Framework for control and measurement systems

Graduation Report

Pieter de Goeje (110585)

Contents

1 Abstract

A software framework is designed and implemented to aid programmers developing portable applications and software components to control and monitor lab equipment. The goal of this framework is to unify access to hardware parameters and provide services on top of this hardware abstraction layer, such as I/O clustering, transaction caching, polling and user defined parameters.

2 Background

2.1 About Convergence Industry BV

Convergence makes customized measurement and control systems for liquids and gases. Many customers use these systems to research membrane filtration systems for applications like water desalination. Aside from research departments and laboratories, testing equipment developed by Convergence is used by a wide range of manufacturers.

An important aspect of these systems is that they come with full software support. The software allows the user to directly control the various system components, run fully automated, or both, depending on the user requirements.

2.2 Hardware Architecture

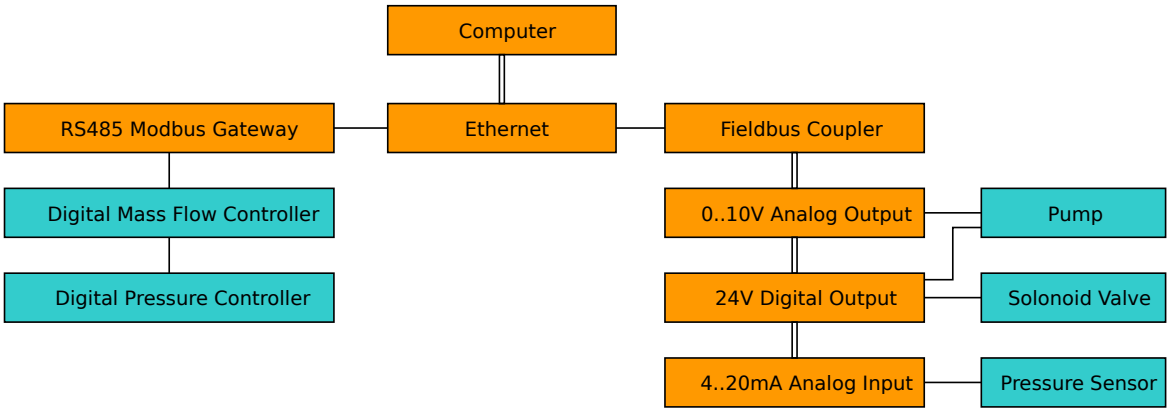

Figure 1: Hardware Architecture

This diagram of a typical system shows a number of commonly used components, and how they're hooked up to the central controller, the computer. The blue components are physical devices that

measure or control the flow of the fluids within the system. Generally, these are the devices that the user wants to control.

Depending on the type of system, the computer can be a low-powered embedded computer or a desktop PC.

Figure 2: Embedded System

The Liquid Entry Pressure (LEP) tester (fig. 2) is an example of lab-scale embedded system. The LEP is a membrane characteristic that is measured by applying an increasing water pressure to a membrane, until the membrane allows some liquid through its pores. This results in a measurable pressure drop at the LEP point.

In this example, a BeagleBone Black controls the display, the pressure sensors, and the pump. Some kind of electrical interface is necessary to translate the 4.20mA output signal from the analog pressure sensor to something the computer understands. This is generally done using an industrial I/O system (Fieldbus Coupler in fig. 1), but in this particular case a specialized PCB attached to the BeagleBone Black converts the analog signal to a digital signal.

In any case, the hardware components can be roughly divided into two groups: digital and analog components. Digital components are typically attached to a bus; Modbus over RS485 or Ethernet is commonly used. Some devices use a peer-to-peer serial connection, like RS232. Analog components' signals are translated to digital using custom electronics or industrial I/O systems. These I/O systems may in turn be attached to a digital bus and become part of a large star shaped network of devices - a device tree, with the computer at its root.

2.3 Software Architecture

Depending on user requirements, either an embedded software package or a desktop software package controls the system.

1. LabView desktop software. This software suite is primarily aimed at users who need full control over their system. It allows the user to specify their own experiments based on a pre-configured set of accessible device parameters. This software was programmed using the LabView visual development environment because of historical reasons. Except for maintenance, no further development of this software will take place.
2. Embedded software. This software was developed due to demand for small, portable systems. These systems tend to have a single, well defined purpose. Both the accessible device parameters and the experiment (if any) are pre-configured and pre-programmed. Generally the user adjusts a couple of parameters, and presses a button to start a fixed sequence of measurements and control commands. Because of the limited computing power available and other platform restrictions, this software was developed in C++.

Due to the differences between the two software packages' underlying platforms and programming languages, almost no logic can be shared between the two. This has resulted into two independently developed software systems, even though these systems control the same hardware architecture.

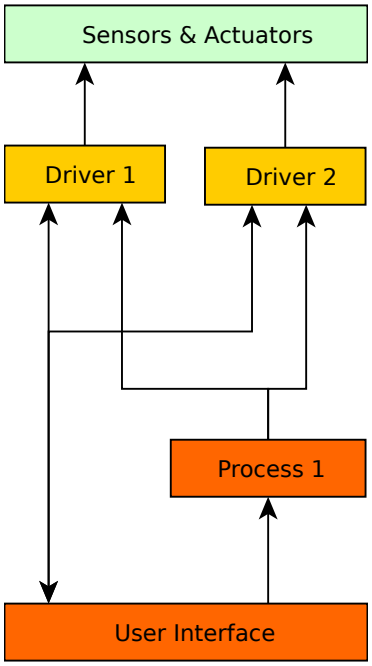

Figure 3: Existing Embedded Software Architecture

A wide variety of devices necessitates a wide variety of software drivers. These drivers were developed in an ad-hoc manner, and have no formally specified common interface as a result. As a consequence, software is written for a specific system only. Drivers are re-used between systems, but the "business logic", shown in fig. 3 as "Process 1", is tightly bound to the set of drivers a particular system uses. The user interface controls these processes and the drivers directly. Changing vendors for a particular component in a system can result in extensive changes to both the user interface as well as dependent processes if the driver for that component has to change as well.

