# Reducing the Power Consumption of IoT with Task-Oriented Programming

Sjoerd Crooijmans, Mart Lubbers(✉), and Pieter Koopman

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

**Abstract.** Limiting the energy consumption of IoT nodes is a hot topic in green computing. For battery-powered devices this necessity is obvious, but the enormous growth of the number of IoT nodes makes energy efficiency important for every node in the IoT. In this paper, we show how we can automatically compute execution intervals for our task-oriented programs for the IoT. These intervals offer the possibility to save energy by bringing the microprocessor driving the IoT node into a low-power sleep mode until the task need to be executed. Furthermore, they offer an elegant way to add interrupts to the system. We do allow an arbitrary number of tasks on the IoT nodes and achieve significant reductions of the energy consumption by bringing the microprocessor in sleep mode as much as possible. We have seen energy reductions of an order of magnitude without imposing any constraints on the tasks to be executed on the IoT nodes.

**Keywords:** Sustainable IoT · Green computing · Task-oriented programming · Edge computing

## 1 Introduction

The Internet of Things (IoT) is omnipresent and powered by software. Depending on whom you ask, the estimated number of connected IoT devices reached between 25 and 100 billion in 2021. IoT systems are traditionally designed according to multi-layered or tiered architectures. As a consequence, discrete programs written in distinct languages with different abstraction levels power the individual layers, forming a heterogeneous system. The variation in components makes programming IoT systems complicated, error-prone and expensive.

The edge layer of IoT contains the small devices that sense and interact with the world and it is crucial to lower their energy consumption. While individual devices consume little energy, the sheer number of devices in total amounts to a lot. Furthermore, many IoT devices operate on batteries and higher energy consumption increases the amount of e-waste as IoT devices are often hard to reach and consequently hard to replace [13].

To reduce the power consumption of an IoT device, the specialized low-power sleep modes of the microprocessors can be leveraged. Different sleep modes achieve different power reductions because of their different run time characteristics. These specifics range from disabling or suspending Wi-Fi; stopping powering (parts) of the RAM; disabling peripherals; or even turning off the processor completely, requiring an external signal to wake up again. Determining when exactly and for how long it is possible to sleep is expensive in the general case and often requires annotations in the source code, a real-time operating system or a handcrafted scheduler.

Task-oriented programming (TOP) is a novel declarative programming paradigm with the potential to solve many of the aforementioned problems. In this paradigm, tasks are the basic building blocks and they can be combined using combinators to describe workflows [15]. This declarative specification of the program only describes the *what* and not the *how* of execution. The system executing the tasks takes care of the gritty details such as the user interface, data storage and communication [16]. An example of a TOP language is the iTask system, a general-purpose framework for specifying multi-user distributed web applications for workflows [14]. iTask is implemented as an embedded domain-specific language (DSL) in the functional programming language Clean [4,6].

mTask lies on the other side of the spectrum and aims to solve semantic friction in IoT. It is a domain-specific TOP language and system specifically designed for IoT devices, implemented as an embedded DSL in iTask. Where iTask abstracts away from details such as user interfaces, data storage, and persistent workflows, mTask offers abstractions for edge layer-specific details such as the heterogeneity of architectures, platforms and frameworks; peripheral access; and multitasking. Yet, it lacks abstractions 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. 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 run-time system (RTS) knows which task is waiting on which triggers and is thus able to determine the next execution time for each task automatically. 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.

## 1.1 Research Contribution

This paper shows that with minor changes to the mTask language from the perspective of the TOP programmer, the energy consumption of the program's execution can be significantly reduced. We show that with an intensional analysis of the task trees at run time, the mTask scheduler can automatically determine the optimal sleep time and sleep mode. Not all tasks have a default rewrite rate that works in all situations, so variants of tasks are added in which the programmer can fine-tune the polling rate. Furthermore, we add an interface to (hardware) interrupts to the mTask language, allowing the program to be notified in case of an external event, resulting in more reactive programs.

## 2 Task-Oriented Programming

TOP is a high-level declarative programming paradigm to specify distributed interactive multi-user systems [14,15]. Developers describe in TOP the work to be done by the systems or users in the form of abstract tasks. Implementation details, like the encoding of data during communication, are handled by the system rather than by the TOP programmer. Tasks describe a unit of work ranging from reading a single sensor to an entire IoT system. Tasks are rewrite systems that produce a result after each step. Possible task results are:

**No value** if the task has no observable value for other tasks. For example, a web-editor that is empty or incomplete is a task with a `NoValue` result.

**Unstable** if the task has an intermediate observable value. This value is by construction properly typed and can change in the future. Examples are a properly filled out web-editor or a reading a sensor.

**Stable** if the task has a final observable value. This value is by construction properly typed and fixed. Examples are a properly filled out web-editor after pressing the `Continue` button or a task that determines that a temperature sensor has passed a given threshold.

Basic tasks are the primitive building blocks of a TOP program. Typical examples are web-editors, reading sensors, waiting some time and controlling peripherals. Tasks can be composed into bigger tasks by combinators. There are combinators for the parallel and sequential composition of tasks.

Apart from task results, tasks can also communicate via shared data sources (SDSs). There are basic tasks to read, write and update such a typed SDS.

### 2.1 mTask

TOP is also very suited to program nodes in the IoT. However, typical nodes in the IoT are cheap and small microprocessors with a very limited amount of processing power and memory. Such a system cannot run a web server nor a browser as a client to execute an iTask program. This is also not necessary, typical IoT nodes just control a few sensors and actuators. Rather than sending all sensor readings to some server and the actuator control back from the server to the IoT node, we want to do a significant part of the computing on the IoT nodes. This concept is popular under the name edge computing.

