Radboud University

Computing Science: Master's Thesis

Task Oriented Programming and the Internet of Things

Author:
Mart Lubbers BSc.
mart@martlubbers.net

Supervisor:
prof. dr. dr.h.c. ir. M.J. Plasmeijer

Second reader:
dr. P.W.M. Koopman

10th July, 2017

Abstract

This thesis introduces an innovative way to connect small Internet of Things devices to high level Task Oriented Programming implementations languages. The existing class-based shallowly Embedded Domain Specific Language called mTask by Koopman et al. — written in Clean — is extended with a new view to allow compilation of Internet of Things Tasks on the fly and send them to devices as interpretable bytecode. All introduced functionality adheres to the Task Oriented Programming philosophy where common concepts such as Shared Data Sources and Task-combinators are available at ease.

Acknowledgements

I would like to thank everyone who supported me during this thesis. This includes but is not limited to Rinus and Pieter for the supervision and weekly meetings. Bas, for providing iTasks *insight. Arjan, for sparring and coffee and George, Camil and Roy for proofreading the thesis. Finally, I would like to thank my family, friends and wife for listening to my year of endless rambling on the subject.*

Contents

Chapter 1

Introduction

1.1 Introduction

1.1.1 Internet of Things

Internet of Things (IoT) technology is emerging rapidly. It offers myriads of solutions and is transforming the way people interact with technology.

The term IoT was coined around 1999 to describe Radio-Frequency Identification (RFID) devices and the communication between them. After a small slumber of the term, it resurfaced recently and has changed definition slightly. In the current day and age, IoT encompasses all small devices that communicate with each other and — most of all — with the world. It has been estimated that there will be around 30 billion IoT devices online in 2020. Even today, IoT devices are already in everyone's household in the form of smart electricity meters, smart fridges, smartphones, smart watches. These devices are often equipped with sensors, Global Navigation Satellite System (GNSS) modules[1] and actuators [1]. With these new technologies, information can be tracked accurately using little power, bandwidth and money. Moreover, IoT technology is coming into healthcare as well [2]. For example, for a few euros a consumer ready fitness tracker watch can be bought that tracks heartbeat and respiration levels.

The architecture of IoT systems is often divided into layers. A very popular division is the four layer architecture but there are also proponents of a five layer structure. The first layer of the four layer architecture is the sensing layer. This layer contains the actual sensing and acting hardware. In a smart electricity meter, this layer would contains the sensors detecting the current drawn. There are myriads of device available to use in this layer and they can be programmed using a variety of different low level programming languages such as C++, C but also higher level languages such as *Python* and *LUA*. The second layer of IoT is the networking layer and is responsible for connecting the first layer with the outer world. In a smart electricity meter, this would be the GSM modem connecting the meter to a server. Existing networking techniques — such as WiFi and GSM — are used to convey IoT information but there are also specialized communication techniques devised for IoT such as ZigBee, LoRa and Bluetooth Low Energy. The third layer is called the service layer. This layer is responsible for all the servicing and business rules surrounding the application. It provides Application Programming Interfaces (APIs) and interfaces to, and storage of the data. Finally, the fourth layer is the application layer. This final layer provides the applications that the user can use to interact with the IoT

[1] e.g. the American Global Positioning System (GPS) or the Russian Global Navigation Satellite System (GLONASS).

devices and their data. In a smart electricity meter, this layer would be the app that can be used to monitor the electricity consumption.

The separation of IoT in layers is a difficulty when developing IoT applications. All layers use different paradigms, languages and architectures which leads to isolated logic which makes it difficult to integrate. Rolling out changes to the system is also complicated since reprogramming microcontrollers in the field is very expensive. Even the changing a few parameters on a device requires often a full reprogram.

1.1.2 Task Oriented Programming

The Task Oriented Programming (TOP) paradigm and the corresponding *iTasks* implementation offer a high abstraction level for real world workflow tasks [3]. These workflow tasks can be described through an Embedded Domain Specific Language (EDSL) hosted in the purely functional programming language *Clean*. *Task*s are the basic building blocks of the language and they resemble actual workflow tasks. For the specification, the system will generate a multi-user web application. This web service can be accessed through a browser and is used to complete these *Task*s. Familiar workflow patterns like sequential, parallel and conditional *Task*s chaining can be modelled in the language.

ITasks has shown to be useful in many fields of operation such as incident management [4]. *ITasks* is highly type driven and is built on generic functions that generate functionality for the given types. This results in the programmer having to do very little implementation work on details such as user interfaces. It is possible to change the derived functions and adapt them to needs.

1.2 Problem statement

*Task*s in the *iTasks* system are modelled after real life workflow tasks but the modelling is applied on a high level. Therefore, it is difficult to connect *iTasks*-*Task*s to real world tasks and allow them to interact. A lot of the actual tasks could very well be performed by IoT devices. Nevertheless, adding such devices to the current system is difficult to say the least as it was not designed to cope with these devices.

In the current system such adapters connecting devices to *iTasks* — in principle — can be written in two ways.

First, an adapter for a specific device can be written as a Shared Data Sources (SDSs)[2]. SDSs can interact with the world and thus with hardware, allowing communication with any type of device. However, this requires a tailor-made SDS for every specific device and functionality and does not allow logic to be changed. Once a device is programmed to serve as an SDS, it has to behave like that forever. Thus, this solution is not suitable for systems that can send *Task*s to the device dynamically.

The second method uses the novel contribution to *iTasks* by Oortgiese et al. They lifted *iTasks* from a single server model to a distributed server architecture [5]. As a proof of concept, an android app has been created that runs an entire *iTasks* core and is able to receive *Task*s from a different server and execute them. While android often runs on small Acorn RISC Machine (ARM) devices, they are a lot more powerful than the average IoT microcontroller. The system is suitable for dynamically sending *Task*s but running the entire *iTasks* core on a microcontroller is not feasible. Even if it would be possible, this technique would still not be suitable because a lot of communication overhead is needed to transfer the *Task*s. IoT devices are often connected

[2]Similar as to resources such as time are available in the current *iTasks* implementation.

to the server through *Low Power Low Bandwidth* which is unsuitable for transferring a lot of data.

The novel system that has been devised bridges the gap between the aforementioned solutions for adding IoT to *iTasks*. The system consists of updates to the *mTask*-EDSL [6], a new communication protocol, device application and an *iTasks* server application. The system supports devices as small as *Arduino* microcontrollers [7] and operates via the same paradigms and patterns as regular *Task*s in the TOP paradigm. Devices in the *mTask*-system can run small imperative programs written in the EDSL and have access to SDSs. *Task*s are sent to the device at runtime, avoiding recompilation and thus write cycles on the program memory. This solution extends the reach of *iTasks* and allows closer resemblance of *Task*s to actual tasks. Moreover, it tries to solve some integration problems in IoT by allowing all components to be programmed from one source.

1.3 Document structure

The structure of this thesis is as follows.

Chapter 1 contains the problem statement, motivation, related work and the structure of the document. Chapter 2 introduces the reader to the basics of TOP and *iTasks*. Chapter 3 discusses the pros and cons of different embedding methods to create EDSL. Chapter 4 shows the existing *mTask*-EDSL which is extended upon in this dissertation. Chapter 5 gives an overview of the proposed system in the broadest sense. Chapter 6 describes the added view and functionality for the *mTask*-EDSL that were added and used in the system. Chapter 7 shows the implementation and considerations for entire system. It covers the client software running on the device and the server written in *iTasks*. Chapter 8 concludes by answering the research questions and discusses future research. Appendix A shows the concrete protocol used for communicating between the server and client. Appendix B shows the concrete interface for the devices.

Some conventions have been kept throughout the document. Text written using the `Teletype` font indicates code and is often referring to section of a listing. *Emphasized* text is used for proper nouns and words that have an unexpected meaning. SMALL CAPS is used for branded acronyms. When the word Tasks is emphasized and capitalized, it refers to the task-entities from either the *mTask* or the *iTasks* system.

The complete source code of this thesis can be found in the following git repository:
`https://git.martlubbers.net/msc-thesis1617.git`

The complete source code of the *mTask*-system can be found in the following git repository:
`https://git.martlubbers.net/mTask.git`

1.4 Related work

Similar research has been conducted on the subject. For example, microcontrollers such as the *Arduino* can be remotely controlled very directly using the *Firmata*-protocol[3]. This protocol is designed to allow control of the peripherals — such as sensors and actuators — directly through commands sent via a communication channel such as a serial port. This allows very fine grained control but with the cost of excessive communication overhead since no code is executed on the device itself, only the peripherals are queried. A *Haskell* implementation of the protocol is also available[4]. The hardware requirements for running a *Firmata* client are very low. However, the

[3] "firmata/protocol: Documentation of the Firmata protocol." (`https://github.com/firmata/protocol`). [Accessed: 23-May-2017].

[4] "hArduino by LeventErkok." (`https://leventerkok.github.io/hArduino`). [Accessed: 23-May-2017].

communication requirements are high and therefore it is not suitable for IoT applications that operate through specialized IoT networks which often only support low bandwidth.

Clean has a history of interpretation and there is a lot of research happening on the intermediate language SAPL. SAPL is a purely functional intermediate language that is designed to be interpreted. It has interpreters written in C++ [8] and *Javascript* [9]. Compiler backends exist for and *Clean* and *Haskell* which compile the respective code to SAPL [10]. The SAPL language is still a functional language and therefore requires big stacks and heaps to operate and is therefore not directly suitable for devices with little RAM such as the *Arduino* Uno which only boasts $2K$ of RAM. It might be possible to compile the SAPL code into efficient machine language or C but then the system would lose its dynamic properties since the microcontroller then would have to be reprogrammed every time a new *Task* is sent to the device.

EDSLs have often been used to generate C code for microcontroller environments. This work uses parts of the existing *mTask*-EDSL which generates C code to run a TOP-like system on microcontrollers [11] [6]. Again, this requires a reprogramming cycle every time the *Task*-specification is changed. Hence, the EDSL is used but the backend is not suitable for the purpose of dynamic IoT solutions.

Another EDSL designed to generate low-level high-assurance programs is called *Ivory* and uses *Haskell* as a host language [12]. The language uses the *Haskell* type-system to make unsafe languages type safe. For example, *Ivory* has been used in the automotive industry to program parts of an autopilot [13] [14]. *Ivory*'s syntax is deeply embedded but the type system is shallowly embedded. This requires several *Haskell* extensions that offer dependent type constructions. The process of compiling an *Ivory* program happens in two stages. The embedded code is transformed into an Abstract Syntax Tree (AST) that is sent to a chosen backend. The technique used in the novel system using the *mTask*-EDSL is different, in the new system, the EDSL is transformed directly into functions. There is no intermediate AST. Moreover, *Ivory* generates static programs and thus it is necessary to reprogram the devices when they need to be repurposed. It would be interesting to explore the possibilities of writing the client software in an EDSL as well.

Not all IoT devices run solely compiled code. The popular *ESP8266* powered NodeMCU is able to run interpreted *LUA* code. Moreover, there is a variation on *Python* called *micropython* that is suitable for running on microcontrollers. However, the overhead of the interpreter for such rich languages often results into limitations on the program size. It would not be possible to repurpose a device with IoT because implementing this extensibility in the interpreted language leaves no room for the actual programs. Also, some devices only have $2K$ of ram, which is not enough for this.

Chapter 2

Task Oriented Programming

2.1 iTasks

TOP is a novel programming paradigm implemented as *iTasks* [15] in the pure lazy functional language *Clean* [16]. *iTasks* is an EDSL to model workflow tasks in the broadest sense. A *Task* is just a function that — given some state — returns the observable `TaskValue`. The `TaskValue` of a `Task` can have different states. Not all state transitions are possible as shown in Figure 2.1. Once a value is stable it can never become unstable again. Stability is often reached by pressing a confirmation button. *Task*s yielding a constant value are immediately stable.

A simple *iTasks* example illustrating the route to stability of a *Task* in which the user has to enter a full name is shown in Listing 2.1. The code is accompanied by screenshots showing the user interface in Figure 2.2a, 2.2b and 2.2c. The `TaskValue` of the *Task* is in the first image in the `NoValue` state, the second image does not have all the fields filled in and therefore the `TaskValue` remains `NoValue`. In the third image all fields are entered and the `TaskValue` transitions to the `Unstable` state. When the user presses *Continue* the value becomes `Stable` and cannot be changed any further.

Figure 2.1: The states of a `TaskValue`

```
:: Name = { firstname :: String
          , lastname  :: String
          }

derive class iTask Name

enterInformation :: String [EnterOption m] -> (Task m) | iTask m

enterName :: Task Name
enterName = enterInformation "Enter your name" []
```

Listing 2.1: An example *Task* for entering a name

(a) Initial interface (b) Incomplete entrance (c) Complete entry

Figure 2.2: Example of a generated user interface

For a type to be suitable, it must have instances for a collection of generic functions that is captured in the class `iTask`. Basic types have specialization instances for these functions and show an interface accordingly. Derived interfaces can be modified with decoration operators or specializations can be created.