3 Unifying and Improving the Software Stack

The problems with the current software stack are clear as day. Scattered implementations of duplicate functionality lead to wasted time and effort when developing a new system. To increase software reuse, we must make sure that the boundary between system specific business logic and common hardware control become well defined. Reducing the coupling between these two layers increases the portability of both.

Aside from straight efficiency improvements in terms of development time, uncoupled software layers using well defined interfaces allow us to add features that were previously hard to implement. Examples include runtime hardware changes, exploration of hardware parameters by an end user and unified configuration of a system by non-programmers.

3.1 Assignment

To that end, the primary objective of this project is to design a framework that separates the concerns of the drivers with those of the user facing side (front-end); whether that is business logic implemented by a programmer or an end-user that manipulates the system using a HMI.

By designing a hardware abstraction layer, we allow the user to discover the functionality of the underlying hardware dynamically - no preexisting knowledge of the system is necessary just to control the various components.

Business logic still must take into account the physical purpose of each component, but beyond that everything should be self-descriptive - that means that a device object should know which parameters it can operate upon, parameter objects should know their name, their unit and limits if applicable, and this can be queried by both an end-user and any processes running on top of the framework at runtime.

Secondary objectives include the design and implementation of various middleware services on top of the hardware abstraction layer for commonly used functionality.

Once this task is complete, middleware services, processes and user interfaces developed on top of this framework can be reused, which should result in decreased development time and therefore potentially higher quality projects. These projected results are however impossible to quantify without further research after this project has concluded.

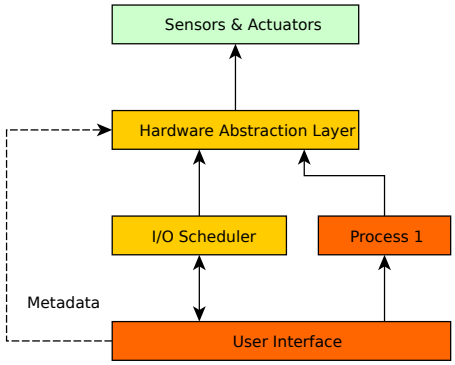

Figure 4: New Software Architecture

Shown in fig. 4 is a high level view of the planned software architecture. Notable is the "I/O Scheduler", an example middleware component that regularly requests updates to various monitored device parameters for use in the user interface. This component can only exist if there's a shared interface to the hardware components, as depicted by the hardware abstraction layer.

3.2 Design Principles

The goal of this framework is primarily to serve as a building block for future projects. The primary user of the framework is the programmer, so focusing on ease of use for the programmer is therefore of the utmost importance. That means making sure that the behavior of the framework is predictable and easy to understand, and maps cleanly to well known programming paradigms in the language of choice: C++ and QML.

Where possible, features of the framework should be "opt-in", that is, if the programmer doesn't want to use something, for instance parameter caching, it shouldn't be forced on them, to reduce the complexity of the resulting software. Ideally, the framework consists of a set of libraries from which the programmer can pick and choose whatever functionality is necessary to accomplish the task of driving a particular piece of equipment.

These design principles are the result of personal experiences working with external libraries. A cur-

sory search on the internet shows[1][2][3] that these guidelines match other programmer's ideals for good library and framework design. The intent is to guide the design when multiple viable solutions to a particular problem are possible. Whether or not the framework is successful in this endeavor, can only be evaluated after the fact.

3.3 Use-Cases

Listed are some use cases that the framework should be able to support. Most of these are based on features found in currently existing systems. Some were collected by looking at customer requirements for their systems. Others were obtained by interviewing assembly and testing engineers.

- Display & manipulation of device parameters in a GUI.
- Device/parameter discovery. The user should be able to browse through the available devices and their parameters. Eventually this feature could be part of a fully user configurable software/hardware platform - a feature that was prototyped previously and found to be very useful by testing engineers working at Convergence.
- Derived parameters, potentially dependent on one or more other "hardware" parameters, and statically configured. An example of why this is useful is shown in section 5.11.1.
- User-defined parameters. This is similar to derived parameters, except that the end user can add these dynamically.
- Sequential programs. These programs manipulate device parameters in a sequential fashion, sometimes with branching. Examples are initialization sequences, timed experiments, and state machines that manage some aspect of a system. This use-case was lifted from existing in-house software that has the same functionality.
- Real-time graphs. Graphs require frequent updates to the monitored parameters, but these updates must not interfere with the performance of critical processes in the system, like control loops.
- Parameter should support metadata, like limits and units, which in turn can be dependent on other parameters. This use-case is the result of experience working with various sensors that allow changing the primary measurement unit at run time.

3.4 Previous work

Two partial prototypes of this framework have been developed previously at Convergence to demonstrate various design aspects with varying levels of success.

[1] https://nordicapis.com/how-to-design-frictionless-apis/
[2] https://www.thereformedprogrammer.net/what-makes-a-good-software-library/
[3] http://www.tldp.org/LDP/LG/issue81/tougher.html

The first prototype was based on the Java platform. This prototype was fairly complete in that it supported a hardware abstraction layer, and experimented with ways to allow a user to define a custom experiment. Some design aspects of this prototype can be found in this framework in the sense that they were improved upon. The Java prototype used generics to support static typing of device parameters. This was found to be extremely cumbersome and error prone by the developers of that framework. Secondly, it supported user defined parameters and programs by implementing an explicitly editable (by the end user) expression tree similar to the way scratch[4] implements visual expression trees. This turned out to be too complex for the average user and not expressive enough for an experienced programmer. Editing that tree required the creation of an internally highly complex GUI, which proved to be difficult to maintain.

The decision was taken to abandon the Java platform in favor of a more portable software stack with an actively developed GUI library, C++/Qt, because multiple embedded projects were already running on that platform.

A second prototype was created using the C++/Qt/QML stack, with reduced scope. This prototype experimented with a custom visual programming language to allow the user to create simple sequential programs. This prototype used a custom interpreter to run expressions and commands. The combination of that interpreter with asynchronous I/O proved to be difficult to maintain. However, the sequential programming interface itself was fairly successful from a usability perspective.