To enable TOP on small microprocessors we have created mTask [10,12]. The domain-specific language mTask is also embedded in Clean. In contrast to iTask, it is not a shallowly embedded DSL but it is a tagless-final style DSL [5]. The target language of iTask is equal to the host language Clean, this is called a homogeneous DSL [17]. The target code of mTask is byte code which is shipped to a microprocessor in the IoT and interpreted by the mTask runtime system running there. Hence, mTask is a heterogeneous embedded DSL. The big advantage of this approach is that we can carefully control which parts of the application is mTask code and needs to be shipped to the IoT node. This prevents that all Clean language machinery ends up in such a small system. Moreover, we can dynamically ship tasks to the IoT nodes without the need to reprogram the flash memory of those systems. Reprogramming this memory is slow and can be done only a few thousand times.

The archetypal example of programs running on microprocessors is the blink example that blinks the single bit *display*, just a LED, of such systems. In Listing 1 we give a slightly more complex example in mTask that blinks two LEDs at their own slowly changing speed.

```
blinkTask :: Main (MTask v Bool) | mtask v
blinkTask = declarePin BuiltinLEDPin PMOutput λled1 →
            declarePin D0 PMOutput λled2 →
            fun λblink = (λ(led, interval, state) →
                     delay interval
                 ≫|. writeD led state
                 ≫|. blink (led, interval +. lit 1, Not state)
            In {main = blink (led1, lit 496, true) .||. blink (led2, lit 8128, true)}
```

**Listing 1.** An mTask task to blink two LEDs at their own rate.

The Clean function `blinkTask` contains the mTask code for the blink task. It starts by declaring two `led` objects to represent the output general-purpose input/output pin to control the LED as output. Next, it declares an mTask function called `blink`. This function has the `led`, delay `time` and new `state` of the LED as arguments. The task in this function is composed of three subtasks. First, it waits `time` millisecond by `delay`. Second, it writes the given `state` to the declared output `led`. Finally, it calls itself recursively with the inverted `state` as the argument. In the `main` expression two of these `blink` tasks are composed in parallel with the `.||.` combinator. The `lit` in the arguments of the subtasks lifts the given constant from the host language to the embedded DSL.

There are some important differences to the usual C-programs controlling microprocessors. First, tail-recursive functions are perfectly safe in mTask. Next, the `delay` task is not blocking the entire program, but just producing a `NoValue` result until the given `time` has passed.

## 3 Energy Efficient IoT Nodes

There are various estimates about the number of IoT nodes. What the estimates have in common is that the number is relatively large and rapidly increasing. Figure 1 contains a popular estimate of the worldwide number of nodes[1]. Currently, there are worldwide just above 40 billion connected IoT devices.

There is a wide variety of IoT nodes. Relative simple single board computers, like a Raspberry Pi 4, are often used. These are full-fledged computers running Linux. This operating system enables multithreading and the dynamic uploading of new applications via remote access. Such a single board computer costs about €70 and consumes 4–6W.

[1] https://statinvestor.com/data/33967/iot-number-of-connected-devices-worldwide/.

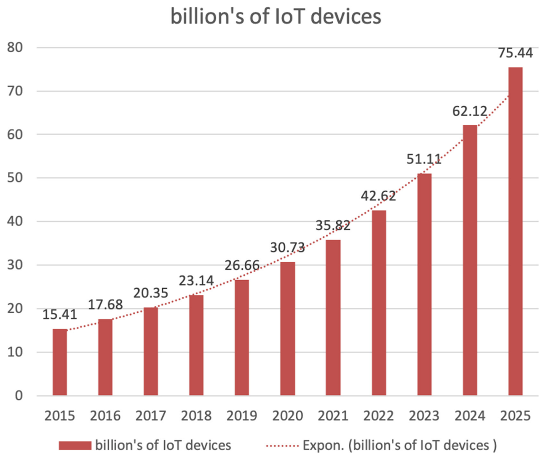

**Fig. 1.** Number of connected IoT nodes worldwide.

For battery powered nodes, the energy consumption of such a single board computer is obviously inconveniently high. Considering the number of IoT nodes, the total energy consumption for all nodes would also be rather high for single board computers connected to a power outlet. The total annual energy consumption of all IoT nodes would be 1.7 PWh/year if they consume 5W each and are always available. This is about 6% of the annual electrical energy production of 26.8 PWh/year[2], about 500 times the production of the only Dutch nuclear power plant[3], and $\frac{2}{3}$ of the annual production of renewable electricity[4].

Fortunately, there are microprocessor based systems that require significantly less energy, typically 1–500 mW, than single board computers and are very cheap as well. The price to be paid is that microprocessors are much slower and have a very limited amount of memory. As a consequence of these bordered capacities, a microprocessor typically has no operating system or at most special purpose real-time operating system like FreeRTOS. Despite their limited capabilities, microprocessors are often more powerful than needed for IoT tasks. This implies that we can switch them off, or partially off, for some fraction of the time. Microprocessors have special low power sleep modes to enable energy saving. The sleep modes as well as the associated energy consumption are model specific. Table 1 lists some examples for two popular microprocessor boards. The Wemos D1 mini[5] is powered by an ESP8266 and costs about €6. The Adafruit Feather

[2] https://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy/electricity.html.
[3] https://nl.wikipedia.org/wiki/Kerncentrale_Borssele.
[4] https://www.bp.com/content/dam/bp/business-sites/en/global/corporate/pdfs/energy-economics/statistical-review/bp-stats-review-2021-electricity.pdf.
[5] https://www.wemos.cc/en/latest/d1/d1_mini.html.