2.2 Combinators

*Task*s in *iTasks* can be combined using so called *Task*-combinators. Combinators describe relations between *Task*s. There are only two basic types of combinators; parallel and sequence. All other combinators are derived from the basic combinators. Type signatures of simplified versions of the basic combinators and their derivations are given in Listing 2.2

```
//Step combinator
(>>=)  infixl 1 :: (Task a) (a -> Task b)         -> Task b   | iTask a & iTask b
(>>*)  infixl 1 :: (Task a) [TaskCont a (Task b)] -> Task b    | iTask a & iTask b
:: TaskCont a b
   =     OnValue             ((TaskValue a)  -> Maybe b)
   |     OnAction    Action  ((TaskValue a)  -> Maybe b)
   | ∃e: OnException         (e              -> b)       & iTask e
   |     OnAllExceptions     (String         -> b)
:: Action = Action String

//Parallel combinators
(-||-) infixr 3 :: (Task a) (Task a)              -> Task a     | iTask a
(||-)  infixr 3 :: (Task a) (Task b)              -> Task b     | iTask a & iTask b
(-||)  infixl 3 :: (Task a) (Task b)              -> Task a     | iTask a & iTask b
(-&&-) infixr 4 :: (Task a) (Task b)              -> Task (a,b) | iTask a & iTask b
```

Listing 2.2: *Task*-combinators

Sequence: The implementation for the sequence combinator is called the `step` (>>*). This combinator runs the left-hand *Task* and starts the right-hand side when a certain predicate holds. Predicates can be propositions about the `TaskValue`, user actions from within the web browser or a thrown exception. The familiar bind-combinator is an example of a sequence combinator. This combinator runs the left-hand side and continues to the right-hand *Task* if there is an `UnStable` value and the user presses continue or when the value is `Stable`. The combinator could have been implemented as follows:

```
(>>=) infixl 1 :: (Task a) (a -> (Task b)) -> (Task b) | iTask a & iTask b
(>>=) ta f = ta >>* [OnAction "Continue" onValue, OnValue onStable]
   where
      onValue (Value a _)     = Just (f a)
      onValue _               = Nothing

      onStable (Value a True) = Just (f a)
      onStable _              = Nothing
```

Parallel: The parallel combinator allows for concurrent *Task*s. The *Task*s combined with these operators will appear at the same time in the web browser of the user and the results are combined as the type dictates. All parallel combinators used are derived from the basic parallel combinator that is very complex and only used internally.

2.3 Shared Data Sources

SDSs are an abstraction over resources that are available in the world or in the *iTasks* system. The shared data can be a file on disk, the system time, a random integer or just some data stored in memory. The actual SDS is just a record containing functions on how to read and write the source. In these functions the `*IWorld` — which in turn contains the real `*World` — is available. Accessing the outside world is required for interacting with it and thus the functions can access files on disk, raw memory, other SDSs and hardware.

The basic operations for SDSs are get, set and update. The signatures for these functions are shown in Listing 2.3. By default, all SDSs are files containing a JavaScript Object Notation (JSON) encoded version of the object and thus are persistent between restarts of the program. Library functions for shares residing in memory are available as well. The three main operations on shares are atomic in the sense that during reading no other *Task*s are executed. The system provides useful functions to transform, map and combine SDSs using combinators. The system also provides functionality to inspect the value of an SDS and act upon a change. *Task*s waiting on an SDS to change are notified when needed. This results in low resource usage because *Task*s are never constantly inspecting SDS values but are notified.

```
:: RWShared p r w = ...
:: ReadWriteShared r w :== RWShared () r w
:: ROShared p r :== RWShared p () r
:: ReadOnlyShared r :== ROShared () r

:: Shared r :== ReadWriteShared r r

get ::          (ReadWriteShared r w)          -> Task r | iTask r
set :: w        (ReadWriteShared r w)          -> Task w | iTask w
upd :: (r -> w) (ReadWriteShared r w)          -> Task w | iTask r & iTask w

sharedStore :: String a -> Shared a | JSONEncode{|*|}, JSONDecode{|*|}
```

Listing 2.3: SDS functions

2.4 Parametric Lenses

SDSs can contain complex data structures such as lists, trees and even resources in the outside world. Sometimes, an update action only updates a part of the resource. When this happens, all waiting *Task*s looking at the resource are notified of the update. However, it may be the case that *Task*s were only looking at parts of the structure that was not updated. To solve this problem, parametric lenses were introduced [17].

Parametric lenses add a type variable to the SDS. This type variable is fixed to the void type (i.e. `()`) in the given functions. When an SDS executes a write operation, it also provides the system with a notification predicate. This notification predicate is a function `p -> Bool` where `p` is the parametric lens type. This allows programmers to create a big SDS, and have *Task*s only look at parts of the big SDS. This technique is used in the current system in memory shares. The `IWorld` contains a map that is accessible through an SDS. While all data is stored in the

map, only *Task*s looking at a specific entry are notified when the structure is updated. The type of the parametric lens is the key in the map.

Functionality for setting parameters is available in the system. The most important functions are the sdsFocus and the sdsLens function. These functions are listed in Listing 2.4. sdsFocus allows the programmer to fix a parametric lens value. sdsLens is a kind of mapReadWrite including access to the parametric lens value. This allows the creation of, for example, SDSs that only read and write to parts of the original SDS.

```
sdsFocus :: p (RWShared p r w) -> RWShared p` r w | iTask p

:: SDSNotifyPred p :== p -> Bool

:: SDSLensRead p r rs     = SDSRead        (p -> rs -> MaybeError TaskException r)
                          | SDSReadConst   (p -> r)
:: SDSLensWrite p w rs ws = SDSWrite       (p -> rs -> w -> MaybeError TaskException (Maybe ws))
                          | SDSWriteConst  (p -> w -> MaybeError TaskException (Maybe ws))
:: SDSLensNotify p w rs   = SDSNotify      (p -> rs -> w -> SDSNotifyPred p)
                          | SDSNotifyConst (p -> w -> SDSNotifyPred p)

sdsLens :: String (p -> ps) (SDSLensRead p r rs) (SDSLensWrite p w rs ws) (SDSLensNotify p w rs)
    (RWShared ps rs ws) -> RWShared p r w | iTask ps
```

Listing 2.4: Parametric lens functions

Chapter 3

Embedded Domain Specific Languages

An EDSL is a language embedded in a host language. EDSLs can have one or more backends or views. Commonly used views are pretty printing, compiling, simulating, verifying and proving the program. There are several techniques available for creating EDSLs. They all have their own advantages and disadvantages in terms of extendability, typedness and view support. In the following subsections each of the main techniques are briefly explained.

3.1 Deep Embedding

A deep EDSL is a language represented as an Algebraic Datatype (ADT). Views are functions that transform something to the datatype or the other way around. As an example, take the simple arithmetic EDSL shown in Listing 3.1.

```
:: DSL
   = LitI  Int
   | LitB  Bool
   | Var   String
   | Plus  DSL DSL
   | Minus DSL DSL
   | And   DSL DSL
   | Eq    DSL
```

Listing 3.1: A minimal deep EDSL

Deep embedding has the advantage that it is easy to build and views are easy to add. To the downside, the expressions created with this language are not type-safe. In the given language it is possible to create an expression such as `Plus (LitI 4) (LitB True)` that adds a boolean to an integer. Evermore so, extending the ADT is easy and convenient but extending the views accordingly is tedious and has to be done individually for all views.

The first downside of this type of EDSL can be overcome by using Generalized Algebraic Data types (GADTs) [18]. Listing 3.2 shows the same language, but type-safe with a GADT. GADTs are not supported in the current version of *Clean* and therefore the syntax is hypothetical. However, it has been shown that GADTs can be simulated using bimaps or projection pairs [19]. Unfortunately the lack of extendability remains a problem. If a language construct is added, no compile time guarantee is given that all views support it.

```
:: DSL a
   =    LitI  Int                 -> DSL Int
   |    LitB  Bool                -> DSL Bool
   | ∃e: Var   String              -> DSL e
   |    Plus  (DSL Int) (DSL Int)  -> DSL Int
   |    Minus (DSL Int) (DSL Int)  -> DSL Int
   |    And   (DSL Bool) (DSL Bool) -> DSL Bool
   | ∃e: Eq (DSL e) (DSL e)        -> DSL Bool & == e
```

Listing 3.2: A minimal deep EDSL using GADTs

3.2 Shallow Embedding

In a shallow EDSL all language constructs are expressed as functions in the host language. An evaluator view for the example language then can be implemented as the code shown in Listing 3.3. Note that much of the internals of the language can be hidden using monads.

```
:: Env   = ...           // Some environment
:: DSL a = DSL (Env -> a)

Lit :: a -> DSL a
Lit x = λe -> x

Var :: String -> DSL Int
Var i = λe -> retrEnv e i

Plus :: (DSL Int) (DSL Int) -> DSL Int
Plus x y = λe -> x e + y e

...

Eq :: (DSL a) (DSL a) -> DSL Bool | == a
Eq x y = λe -> x e + y e
```

Listing 3.3: A minimal shallow EDSL

The advantage of shallowly embedding a language in a host language is its extendability. It is very easy to add functionality and compile time checks of the host language guarantee whether or not the functionality is available when used. Moreover, the language is type safe as it is directly typed in the host language.

The downside of this method is extending the language with views. It is nearly impossible to add views to a shallowly embedded language. The only way of achieving this is by decorating the datatype for the EDSL with all the information for all the views. This will mean that every component will have to implement all views rendering it slow for multiple views and complex to implement.

3.3 Class Based Shallow Embedding

The third type of embedding is called class-based shallow embedding and has the advantages of both shallow and deep embedding [20]. In class-based shallow embedding the language constructs are defined as type classes. This language is shown with the new method in Listing 3.4.

This type of embedding inherits the ease of adding views from shallow embedding. A view is just a different data type implementing one or more of the type classes as shown in the aforementioned Listing where an evaluator and a pretty printer are implemented.

Just as with GADTs, type safety is guaranteed in deep embedding. Type constraints are enforced through phantom types. One can add as many phantom types as necessary. Lastly, extensions can be added easily, just as in shallow embedding. When an extension is made in an existing class, all views must be updated accordingly to prevent possible runtime errors. When an extension is added in a new class, this problem does not arise and views can choose to implement only parts of the collection of classes.

In contrast to deep embedding, it is very well possible to have multiple views applied on the same expression. This is also shown in the following listing.

```
:: Env   = ...           // Some environment
:: Evaluator a = Evaluator (Env -> a)
:: PrettyPrinter a = PP String

class intArith where
    lit   :: t -> v t             | toString t
    add   :: (v t) (v t) -> (v t) | + t
    minus :: (v t) (v t) -> (v t) | - t

class boolArith where
    and :: (v Bool) (v Bool) -> (v Bool)
    eq  :: (v t)    (v t)    -> (v Bool) | == t

instance intArith Evaluator where
    lit x   = Evaluator λe->x
    add x y = Evaluator ...

instance intArith PrettyPrinter where
    lit x   = PP $ toString x
    add x y = PP $ x +++ "+" +++ y
    ...

...

Start :: (PP String, Bool)
Start = (print e0, eval e0)
where
    e0 :: a Bool | intArith, boolArith a
    e0 = eq (lit 42) (lit 21 +. lit 21)

    print (PP p) = p
    eval (Evaluator e) env = e env
```

Listing 3.4: A minimal class based shallow EDSL

Chapter 4

The *mTask*-EDSL

The *mTask*-EDSL was created by Koopman et al. and supports several views such as an *iTasks* simulation and a C-code generator. The EDSL was designed to generate a ready-to-compile TOP-like program for microcontrollers such as the *Arduino* [6][11].

The *mTask*-EDSL is a shallowly embedded class based EDSL and therefore it is very suitable to have a new backend that partly implements the classes given. The following sections show the details of the EDSL that is used in this extension. The parts of the EDSL that are not used will not be discussed and the details of those parts can be found in the cited literature.

A view for the *mTask*-EDSL is a type with two free type variables[1] that implements some of the classes given. The types do not have to be present as fields in the view and can, and will most often, be exclusively phantom types. Thus, views are of the form: `:: v t r = ...`. The first type variable will be the type of the view. The second type variable will be the type of the EDSL-expression and the third type variable represents the role of the expression. Currently the role of the expressions form a hierarchy. The three roles and their hierarchy are shown in Listing 4.1. This implies that everything is a statement, only an `Upd` and an `Expr` are expressions. The `Upd` restriction describes updatable expressions such as General-Purpose Input/Output (GPIO) pins and SDSs. The roles are used to constrain certain classes. For example, without the roles for `Upd`. Assignment would be possible to a non-assignable expression such as a literal integer.

```
:: Upd  = Upd
:: Expr = Expr
:: Stmt = Stmt

class isExpr a :: a -> Int
instance isExpr Upd
instance isExpr Expr
```

Listing 4.1: Expression role hierarchy

4.1 Expressions

Expressions in the *mTask*-EDSL are divided into two types, namely boolean expressions and arithmetic expressions. The class of arithmetic language constructs also contains the function `lit` that lifts a host-language value into the EDSL domain. All standard arithmetic functions are included in the EDSL but are omitted in the example for brevity. Moreover, the class restrictions

[1] kind *->*->*.

are only shown in the first functions and are omitted in subsequent functions. Both the boolean expression and arithmetic expression classes are shown in Listing 4.2.

```
class arith v where
  lit          :: t -> v t Expr
  (+.) infixl 6 :: (v t p) (v t q) -> v t Expr          | +, zero t    & isExpr p & isExpr q
  (-.) infixl 6 :: (v t p) (v t q) -> v t Expr          | -, zero t    & ...
  ...
class boolExpr v where
  Not          :: (v Bool p) -> v Bool Expr            | ...
  (&.) infixr 3 :: (v Bool p) (v Bool q) -> v Bool Expr | ...
  ...
  (==.) infix 4 :: (v a p) (v a q) -> v Bool Expr       | ==, toCode a & ...
```

Listing 4.2: Basic classes for expressions

4.2 Control flow

Looping of *Task*s happens because *Task*s are executed after waiting a specified amount of time or when they are launched by another *Task* or even themselves. Therefore there is no need for loop control flow functionality such as *while* or *for* constructions. The main control flow operators are the sequence operator and the *if* statement. Both are shown in Listing 4.3. The first class of *If* statements describes the regular *if* statement. The expressions given can have any role. The functional dependency on `s` determines the return type of the statement. The listing includes examples of implementations that illustrate this dependency. A special *If* statement — only used for statements — is also added under the name `IF`, of which the `?` is a conditional statement to execute.