In terms of external frameworks, to the author's knowledge there are none that cover the use-cases listed previously. A quick google search reveals that there exist many driver libraries for communication with sensors that use a particular industrial automation protocol (Modbus, CANbus, etc), and on the other end of the stack there are robotics, computer vision and IoT libraries that implement functionality on presumed available sensor data. But those don't provide the glue that binds the user software (controllers) and graphical user interface to the hardware, and that is what this project is all about.

3.5 Planning

After the preliminary research is completed, work on this framework can start in earnest. The work itself takes place over six two week sprints. Due to the nature of software development, we're only going to present a very coarse planning. The first 3 sprints activities will focus on architectural work. Getting the details (mostly) right is important to reduce the amount of rework later. Much of the architecture discussed later is inspired by previous experience in this domain, as well as experiments done during the preliminary investigation.

[4] A visual programming language. https://scratch.mit.edu/

Date	Activity
10-02-2020	Administration
17-02-2020	Preliminary investigation
09-03-2020	Document plan of approach
10-03-2020	Start sprint 1 - HAL Design
24-03-2020	Start sprint 2 - Polling & I/O Clustering
07-04-2020	Start sprint 3 - User Defined Parameters
21-04-2020	Start sprint 4 - User Defined Parameters
05-05-2020	Start sprint 5 - Caching & Testing
19-05-2020	Start sprint 6 - Documentation
02-06-2020	Finish concept documentation for review
15-06-2020	Complete documentation
29-06-2020 - 03-07-2020	Presentation

3.6 Tools & Languages

The choice for C++ and Qt+QML was made before this project was started. Some of the historical reasons why this software stack was chosen include interoperability with existing software and preexisting experience with these libraries among software developers working at Convergence. Listed are some of the historical reasons C++/Qt+QML was chosen as the implementation language.

- Predictable and ideally low memory use.
- Predictable latency.
- Portable between embedded and desktop.
- Availability of GUI toolkits.
- Availability of developers.

Qt, while traditionally a portable desktop oriented GUI library, has recently (since 5.7) grown support for modern controls using a hardware accelerated renderer, which makes it suitable for use on embedded devices like for instance the Raspberry Pi. These user interfaces are created using QML[5], which is a domain specific language for graphical user interfaces. A GUI created with QML consists of a tree structure of QML components, which can be user controls like buttons and labels, and containers like lists.

[5] https://doc.qt.io/qt-5/qtqml-index.html

Javascript bindings allow the UI to react to changes in the underlying data model. Bindings can exist between C++ components and QML components and reuse Qt's signals and slots mechanism[6]. QML components can be defined in C++ and/or QML, and extended with a mixture of QML and Javascript code.

Qt signals implement a publish/subscribe pattern using a convenient syntax. Each signal maintains a list of slots that are connected to that signal. Signals (senders) and slots (receivers) live on `QObject` derived classes, and if such an object is destroyed, any bindings that involve that object are automatically removed. Signals can be synchronously emitted, in which case the emitter waits for the slots to finish processing (as a regular function call), or they can be scheduled asynchronously on an event loop. In that case, any parameters that are passed from signal to slot are serialized, stored and deserialized automatically.

4 Framework Overview

The framework provides an abstraction layer (section 5.2) for hardware sensors and actuators, so that user software can consume the associated parameters in a unified manner. On top of this, services such as polling (section 5.9), automatic I/O clustering (section 5.10), user defined parameters (section 5.11) and transaction caching (section 5.12) are provided. The framework is meant to integrate well into existing or new embedded Qt/QML applications.

Listed are the core components of the framework. Examples of actual use are shown in section 11.2. Note that in the rest of the documentation the prefix "Fcp" is left out for readability.

Fcp (Class) Helper class to make the components listed below available for use with QML when constructed.

FcpDevice (Interface) A collection of device parameters. Concrete implementations of this interface (device drivers) must implement the `read()` and `write()` methods that take each take parameter to read and write respectively. Example implementation are provided with the demo application source. See section 5.2 for more information on its design.

FcpParameter (Class) A representation of a hardware (or software) parameter. Each parameter holds a single value. This value can be of any type. Values can be written to and read from parameters - the device driver that holds the parameter is responsible for writing that value to the actual hardware.

FcpUserDefinedParameter (Subclass of FcpParameter) A parameter driven by a user defined expression. Read only. Has an expression property that can be set and updated by the user.

FcpAsync (Class) User object passed to all asynchronous methods such as `read()` and `write`. It has a single signal, `finished()`, that is called when the operation completes.

[6] https://doc.qt.io/qt-5/signalsandslots.html

FcpScheduler (Class) Executes a batched read. There are two methods: `poll()`, which takes a list of parameters to add to the set of the parameters to read. `schedule()` which reads all the parameters, previously added with `poll()`. It uses I/O clustering and transaction caching to minimize the number of I/O requests that get send to the hardware. See section 5.9 for more information on its implementation.

FcpPoller (Class) A QML component that registers a parameter with an FcpScheduler, for the lifetime of the FcpPoller object. Parameters registered this way are always read whenever the `schedule()` function is called on FcpScheduler. A possible use of this class is in QML list delegates to automatically poll only the parameters that are visible in that list.

FcpBulkIo (Stateless singleton) Subsystem that allows the user to read multiple parameters at once. Uses I/O clustering optimizations if possible. Has a single method `readList()` that takes a list of parameters tor read. Returns the result a map of the results in the `finished()` signal. FcpDevice implementations must implement the FcpBulkIoTrait if they want to take advantage of I/O clustering. See section 5.10 for more information on its implementation.

FcpBulkIoTrait (Interface) Optional trait that can be implemented by device drivers that want to support clustered I/O. Users of the framework should not need to call this directly (and in fact this trait is not exposed to QML).

FcpUnits (Stateless singleton) Subsystem that acts as the front end API to drivers that implement (dynamic) units for their parameters. Has a single method, `unit()`, that takes a parameter and returns a parameter that contains the unit. See section 5.8 for more information on its implementation.

FcpUnitsTrait (Interface) Optional trait that can be implemented by device drivers that want to support units for their parameters.