**Table 1.** Current use in mA of two microprocessor boards in various sleep modes.

| Component | Wemos D1 mini | | | | Adafruit Feather M0 Wifi | | | |
|---|---|---|---|---|---|---|---|---|
| | Active | Modem sleep | Light sleep | Deep sleep | Active | Modem sleep | Light sleep | Deep sleep |
| Wi-Fi | On | Off | Off | Off | On | Off | Off | Off |
| CPU | On | On | Pending | Off | On | On | Idle | Idle |
| RAM | On | On | On | Off | On | On | On | Low power |
| Current | 100–240 | 15 | 0.5 | 0.002 | 90–300 | 5 | 2 | 0.005 |

M0 WiFi[6] is powered by an ATSAMD21 and an ATWINC1500 for Wi-Fi, it costs about €50.

This table shows that switching the Wi-Fi radio off yields the biggest energy savings. In most IoT applications, we need Wi-Fi for communications. It is fine to switch it off, but after switching it on, the Wi-Fi protocol needs to transmit a number of messages to re-establish the connection. This implies that it is only worthwhile to switch the radio off when this can be done for some time. The details vary per system and situation. As a rule of thumb, it is only worthwhile to switch the Wi-Fi off when it is not needed for at least some tens of seconds.

The data in Table 1 shows that it is worthwhile to put the system in some sleep mode when there is temporarily no work to be done. A deeper sleep mode saves more energy, but also requires more work to restore the software to its working state. A processor like the ESP8266 driving the Wemos D1 mini loses the content of its RAM in deep sleep mode. This implies that the program itself is preserved, since it is stored in flash memory, but the entire program state is lost. When there is a program state to be preserved, we must either store it elsewhere, limit us to light sleep, or use a microprocessor that keeps the RAM intact during deep sleep.

For IoT nodes executing a single task, explicit sleeping to save energy can be achieved without too much hassle. This becomes much more challenging as soon as multiple independent tasks run on the same node. Sleeping of the entire node induced by one task prevents progress of all tasks. This is especially annoying when the other tasks are executing time critical parts, like communication protocols. Such protocols control the communication with sensors and actuators. Without the help of an OS, the programmer is forced to combine all subtasks into one big system that decides if it is safe to sleep for all subtasks.

## 4 Scheduling Tasks Efficiently

The mTask system enables an elegant solution for these dynamic scheduling and sleeping problems. To understand the solution, we have to reveal some details about the mTask implementation. This system consists of two components. First,

[6] https://www.adafruit.com/product/3010.

there is the embedded DSL as introduced in Sect. 2.1. Next, there is an execution environment for those programs running on IoT nodes. This execution environment is able to execute multiple mTask programs simultaneously, it is a featherlight domain-specific operating system.

Every mTask program is embedded in an iTask program, a TOP DSL very similar to mTask. In contrast to mTask, iTask is geared to client server web-applications. The iTask and mTask components form a single source that is statically type-checked by the compiler of the host language Clean. The mTask components determine what is done by the IoT nodes. The surrounding iTask program determines the IoT node to be used and the moment it is shipped. The communication between iTask and mTask programs is outside the scope of this paper, see [11] for details.

A mTask program is dynamically transformed to byte code. This byte code and the initial mTask expression are shipped to mTask IoT node. For the example in Listing 1 there is byte code representing the `blink` function and `main` determines the initial expression.

The mTask rewrite engine rewrites the current expression just a single rewrite step at a time. When subtasks are composed in parallel, like blink tasks in Listing 1, all subtasks are rewritten unless the result of the first rewrite step makes the result of the other tasks superfluous. This yields for our example:

```
{main =  delay 496  >>|. writeD led1 true >>|. blink (led1, 497, false)
     .||. delay 8128 >>|. writeD led2 true >>|. blink (led2, 8129, false)}
```

**Listing 2.** State of the task from Listing 1 after a single rewrite step.

The mTask node can inspect this expression. It is obvious that nothing will change the next 496 ms. The system can therefore go to sleep for this period when there are no other tasks[7].

The task design ensures such that all time critical communication with peripherals is within a single rewrite step. This is very convenient, since the system can inspect the current state of all mTask expressions after a rewrite and decide if sleeping and how long is possible. As a consequence, we cannot have fair multitasking. When a single rewrite step would take forever due to an infinite sequence of function calls, this would block the entire IoT node. Infinite sequences rewrite steps, as in the `blink` example above, are perfectly fine. The mTask system does proper tail-call optimizations to facilitate this.

## 4.1 Evaluation Interval

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

[7] In the implementation a `delay d` is replaced by a `waitUntil (now + d)` that compares the clock time of the system with the given start time.

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

**Listing 3.** A basic thermostat task.

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, they will make 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 will change significantly within one minute, let alone within some milliseconds. 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, similar to the blink example in Listing 1. 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.

In this paper, we propose a solution that is independent of the help of the programmer. 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 milliseconds in that range. Such an interval is just a hint for the run time system. 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 refresh 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.

### 4.2 Basic Refresh Rates

We start by assigning default refresh rates to basic tasks. These refresh rates reflect the expected change rates of sensors and other inputs. Writing to basic GPIO pins and actuators has refresh rate $\langle 0, 0 \rangle$, this is never delayed.

### 4.3 Deriving Refresh Rates

Based on these refresh rates, the system can automatically derive refresh rates for composed mTask expressions using $\mathcal{R}$. We use the operator $\cap_{safe}$ to compose refresh ranges. When the ranges overlap the result is the overlapping range. Otherwise, the result is the range with the lowest numbers. The rationale is that

**Table 2.** Default refresh rates of basic tasks.

| Task | Default interval |
|---|---|
| Reading an SDS | $\langle 0, 2000 \rangle$ |
| Slow sensor, like temperature | $\langle 0, 2000 \rangle$ |
| Gesture sensor | $\langle 0, 1000 \rangle$ |
| Fast sensor, like sound or light | $\langle 0, 100 \rangle$ |
| Reading GPIO pins | $\langle 0, 100 \rangle$ |

subtasks should not be delayed longer than their refresh range. Evaluating a task earlier should not change its result but can consume more energy.