The sequence operator is straightforward and its only function is to tie two expressions together. The left expression is executed first, followed by the right expression.

```
class IF v where
  IF :: (v Bool p) (v t q) (v s r) -> v () Stmt | ...
  (?) infix 1 :: (v Bool p) (v t q) -> v () Stmt | ...

class seq v where
  (:.) infixr 0 :: (v t p) (v u q) -> v u Stmt | ...
```

Listing 4.3: Control flow operators

4.3 Input/Output

Values can be assigned to all expressions that have an `Upd` role. Examples of such expressions are SDSs and GPIO pins. Moreover, class extensions can be created for specific peripherals such as built-in LEDs. The classes facilitating this are shown in Listing 4.4. In this way the assignment is the same for every assignable entity.

```
:: DigitalPin = D0 | D1 | D2 | D3 | D4 | D5 |D6 | D7 | D8 | D9 | D10 | D11 | D12 | D13
:: AnalogPin  = A0 | A1 | A2 | A3 | A4 | A5
:: UserLED = LED1 | LED2 | LED3

class dIO v where dIO :: DigitalPin -> v Bool Upd
class aIO v where aIO :: AnalogPin -> v Int Upd
class analogRead v where
    analogRead :: AnalogPin -> v Int Expr
    analogWrite :: AnalogPin (v Int p) -> v Int Expr
```

```
class digitalRead v where
    digitalRead :: DigitalPin -> v Bin Expr
    digitalWrite :: DigitalPin (v Bool p) -> v Int Expr

:: UserLED = LED1 | LED2 | LED3
class userLed v where
    ledOn  :: (v UserLED q) -> (v () Stmt)
    ledOff :: (v UserLED q) -> (v () Stmt)

class assign v where
    (=.) infixr 2 :: (v t Upd) (v t p) -> v t Expr | ...
```

Listing 4.4: Input/Output classes

One way of storing data in *mTask*-*Task*s is using SDSs. SDSs serve as global variables in *mTask* and maintain their value across executions. SDSs can be used by multiple *Task*s and can be used to share data. The classes associated with SDSs are listed in Listing 4.5. The `Main` type is introduced to box an *mTask* and make it recognizable by the type system by separating programs and decorations such as SDSs. The type signature is complex and uses infix type constructors and therefore, an implementation example is also given.

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

class sds v where
    sds :: ((v t Upd) -> In t (Main (v c s))) -> (Main (v c s)) | ...

sdsExample :: Main (v Int Stmt)
sdsExample = sds λx.0 In
    {main= x =. x +. lit 42 }
```

Listing 4.5: SDSs in *mTask*

4.4 Class Extensions

In the *Arduino* ecosystem, shields are available to plug into the microcontroller and add functionality. These shields range from Bluetooth, WiFi, Ethernet, LoRa, LCD screens and much more. Often the functionality available in these shields is housed in a C++ class. This functionality is ported using little work to *mTask* by just creating a corresponding class with the same functions. As an example, Listing 4.6 shows parts of the Liquid Crystal Display (LCD) class as an *mTask* class functions and as Listing 4.7 shown the corresponding *Arduino* class functions.

```
:: LCD = ...

class lcd v where
  begin      :: (v LCD Expr) (v Int p) (v Int q) -> v () Expr
  LCD        :: Int Int [DigitalPin] ((v LCD Expr) -> Main (v b q)) -> Main (v b q)
  ...
  scrollLeft  :: (v LCD Expr) -> v () Expr
  scrollRight :: (v LCD Expr) -> v () Expr
  ...
```

Listing 4.6: Adding the LCD to the *mTask* language

```
class LiquidCrystal {
public:
  void begin(uint8_t cols, uint8_t rows);
  ...
```

```
  void scrollDisplayLeft();
  void scrollDisplayRight();
  ...
}
```

Listing 4.7: Functions from the *Arduino* LCD library

4.5 Scheduling Strategy

The C-backend of the *mTask*-system has an engine that is generated alongside the code for the *Task*s. This engine will execute the *mTask*-*Task*s according to certain rules and execution strategies. *mTask*-*Task*s do not behave like functions but more like *iTasks*-*Task*s. An *mTask* is queued when either its timer runs out or when it is launched by another *mTask*. When an *mTask* is queued it does not block the execution and it will return immediately while the actual *Task* will be executed anytime in the future.

The *iTasks*-backend simulates the C-backend and thus uses the same scheduling strategy. This engine expressed in pseudocode is listed as Algorithm 4.1. All the *Task*s are inspected on their waiting time. When the waiting time has not passed; the delta is subtracted and the *Task* gets pushed to the end of the queue. When the waiting has surpassed they are executed. When an *mTask* opts to queue another *mTask* it can just append it to the queue.

```
Data: queue queue, time t, t_p
t ← now();
begin
    while true do
        t_p ← t;
        t ← now();
        if notEmpty(queue) then
            task ← queue.pop();
            task.wait ← task.wait −(t − t_p);
            if task.wait > t_0 then
                queue.append(task);
            else
                run_task(task);
            end
        end
    end
end
```

Algorithm 4.1: Engine pseudocode for the C- and *iTasks*-view

To achieve this in the EDSL a *Task* class is added that work in a similar fashion as the `sds` class. This class is listed in Listing 4.8. *Task*s can have an argument and always have to specify a delay or waiting time. The type signature of the `mtask` is complex and therefore an example is given. The aforementioned Listing shows a simple specification containing one *Task* that increments a value indefinitely every one seconds.

```
class mtask v a where
  task :: (((v delay r) a->v MTask Expr)->In (a->v u p) (Main (v t q))) -> Main (v t q) | ...

count = task λcount = (λn.count (lit 1000) (n +. lit 1)) In
```

```
    {main = count (lit 1000) (lit 0)}
```

Listing 4.8: The classes for defining *Task*s

4.6 Example mTask

Some example *mTask*-*Task*s — using almost all of their functionality — are shown in Listing 4.9. The `blink` *mTask* show the classic *Arduino* blinking LED application that blinks a certain LED every second. The `thermostat` expression will enable a digital pin powering a cooling fan when the analog pin representing a temperature sensor is too high. `thermostat`` shows the same expression but now using the assignment style GPIO technique. The `thermostat` example also shows that it is not necessary to run everything as a `task`. The main program code can also just consist of the contents of the root `main` itself. Finally a thermostat example is shown that also displays the temperature on its LCD while regulating the temperature.

```
a0 = aIO A0
d0 = dIO D0

blink = task λblink.(λx.
        IF (x ==. lit True) (ledOn led) (ledOff led) :.
        blink (lit 1000) (Not x)
    In {main=blink (lit 1000) True}

thermostat = {main = digitalWrite D0 (analogRead A0 >. lit 50) }

thermostat` = {main = d0 =. a0 > lit 50 }

thermostat`` = task λth.(λlcd.
        d0 =. a0 > lit 50 :.
        print lcd a0      :.
        th (lit 1000) lim   ) In
    LCD 16 12 [] λlcd.{main = th (lit 1000) lim }
```

Listing 4.9: Some example *mTask*-*Task*s

Chapter 5

System Overview

A system has been researched and built and will be described in the following chapters. This novel system provides a bridge between to gap present in the current system explained in the introduction. It provides a framework to offer functionality for an *iTasks* server to outsource *Task*s to IoT-devices without needing to recompile the code. The *Task*s targeted at IoT devices are compiled at runtime to bytecode which is sent to the device for interpretation.

The following terms will be used throughout the following chapters:

- Device, Client

  These terms are used interchangeably and denote the actual device connected to the system. This can be a real device such as a microcontroller but it can also just be a program on the same machine as the server functioning as a client.

- Server, *iTasks*-System

  This is the actual executable serving the *iTasks* application. The system contains *Task*s taking care of the communication with the clients and infrastructure to manage the clients.

- System

  The system describes the complete ecosystem, containing both the server and the clients including the communication between them.

- Engine

  The runtime system of the client is called the engine. This program handles communicating with the server and runs the interpreter for the *Task*s on the client.

5.1 EDSL for IoT Tasks

Not all *Task*s are suitable to run on an IoT-device and therefore an EDSL is used to offer a constrained language that expresses *Task*s for the new system. The *mTask*-EDSL shown in Chapter 4 provides the language to create imperative programs that are suitable to run on microcontrollers. The EDSL's main view is a C code generator who's code compiles to *Arduino* compatible microcontrollers. The big downside of this approach is the stiffness of the system. Once the code has been generated and the microcontroller has been programmed, nothing can be changed to it anymore. IoT-devices often have a limited amount of write cycles on their program memory available and therefore it is very expensive to keep recompiling and reprogramming the

chips. To solve this problem, a new view is proposed for the *mTask*-EDSL which compiles the expressions not to C-code, but to a bytecode format which is interpetable and thus the need for reprogramming is removed. To achieve this, several classes have been added to the *mTask*-EDSL. Not all of the functionality of the *mTask* language is needed or not implemented.

The functionality and implementation added to the *mTask*-EDSL is shown in Chapter 6.

5.2 System Overview

The existing C-backend for the *mTask*-EDSL generates a self-contained *iTasks*-like TOP system for microcontrollers. The added view for the *mTask*-EDSL does not result in a self-contained system but compiles the expressions to bytecode representing a single *Task*. The device and the server communicate using the Leader/Follower principle[1]. Only the server can initiate a connection with a device and only the server can produce and send *Task*s to the device. Concepts such as SDSs are available on the device and are the main use of communication between the client and the server.

Chapter 7 elaborates on the considerations and implementation of the system.

[1] Also known as Master/Slave

Chapter 6

Extending the *mTask* EDSL

The *Task*s suitable for a client are called *mTask-Task* and are written in the aforementioned *mTask*-EDSL. Some functionality of the original *mTask*-EDSL will not be used in this system. Conversely, some functionality needed was not available in the existing EDSL. Due to the nature of class based shallow embedding this obstacle is easy to solve. A type — housing the EDSL — does not have to implement all the available classes. Moreover, classes can be added at will without interfering with the existing views.

6.1 *Task* Scheduling Strategy

The current *mTask* engine for devices does not support *Task*s in the sense that the C-view does. *Task*s used with the C-view are a main program that executes code and launches *Task*s. It was also possible to just have a main program. The current *mTask*-system only supports main programs which are the *Task*s in their entirety. However, this results in a problem. *Task*s can not call other *Task*s nor themselves. Therefore, execution strategies have been added. Sending a *Task* always goes together with choosing a scheduling strategy. This strategy can be one of the following three strategies:

- `OneShot`

  The `OneShot` strategy consists of executing the *Task* only once. In IoT applications, often the status of a peripheral or system has to be queried only once on the request of the user. For example in a thermostat, the temperature is logged every 30 minutes. However, the user might want to know the temperature at that exact moment, and then they can just send a `OneShot` *Task* probing the temperature. After execution, the *Task* will be removed from the memory of the client.

- `OnInterval Int`

  `OnInterval` is a execution strategy that executes the *Task* in the given number of milliseconds. This strategy is very useful for logging measurements on an interval. Moreover, the strategy can be (ab)used to simulate recursion. SDSs store global information and is persistent. The `retrn` instruction — as will be shown in Section 6.3.4 — can then be used to terminate. Therefore, *Task*s can be crafted that recursively call themselves using a SDS to simulate arguments.

- `OnInterrupt Int`

Finally, a scheduling method is available that executes a *Task* when a given interrupt is received. This method can be useful to launch a *Task* on the press on hardware events such as the press of a button. Unfortunately, due to time constraints and focus, this functionality is only built in the protocol, none of the current client implementations support this.

6.2 SDS Properties

MTask-SDSs on a client are available on the server as in the form of regular *iTasks*-SDSs. However, the same freedom that an SDS has in the *iTasks*-system is not given for SDSs that reside on the client. Not all types are suitable to be located on a client, simply because it needs to be representable on clients and serializable for communication. Moreover, SDSs behave a little different in an *mTask* device compared to in the *iTasks* system. In an *iTasks* system, when the SDS is updated, a broadcast to all watching *Task*s in the system is made to notify them of the update. SDSs can update often and the update might not be the final value it will get. Implementing the same functionality on the *mTask* client would result in a lot of expensive unneeded bandwidth usage. Therefore a device must publish the SDS explicitly to save bandwidth. Note that this means that the SDS value on the device can be different compared to the value of the same SDS on the server.

To add this functionality, the `sds` class could be extended. However, this would result in having to update all existing views that use the `sds` class. Therefore, an extra class is added that contains the extra functionality. Programmers can choose to implement it for existing views in the future but are not obliged to. The publication function has the following signature:

```
class sdspub v where
  pub :: (v t Upd) -> v t Expr | type t
```

Listing 6.1: The `sdspub` class

SDSs in the *mTask*-EDSL are always attached to a `Main` component. Thus, they are not usable in the *Task* domain. To solve this, the SDSs found in the `Main` object are instantiated as real SDS in the server. This poses a problem of naming because SDSs in the *mTask*-EDSL are always anonymous at runtime. There is no way of labeling it since it is not a real entity, it is just a function. When SDSs is instantiated and communicated with the device, they must be retrievable and identifiable. Internally this identification happens through numeric identifiers, but this is handy for programmers since. Therefore, an added class named `namedsds` is added that provides the exact same functionality as the SDS class but adds a `String` parameter that can later be used to identify an SDS in the bag of instantiated SDSs that result from compilation. The types for this class are shown in Listing 6.2. Again, an example is added for illustration. Retrieving the SDS after compilation is shown in Section 7.4.