FcpLimits (Stateless singleton) Subsystem that acts as the front end API to drivers that implement (dynamic) limits for their parameters. Has two methods, `min()` and `max()`, that take a parameter and return a parameter that contains the minimum and maximum value of the given parameter respectively. See section 5.8 for more information on its implementation.

FcpLimitsTrait (Interface) Optional trait that can be implemented by device drivers that want to support limits for their parameters.

5 Framework Design & Implementation

5.1 Hardware Selection

A list of representative system components was selected to guide the design and implementation of the framework. The list was obtained by reviewing the hardware of existing systems and extracting a set of commonly used components. If multiple components shared characteristics in terms of elec-

trical connection and/or communication protocols, a single representative component was selected. For example, many digital sensors implement a similar communication protocol. This is unsurprising, because these sensors were historically selected to be as similar as possible for ease of integration with the existing software and hardware. The exact components selected are not important - what matters is that the way the software talks to them is sufficiently distinct. The communication direction, digital/analog, and the data types necessary were some of the deciding factors.

After review, we can come to the conclusion that many systems can be characterized as pumping stations with some sensors attached to them, and use varying quantities of the following devices.

List of common devices and some of their parameters.

Device	Parameters
4..20mA Pressure Sensor	Pressure (real), Unit (string), Range min/max (real)
0..10V Pump	Power (real), Pump Enabled (boolean)
24V Solonoid Valve	Valve Enabled (boolean)
Digital Flow Controller	Flow (real), Setpoint (real), Fluid type (real), Unit (string), Range min/max (real)

It can be assumed, based on the reasons outlined above, that if support for these devices can be implemented, then the framework can be extended to support other similar devices.

5.2 Abstracting the hardware

We must define a common interface that can be used by the software to discover and control a heterogeneous system. A complete system consists of a number of hardware devices with various properties, so it makes sense to take the “device” as a starting point for an object in the system. Each device has a number of parameters that can be read/written to. This leads to the following natural design (fig. 5).

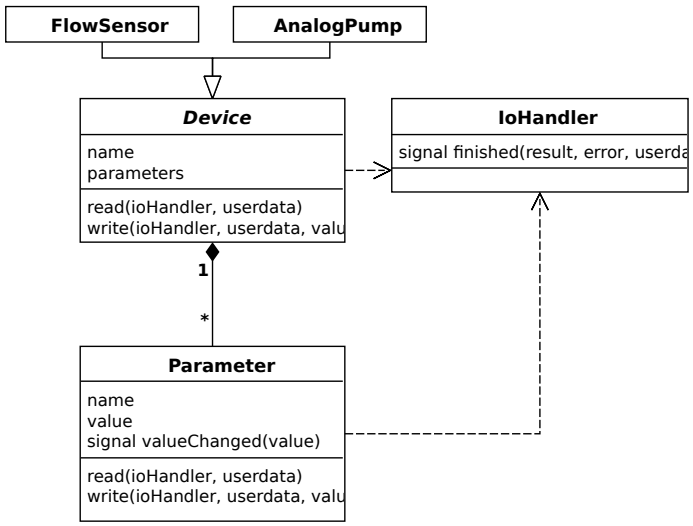

Figure 5: Hardware Abstraction Layer

Of interest in fig. 5 are the `Device` and `Parameter` classes. The concrete `FlowSensor` and `AnalogPump` implementations only serve to demonstrate the role of the `Device` abstract class. Assuming that there's a common set of parameters between devices of the same type, and the upper layers only communicate using the `Device` and `Parameter` abstraction, it should be possible to swap hardware with no impact on software built on this framework. So this design satisfies the portability requirement.

5.3 Getting data in and out of the system

Hardware control requires a special approach to API design because the consumer of the API, the software, is not always in control of the system.

If the hardware needs to communicate with the software, for example because new data is available, it needs a way to notify the software of this fact. Traditionally, this is done by raising an interrupt on the CPU. The CPU detects this condition, and calls a specialized interrupt handler from the operating system, that clears the interrupt and reads the data.

Because the software reacts to a hardware event, we can call this programming interface a *publish-subscribe API*, *reactive API* or *push API*.

On the other hand, some hardware doesn't generate interrupts at all, or the software wants to write or read some data to or from the hardware because of another event in the system. In that case, the software must initiate the communication with the hardware. This is traditional API or *pull API* design.

Pull API's are often combined with polling, to continuously request updates from the hardware in the absence of interrupts.

In both cases, it is possible for data to flow in either direction - from hardware to software and vice-versa. The difference between these API designs is determined solely by the source of the event; API provider (hardware abstraction layer) or API consumer (user software).

5.4 Handling hardware instigated changes to parameters

This diagram shows the flow of information from the device driver, an implementation of `Device`, to the user of the HAL, denoted by "view".

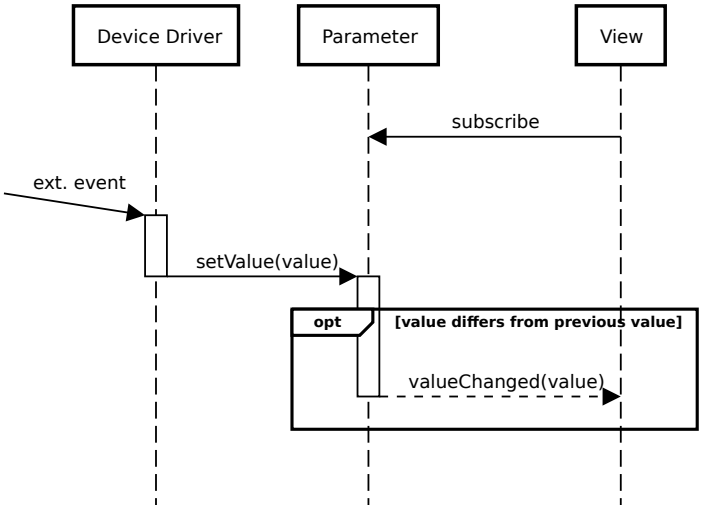

Figure 6: Push control flow

Any external or internal event that causes a change to a parameter's value, should result in setValue() being called on it. This method does two things:

1. It compares and replaces the previously known value with the new value.
2. If it has changed, it emits the `valueChanged()` signal.