$$\cap_{safe} :: \langle Int, Int \rangle\ \langle Int, Int \rangle \to \langle Int, Int \rangle$$

$$R_1 \cap_{safe} R_2 = R_1 \cap R_2 \qquad \text{if } R_1 \cap R_2 \neq \emptyset \tag{1}$$

$$\langle l_1, h_1 \rangle \cap_{safe} \langle l_2, h_2 \rangle = \langle l_2, h_2 \rangle \qquad \text{if } h_2 < l_1 \tag{2}$$

$$R_1 \cap_{safe} R_2 = R_1 \qquad \text{otherwise} \tag{3}$$

$$\mathcal{R} :: (MTask\ v\ a) \to \langle Int, Int \rangle$$

$$\mathcal{R}(t_1\ .||.\ t_2) = \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \tag{4}$$

$$\mathcal{R}(t_1\ .\&\&.\ t_2) = \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \tag{5}$$

$$\mathcal{R}(t_1 \gg|.\ t_2) = \mathcal{R}(t_1) \tag{6}$$

$$\mathcal{R}(t \gg=.\ f) = \mathcal{R}(t) \tag{7}$$

$$\mathcal{R}(t \gg*.\ [a_1 \ldots a_n]) = \mathcal{R}(t) \tag{8}$$

$$\mathcal{R}(rpeat\ t) = \langle 0, 0 \rangle \tag{9}$$

$$\mathcal{R}(rpeatEvery\ d\ t) = \langle 0, 0 \rangle \tag{10}$$

$$\mathcal{R}(delay\ d) = \langle d, d \rangle \tag{11}$$

$$\mathcal{R}(t) = \begin{cases} \langle \infty, \infty \rangle & \text{if } t \text{ is Stable} \\ \langle r_l, r_u \rangle & \text{otherwise} \end{cases} \tag{12}$$

We will briefly discuss the various cases of deriving refresh rates together with the task semantics of the different combinators.

**Parallel Combinators.** For the parallel composition of tasks we compute the intersection of the refresh intervals of the components as outlined in the definition of $\cap_{safe}$. The operator `.||.` in Eq. 4 is the *or*-combinator; the first subtask that produces a stable value determines the result of the composition. The operator `.&&.` in Eq. 5 is the *and*-operator. The result is the tuple containing both results when both subtasks have a stable value.

**Sequential Combinators.** For the sequential composition of tasks we only have to look at the refresh rate of the current task on the left. The sequential composition operator $\gg|$. in Eq. 6 is similar to the monadic sequence operator $\gg|$. The operator $\gg=$. in Eq. 7 provides the stable task result to the function on the right-hand side, similar to the monadic bind. The operator `>>~.` steps on an unstable value and is otherwise equal to $\gg=$.. The step combinator `>>*.` in Eq. 8 has a list of conditional actions that specify a new task.

**Repeat Combinators.** The repeat combinators repeats their argument indefinitely. The combinator `rpeatEvery` guarantees the given delay between repetitions. The refresh rate is equal to the refresh rate of the current argument task. Only when `rpeatEvery` waits between the iterations of the argument the refresh interval is equal to the remaining delay time.

**Other Combinators.** The refresh rate of the `delay` in Eq. 11 is equal to the remaining delay. Refreshing stable tasks can be delayed indefinitely, their value never changes. For other basic tasks, the values from Table 2 apply. The values $r_l$ and $r_u$ in Eq. 12 are the lower and upper bound of the rate.

The refresh intervals associated with various steps of the thermostat program from Listing 3 are given in Table 3. Those rewrite steps and intervals are circular, after step 2 we continue with step 0 again. Only the actual reading of the sensor with `temperature dht` offers the possibility for a non-zero delay.

**Table 3.** Rewrite steps of the thermostat from Listing 3 and associated intervals.

| Step | Expression | Interval |
|---|---|---|
| 0 | `rpeat (`<br>`    temperature dht >>~. \temp.`<br>`    writeD builtInLED (goal <. temp)`<br>`)` | $\langle 0, 0\rangle$ |
| 1 | `temperature dht >>~. \temp.`<br>`writeD builtInLED (goal <. temp) >>\|.`<br>`rpeat (`<br>`    temperature dht >>~. \temp.`<br>`    writeD builtInLED (goal <. temp)`<br>`)` | $\langle 0, 2000\rangle$ |
| 2 | `writeD builtInLED false >>\|.`<br>`rpeat (`<br>`    temperature dht >>~. \temp.`<br>`    writeD builtInLED (goal <. temp)`<br>`)` | $\langle 0, 0\rangle$ |

### 4.4 User Defined Refresh Rates

In some applications, it is necessary to read sensors at a different rate than the default rate given in Table 2. This is achieved by calling the access functions with a custom refresh rate as an additional argument.

```
class dht v where
    temperature‘ :: (TimingInterval v) (v DHT) → MTask v Real
    temperature  ::                    (v DHT) → MTask v Real
    humidity‘    :: (TimingInterval v) (v DHT) → MTask v Real
    humidity     ::                    (v DHT) → MTask v Real

class dio p v | pin p where
    readD‘ :: (TimingInterval v) (v p) → MTask v Bool | pin p
    readD  ::                    (v p) → MTask v Bool | pin p
```

**Listing 4.** Definition for DHT sensors and reading digital values from GPIO pins with a custom timing interval.

A tailor-made algebraic data type determines the timing intervals.

```
:: 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 ⟨0, x × 1000⟩
                    | RangeMs  (v Int) (v Int)  // yields ⟨x, y⟩
                    | RangeS   (v Int) (v Int)  // yields ⟨x × 1000, y × 1000⟩
```