```
class namedsds v where
    namedsds :: ((v t Upd) -> In (Named t String) (Main (v c s))) -> (Main (v c s)) | ...
:: Named a b = Named infix 1 a b

sdsExample :: Main (v Int Stmt)
sdsExample = sds λx.0 Named "xvalue" In
    {main= x =. x +. lit 42 }
```

Listing 6.2: The `namedsds` class

6.3 Bytecode Compilation View

The *mTask-Task*s are sent to the device in bytecode and are saved in the memory of the device. To compile the EDSL code to bytecode, a view is added to the *mTask*-system encapsulated in the type ByteCode. As shown in Listing 6.3, the ByteCode view is a boxed Reader Writer State Transformer Monad (RWST) that writes bytecode instructions (BC, Subsection 6.3.1) while carrying around a BCState. The state is kept between compilations and is unique to a device. The state contains fresh variable names and a register of SDSs that are used.

Types implementing the *mTask* classes must have two free type variables. Therefore the RWST is wrapped with a constructor and two phantom type variables are added. This means that the programmer has to unbox the ByteCode object to be able to make use of the RWST functionality such as return values. Tailor made access functions are used to achieve this with ease. The fresh variable stream in a compiler using a RWST is often put into the *Reader* part of the monad. However, not all code is compiled immediately and later on the fresh variable stream cannot contain variables that were used before. Therefore this information is put in the state which is kept between compilations.

Not all types are suitable for usage in bytecode compiled programs. Every value used in the bytecode view must fit in the BCValue type which restricts the content. Most notably, the type must be bytecode encodable. A BCValue must be encodable and decodable without losing type or value information. At the moment a simple encoding scheme is used that uses single byte prefixes to detect the type of the value. The devices know these prefixes and can apply the same detection if necessary. Note that BCValue uses existentially quantified type variables and therefore it is not possible to derive class instances such as iTasks. Tailor-made instances for these functions have been made.

```
:: ByteCode a p = BC (RWS () [BC] BCState ())
:: BCValue = ∃e: BCValue e & mTaskType, TC e
:: BCShare =
    { sdsi    :: Int
    , sdsval  :: BCValue
    , sdsname :: String
    }
:: BCState =
    { freshl :: Int
    , freshs :: Int
    , sdss   :: [BCShare]
    }

class toByteCode a :: a -> String
class fromByteCode a :: String -> a
class mTaskType a | toByteCode, fromByteCode, iTask, TC a

instance toByteCode Int, ... , UserLED, BCValue
instance fromByteCode Int, ... , UserLED, BCValue

instance arith ByteCode
...
instance serial ByteCode
```

Listing 6.3: Bytecode view

6.3.1 Instruction Set

The instruction set is given in Listing 6.4. The instruction set is kept large, but the number of instructions stays under 255 to get as much expressive power while keeping all instruction within

one byte.

The interpreter running in the client is a stack machine. The virtual instruction BCLab is added to allow for an easy implementation of jumping. However, this is not a real instruction and the labels are resolved to actual program memory addresses in the final step of compilation to save instructions and avoid label lookups at runtime.

```
:: BC = BCNop
    | BCLab Int          | BCPush BCValue     | BCPop
    //SDS functions
    | BCSdsStore BCShare | BCSdsFetch BCShare | BCSdsPublish BCShare
    //Unary ops
    | BCNot
    //Binary Int ops
    | BCAdd              | BCSub              | BCMul
    | BCDiv
    //Binary Bool ops
    | BCAnd              | BCOr
    //Binary ops
    | BCEq               | BCNeq              | BCLes             | BCGre
    | BCLeq              | BCGeq
    //Conditionals and jumping
    | BCJmp Int          | BCJmpT Int         | BCJmpF Int
    //UserLED
    | BCLedOn            | BCLedOff
    //Pins
    | BCAnalogRead Pin   | BCAnalogWrite Pin  | BCDigitalRead Pin | BCDigitalWrite Pin
    //Return
    | BCReturn
```

Listing 6.4: Bytecode instruction set

All single byte instructions are converted automatically using a generic function which returns the index of the constructor. The index of the constructor is the byte value for all instructions. Added to this single byte value are the encoded parameters of the instruction. The last step of the compilation is transforming the list of bytecode instructions to actual bytes.

6.3.2 Helper functions

Since the ByteCode type is just a boxed RWST, access to the whole range of RWST functions is available. However, to use this, the type must be unboxed. After application the type must be boxed again. To achieve this, several helper functions have been created. They are given in Listing 6.5. The op and op2 functions is hand-crafted to make operators that pop one or two values off the stack respectively. The `tell`` function is a wrapper around the RWST function `tell` that appends the argument to the *Writer* value.

```
op2 :: (ByteCode a p1) (ByteCode a p2) BC -> ByteCode b Expr
op2 (BC x) (BC y) bc = BC (x >>| y >>| tell [bc])

op :: (ByteCode a p) BC -> ByteCode b c
op (BC x) bc = BC (x >>| tell [bc])

tell` :: [BC] -> (ByteCode a p)
tell` x = BC (tell x)

unBC :: (ByteCode a p) -> RWS () [BC] BCState ()
unBC (BC x) = x
```

Listing 6.5: Some helper functions

6.3.3 Arithmetics & Peripherals

Almost all of the code from the simple classes exclusively use helper functions. Listing 6.6 shows some implementations. The boolExpr class and the classes for the peripherals are implemented using the same strategy.

```
instance arith ByteCode where
    lit x = tell` [BCPush (BCValue x)]
    (+.) x y = op2 x y BCAdd
    ...

instance userLed ByteCode where
    ledOn  l = op l BCLedOn
    ledOff l = op l BCLedOff
```

Listing 6.6: Bytecode view implementation for arithmetic and peripheral classes

6.3.4 Control Flow

Implementing the sequence operator is very straightforward in the bytecode view. The function just sequences the two RWSTs. The implementation for the *If* statement speaks for itself in Listing 6.7. First, all the labels are gathered after which they are placed in the correct order in the bytecode sequence. It can happen that multiple labels appear consecutively in the code. This is not a problem since the labels are resolved to real addresses later on anyway.

```
freshlabel = get >>= λst=:{freshl}->put {st & freshl=freshl+1} >>| tell freshl

instance IF ByteCode where
    IF b t e = BCIfStmt b t e
    (?) b t = BCIfStmt b t (tell` [])

BCIfStmt (BC b) (BC t) (BC e) = BC (
    freshlabel >>= λelse->freshlabel >>= λendif->
    b >>| tell [BCJmpF else] >>|
    t >>| tell [BCJmp endif, BCLab else] >>|
    e >>| tell [BCLab endif]
    )

instance noOp ByteCode where
    noOp = BC (pure ())
```

Listing 6.7: Bytecode view for the IF class

The scheduling in the *mTask*-*Task*s bytecode view is different from the scheduling in the C view. *Task*s in the C view can start new *Task*s or even start themselves to continue, while in the bytecode view, *Task*s run indefinitely, one-shot or on interrupt. To allow interval and interrupt *Task*s to terminate, a return instruction is added. This class was not available in the original system and is thus added. It just writes a single instruction so that the interpreter knows to stop execution. Listing 6.8 shows the classes and implementation for the return expression.

```
class retrn v where
  retrn :: v () Expr

instance retrn ByteCode where
    retrn = tell` [BCReturn]
```

Listing 6.8: Bytecode view for the return instruction

6.3.5 Shared Data Sources & Assignment

Fresh SDS are generated using the state and constructing one involves multiple steps. First, a fresh identifier is grabbed from the state. Then a `BCShare` record is created with that identifier. A `BCSdsFetch` instruction is written and the body is generated to finally add the SDS to the actual state with the value obtained from the function. The exact implementation is shown in Listing 6.9. The implementation for the `namedsds` class is exactly the same other than that it stores the given name in the `BCShare` structure as well.

```
freshshare = get >>= λst=:{freshs}->put {st & freshs=freshs+1} >>| pure freshs

instance sds ByteCode where
    sds f = {main = BC (freshshare
            >>= λsdsi->pure {BCShare|sdsname="",sdsi=sdsi,sdsval=BCValue 0}
            >>= λsds->pure (f (tell` [BCSdsFetch sds]))
            >>= λ(v In bdy)->modify (addSDS sds v)
            >>| unBC (unMain bdy))
        }
instance sdspub ByteCode where
    pub (BC x) = BC (censor (λ[BCSdsFetch s]->[BCSdsPublish s]) x)

addSDS sds v s = {s & sdss=[{sds & sdsval=BCValue v}:s.sdss]}
```

Listing 6.9: Bytecode view for `arith`

All assignable types compile to an RWST which writes the specific fetch instruction(s). For example, using an SDS always results in an expression of the form `sds x=4 In ...`. The actual `x` is the RWST that always writes one `BCSdsFetch` instruction with the correctly embedded SDS. Assigning to an analog pin will result in the RWST containing the `BCAnalogRead` instruction. When the operation on the assignable is not a read operation from but an assign operation, the instruction(s) will be rewritten accordingly. This results in a `BCSdsStore` or `BCAnalogWrite` instruction respectively. The implementation for this is given in Listing 6.10.

```
instance assign ByteCode where
    (=.) (BC v) (BC e) = BC (e >>| censor makeStore v)

makeStore [BCSdsFetch i]    = [BCSdsStore i]
makeStore [BCDigitalRead i] = [BCDigitalWrite i]
makeStore [...]             = [...]
```

Listing 6.10: Bytecode view implementation for assignment.

6.3.6 Actual Compilation

All the previous functions are tied together with the `toMessages` function. This function compiles the bytecode and transforms the *Task* to a message. The SDSs that were not already sent to the device are also added as messages to be sent to the device. This functionality is shown in Listing 6.11. The compilation process consists of two steps. First, the RWST is executed. Then, the *Jump* statements that jump to labels are transformed to jump to program memory addresses. The translation of labels to program addresses is straightforward. The function consumes the instructions one by one while incrementing the address counter with the length of the instruction. The generic function `consNum` is used which gives the arity of the constructor. However, when it encounters a `BCLab` instruction, the counter is not increased because the label will not result in an actual instruction. The label is removed and the position of the label is stored in the resulting map. When all labels are removed, the jump instructions are transformed using the `implGotos` function that looks up the correct program address in the map resulting from the aforementioned

function. This step is followed by comparing the old compiler state to the new one to find new instantiated SDSs. The compilation concludes with converting the bytecode and SDSs to actual messages ready to send to the client.

```
bclength :: BC -> Int
bclength (BCPush s) = 1 + size (toByteCode s)
bclength ... = ...
bclength x = 1 + consNum{|*|} x

computeGotos :: [BC] Int -> ([BC], Map Int Int)
computeGotos [] _ = ([], newMap)
computeGotos [BCLab l:xs] i
# (bc, i) = computeGotos xs i
= (bc, put l i)
computeGotos [x:xs] i
# (bc, i) = computeGotos xs (i + bclength x)
= ([x:bc], i)

toRealByteCode :: (ByteCode a b) BCState -> (String, BCState)
toRealByteCode x s
# (s, bc) = runBC x s
# (bc, gtmap) = computeGotos bc 1
= (concat (map (toString o toByteVal) (map (implGotos gtmap) bc)), s)

implGotos map (BCJmp t) = BCJmp $ fromJust   (get t map)
implGotos map (BCJmpT t) = BCJmpT $ fromJust (get t map)
implGotos map (BCJmpF t) = BCJmpF $ fromJust (get t map)
implGotos _ i = i

toMessages :: MTaskInterval (Main (ByteCode a b)) BCState -> ([MTaskMSGSend], BCState)
toMessages interval x oldstate
# (bc, newstate) = toRealByteCode (unMain x) oldstate
# newsdss = difference newstate.sdss oldstate.sdss
= ([MTSds sdsi e\\{sdsi,sdsval=e}<-newsdss] ++ [MTTask interval bc], newstate)
```

Listing 6.11: Actual compilation.

6.4 Examples

As an example for the bytecode compilation the following listing shows the thermostat example given in Listing 4.9 compiled to bytecode. The left column indicates the position in the program memory. The `endif` label is resolved to an address outside of the program space. This is not a problem since this is included in the stopping condition of the interpreter. When the program counter exceeds the length of the program, the task terminates.

```
 0-1 : BCAnalogRead (Analog A0)
 2-5 : BCPush (Int 50)
 6   : BCGre
 7-8 : BCJmpF 17                  //Jump to else
 9-11: BCPush (Bool 1)
12-13: BCDigitalWrite (Digital D0)
14-15: BCJmp 21                   //Jump to endif
16-18: BCPush (Bool 0)            //Else label
19   : BCDigitalWrite (Digital D0)
20   :                            //Endif label
```

Listing 6.12: Thermostat bytecode

The factorial function can be expressed as an *mTask-Task* and uses the `sds` and the `return` functionality. Typically this *Task* is called with the `OnInterval` scheduling strategy and will calculate the factorial after which is will return. The following listings shows the actual *mTask* and the generated messages followed by the actual bytecode in a readable form.