If the GUI has a binding to this parameter's value for instance, it can then update the text on the screen.

5.5 Reading from parameters

This diagram shows how the controller of the system can request the latest value of a parameter. An important use of this mechanism is in a finite state machine, which can be part of device initialization

for example. When the asynchronous `read()` completes, it emits a `finished()` signal on the provided handler. Because the user is able to associate the request with its own data, it is easy to build a finite state machine that handles for instance device initialization using a combination of `read()` and `write()` (see section 5.6) calls.

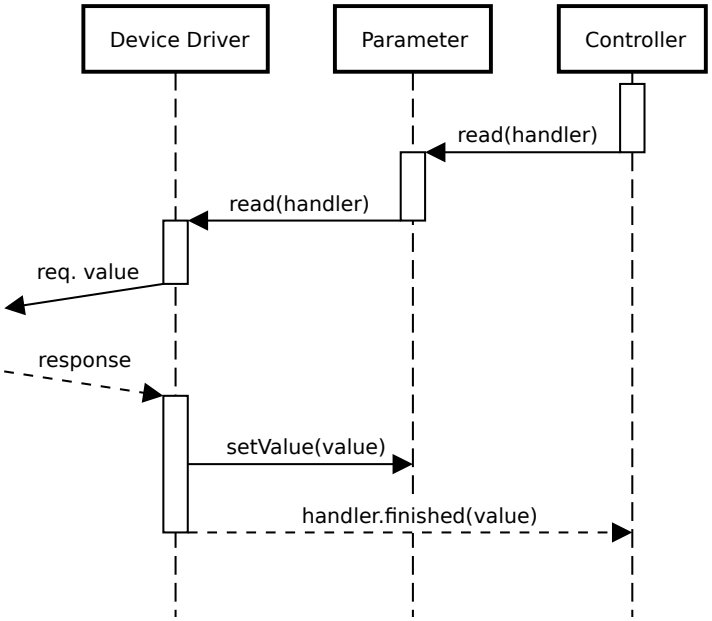

Figure 7: Read control flow

Worth noting is that the `setValue()` call results in the same logic as shown in fig. 6, which causes it to update GUI elements with bindings to this parameter.

5.6 Writing to parameters

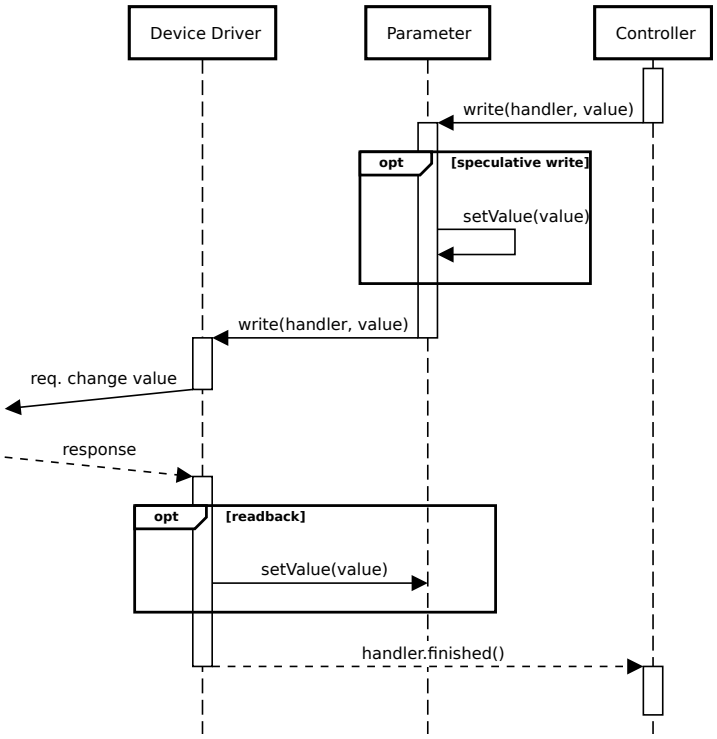

Figure 8: Write control flow

This diagram (fig. 8) shows the path a value takes through the system as the controller requests that a value should be written to a particular parameter. The path is similar to that of a `read()`, except that there are two extra, optional steps.

To ensure a responsive user interface, it is often desirable that a parameter's value gets updated as soon as possible, after a user (controller) requested its change to a new value. By calling `setValue()` immediately after `write()`, the user interface gets updated immediately - but possibly with a value the underlying hardware cannot accept. This could happen for instance if the specified value was out of range for that parameter. After the write completes, the device driver can optionally issue a read to verify that the value was correctly written - and if that is not the case correct the value. Alternatively, if the protocol and device allow the driver to detect conditions like this, the device driver can update the value without doing a `read()`.

Support for speculative updates to parameters is entirely up to the implementation of a particular device driver. Whether or not the device driver must issue a parameter read back depends on the specifics of the communication protocol and the robustness of the device's error handling.

In any case, the contract is that `setValue()` must be called somewhere between `write()` and the

point where the driver emits the `finished()` signal on the given handler.

5.7 Parameter data types

In the previous sections we discussed reading and writing data from and to device parameters. The type of the data was a detail left unspecified. As shown in section 5.1, the data can be of various primitive types. As C++ is (mostly) statically typed, we must extend the API to handle all these cases, and ideally handle more than just those listed, to allow the framework to be extended in the future.

C++ can handle generic data types in multiple ways:

- Function overloading. By duplicating the API for each possible datatype we can ensure type safety while supporting multiple datatypes. Every time a new type is added to the system, a new overload must be implemented.
- With C++ templates. Unfortunately this does not work when combined with virtual inheritance, which is used throughout the system. In addition, each template specialization must be generated and registered individually with the Qt/QML type system to allow for interoperability between the two.
- Using tagged unions[7]. `QVariant`[8] is a tagged union implemented by Qt with support for complex objects as well as primitive types. In essence, this turns C++ into a dynamically typed language.

We chose to use `QVariant` to hold parameter values, mainly because it keeps the API surface as small as possible and because it provides maximum forward compatibility.