**Listing 5.** The ADT for timing intervals in mTask.

As example, we define an mTask that updates the SDS `tempSDS` in iTask in a tight loop. The `temperature‘` reading requires that this happens at least once per minute. Without other tasks on the IoT node, the temperature SDS will be updated once per minute. Other tasks can cause a slightly more frequent update.

```
delayTime = BeforeS (lit 60) // 1 minute in seconds

devTask :: Main (MTask v Real) | mtask, dht, liftsds v
devTask = DHT (DHT_DHT pin DHT11) λdht =
          liftsds λlocalSds = tempSDS
          In {main = rpeat (temperature‘ delayTime dht >>~. setSds localSds)}
```

**Listing 6.** Updating an SDS in iTask at most once per minute.

## 5 Running Tasks on Interrupts

Interrupts can be used in IoT programming to prevent polling of input pins. Most microprocessors offer hardware support to execute a piece of code, called an interrupt service routine (ISR) or handler, associated with a change of input on an input pin. The interrupt handlers are regular functions that typically come with some restrictions, they must be short and should not do any communication.

Invoking an ISR often works even when the microprocessor is in sleep mode. Upon the specified change of input level, the system becomes active and executes the ISR. This interrupt mechanism is obviously very suited to reduce the energy consumption of IoT nodes. We can replace polling of an input by installing the appropriate ISR and we do not have to spend any energy to monitor this input. Moreover, it eliminates the possibility that the program misses an event when it occurs in between measurements of the polling cycle. Unfortunately, it is not on all kind of microprocessors possible to determine the reason the system wakes up from deep sleep. Since we cannot uniquely identify interrupts on such a system, the mTask system is limited to light sleep when it has to monitor interrupts.

The refresh rate mechanism offers a suitable way to integrate interrupts in mTask. MTask is extended with a basic task called `interrupt`. It is parameterized by the interrupt mode and the GPIO pin to watch.

```
class interrupts v where
    interrupt :: InterruptMode (v p) → MTask v Bool | pin p

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

**Listing 7.** Definition of the interrupt class in mTask.

When the mTask node executes this task, it installs a ISR and sets the refresh rate of the task to infinity, $\langle\infty,\infty\rangle$. The interrupt handler changes the refresh rate to $\langle 0,0\rangle$ when the interrupt occurs. As a consequence, the task is executed on the next execution cycle.

The `pirSwitch` task in Listing 8 reacts to motion detection by a Passive InfraRed (PIR) sensor by lighting the onboard LED for the given interval. The system lightens the LED again when there is still motion detected after this interval.

```
pirSwitch :: (v Int) → Main (MTask v Bool) | mtask v
pirSwitch interval = {main = rpeat (
                         interrupt High pirPin >>|.
                         writeD BuiltinLEDPin false >>|.
                         delay interval >>|.
                         writeD BuiltinLEDPin true
                     )}
```

**Listing 8.** Example of a toggle switch with interrupts

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 `pirPin`. This example shows that this abstraction of interrupts is very easy to use and blends well in the TOP design.

## 6 Implementing Refresh Rates

The refresh rates from the previous section tell us how much the next evaluation of the task can be delayed. An IoT node executes multiple tasks interleaved. In addition, it has to communicate with a server to collect new tasks and updates of SDS. Hence, we cannot use those refresh intervals directly to let the microprocessor sleep. Our scheduler has the following objectives.

– Meet the deadline whenever possible, i.e. the system tries to execute every task before the end of its refresh interval. Only too much work on the device might cause an overflow of the deadline.
– Achieve long sleep times. Waking up from sleep consumes some energy and takes some time. Hence, we prefer a single long sleep over splitting this interval into several smaller pieces.
– The scheduler tries to avoid unnecessary evaluations of tasks as much as possible. A task should not be evaluated now when its execution can also be delayed until the next time that the device is active. That is, a task should preferably not be executed before the start of its refresh interval. Whenever possible, task execution should even be delayed when we are inside the refresh interval as long as we can execute the task before the end of the interval.
– The optimal power state should be selected. Although a system uses less power in a deep sleep mode, it also takes more time and energy to wake up from deep sleep. When the system knows that it can sleep only a short time it is better to go to light sleep mode since waking up from light sleep is faster and consumes less energy.

The algorithm $\mathcal{R}$ from Sect. 4 computes the evaluation rate of the current tasks. For the scheduler, we transform this interval to an absolute evaluation interval; the lower and upper bound of the start time of that task measured in the time of the IoT node. We obtain those bounds by adding the current system time to the bounds of the computed refresh interval by algorithm $\mathcal{R}$.

For the implementation, it is important to note that the evaluation of a task takes time. Some tasks are extremely fast, but other tasks require long computations and time-consuming communication with peripherals as well as with the server. These execution times can yield a considerable and noticeable time drift in mTask programs. For instance, a task like `rpeatEvery (ExactMs 1) t` should repeat `t` every millisecond. The programmer might expect that `t` will be executed for the $(N+1)^{th}$ time after $N$ milliseconds. Uncompensated time drift might make this considerably later. MTask does not pretend to be a hard real-time operating system, and cannot give firm guarantees with respect to evaluation time. Nevertheless, we try to make time handling as reliable as possible. This is achieved by adding the start time of this round of task evaluations rather than the current time to compute absolute execution intervals.

## 7 Scheduling Tasks

Apart from the task to execute, the IoT device has to maintain the connection with the server and check there for new tasks and updates of SDS. When the microprocessor is active, it checks the connection and updates from the server and executes the task if it is in its execution window. Next, the microprocessor goes to light sleep for the minimum of a predefined interval and the task delay.

In general, the microprocessor node executes multiple mTask tasks. Our mTask nodes repeatedly check for inputs from servers and execute all tasks that cannot be delayed to the next evaluation round one step. We store tasks in a priority queue to check efficiently which of them need to be stepped. The mTask tasks are ordered at their absolute latest start time in this queue; the earliest deadline first. We use the earliest deadline to order tasks with equal latest deadline.

It is very complicated to make an optimal scheduling algorithm for tasks to minimize the energy consumption. We use a simple heuristic to evaluate tasks and determine sleep time rather than wasting energy on a fancy evaluation algorithm. Listing 9 gives this algorithm in pseudo code. First the mTask node checks for new tasks and updates of SDS. This communication adds any task to the queue. The `stepped` set contains all tasks evaluated in this evaluation round. Next, we evaluate tasks from the queue until we encounter a task that has an evaluation interval that is not started. This might evaluate tasks earlier than required, but maximizes the opportunities to sleep after this evaluation round. Executed tasks are temporarily stored in the `stepped` set instead of inserted directly into the queue to ensure that they are evaluated at most once in a evaluation round to ensure that there is frequent communication with the server. A task that produces a stable value is completed and is not queued again.