```
factorial :: Int -> Main (ByteCode () Stmt)
factorial i = sds λy=i In
    namedsds λx=1 Named "result" In
    {main =
        IF (y <=. lit 1) (
            pub x :. retrn
        ) (
            x =. x *. y :. y =. y -. lit 1
        )}

//Generating the actual messages with:
Start = fst $ toMessages (OnInterval 500) (factorial 5) zero

//The output will be
//[MTSds 2 (BCValue 5), MTSds 1 (BCValue 1), MTTask (OnInterval 500) ...]
```

Listing 6.13: Factorial as an *mTask-Task*

```
 0-2 : BCSdsFetch 1
 3-6 : BCPush (Int 1)
 7   : BCLeq
 8-9 : BCJmpF 16                  //Jump to else
10-12: BCSdsPublish 2 ("result")
13   : BCReturn
14-15: BCJmp 37                   //Jump to endif
16-18: BCSdsFetch 2 ("result")    //Else label
19-21: BCSdsFetch 1
22   : BCMul
23-25: BCSdsStore 2 ("result")
26-28: BCSdsFetch 1
29-32: BCPush (Int 1)
33   : BCSub
34-36: BCSdsStore 1
37   :                             //Endif label
```

Listing 6.14: The resulting bytecode for the factorial function

Chapter 7

System Considerations & Implementation

The system provides a framework of functions with which an *iTasks*-system can add, change and remove devices at runtime. Moreover, the *iTasks*-system can send *mTask*-*Task*s — compiled at runtime to bytecode by the *mTask*-view — to the device. The device runs an interpreter which executes the *Task*'s bytecode following the provided scheduling strategy. Devices added to the system are stored and get a profile for identification. These profiles are persistent during reboots of the *iTasks*-system to allow for easy reconnecting with old devices. The way of interacting with *mTask*-*Task*s is analogous to interacting with *iTasks*-*Task*s. This means that programmers can access the SDSs made for a device in the same way as regular SDSs and they can execute, combine and transform *mTask*-*Task*s as if they where normal *iTasks*-*Task*s.

7.1 Devices

A device is suitable for the system as a client if it can run the engine. The engine is compiled from one codebase and devices implement (part of) the device specific interface. The shared codebase only uses standard C and no special libraries or tricks are used. Therefore, the code is compilable for almost any device or system. The full interface — excluding the device specific settings — is listed in Appendix B. The interface works in a similar fashion as the EDSL. Devices do not have to implement all functionality, this is analogous to the fact that views do not have to implement all type classes in the EDSL. When the device connects with the server for the first time, the specifications of what is implemented is communicated. Devices must be available throughout sessions and cannot always be kept in scope and therefore they are stored in an SDS.

At the time of writing the following device families are supported and can run the device software. Porting the client software to a new device does not require a lot of work. For example, porting to the MBED device family only took about an hour.

- `POSIX` compatible systems connected via the Transmission Control Protocol (TCP).

  This port only uses functionality from the standard C library and therefore runs on *Linux* and *MacOS*.

- Microcontrollers supported by the MBED[1] environment.

[1] `https://mbed.com`

This is tested in particular on the `STM32f7x` series ARM development board.

- Microcontrollers family supported by `ChibiOS`[2] connected via serial communication.

  This is also tested in particular on the `STM32f7x` series ARM development board.

- Microcontrollers which are programmable in the *Arduino* Integrated Development Environment (IDE) connected via serial communication or via TCP over WiFi or Ethernet.

  This does not only include *Arduino* compatible boards but also other boards capable of running *Arduino* code. A port of the client has been made for the ESP8266 powered NodeMCU that is connected via TCP over WiFi. A port also has been made for the regular *Arduino* Uno board which only boasts a meager *2K* RAM. The stack size and storage available for devices boasting this little RAM has to be smaller than default but are still suitable to hold a hand full of *Task*s.

7.1.1 Client

7.1.1.1 Engine

The client software is responsible for maintaining the communication and executing the *Task*s when scheduled. In practise, this means that the client is in a constant loop, checking communication and executing *Task*s. The pseudocode for this is shown in Algorithm 7.1. The `input_available` function waits for input, but has a timeout set which can be interrupted. The timeout of the function determines the amount of loops per time interval and is a parameter that can be set during compilation for a device.

```
Data: list tasks, time tm
begin
    while true do
        if input_available() then
            receive_data();
        end
        tm ← now();
        foreach t ← tasks do
            if is_interrupt(t) and had_interrupt(t) then
                run_task(t);
            else if tm − t.lastrun > t.interval then
                run_task(t);
                if t.interval == 0 then
                    delete_task(t);
                else
                    t.lastrun ← t;
                end
            end
        end
    end
end
```

Algorithm 7.1: Engine pseudocode

[2] `https://chibios.org`

```
struct task {
    uint16_t tasklength;
    uint16_t interval;
    unsigned long lastrun;
    uint8_t taskid;
    uint8_t *bc;
};

struct task *task_head(void);
struct task *task_next(struct task *t);

struct sds {
    int id;
    int value;
    char type;
};

struct sds *sds_head(void);
struct sds *sds_next(struct sds *s);
```

Listing 7.1: The data type storing the *Task*s

7.1.1.2 Storage

*Task*s and SDSs are stored on the client not in program memory but in memory. Some devices have very little memory and therefore memory space is very expensive and needs to be used optimally. Almost all microcontrollers support heaps nowadays, however, the functions for allocating and freeing the memory on the heap are not very space optimal and often leave holes in the heap if allocations are not freed in a last in first out fashion. To overcome this problem, the client will allocate a big memory segment in the global data block. This block of memory resides under the stack and its size can be set in the interface implementation. This block of memory will be managed in a similar way as the entire memory space of the device is managed. *Task*s will grow from the bottom up and SDSs will grow from the top down.

When a *Task* is received, the program will traverse the memory space from the bottom up, jumping over all *Task*s. A *Task* is stored as the structure followed directly by its bytecode. Therefore it only takes two jumps to determine the size of the *Task*. When the program arrived at the last *Task*, this place is returned and the newly received *Task* can be copied to there. This method is analogously applied for SDSs, however, the SDSs grow from the bottom down.

When a *Task* or SDS is removed, all the remaining objects in the memory space are reordered in such a way that there are no holes left. In practice this means that if the first received *Task* is removed, all *Task*s received later will have to move back. Obviously, this is quite time intensive but it can not be permitted to leave holes in the memory since the memory space is so limited. With this technique, even the smallest tested microcontrollers with only $2K$ RAM can hold several *Task*s and SDSs. Without this technique, the memory space will decrease over time and the client can then not run for very long since holes are evidently created at some point.

The structure instances and helper functions for traversing for *Task*s and SDSs are shown in Listing 7.1.

7.1.1.3 Interpretation

The execution of a *Task* is started by running the `run_task` function and always starts with setting the program counter and stack pointer to zero and the bottom respectively. When finished, the interpreter executes one step at the time while the program counter is smaller than the program

length. This code is listed in Listing 7.2. One execution step is basically a switch statement going over all possible bytecode instructions. The implementation of some instructions is shown in the listing. The `BCPush` instruction is a little more complicated in the real code because some decoding will take place as not all `BCValues` are of the same length and are encoded.

```
#define f16(p) program[pc]*265+program[pc+1]

void run_task(struct task *t){
    uint8_t *program = t->bc;
    int plen = t->tasklength;
    int pc = 0;
    int sp = 0;
    while(pc < plen){
        switch(program[pc++]){
        case BCNOP:
            break;
        case BCPUSH:
            stack[sp++] = pc++ //Simplified
            break;
        case BCPOP:
            sp--;
            break;
        case BCSDSSTORE:
            sds_store(f16(pc), stack[--sp]);
            pc+=2;
            break;
        // ...
        case BCADD:
            stack[sp-2] = stack[sp-2] + stack[sp-1];
            sp -= 1;
            break;
        // ...
        case BCJMPT:
            pc = stack[--sp] ? program[pc]-1 : pc+1;
            break;
        // ...
        }
    }
}
```

Listing 7.2: Rough code outline for interpretation

7.2 iTasks

The server part of the system is written in *iTasks*. Functions for managing are added. This includes functionality for adding, removing and updating devices. Functions for sending *Task*s and SDSs to devices and functionality to remove them. Furthermore, an interactive web application has been created that provides an interactive management console for these manageming tasks. This interface provides functionality to show *Task*s and SDSs and their according status. It also provides the user with a library of example *mTask-Task*s that can be sent interactively to the device.

7.2.1 Device Storage

Everything that a device encompasses is stored in the `MTaskDevice` record type which is in turn stored in an SDS. This includes management for the SDSs and *Task*s stored on the device. The

MTaskDevice definition is shown in Listing 7.3 accompanied with the necessary classes and sub types. Devices added to the system must be reachable asynchronously. This implies that the programmer only needs to keep hold of the reference to the device and not the actual device record.

```
:: Channels :== ([MTaskMSGRecv], [MTaskMSGSend], Bool)
:: MTaskDeviceSpec = ... // Explained in a later section
:: MTaskMSGRecv = ...    // Message format, explained in a later section
:: MTaskMSGSend = ...    // Also explained in a later section
:: MTaskResource
    = TCPDevice TCPSettings
    | SerialDevice TTYSettings
    | ...
:: MTaskDevice =
    { deviceTask     :: Maybe TaskId
    , deviceError    :: Maybe String
    , deviceChannels :: String
    , deviceName     :: String
    , deviceState    :: BCState
    , deviceTasks    :: [MTaskTask]
    , deviceResource :: MTaskResource
    , deviceSpec     :: Maybe MTaskDeviceSpec
    , deviceShares   :: [MTaskShare]
    }

channels :: MTaskDevice -> Shared Channels

class MTaskDuplex a where
    synFun :: a (Shared Channels) -> Task ()
```

Listing 7.3: Device type

The deviceResource component of the record must implement the MTaskDuplex interface that provides a function that launches a *Task* used for synchronizing the channels. The deviceChannels field can be used to get the memory SDS containing the channels. This field does not contain the channels itself because they update often. The field is used to get a memory SDS containing the actual channel data when calling the channels function. The deviceTask stores the *Task*-id for this *Task* when active so that it can be checked upon. This top-level task has the duty to report exceptions and errors as they are thrown by setting the deviceError field. All communication goes via these channels. To send a message to the device, the system just puts it in the channels. Messages sent from the client to the server are also placed in there. In the case of the TCP device type, the *Task* is just a simple wrapper around the existing tcpconnect function in *iTasks*. In case of a device connected by a serial connection, it uses the newly developed serial port library of *Clean*[3]. The implementation and semantics for the MTaskMSGRecv and MTaskMSGSend types are given in Section 7.3.

Besides all the communication information, the record also keeps track of the *Task*s currently on the device, the compiler state (see Section 6.3) and the according SDSs. Finally, it stores the specification of the device that is received when connecting. All of this is given in Listing 7.3. The definitions of the message format are explained in the following section.

7.2.2 SDSs

SDSs on the device can be accessed by both the device and the server. While it would be possible to only store the SDSs on the device, this would require a lot of communication because every read

[3] https://gitlab.science.ru.nl/mlubbers/CleanSerial

operation will then result in sending messages to-and-fro the device. Thus, the *Task* requesting the shared information can just be provided with the synchronized value. As mentioned before, the device has to explicitly publish an update. This has the implication that the server and the client can get out of sync. However, this is by design and well documented. In the current system, an SDS can only reside on a single device.

There are several possible approaches for storing SDSs on the server each with their own level of control. A possible way is to — in the device record — add a list of references to *iTasks*-SDSs that represent the SDS on the device. The problem with this is the fact that an SDS can become an orphan. The SDS is still accessible even when the device is long gone. There is no way of knowing whether the SDS is unreachable because of the device being gone, or the SDS itself is gone on the device. Accessing the SDS happens by calling the `get`, `set` and `upd` functions directory on the actual SDS.

Another approach would be to have reference to an SDS containing a table of SDS values per device. This approach suffers the same orphan problem as before. Accessing a single SDS on the device happens by calling the `get`, `set` and `upd` functions on the actual table SDS with an applied `mapReadWrite`. Using parametric lenses can circumvent the problem of watchers getting notified for other shares that are written. Error handling is better than the previously mentioned approach because an SDS can know whether the SDS has really gone because it will not be available anymore in the table. It still does not know whether the device is still available.

Finally, all devices containing all of their SDSs in a table could be stored in a single big SDS. While the `mapReadWrite` functions require a bit more logic, they can determine the source of the error and act upon it. Also, the parametric lenses must contain more logic. A downside of this approach is that updating a single SDS requires an update of the entire object. In practise, this is not a real issue since almost all information can be reused and SDS residing on devices are often not updated with a very high frequency.

7.2.3 Parametric Lenses

The type for the parametric lens of the SDS containing all devices is `Maybe (MTaskDevice, Int)`. There are several levels of abstraction that have to be introduced. First, the SDS responsible for storing the entire list of devices is called the global SDS. Secondly, an SDS can focus on a single device, such SDSs are called local SDSs. Finally, an SDS can focus on a single SDS on a single device. These SDSs are called share SDSs. Using parametric lenses, the notifications can be directed to only the watchers interested. Moreover, using parametric lenses, the SDS can know whether it is updating a single SDS on a single device and synchronize the value with the actual device. This means that when writing to a share SDS the update is also transformed to messages that are put in the channels of the corresponding device to also notify the device of the update. The SDS is tailor-made and uses an actual standard SDS that writes to a file or memory as the storage. The tailor-made read and write functions are only used to detect whether it is required to send an update to the actual device.

Listing 7.4 shows the implementation of the big SDS. From this SDS all other SDSs are derived. The following paragraphs show how this is achieved for the global SDS local SDS and the share SDS. In the big SDS, reading the value is just a matter of reading the standard SDS that serves as the actual storage of the SDS. The derived shares will filter the output read accordingly. Writing the share requires some extra work because it might be possible that an actual device has to be notified. First, the actual storage of the SDS is written. If the parameter was `Nothing` — the global SDS — the write operation is done. If the parameter was `Just (d, -1)` — a local SDS — nothing has to be done as well. The final case is the special case, when the parameter is `Just (d, i)`, this means that the SDS was focussed on device `d` and SDS `i` and thus it needs to

write to it. First it locates the device in the list, followed by the location of the share to check whether is still exists. Finally the actual update messages are added to the device channels using the `sendMessagesIW` function.

All of the methods share the same `SDSNotifyPred p` which is a function `p -> Bool` and determines for the given `p` whether a notification is required. The predicate function has the `p` of the writer curried in and can determine whether the second argument — the reader — needs to be notified. In practice, the reader only needs to be notified when the parameters are exactly the same.