By using a `QVariant` we avoid the problem of duplicated APIs and the interoperability problems with QML. The main disadvantage is that the type of a value is now no longer guaranteed to be stable, and that a user of the framework can no longer tell which type a parameter has without directly inspecting its value. A benefit of `QVariant` in particular is that it can hold an "uninitialized" value. This can be used regardless of the underlying datatype, in contrast with regular C++ primitive types, which don't have a representation for "uninitialized".

As an application loads, it is desirable to show the user interface as soon as possible, even if that means that some data has not yet been loaded. Using a QVariant, it is possible for the user interface to distinguish between the initialized and uninitialized cases. This avoids the situation where invalid data is shown to the user during the initialization process.

[7] Stroustrup, B. (2013) *The C++ Programming Language 4th Edition*. Addison-Wesley. pp 217.
[8] https://doc.qt.io/qt-5/qvariant.html

5.8 Parameter metadata

Many parameters have additional associated data, which may in turn be sourced from hardware. Some commonly used ones are units and parameters limits. For example, a certain flow controller's setpoint may have an associated unit, kg/h, a minimum, 0, and a maximum value, 1000.

5.8.1 Naive metadata API

The most straightforward solution is to add this "metadata" to the `Parameter` class itself. The drawback being that every parameter must pay the cost for this additional functionality in terms of memory use and program complexity, even though only a subset of the parameters actually have metadata. Then, when it later turns out that we need to add additional metadata, all implementations of `Parameter` and `Device` must be changed to deal with this fact, causing many changes where none should be required.

5.8.2 API design using the decorator pattern

Another solution could be constructed using the decorator pattern [9]. This pattern facilitates runtime class inheritance by creating a decorator for each desired behavioral change. The decorator holds a reference to the original object, `Parameter` in this case, and modifies the behavior by adding additional methods and/or by intercepting calls to the original object. Decorators can be combined to create any desired combination of behavioral changes; in this instance, there would be a LimitDecorator, a UnitDecorator, and by combining both, you'd have the effective functionality of a LimitUnitParameter without actually having to create such a class.

However, this flexibility comes at a high cost; each decorator must forward method calls to the original object, which means that once this interface is expanded or changed, each decorator needs to be adjusted to match. This would be easy if there was language support for automatic method forwarding, as is the case in some languages that don't use static class inheritance, like python and lua; essentially, these languages allow the user to construct the *vtable* at run time, which incidentally almost completely covers the use case for this pattern. In C++ however, each change to the interface of `Parameter` or a `Decorator` interface, would require the programmer to alter every concrete implementation of that decorator. `Device` implementations can live in different projects. Requiring changes to these implementations to change the interface seems impractical.

Decorators are helpful if they alter each other's behavior - but in this case, there is no (anticipated) overlap in behavior. Each piece of metadata has an independent interface.

[9] Design Patterns: *Elements of Reusable Object-Oriented Software* (1995). Gamma, Erich et al. pp 175.

Additionally, this would result in a complex object hierarchy at run time, especially when multiple decorators are stacked. One consequence of this is that memory management becomes more complex and memory use harder to quantify.

The final nail in the coffin of this pattern is that in practice each parameter doesn't live in its own world, but is part of the device that holds it. Drivers should be able to manipulate whole groups of parameters at once, to maintain efficiency. This means that the actual logic implementing the desired functionality lives in concrete `Device`s, and not in the `Parameter` class.

In general it is desirable to leave the decision of whether or not to implement metadata logic in a subclass of `Parameter` or in the `Device` subclass up to the programmer of that driver. If a programmer chooses to do so, he or she can still use the decorator pattern to layer functionality for a particular device/parameter combination - this should be invisible to users of those devices/parameters and therefore not part of the public API.

5.8.3 Final API design

From an API design perspective, it is extremely important that a user of that API can count on a stable interface, maximizing forward and backward compatibility. To achieve this, the API should be as compact and flexible as possible.

To that end, keeping in mind the previous discussions, the following pattern for adding support for optional metadata was designed:

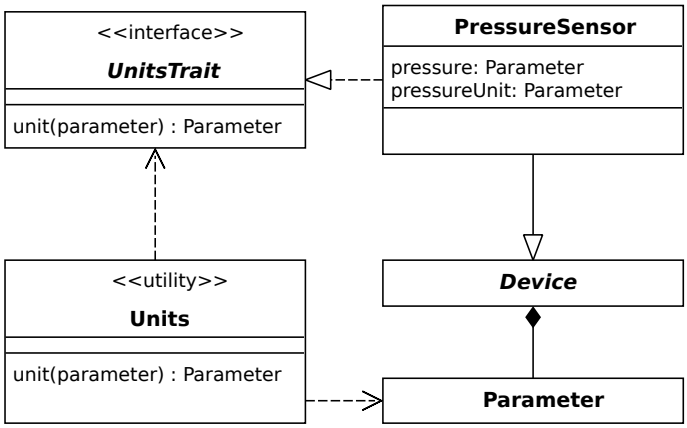

Figure 9: Metadata

Figure 9 shows the relation between the `Units` provider, which is the API entry point for metadata related to Units, and an example device `PressureSensor` which provides unit metadata for some of

its parameters. The `Units.unit()` method returns the unit in `Parameter` form, based on the input parameter. The returned parameter may be a constant value, an actual device parameter, or anything else.

The `Units.unit()` method does no actual work; it only delegates the call to the parameter's `Device` *if* it implements the `UnitsTrait` interface. To do this, we can leverage Qt's RTTI system, resulting in very succinct code:

```
1 Parameter *Units::unit(Parameter *p) {
2   if(UnitsTrait *trait = qobject_cast<UnitsTrait *>(p->device())) {
3     return trait->unit(p);
4   }
5   return &constUndefined;
6 }
```

The `Device` implementation of `UnitsTrait` is then free to use whatever method it sees fit to actually return the correct metadata `Parameter`. Some devices may have internal databases, others can probably get away with a simple switch statement.