```
repeat {
    queue += communicateWithServer;
    stepped = empty                    // tasks stepped in this round
    while (notEmpty queue && earliestDeadline (top queue) ≤ currentTime ) {
        (task, queue) = pop queue;
        task2 = step task;             // includes computation of new execution interval
        if (not (isStable task2))      // not finished after step
            stepped += task2;
    }
    queue = merge queue stepped;
    sleep (queue);
}
```

**Listing 9.** Pseudo code for the evaluation round of tasks in the queue.

The `sleep` function determines the maximum sleep time based on the top of the queue. The computed sleep time and the characteristics of the microprocessor determine the length and depth of the sleep. For very short sleep times it might not be worthwhile to sleep. In the current mTask RTS, the thresholds are determined by experimentation but can be tuned by the programmer. On systems that lose the content of their RAM it is not possible to go to deep sleep mode.

## 8 Resulting Power Reductions

For the measurements we used the Adafruit Feather M0 WiFi since it is able to preserve its RAM memory and hence the mTask code and the task queue during deep sleep. The results are representative for any micro controller with similar sleeping characteristics. For the power measurements we used an INA219 current sensor[8] configured to measure currents from 0 to 400 mA with a resolution of

[8] See https://www.adafruit.com/product/904 for the sensor and https://github.com/adafruit/Adafruit_INA219 for the associated library.

0.1 mA. An Arduino Uno controls this sensor and measures the current every two milliseconds. We measure the currents in the power lines of the USB cable powering the microprocessor.

The mTask system uses the automatic Arduino power-saving mode for the Wi-Fi module in the old situation, but no other power savings. We compare this with the mTask system extended as described in this paper.

*Blink Example.* The simplest mTask program is the blink function from Listing 1. The microprocessor is always active in the old implementation and hence uses constantly about 15 mA. The LED adds about 1 mA when it is on. The figure on the bottom shows the new implementation, it shows that the system is in light sleep for most of the time. Periodically, the Wi-Fi stack needs to maintain the connection, resulting in current spikes in the graph (Fig. 2).

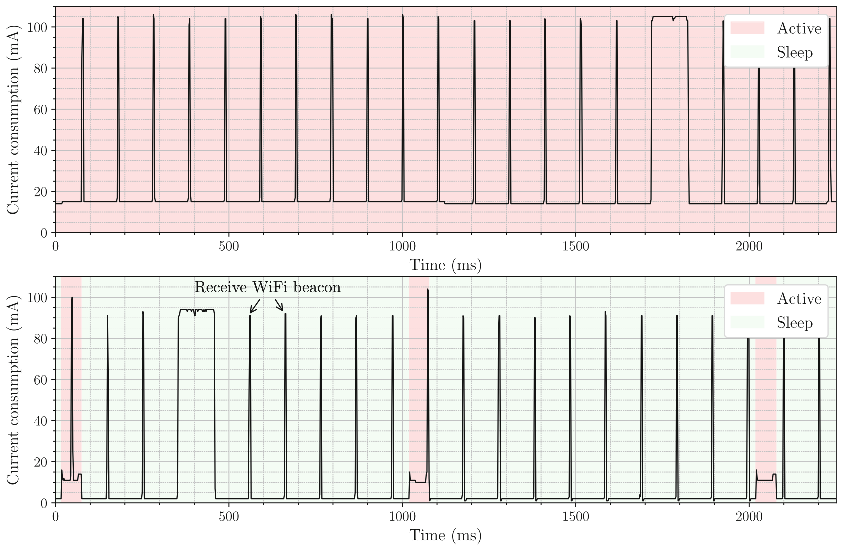

**Fig. 2.** Current draw of the blink task with the old implementation on top.

*Thermometer Example.* The thermometer task just reads the temperature from a Wemos SHT30 sensor[9].