```
($<) :: a (f a) -> (f b)
($<) a fb = fmap (const a) fb

deviceStore :: RWShared (Maybe (MTaskDevice, Int)) [MTaskDevice] [MTaskDevice]
deviceStore = SDSSource {SDSSource | name="deviceStore", read=realRead, write=realWrite}
where
    realRead :: (Maybe (MTaskDevice,Int)) *IWorld -> (MaybeError TaskException [MTaskDevice], *IWorld)
    realRead p iw = read realDeviceStore iw

    realWrite :: (Maybe (MTaskDevice,Int)) [MTaskDevice] *IWorld -> (MaybeError TaskException (SDSNotifyPred (
     Maybe (MTaskDevice,Int))), *IWorld)
    realWrite mi w iw
    # (merr, iw) = write w realDeviceStore iw
    | isError merr || isNothing mi = (merr $> gEq{|*|} mi, iw)
    # (Just (dev, ident)) = mi
    | ident == -1 = (merr $> gEq{|*|} mi, iw)
    = case find ((==)dev) w of
        Nothing = (Error $ exception "Device lost", iw)
        Just {deviceShares} = case find (λd->d.identifier == ident) deviceShares of
            Nothing = (Error $ exception "Share lost", iw)
            Just s = case sendMessagesIW [MTUpd ident s.MTaskShare.value] dev iw of
                (Error e, iw) = (Error e, iw)
                (Ok _, iw) = (Ok $ gEq{|*|} mi, iw)

    realDeviceStore :: Shared [MTaskDevice]
    realDeviceStore = sharedStore "mTaskDevices" []
```

Listing 7.4: Device SDS

7.2.3.1 Global SDSs

Accessing the global SDS is just a matter of focussing the `deviceStore` to `Nothing`. In this way, *Task*s watching the SDS will only be notified if a device is added or removed. The actual code is as follows:

```
deviceStoreNP :: Shared [MTaskDevice]
deviceStoreNP = sdsFocus Nothing deviceStore
```

Listing 7.5: Global SDS

7.2.3.2 Local SDSs

Accessing a single device can be done using the `deviceShare` function. Since device comparison is shallow, the device that is given is allowed to be an old version. The identification of devices is solely done on the name of the channels and is unique throughout the system. This type of SDS will only be notified if the device itself changed. It will not be notified when only a single SDS on the device changes. The implementation is as follows:

```
deviceShare :: MTaskDevice -> Shared MTaskDevice
deviceShare d = mapReadWriteError
    ( λds->case find ((==)d) ds of
        Nothing = exception "Device lost"
        Just d = Ok d)
    , λw ds->case splitWith ((==)d) ds of
        ([], _) = Error $ exception "Device lost"
        ([_:_], ds) = Ok $ Just [w:ds])
    $ sdsFocus (Just (d, -1)) deviceStore
```

Listing 7.6: Local SDS

7.2.3.3 Local-SDS specific SDSs

A single SDS on a single device can be accessed using the shareShare function. This function focusses the global SDS on a single SDS from a single device. It can use old share references in the same fashion as the local SDS only treating it as references. It uses the mapReadWrite functions to serve the correct part of the information. When a *Task* writes to this SDS, the global SDS will know this through the parameter and propagate the value to the device.

```
shareShare :: MTaskDevice MTaskShare -> Shared BCValue
shareShare dev share = sdsFocus ()
    $ mapReadWriteError (read, write)
    $ sdsFocus (Just (dev, share.identifier))
    $ deviceStore
where
    read :: [MTaskDevice] -> MaybeError TaskException BCValue
    read devs = case find ((==)dev) devs of
        Nothing = exception "Device lost"
        Just d = case find ((==)share) d.deviceShares of
            Nothing = exception "Share lost"
            Just s = Ok s.MTaskShare.value

    write :: BCValue [MTaskDevice] -> MaybeError TaskException (Maybe [MTaskDevice])
    write val devs = case partition ((==)dev) devs of
        ([], _) = Error $ exception "Device doesn't exist anymore"
        ([_,_:_], _) = Error $ exception "Multiple matching devices"
        ([d=:{deviceShares}], devs) = case partition ((==)share) deviceShares of
            ([], _) = Error $ exception "Share doesn't exist anymore"
            ([_,_:_], _) = Error $ exception "Multiple matching shares"
            ([s], shares) = Ok $ Just [{MTaskDevice | d &
                deviceShares=[{MTaskShare | s & value=val}:shares]}:devs]
```

Listing 7.7: Local SDS

7.3 Communication

The communication from the server to the client and vice versa is just a character stream containing encoded *mTask* messages. The synFun belonging to the device is responsible for sending the content in the left channel and putting received messages in the right channel. Moreover, the boolean flag in the channel type should be set to True when the connection is terminated. The specific encoding of the messages is visible in Appendix A. The type holding the messages is shown in Listing 7.8. Detailed explanation about the message types and according actions will be given in the following subsections.

```
:: MTaskId :== Int
:: MSDSId :== Int
```

```
:: MTaskFreeBytes :== Int
:: MTaskMSGRecv
    = MTTaskAck MTaskId MTaskFreeBytes | MTTaskDelAck MTaskId
    | MTSDSAck MSDSId                  | MTSDSDelAck MSDSId
    | MTPub MSDSId BCValue             | MTMessage String
    | MTDevSpec MTaskDeviceSpec        | MTEmpty

:: MTaskMSGSend
    = MTTask MTaskInterval String | MTTaskDel MTaskId
    | MTShutdown                  | MTSds MSDSId BCValue
    | MTUpd MSDSId BCValue        | MTSpec

:: MTaskInterval = OneShot | OnInterval Int | OnInterrupt Int
```

Listing 7.8: Available messages

7.3.1 Device Specification

The server stores a description for every device available in a record type. From the macro settings in the client — in the interface file— a profile is created that describes the specification of the device. When the connection between the server and a client is established, the server will send a request for specification. The client serializes its specification and send it to the server so that the server knows what the client is capable of. The exact specification is shown in Listing 7.9 and stores the peripheral availability, the memory available for storing *Task*s and SDSs and the size of the stack. Not all peripheral flags are shown for brevity.

```
:: MTaskDeviceSpec =
    { haveLed     :: Bool
    , haveLCD     :: Bool
    , have...
    , bytesMemory :: Int
    , stackSize   :: Int
    , aPins       :: Int
    , dPins       :: Int
    }
```

Listing 7.9: Device specification for *mTask*-*Task*s

The code on the device generates the specification. When a device does not have a specific peripheral, the code will also not be on the device. In the interface file, the code for peripherals is always guarded by macros. Thus, if the peripheral is not there, the macro is set accordingly and the code will not be included. To illustrate this, Listings 7.10-7.11 show parts of the interface file and device specification generation function for the *NodeMCU* microcontroller which only boasts a single analog pin and eight digital pins.

```
...
#elif defined ARDUINO_ESP8266_NODEMCU
#define APINS 1
#define DPINS 8
#define STACKSIZE 1024
#define MEMSIZE 1024
#define HAVELED 0
#define HAVEHB 0

#if APINS > 0
void write_apin(uint8_t p, uint8_t v);
uint8_t read_apin(uint8_t pin);
#endif
```

Listing 7.10: Specification in the interface

```
...
void spec_send(void) {
    write_byte('c');
    write_byte(0 | (HAVELED << 0)
                 | (HAVELCD << 1)
                 | (HAVEHB  << 2)
                 | ...);
    write16(MEMSIZE);
    write16(STACKSIZE);
    write_byte(APINS);
    write_byte(DPINS);
    write_byte('\n');
}
```

Listing 7.11: Actual generation

7.3.2 Add a device

A device can be added by filling in the `MTaskDevice` record as much as possible and running the `connectDevice` function. This function grabs and clears the channels, starts the synchronization *Task* (`synFun`), makes sure the errors are handled when needed and runs a processing function in parallel to react on the incoming messages. Moreover, it sends a specification request to the device in question to determine the details of the device and updates the record to contain the top-level *Task*-id. All device functionality heavily depends on the specific `deviceShare` function that generates an SDS for a specific device. This allows giving an old device record to the function and still update the latest instance. Listing 7.12 shows the connection function.

```
process :: MTaskDevice (Shared Channels) -> Task ()
process device ch = forever $ wait "process" (not o isEmpty o fst3) ch
    >>= λ(r,s,ss)->upd (appFst3 (const [])) ch >>| proc r
where
    proc :: [MTaskMSGRecv] -> Task ()
    proc [] = treturn ()
    proc [m:ms] = (case m of
        MTPub i val = updateShareFromPublish device i val @! ()
        ...
        MTDevSpec s = deviceAddSpec device s @! ()
        ) >>| proc ms

connectDevice :: MTaskDevice -> Task MTaskDevice
connectDevice device = set ([], [], False) ch
    >>| appendTopLevelTask 'DM'.newMap True
    (   process device ch -||- catchAll (getSynFun device.deviceData ch) errHdl)
    >>= λtid->upd (λd->{d&deviceTask=Just tid,deviceError=Nothing}) (deviceShare device)
    >>| set (r,[MTSpec],ss) ch
    >>| treturn device
where
    errHdl e = upd (λd->{d & deviceTask=Nothing, deviceError=Just e}) (deviceShare device) @! ()
    ch = channels device
```

Listing 7.12: Connect a device

Figure 7.1 shows the connection diagram. The client responds to the server with their device specification. This is detected by the processing function and the record is updated accordingly.

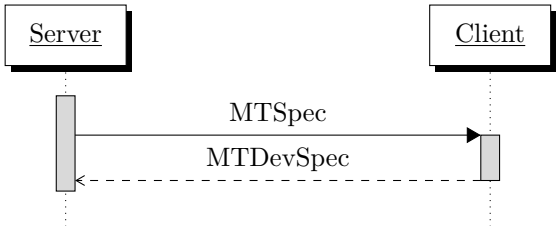

Figure 7.1: Connect a device

7.3.3 *Task*s & SDSs

When a *Task* is sent to the device it is added to the device record without an identifier. The actual identifier is added to the record when the acknowledgement of the *Task* by the device is received. The connection diagram is shown in Figure 7.2.

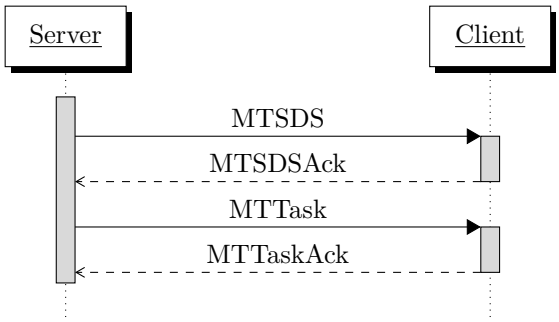

Figure 7.2: Sending a *Task* to a device

The function for sending a *Task* to the device is shown in Listing 7.13. First the *Task* is compiled into messages. The details of the compilation process are given in Section 6.3. The new SDSs that were generated during compilation are merged with the existing device's SDSs. Furthermore the messages are placed in the channel SDS of the device. This will result in sending the actual SDS specification and *Task* specifications to the device. A *Task* record is created with the identifier -1 to denote a *Task* not yet acknowledged. Finally the device itself is updated with the new state and with the new *Task*. After waiting for the acknowledgement the device is updated again and the *Task* returns.

```
makeTask :: String Int -> Task MTaskTask
makeTask name ident = get currentDateTime @ λdt->{MTaskTask | name=name, ident=ident, dateAdded=dt}

makeShare :: String Int BCValue -> MTaskShare
makeShare withTask identifier value = {MTaskShare | withTask=[withTask], identifier=identifier, value=value}

sendTaskToDevice :: String (Main (ByteCode a Stmt)) (MTaskDevice, MTaskInterval) -> Task (MTaskTask, [
     MTaskShare])
sendTaskToDevice wta mTask (device, timeout)
# (msgs, newState=:{sdss}) = toMessages timeout mTask device.deviceState
# shares = [makeShare wta "" sdsi sdsval\\{sdsi,sdsval}<-sdss, (MTSds sdsi` _)<-msgs | sdsi == sdsi`]
= updateShares device ((++) shares)
    >>| sendMessages msgs device
    >>| makeTask wta -1
    >>= λt->upd (addTaskUpState newState t) (deviceShare device)
```

```
    >>| wait "Waiting for task to be acked" (taskAcked t) (deviceShare device)
    >>| treturn (t, shares)
where
    addTaskUpState :: BCState MTaskTask MTaskDevice -> MTaskDevice
    addTaskUpState st task device = {MTaskDevice | device & deviceState=st, deviceTasks=[
     task:device.deviceTasks]}
    taskAcked t d = maybe True (\t->t.ident <> -1) $ find (eq t) d.deviceTasks
    eq t1 t2 = t1.dateAdded == t2.dateAdded && t1.MTaskTask.name == t2.MTaskTask.name
```

Listing 7.13: Sending a *Task* to a device

7.3.4 Miscellaneous Messages

One special type of message is available which is sent to the device only when it needs to reboot. When the server wants to stop the bond with the device it sends the `MTShutdown` message. The device will then clear its memory, thus losing all the SDSs and *Task*s that were stored and reset itself. Shortly after the shutdown message a new server can connect to the device because the device is back in listening mode.

7.3.5 Integration

When the system starts up, the devices from the previous execution still residing in the SDS must be cleaned up. It might be the case that they contain *Task*s, SDSs or errors that are no longer applicable in this run. A user or programmer can later choose to reconnect to some devices.