If, in the future, it turns out that the design must change and that a `Parameter` must keep track of its own unit, then we can change the implementation of `Units::unit()` without affecting users of the framework, because the public API remains exactly the same.

5.9 Polling

In many cases, the software needs to communicate to the device layer that it is interested in changes to a number of parameters. We've seen in previous chapters that the hardware is capable of notifying the observers of changes to parameters, but how do the hardware drivers know which parameters are interesting for the software? The obvious solution of monitoring all parameters doesn't scale as it will quickly overwhelm the communication buses as the number of devices and parameters increases. Instead, it must be possible to somehow communicate to the HAL that we're interested in changes only to the currently *visible* parameters. Visible parameters are those parameters that directly or indirectly affect the GUI and/or controlling processes.

To solve this problem we need a mechanism that manages the following tasks:

- Keeps track of visible parameters.
- Allow the user to request updates to each unique parameter in the visible parameter set with a single call.
- Group parameters and issue batch requests if possible for efficiency.

To maintain a set of visible parameters, we can use reference counting. Each observer increases the reference count by one. Once an observer is no longer interested, it decreases the reference count by

one. Then, if the count is zero, the parameter is eliminated from the visible set.

This method has one particular disadvantage in practice. It is easy to forget to “unsubscribe” from a particular parameter, which causes invisible parameters to be polled anyway, eventually leading to the problem that we tried to prevent in the first place - an overloaded bus. To that end, it is important to ensure that each subscription is matched by an unsubscription. We can do that by preventing direct access to the reference count. Instead we rely on a `Poller` object, which lifetime is bound to the scope and thus the lifetime of the observer. On construction of this object we increase the reference count to the supplied parameter, and on destruction we decrease the reference count.

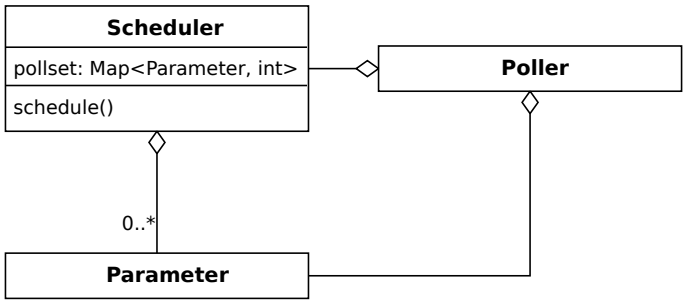

Figure 10: Polling architecture

Another approach might be to have the `Parameter` itself ask the HAL for updates if there are any observers of it. Essentially, every `Parameter` would then be responsible for its own updates. The `Scheduler` class in the above diagram would no longer be necessary as we can move the observer count to the `Parameter` class. This simpler design has an important drawback: it is no longer possible to request the set of visible parameters efficiently. We need access to the entire visible set at once to efficiently implement read clustering, the topic of the next chapter.

5.10 Read clustering

One limitation of per-parameter read/writes is that there is limited opportunity for the underlying driver to combine multiple I/Os into a single I/O operation. This is undesirable because many transport protocols incur a hefty per-I/O penalty. Half-duplex communication protocols for instance must wait for the hardware to acknowledge each packet, and on slow physical links like RS485 this can result in very poor bus utilization. In practice, a packet round-trip time of up to 100ms can be expected.

The following optimizations, dependent on the underlying transport fabric and protocols, are possible if we can cluster multiple related I/O operations. These optimizations are similar to how filesystems

use I/O clustering[10] to optimize throughput.

- Coalescing reads/writes of adjacent memory locations. This optimization works for some Modbus devices that map each parameter to a specific location in device memory. Modbus allows the user to request an entire range of memory at once, potentially covering multiple parameters.
- Clustering multiple unrelated I/O operation into a single operation for protocols that support this.

In both cases the objective is to improve throughput by reducing the per-I/O overhead. I/O in a control system is generally heavily skewed towards read operations, so it makes sense to try to optimize that path first. The previously defined polling architecture is in an ideal position to take advantage of these optimizations, because it has access to an entire list of parameters at once that must be read very frequently. Instead of asking each individual parameter to update its value, it can group parameters by device, and then issue a single read request for all parameters on each device. Once the request arrives at the hardware driver, it can then update all parameters at once if possible.

Because not every `Device` implementation might want to pay the additional complexity cost, the clustered I/O API must be able to deal with `Device`s that don't implement that API. To solve that we add a new optional trait to `Device`, `BulkIoTrait`. The `Scheduler` can then decide at run time if a `Device` supports this interface, and if it does, take advantage of it.

5.10.1 Modbus read clustering

Some Modbus devices expose their parameters as a continuous (virtual) memory space. A modbus I/O request consists of an address and a length parameter. If multiple requested parameters are adjacent in memory, a single I/O request can be used to retrieve an entire cluster of parameters in one go. This doesn't work for all devices, and should probably only apply to memory ranges that are explicitly white listed inside the driver.

Presented here is a possible strategy for optimizing access to these devices.

1. Sort the requested parameters by address. This results in a number of clusters with holes in between.
2. Find any adjacent clusters within a small distance, according to some device dependent heuristic. Merge these clusters.
3. Split clusters based on the maximum allowable I/O size[11]. If a split happens on an inserted memory range from step 2, discard that memory from the start and end of the two resulting clusters.

[10] *The Design and Implementation of the FreeBSD Operating System* (2014). McKusick et al. pp 515. Detailing the design of the Fast File System.

[11] A Modbus/TCP read command can retrieve up to 123 (2-byte) registers. Many parameters require 4 bytes of memory to support the float32 and int32 datatypes, spanning two registers.

4. Trim each cluster that ends with useless memory inserted in step 2.
5. Issue I/Os for each cluster.
6. Map the incoming data back onto the parameters using the address as a key.

5.11 User Defined Parameters

In many cases, raw sensor data is of limited use. The data needs to be processed before it can be presented to the user in a meaningful way.

5.11.1 Use-Case: Transmembrane Pressure

An example of processed data is the transmembrane pressure (TMP), which describes the pressure difference between two sides of a membrane.

