (!INA.begin()) {
        Serial.println("Couldn't find INA260 chip");
        while(1);
    }
    INA.setAveragingCount(INA260_COUNT_4);
#else
    INA.init();
    INA.setAverage(AVERAGE_4);
#endif
}

void loop() {
    Serial.println("Power(W)");
    for (int i = 0; i < 100; i += 1) {
        Serial.printf("%.3f\n", INA.READPOWER());
        delay(100);
    }
}
```

Listing 1.10. Arduino C++ sketch for the power monitor.



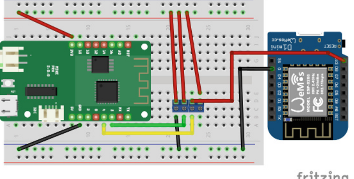


Fig. 1. Wiring instructions for the powermonitor using the Wemos D1 mini (left) or pro variant (right).

C Solutions

This appendix contains a possible solution for the exercises in this paper. Other solutions might be fine, like usual in programming.

Solution 1 Hello world! (`blink.icl`)

The full program was given in Exercise 1.

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 0.0 -. 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 (0.0) -. 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)}
```

References

1. Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J.: Clean A language for functional graph rewriting. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture, pp. 364–384. Springer, Heidelberg (1987)
2. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509 543 (2009). https://doi.org/10.1017/S0956796809007205
3. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, pp. 143–156. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1411204.1411226
4. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, pp. 266–296. Springer, Heidelberg (2007)
5. Koopman, P., Lubbers, M., Plasmeijer, R.: A task-based DSL for microcomputers. In: Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018, pp. 1–11. ACM Press, Vienna, Austria (2018). https://doi.org/10.1145/3183895.3183902. http://dl.acm.org/citation.cfm?doid=3183895.3183902
6. Lubbers, M., Koopman, P., Plasmeijer, R.: Multitasking on microcontrollers using task oriented programming. In: 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 1587 1592. Opatija, Croatia (2019). https://doi.org/10.23919/MIPRO.2019.8756711
7. Lubbers, M., Koopman, P.: Code for the lecture notes: “Green Computing for the Internet of Things” (2023). https://doi.org/10.5281/zenodo.7643316
8. Lubbers, M., Koopman, P., Plasmeijer, R.: Interpreting task oriented programs on tiny computers. In: Stutterheim, J., Chin, W.N. (eds.) Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, Singapore, Singapore, p. 12. IFL 19, ACM, New York, NY, USA (2021). https://doi.org/10.1145/3412932.3412936
9. Lubbers, M., Koopman, P., Plasmeijer, R.: Writing internet of things applications with task oriented programming. In: Porkoláb, Z., Zsók, V. (eds.) Composability, Comprehensibility and Correctness of Working Software, pp. 3–52. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-42833-3_1
10. Lubbers, M., Koopman, P., Ramsingh, A., Singer, J., Trinder, P.: Tiered versus Tierless IoT stacks: comparing smart campus software architectures. In: Proceedings of the 10th International Conference on the Internet of Things, Malm , Sweden. IoT 20, ACM, Malm (2020). https://doi.org/10.1145/3410992.3411002

11. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA, pp. 199–208. PLDI 88, ACM, New York, NY, USA (1988). https://doi.org/10.1145/53990.54010
12. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. ACM SIGPLAN Not. **42**(9), 141–152 (2007)
13. Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean Language Report version 3.1. Technical report, Institute for Computing and Information Sciences, Nijmegen (2021)
14. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, Leuven, Belgium, pp. 195–206. PPDP 12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2370776.2370801
15. Serrano, M., Gallesio, E., Loitsch, F.: Hop: a language for programming the web 2.0. In: OOPSLA Companion, pp. 975–985. ACM, Portland, Oregon, USA (2006)
16. Tratt, L.: Domain specific language implementation via compile-time meta- programming. ACM Trans. Program. Lang. Syst. **30**(6) (2008). https://doi.org/10.1145/1391956.1391958