```
startupDevices :: Task [MTaskDevice]
startupDevices = upd (map reset) deviceStoreNP
    where reset d = {d & deviceTask=Nothing, deviceTasks=[], deviceError=Nothing}
```

Listing 7.14: Starting up the devices

The system's management is done through the interface of a single *Task* called `mTaskManager`. To manage the system, a couple of different functionalities are necessary and are launched. An image of the management interface is shown in Figure 7.3. The left sidebar of the interface shows the list of example *Task*s that are present in the system. When clicking a *Task*, a dialog opens in which a device can be selected to send the *Task* to. The dialog might contain user specified variables. All example *mTask-Task*s are of the type `Task (Main (ByteCode () Stmt))` and can thus ask for user input first if needed for parameterized *mTask-Task*s. The bottom panel shows the device information. In this panel, the devices can be created and modified. Moreover, this panel allows the user to reconnect with a device after a restart of the server application.

7.4 Example

7.4.1 Framework

Systems built with support for *mTask* often follow the same design pattern. First the devices are created — with or without the interaction of the user — and they are then connected. When all devices are registered, the *mTask-Task*s can be sent and *iTasks-Task*s can be started to monitor the output. When everything is finished, the devices are removed and the system is shut down. To illustrate this, a demo blinking application is shown in Listing 7.15. The application is a complete *iTasks* application.

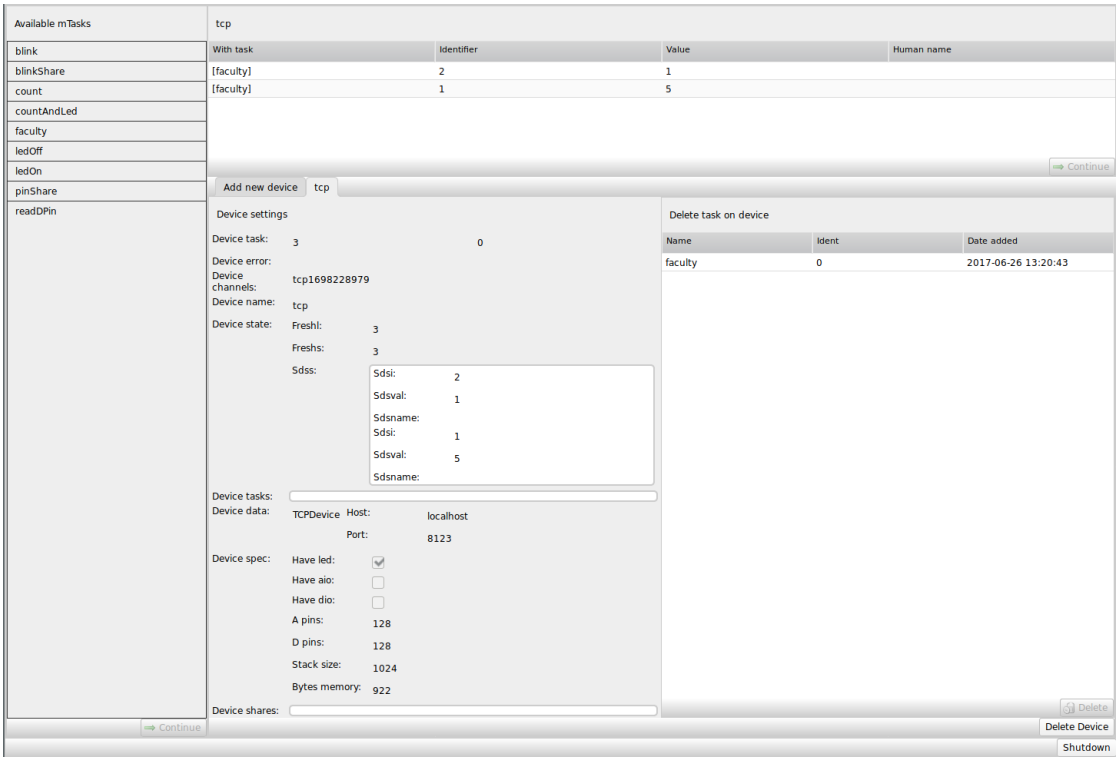

Figure 7.3: The device management interface

```
module blinkdemo

import iTasks
import mTask
import Devices.mTaskDevice

from Data.Func import $

Start world = startEngine blink world

blink :: Task ()
blink =       addDevice
    >>=       connectDevice
    >>= λstm->sendTaskToDevice "blink" blinkTask (stm, OnInterval 1000)
    >>= λ(st, [_,t])->forever (
        updateSharedInformation "Which led to blink" [] (shareShare stm t)
    ) >>* [OnAction (Action "Shutdown") $ always
        $ deleteDevice stm >>| shutDown 0
    ]
where
    blinkTask = sds λled=LED1 In sds λx=True In {main =
        ledOff led1 :. ledOff led2 :. ledOff led3 :.
        IF x (ledOff led) (ledOn led) :.
        x =. Not x}
```

Listing 7.15: *mTask* framework for building applications

7.4.2 Thermostat

The thermostat is a classic example program for showing interactions between peripherals. The following program shows a system containing two devices. The first device — the sensor — contains a temperature sensor that measures the room temperature. The second device — the actor — contains a heater, connected to the digital pin D5. Moreover, this device contains an LED to indicate whether the heater is on. The following code shows an implementation for this. The code makes use of all the aspects of the framework. Note that a little bit of type twiddling is required to fully use the result from the SDS. This approach is still type safe due to the type safety of Dynamics.

```
thermos :: Task ()
thermos =           makeDevice "nodeM" nodeMCU >>= connectDevice
    >>= λnod->      makeDevice "stm32" stm32   >>= connectDevice
    >>= λstm->      sendTaskToDevice "sensing" sensing (nod, OnInterval 1000)
    >>= λ(st, [t])->sendTaskToDevice "acting" acting (stm, OnInterval 1000)
            (λ(BCValue s)->set (BCValue $ dynInt (dynamic s) > 0) (shareShare nod a))
    >>* [OnAction (Action "Shutdown") $ always $ deleteDevice nod >>| deleteDevice stm >>| shutDown 0]
where
    dynInt :: Dynamic -> Int
    dynInt (a :: Int) = a

    sensing = sds λx=0 In {main=
            x =. analogRead A0 :. pub x
        }
    acting = sds λcool=False In {main=
            IF cool (ledOn LED1) (ledOff LED1) :.
            digitalWrite D5 cool
        }
    nodeMCU = makeDevice "NodeMCU"
        (TCPDevice {host="192.168.0.12", port=8888})
    stm32   = makeDevice "Stm32"
        (SerialDevice {devicePath="/dev/ttyUSB0", baudrate=B9600, ...}
```

Listing 7.16: Thermostat example

7.4.3 Lifting *mTask-Task*s to *iTasks-Task*s

If the user does not want to know where and when an *mTask* is actually executed and is just interested in the results, it can lift the *mTask* to an *iTasks-Task*. The function is called with a name, *mTask*, device and interval specification and it will return a *Task* that finishes if and only if the *mTask* has returned.

```
liftmTask :: String (Main (ByteCode () Stmt)) (MTaskDevice, MTaskInterval) -> Task [MTaskShare]
liftmTask wta mTask c=:(dev, _)= sendTaskToDevice wta mTask c
    >>= λ(t, shs)->wait "Waiting for mTask to return" (taskRemoved t) (deviceShare dev)
    >>| viewInformation "Done!" [] ()
    >>| treturn shs
where
    taskRemoved t d = isNothing $ find (λt1->t1.ident==t.ident) d.deviceTasks
```

Listing 7.17: Lifting *mTask-Task*s to *iTasks*

The factorial function example from Chapter 6 can then be lifted to a real *iTasks-Task* with the following code:

```
factorial :: MTaskDevice -> Task BCValue
factorial dev = enterInformation "Factorial of ?" []
    >>= λfac->liftmTask "fact" (fact fac) (dev, OnInterval 100)
```

```
    @         fromJust o find (λx->x.humanName == "result")
    @         λs->s.MTaskShare.value
where
    fact i = sds λy=i
        In namedsds λx=(1 Named "result")
        In {main = IF (y <=. lit 1)
            ( pub x :. return                 )
            ( x =. x *. y :.  y =. y -. lit 1 )}
```

Listing 7.18: Lifting the factorial *Task* to *iTasks*

7.4.4 Heartbeat & Oxygen Saturation Sensor

As an example, the addition of a new sensor will be demonstrated. The heartbeat and oxygen saturation sensor add-on is a PCB the size of a fingernail with a red LED and a light sensor on it. Moreover, it contains an I2C chip to communicate. The company producing the chip provides the programmer with example code for *Arduino* and MBED. The sensor emits red light and measures the intensity of the light returned. The microcontroller hosting the device has to keep track of four seconds of samples to determine the heartbeat. In the *mTask*-system, an abstraction is made. The current implementation runs on MBED supported devices.

7.4.4.1 *mTask* Classes

First, a class has to be devised to store the functionality of the sensor. The heartbeat sensor updates four values continuously, namely the heartbeat, the oxygen saturation and the validity of the two. The real value and the validity are combined in an ADT and functions are added for both of them in the new `hb` class. The values are combined in such a way that they fit in a 16 bit integer with the last bit representing the validity of the reading. The introduced datatype housing the values should implement the `mTaskType` classes. The definition is as follows:

```
:: Heartbeat = HB Int Bool
:: SP02      = SP02 Int Bool

instance toByteCode Heartbeat
    where toByteCode (HB i b) = "h" +++ (to16bit $ (i << 1) bitand (if b 1 0))
instance toByteCode SP02 where ...

instance fromByteCode Heartbeat
    where fromByteCode s = let i = fromByteCode s //The Int from bytecode
        in HB (i >> 1) (i bitand 1 > 0)
instance fromByteCode SP02 where ...

derive class iTask Heartbeat, SP02

class hb v where
    getHb     :: (v Heartbeat Expr)
    getSp02   :: (v SP02 Expr)
```

Listing 7.19: The `hb` class and class implementations

7.4.4.2 Bytecode Implementation

The class is available now, and the implementation can be created. The implementation is trivial since the functionality is limited to retrieving single values and no assignment is possible. The following code shows the implementation. Dedicated bytecode instructions have been added to support the functionality.

```
:: BC
    = BCNop
    | ...
    | BCGetHB
    | BCGetSPO2

instance hb ByteCode where
    getHb = tell` [BCGetHB]
    getSp02 = tell` [BCGetSP02]
```

Listing 7.20: The `hb` bytecode instance

7.4.4.3 Device Interface

The bytecode instructions are added but still the functionality needs to be added to the device interface to be implemented by clients. The following addition to `interface.h` and the interpreter shows the added instructions. When adding a peripheral, the devices not having the peripheral do not need to have their code recompiled. New instructions always get a higher bytecode number if added correctly. The peripheral byte in the device specification by default shows a negative flag for every peripheral. Only the peripherals added will be flagged positive.

```
// interface.h
...
#if HAVEHB == 1
uint16_t get_hb();
uint16_t get_spo2();
#endif
...

// interpret.c
    while(pc < plen){
        switch(program[pc++]){
        ...
#if HAVEHB == 1
        case BCGETHB:
            stack[sp++] = get_hb();
            break;
        case BCGETSPO2:
            stack[sp++] = get_spo2();
            break;
#endif
        ...
```

Listing 7.21: Adding the device interface

7.4.4.4 Client Software

The device client software always executes the `real_setup` in which the client software can setup the connection and peripherals. In the case of the heartbeat peripheral it starts a thread running the calculations. The thread started in the setup will set the global heartbeat and oxygen level variables so that the interface functions for it can access it. This is listed in Listing 7.22. If interrupts were implemented, the *Task*s using the heartbeat sensor could be executed on interrupt. The heartbeat thread can fire an interrupt everytime it calculated a new heartbeat.

```
Serial pc;
Thread thread;

void heartbeat_thread(void) {
```

```
    // Constant heartbeat calculations
}

void real_setup(void) {
    pc.baud(19200);
    thread.start(heartbeat_thread);
}
```

Listing 7.22: Heartbeat code in the client

7.4.4.5 Example

The following code shows a complete example of a *Task* controlling an *STM* microcontroller containing a heartbeat sensor. The web application belonging to the server shows the heartbeat value and starts an alert *Task* when it exceeds the value given or is no longer valid. This example also shows how named SDS are handled.

```
hbwatch :: (Task a) Int -> Task ()
hbwatch alert lim
      =           makeDevice "stm32" stm32
    >>=           connectDevice
    >>= λstm    ->sendTaskToDevice "monitor" monitor (stm, OnInterval 200)
    >>= λ(t, sh)->mon (fromJust $ find (λx->x.name == "hb") sh)
    >>* [OnAction (Action "Shutdown") $ always $ deleteDevice stm >>| shutDown 0]
where
    mon :: (Shared BCValue) -> Task ()
    mon b = whileUnchanged (mapRead dynHB b)
        λhb=:(HB i valid)->if (not valid || i > lim)
            alert (viewInformation "HB Okay" [] hb)