```
thermometer :: Main (MTask v Bool) | mtask v
thermometer = DHT I2Caddr λsensor.
                {main = temperature' (BeforeSec (lit 60)) sensor}
```

**Listing 10.** A basic thermometer task.

The measurement will be repeated since this produces an unstable value. Figure 3 depicts the current consumption of this example. The longer upper

[9] See https://www.wemos.cc/en/latest/d1_mini_shield/sht30.html.

bound of the refresh interval enables deep sleep between individual measurements. As a consequence, the Wi-Fi connection needs to be re-established after waking up, resulting in a longer current spike.

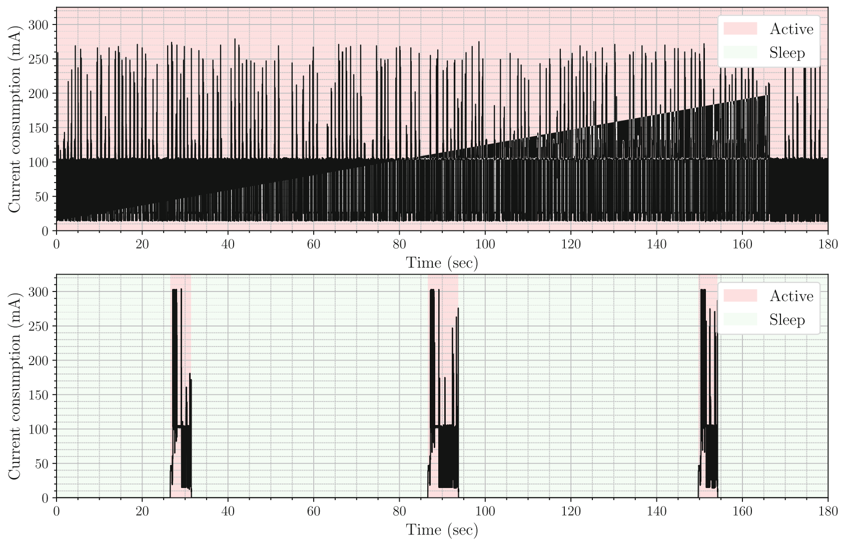

**Fig. 3.** Current draw of the temperature task with the old implementation on top.

*PIR Switch Example.* Next, we consider the PIR switch example from Listing 8. Since the old version of mTask does not support interrupts, we cannot execute the program directly. To mimic the effect as well as possible, we replaced `interrupt High pirPin` by a polling solution that waits until the pin reading is high; `readD pirPin >>*. [IfValue id rtrn]`. The measurement results are similar to the previous example, constant activity in the old implementation and sleep with small bursts in the new implementation. The actual interrupts determine the number of bursts seen so it does not make sense to plot the current.

*Plant Monitor.* This task observes a plant's environment and monitors two values, the amount of water in the soil and the light intensity. It becomes stable when one of the monitored values exceeds a threshold. The value of the soil sensor should be read at least every 5 min, 300 s, and the light intensity at least every 90 s.