    dynHB :: Dynamic -> HeartBeat
    dynHB (a :: HeartBeat) = a

    monitor = namedsds λhb=(0 Named hb) In
        {main= hb = getHB :.  pub hb }
```

Listing 7.23: Heartbeat example

Chapter 8

Discussion & Conclusion

8.1 Discussion & Future Research

The novel system is functional but still a crude prototype and a proof of concept. The system shows potential but improvements and extensions for the system are amply available in several fields of study.

8.1.1 Simulation

An additional simulation view to the *mTask*-EDSL could be added that works in the same way as the existing C-backed simulation. It simulates the bytecode interpretation. Moreover, it would also be possible to let the simulator function as a real device, thus handling all communication through the existing SDS-based systems. At the moment the *POSIX*-client is the reference client and contains debugging code. Adding a simulation view to the system allows for easy interactive debugging. However, it might not be easy to devise a simulation tool that accurately simulates the *mTask* system on some levels. The execution strategy can be simulated but timing and peripheral input/output are more difficult to simulate properly.

8.1.2 Optimization

Multitasking on the client: True multitasking could be added to the client software. This allows *mTask*-*Task*s to run truly parallel. All *mTask*-*Task*s get slices of execution time and will each have their own interpreter state instead of a single system-wide state which is reset after am *mTask* finishes. This does require separate stacks for each *Task* and therefore increases the system requirements of the client software. However, it could be implemented as a compile-time option and exchanged during the handshake so that the server knows the multithreading capabilities of the client. Multithreading allows *Task*s to be truly interruptible by other *Task*s. Furthermore, this allows for more fine-grained timing control of *Task*s.

Optimizing the interpreter: Due to time constraints and focus, hardly any work has been done in the interpreter. The current interpreter is a no nonsense stack machine. A lot of improvements can be done in this part. For example, precomputed *gotos* can improve jumping to the correct part of the code corresponding to the correct instruction. Moreover, the stack currently consists of 16-bit values. All operations work on 16-bit values and this simplifies the

interpreter implementation. A memory improvement can be made by converting the stack to 8-bit values. This does pose some problems since an equality instruction must work on single-byte booleans *and* two-byte integers. Adding specialized instructions per word size could overcome this problem.

8.1.3 Resources

Resource analysis: Resource analysis during compilation can be useful to determine if an *mTask*-*Task* is suitable for a specific device. If the device does not contain the correct peripherals — such as an LCD — then the *mTask*-*Task* should be rejected and feedback to the user must be given. It might even be possible to do this statically on the type level. The current system does not have any of this built-in. Sending a *Task* that uses the LCD to a device not containing one will result in the device just skipping the LCD related instructions.

Extended resource analysis: The previous idea could be extended to the analysis of stack size and possibly communication bandwidth. With this functionality ever more reliable fail-over systems can be designed. When the system knows precise bounds it can allocate more *Task*s on a device whilst staying within safe memory bounds. The resource allocation can be done at runtime within the backend itself or a general backend can be devised that can calculate the resources needed for a given *mTask*. A specific *mTask* cannot have multiple views at the same time due to the restrictions of class based shallow embedding. It might even be possible to encode the resource allocation in the type system itself using forms of dependant types.

8.1.4 Functionality

Add more combinators: More *Task*-combinators — already existing in the *iTasks*-system — could be added to the *mTask*-system to allow for more fine-grained control flow between *mTask*-*Task*s. In this way the new system follows the TOP paradigm even more and makes programming *mTask*-*Task*s for TOP-programmers more seamless. Some of the combinators require previously mentioned extension such as the parallel combinator. Others might be achieved using simple syntactic transformations.

Launch *Task*s from a *Task*: Currently the C-view allows *Task*s to launch other *Task*s. In the current system this type of logic has to take place on the server side. Adding this functionality to the bytecode-view allows greater flexibility, easier programming and less communication resources. Adding this type of scheduling requires modifications to the client software and extensions to the communication protocol since relations between *Task*s also need to be encoded and communicated. A similar technique as used with SDSs has to be used to overcome the scoping problem.

The SDS functionality in the current system is bare. There is no easy way of reusing an SDS for another *Task* on the same device or on another device. Such functionality can be implemented in a crude way by tying the SDSs together in the *iTasks* environment. However, this will result in a slow updating system. Functionality for reusing shares from a device should be added. This requires rethinking the storage because some typedness is lost when the SDS is stored after compilation. A possibility would be to use runtime typing with `Dynamic`s or the encoding technique currently used for `BCValue`s. Using SDSs for multiple *Task*s within one device is solved when the previous point is implemented.

Another way of improving on SDS handling is to separate SDSs from devices. In this implementation, the SDS not only needs to know on which device it is, but also which internal device

SDS id it has. A pro of this technique is that the SDS can be shared between *Task*s that are not defined in the same scope because they are separated. A con of this implementation is that the mechanisms for implementing SDSs have to be more complex, they have to keep track of the devices containing or sharing an SDS. Moreover, when the SDS is updated, all attached devices must be updated which requires some extra work.

8.1.5 Robustness

Reconnect with lost devices: The robustness of the system can be greatly improved. Devices that lose connection are not well supported in the current system. The device will stop functioning and has to be emptied for a reconnect. *Task*s residing on a device that disconnected should be kept on the server to allow a swift reconnect and restoration of the *Task*s. This holds the same for the client software. The client drops all existing *Task*s on a shutdown request. An extra specialization of the shutdown could be added that drops the connection but keeps the *Task*s in memory. During the downtime the *Task*s can still be executed but publications need to be delayed. If the same server connects to the client the delayed publications can be sent anyways.

Reverse *Task* sending: Furthermore, devices could send their current *Task*s back to the server to synchronize it. This allows interchanging servers without interrupting the client. Allowing the client to send *Task*s to the server is something to handle with care because it can easily cause high bandwidth usage.

8.2 Conclusion

This thesis introduces a novel system for adding IoT functionality to the TOP implementation *iTasks*. A new view for the existing *mTask*-EDSL has been created which compiles the program into bytecode that can be interpreted by a client. Clients have been written for several microcontrollers and consumer architectures which can be connected through various means of communication such as serial port, wifi and wired network communication. The bytecode on the devices is interpreted using a stack machine and provides the programmer with interfaces to the peripherals. The semantics for *mTask* try to resemble the *iTasks* semantics as close as possible.

The host language has a proven efficient compiler and code generator. The compilation is linear in the amount of instructions generated and is therefore also scalable. Moreover, compiling *Task*s is fast because it is nothing more than running some functions native to the host language and there is no intermediate AST.

The dynamic nature of the client allows the microcontroller to be programmed once and used many times. The program memory of microcontrollers often guarantees around 10.000 write or upload cycles and therefore existing techniques such as generating C code are not suitable for dynamic *Task* environments. The dynamic nature also allows the programmer to design fail-over mechanisms. When a device is assigned a *Task* but another device suddenly becomes unusable, the *iTasks* system can reassign a new *mTask*-*Task* to another device that is also suitable for running the *Task* without needing to recompile the code. It also showed that adding peripherals is not a time consuming task and does not even requires recompilation of clients not having the peripheral.

The new functionality extends the reach of *iTasks* by adding IoT functionality and allowing devices to run *mTask*-*Task*s. With this extension, a programmer can create an entire IoT system from one source that reaches all layers of the IoT architecture. However, this does not limit

the applications and makes them static. Components can be updated individually without causing integration problems. Devices can be repurposed just by sending new *Task*s to it. Most importantly, it gives an insight in the possibilities of adding IoT to TOP programs.

Appendix A

Communication Protocol

General Message Format

Messages are delimited by newlines to make processing by line based devices easier. Message exchanges have a *Request* and *Response* header. The *Request* header means that the server is sending to the client. The *Response* header means that the client is sending to the server. In some cases either the *Request* or *Response* is empty. This means that the message is not acknowledged or responded upon. Multibyte values are interpreted as Most Significant Byte (MSB) first integers.

Handshake

Request	
byte	value
1	`'c'`

Response	
byte	value
1	`'c'`
2	Peripheral bitmask
3,4	Bytes of memory
5,6	Size of the stack
7	Number of analog pins
8	Number of digital pins

Table A.1: Send a device specification

*mTask-Task*s

Request	
byte	value
1	`'t'`
2,3	interval or interrupt
4,5	length (n)
6 to n+6	bytecode
Response	
byte	value
1	`'t'`
2,3	*Task* id

(a) Send a *Task*

Request	
byte	meaning
1	`'d'`
2,3	*Task* id
Response	
byte	value
1	`'d'`
2,3	*Task* id

(b) Delete a *Task*

Table A.2: Message protocol for exchanging *Task*s

SDSs

Request	
byte	meaning
1	`'s'`
2,3	id
4,5	value
Response	
byte	meaning
1	`'s'`
2,3	id

(a) Send an SDS specification

Response	
byte	value
1	`'a'`
2,3	SDS id
Request	
byte	value
1	`'a'`
2,3	SDS id

(b) Delete an SDS

Request	
byte	meaning
1	`'u'`
2,3	sdsid
4,5	value
Response	

(c) SDS update

Response
Request
byte value
1 `'p'`
2,3 SDS id
4,5 value

(d) SDS publish

Table A.3: Message protocol for exchanging SDSs

Appendix B

Device Client Interface

```
#ifndef INTERFACE_H
#define INTERFACE_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdbool.h>
#include <stdint.h>
#include <stdarg.h>

#ifdef LINUX
#define APINS 128
#define DPINS 128
#define STACKSIZE 1024
#define MEMSIZE 1024
#define HAVELED 1
#define HAVEHB 1
#elif defined STM
...
#endif

/* Communication */
bool    input_available(void);
uint8_t read_byte(void);
void    write_byte(uint8_t b);

/* Analog and digital pins */
#if DPINS > 0
void write_dpin(uint8_t i, bool b);
bool read_dpin(uint8_t i);
#endif
#if APINS > 0
void    write_apin(uint8_t i, uint8_t a);
uint8_t read_apin(uint8_t i);
#endif

/* UserLED */
#if HAVELED == 1
void led_on(uint8_t i);
void led_off(uint8_t i);
#endif
```

```
#if HAVEHB == 1
uint16_t get_hb();
bool     valid_hb();
uint16_t get_spo2();
bool     valid_spo2();
#endif

/* Delay and communication */
unsigned long getmillis(void);
void msdelay(unsigned long ms);

/* Auxilliary */
void real_setup(void);
void real_debug(char *fmt, ...);
void pdie(char *s);
void die(char *fmt, ...);
void reset(void);

#ifdef __cplusplus
}
#endif
#endif
```

Listing B.1: Full device interface

Bibliography

[1] L. Da Xu, W. He, and S. Li, "Internet of things in industries: a survey," *Industrial Informatics, IEEE Transactions on*, vol. 10, no. 4, pp. 2233–2243, 2014.

[2] S. M. Riazul Islam, Daehan Kwak, M. Humaun Kabir, M. Hossain, and Kyung-Sup Kwak, "The Internet of Things for Health Care: A Comprehensive Survey," *IEEE Access*, vol. 3, pp. 678–708, 2015.

[3] R. Plasmeijer, P. Achten, and P. Koopman, "iTasks: executable specifications of interactive work flow systems for the web," *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 141–152, 2007.

[4] B. Lijnse, *TOP to the rescue: task-oriented programming for incident response applications*. S.l.; Nijmegen: s.n.; UB Nijmegen, 2013. OCLC: 833851220.

[5] A. Oortgiese, *A Distributed Server Architecture for Task Oriented Programming*. Master's thesis, Radboud University, Nijmegen, 2017.

[6] P. Koopman and R. Plasmeijer, "Type-Safe Functions and Tasks in a Shallow Embedded DSL for Microprocessors,"

[7] "Arduino - Open Source Products for Electronic Projects." `http://www.arduino.org/`.

[8] J. M. Jansen, P. Koopman, and R. Plasmeijer, "Efficient Interpretation by Transforming Data Types and Patterns to Functions," *Trends in Functional Programming*, vol. 7, p. 73, 2007.

[9] L. Domoszlai, E. Bruel, and J. M. Jansen, "Implementing a non-strict purely functional language in JavaScript," *Acta Universitatis Sapientiae*, vol. 3, pp. 76–98, 2011.

[10] L. Domoszlai and R. Plasmeijer, "Compiling Haskell to JavaScript through Clean's core," in *Selected papers of 9th Joint Conference on Mathematics and Computer Science (February 2012)*, 2012.

[11] R. Plasmeijer and P. Koopman, "A Shallow Embedded Type Safe Extendable DSL for the Arduino," in *Trends in Functional Programming*, vol. 9547 of *Lecture Notes in Computer Science*, Cham: Springer International Publishing, 2016. DOI: 10.1007/978-3-319-39110-6.

[12] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury, "Guilt free ivory," in *ACM SIGPLAN Notices*, vol. 50, pp. 189–200, ACM, 2015.

[13] L. Pike, P. Hickey, J. Bielman, T. Elliott, T. DuBuisson, and J. Launchbury, "Programming languages for high-assurance autonomous vehicles: extended abstract," pp. 1–2, ACM Press, 2014.

[14] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury, "Building embedded systems with embedded DSLs," pp. 3–9, ACM Press, 2014.

[15] P. Achten, P. Koopman, and R. Plasmeijer, "An Introduction to Task Oriented Programming," in *Central European Functional Programming School*, pp. 187–245, Springer, 2015.

[16] T. H. Brus, M. C. van Eekelen, M. O. Van Leer, and M. J. Plasmeijer, "CleanâĂŤa language for functional graph rewriting," in *Conference on Functional Programming Languages and Computer Architecture*, pp. 364–384, Springer, 1987.

[17] L. Domoszlai, B. Lijnse, and R. Plasmeijer, "Parametric lenses: change notification for bidirectional lenses," in *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, p. 9, ACM, 2014.

[18] J. Cheney and R. Hinze, "First-class phantom types," tech. rep., Cornell University, 2003.

[19] J. Cheney and R. Hinze, "A lightweight implementation of generics and dynamics," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 90–104, ACM, 2002.

[20] J. Svenningsson and E. Axelsson, "Combining deep and shallow embedding for EDSL," in *International Symposium on Trends in Functional Programming*, pp. 21–36, Springer, 2012.

Glossary

Arduino is a cheap and popular microcontroller that is widely used for rapid prototyping.

Clean is a statically typed pure lazy functional programming language based on graph rewriting.

Firmata is standardized protocol for communicating with microcontrollers.

Haskell is a statically typed pure lazy functional programming language.

Ivory is a type-safe EDSL designed to generate C-code for high-assurance low-level systems.

Javascript is an imperative programming language designed to run in web browsers.

LUA is an interpreted scripting language famous for having a very lightweight interpreter that is easy to port..

Python is an interpreted object oriented scripting language. Variants exist that are suitable to run on microcontrollers such as *micropython*.

Task is the basic building block of a TOP system.

iTasks is a TOP implementation written as an EDSL in the *Clean* programming language.

mTask is an abstraction for *Task*s living on IoT devices. Moreover, it is the name of an EDSL.

C++ is low-level imperative and object-oriented programming language suitable for embedded devices based on C.

C is low-level imperative programming language suitable for embedded devices.

SAPL is an intermediate purely functional programming language.

mbed is a programming framework for microcontrollers..

ADT Algebraic Datatype.

API Application Programming Interface.

ARM Acorn RISC Machine.

AST Abstract Syntax Tree.

EDSL Embedded Domain Specific Language.

GADT Generalized Algebraic Data type.

GLONASS Global Navigation Satellite System.

GNSS Global Navigation Satellite System.

GPIO General-Purpose Input/Output.

GPS Global Positioning System.

IDE Integrated Development Environment.

IoT Internet of Things.

JSON JavaScript Object Notation.

LCD Liquid Crystal Display.

LED Lighting Emitting Diode.

MSB Most Significant Byte.

RFID Radio-Frequency Identification.

RWST Reader Writer State Transformer Monad.

SDS Shared Data Source.

TCP Transmission Control Protocol.

TOP Task Oriented Programming.

List of Figures

List of Tables

List of Algorithms

List of Listings