```
monitorPlant :: Main (MTask v Bool) | mtask, dht, LightSensor v
monitorPlant = {main = lightSensor .||. moistureSensor}
where
    moistureSensor = readA` (BeforeSec (lit 300)) MoistureSensorPin
                     >>*. [IfValue ((<=.) moistureThreshold) (λ_. rtrn true)]
    lightSensor    = light` (BeforeSec (lit 90)) lightsensor
                     >>*. [IfValue ((<=.) lightThreshold) (λ_. rtrn true)]
```

**Listing 11.** The plant monitor task.

The combined refresh rate of the combinator $.\|.$ is determined by the shortest interval: $\langle 0, 90 \times 1000 \rangle$. We execute this task and the thermometer from Listing 10 on the same node. The thermometer task has a refresh interval of $\langle 0, 60 \times 1000 \rangle$. Hence, we should see the execution of both tasks every 60 s. This is exactly what is shown in Fig. 4.

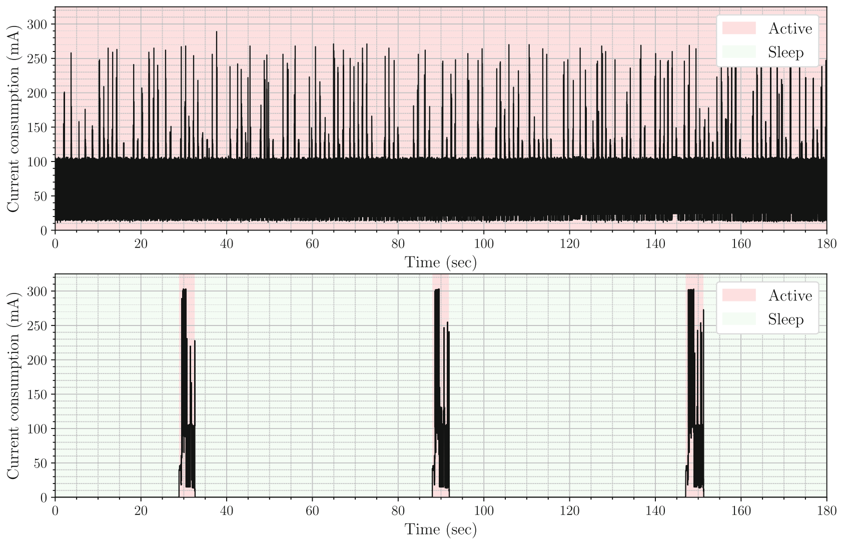

**Fig. 4.** Current draw of the thermometer task and the plant monitor.

### 8.1 Power Saving

By integrating with the composite trapezoidal rule, we calculate the total power consumption from the shown current measurements. The number of interrupts determines the saving for the PIR sensor example. Hence, we have not included these numbers in the table. Table 4 shows that our implementation works as expected. It reduces the energy consumption in all our examples considerably.

## 9 Related Work

Reducing the energy consumption of IoT nodes is currently getting attention from several directions. More modern microprocessors are typically bigger in terms of memory and computation power, while they consume less energy. Our work is complementary to this approach.

**Table 4.** Energy consumption and saving of the example programs.

| Task | Average energy consumption | | Difference |
|---|---|---|---|
| | Old | New | |
| Blink | 111.3 mW | 52.1 mW | −53% |
| Thermometer | 204.2 mW | 28.8 mW | −86% |
| Combined | 190.2 mW | 22.5 mW | −88% |

**Task-Oriented Programming.** Most implementation work for TOP is part of the iTask system [14,15]. The iTask framework generates web servers running on a normal computer. The efficiency of the generated programs focusses on short response times and preventing unnecessary computations. It also implements a sophisticated publish-subscribe system to handle updates caused by SDS. It is worthwhile to investigate if such a system can be implemented within the context of a microprocessor based system.

**Functional Reactive Programming.** Implementations of functional reactive programming (FRP) for microprocessors have similar goals as the mTask implementation of TOP [7]. The TOP system approaches the program from the tasks to be done and the change of those tasks. The FRP approach approaches the program from the streams of data from sensors and other inputs. This results in quite different programming styles [16].

Hae is a DSL for FRP deeply embedded in Haskell that generates C++ code. This code will be loaded in the flash memory of the microprocessors [19]. Hence, it cannot be updated as often and easy as the task in our mTask system.

Belwal et al. present an energy reduction technique for FRP based on Dynamic Voltage and Frequency Scaling (DVFS) [2]. Instead of bringing the system to sleep, they reduce the energy consumption by slowing the processor down.

**Embedded DSLs for Programming Microprocessors.** There are some other DSLs for programming microprocessors embedded in Functional Programming Languages. Haskino comes in various variants [9]. It enables to dynamically execute abstractions of Arduino statements on a small microprocessor. Haski focuses on the secure communication with IoT nodes [18].

**Embedded Operating Systems.** There are several embedded operating systems geared towards microprocessors, like FreeRTOS [1], Mantis OS [3] and Nano RK [8]. Those systems offer pre-emptive multitasking on microprocessors as well as several ways to reduce energy consumption. In contrast to the mTask system, those systems store the programs in flash memory and are hence less dynamically. Moreover, they are based on C-programs and cannot offer the reflection on the current state of tasks. Sleep modes must typically be invoked explicitly in the user program, or are at best invoked when all threads in the program explicitly invoke a sufficient long delay.

## 10 Conclusion

In this paper, we show how we can automatically associate execution intervals to tasks. Based on these intervals, we can delay the executions of those tasks. When all task executions can be delayed, the microprocessor executing those tasks can go to sleep mode to reduce its energy consumption. This is a rather difficult problem that must be solved dynamically, since we make no assumptions on the number and nature of the tasks that will be allocated to an IoT node. Furthermore, the execution intervals offer an elegant and efficient way to add interrupts to the language. Those interrupts offer a more elegant and energy efficient implementation of watching an input than polling this input.

The actual reduction of the energy is of course highly dependent on the number and nature of the task shipped to the IoT node. Our examples show a reduction in energy consumption of two orders of magnitude. Those reductions are a necessity for IoT nodes with battery power. Given the exploding number of IoT nodes, such savings are also mandatory for other nodes to limit the total power consumption of the IoT.

**Acknowledgements.** This research is partly funded by the Royal Netherlands Navy. Furthermore, we would like to thank the anonymous reviewers for their invaluable comments.

## References

1. Barry, R.: Using the FreeRTOS Real Time Kernel - A Practical Guide (2009)
2. Belwal, C., Cheng, A.M.K., Ras, J., Wen, Y.: Variable voltage scheduling with the priority-based functional reactive programming language. In: Proceedings of the 2013 Research in Adaptive and Convergent Systems, pp. 440–445 (2013). https://doi.org/10.1145/2513228.2513271
3. Bhatti, S., et al.: Mantis OS: an embedded multithreaded operating system for wireless micro sensor platforms. Mob. Netw. Appl. **10**(4), 563–579 (2005). https://doi.org/10.1007/s11036-005-1567-8
4. 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.) FPCA 1987. LNCS, vol. 274, pp. 364–384. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_20
5. Carette, J., Kiselyov, O., Shan, 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
6. Clean team: Clean 3.0 language report (2020). https://cloogle.org/doc. Accessed 05 Apr 2022

7. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, pp. 263–273 (1997). https://doi.org/10.1145/258948.258973
8. Eswaran, A., Rowe, A., Rajkumar, R.: Nano-RK: an energy-aware resource-centric RTOS for sensor networks. In: 26th IEEE International Real-Time Systems Symposium, pp. 256–265 (2005). https://doi.org/10.1109/RTSS.2005.30
9. Grebe, M., Gill, A.: Threading the Arduino with Haskell. In: Van Horn, D., Hughes, J. (eds.) TFP 2016. LNCS, vol. 10447, pp. 135–154. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14805-8_8
10. Koopman, P., Lubbers, M., Plasmeijer, R.: A task-based DSL for microcomputers. In: Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL 2018, Vienna, Austria, pp. 1–11. ACM Press (2018). https://doi.org/10.1145/3183895.3183902
11. Lubbers, M., Koopman, P., Plasmeijer, R.: Writing Internet of Things applications with task oriented programming. arXiv preprint arXiv:2212.04193 (2022)
12. 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, IoT 2020. ACM, New York (2020). https://doi.org/10.1145/3410992.3411002
13. Nižetić, S., Šolić, P., López-de-Ipiña González-de-Artaza, D., Patrono, L.: Internet of things (IoT): opportunities, issues and challenges towards a smart and sustainable future. J. Clean. Prod. **274**, 122877 (2020). https://doi.org/10.1016/j.jclepro.2020.122877
14. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), Freiburg, Germany, pp. 141–152. ACM (2007)
15. 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, PPDP 2012, pp. 195–206. ACM, New York (2012). https://doi.org/10.1145/2370776.2370801
16. Stutterheim, J., Achten, P., Plasmeijer, R.: Maintaining separation of concerns through task oriented software development. In: Wang, M., Owens, S. (eds.) TFP 2017. LNCS, vol. 10788, pp. 19–38. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89719-6_2
17. Tratt, L.: Domain specific language implementation via compile-time metaprogramming. ACM Trans. Program. Lang. Syst. **30**(6) (2008). https://doi.org/10.1145/1391956.1391958
18. Valliappan, N., Krook, R., Russo, A., Claessen, K.: Towards secure IoT programming in Haskell. In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell 2020, pp. 136–150. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3406088.3409027
19. Wang, S., Watanabe, T.: Functional reactive EDSL with asynchronous execution for resource-constrained embedded systems. In: Lee, R. (ed.) SNPD 2019. SCI, vol. 850, pp. 171–190. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-26428-4_12