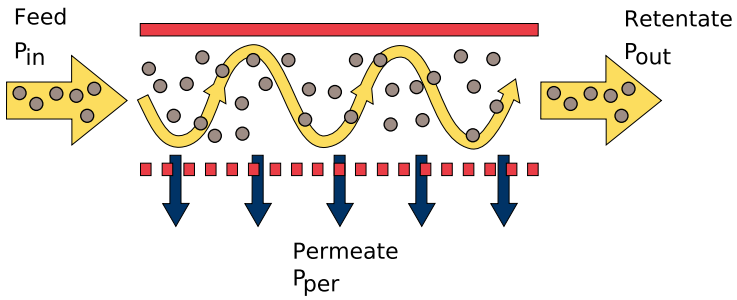

Figure 11: Example: Cross-flow filtration

In this setup (fig. 11), a pump forces a fluid (feed), like water, to flow under pressure over a membrane (the dotted line in the diagram above). Some fluid permeates through the membrane, increasing the concentration of the particulate contained within the feed fluid until it leaves the membrane cell as retentate. This particular setup is extremely common because it increases the longevity of the membrane. The shear forces of the fluid on the membrane prevent particulate from building up on the membrane surface. One of the most important applications of this principle is water desalination using reverse osmosis.

The pressure of the fluid is continuously monitored at three distinct points, shown as P_{in}, P_{out} and P_{per} in the diagram.

To characterize the membrane, an important measurement is the average pressure drop over the membrane: the transmembrane pressure or TMP. We can calculate this as follows:

$$\mathrm{TMP} = \frac{P_{in} + P_{out}}{2} - P_{per}$$

In this use-case, the system must query three parameters from three different sensors before it is able to calculate the TMP.

5.11.2 Requirements

The requirements for user defined parameters are similar to those for regular parameters, with some additions.

- Allow access to existing parameters via user defined algebraic expressions. This means that there must be some way to lookup parameters from a pool of identifiers. Parameters should probably have a name to facilitate this.
- Evaluation should remain asynchronous - hardware queries must not ever block evaluation.
- Persist across runs of the framework. User defined expressions should be able to be stored and reloaded.
- Enable dynamic construction/destruction. These parameters can be created as the result of a user action.
- Track dependent parameters. The system must "know" which parameters a user defined parameter depends on for the purpose of automatic updates and polling.
- Lazy (delayed) updates. This is not a hard requirement, but it would be nice to delay the update to a value of a user defined parameter until after *all* its dependent parameters have been updated. Not doing so can result in flickering in the user interface due to unnecessary updates to GUI bindings.

5.11.3 Design

Evaluation of an expression is going to involve some kind of parser/compiler to shape it into a form that can be evaluated by the system. Ideally it is possible to reuse an existing compiler for this task. Historic prototypes of this particular feature (in Java) involved the use of a manually constructed expression parser, which generated an abstract syntax tree (AST) of the expression. This AST was cached and repeatedly evaluated, using synchronous hardware I/O. At the time, tracking the dependent parameters through this AST proved to be difficult, as parameters could come and go at any time. Aside from the problems inherent to synchronous execution (blocking GUI), this implementation was fairly usable.

We can use the lessons learned from that prototype to guide our design. In particular, some pitfalls like synchronous execution can be avoided if they're part of the design from the beginning.

A generic asynchronous expression evaluator can't be implemented as a recursive function over the

AST in C++, because the evaluator must be suspended whenever a hardware parameter is encountered, for the duration of the I/O, in a non-blocking way.

Some languages allow this through the use of specialized constructs like async/await, co-routines, fibers and continuations. C++ supports none of these natively - at least until C++ 20 is finalized[12]. In essence, these constructs record the current state of the stack, CPU registers, and then suspend execution until an external event restores that state. There exist libraries[13] for C++ that emulate this functionality using platform specific implementations that directly manipulate the CPU registers, or they use the `setjmp()` / `longjmp()` API[14], which is non-portable. Due to this, use of these libraries should be very carefully considered.

Alternatively, if the evaluator is implemented as a virtual machine that executes bytecode generated from the AST, it is possible to suspend the VM whenever a hardware I/O is going to be executed. The VM explicitly keeps track of the call stack, so there's no need for platform specific hacks to store it. In essence, this would emulate fibers/co-routines for a custom language on top of C++.

A prototype was developed to test this, and this turned out to work well. However, it came at a very high cost: code complexity due to the need for a custom interpreter made maintenance and development difficult.

However, we can avoid going through all that trouble by realizing that **a key property of user defined expressions is that they cannot have observable side effects**. In other words, user defined expressions cannot have observable state that depends on the order and/or number of executions. Observable in this case means observable to the user of the system; not to the framework internally.

This follows from the fact that a user cannot reliably use side effects in expressions of user defined parameters, because they're executed an unknown number of times and in a hard to predict order by the QML evaluation engine. This is because bindings to these parameters can be created and destroyed dynamically at runtime. In addition, the order of evaluation is hard to predict in practice.

Instead of initiating I/O *during* evaluation of an expression, we can use another strategy: evaluate an expression to find out which hardware parameters are required, based on the currently known values of these parameters. Then, once the set of parameters is collected, query all parameters asynchronously. Then, run the evaluator again using the updated values for these parameters. If the set of parameters hasn't changed between these evaluations, in other words the evaluation of the expression used up-to-date values for all its parameters, then we have the correct result.

In pseudo-code:

```
1 fresh_set ← { }
```

[12] co_await and friends are defined in Coroutines (C++20)

[13] The most complete library for this purpose is probably boost.context.

[14] See the longjmp(3), setjmp(3) manual pages. Notably, these are not guaranteed to work across function boundaries by the standard, but specific implementations do, so they can be used to implement coroutines.

```
loop {
  {result, dependencies} ← evaluate(expression)
  if(dependencies equal_or_subset_of fresh_set) {
    return result
  }
  await async_read(dependencies - fresh_set)
  fresh_set ← fresh_set union dependencies
}
```

The `await` in this pseudo-code is easily implemented in C++ using a state machine. All that needs to be kept track of during state transitions, is the `fresh_set` set. The complexity of `async_read` itself is hidden by the read clustering subsystem we designed earlier.

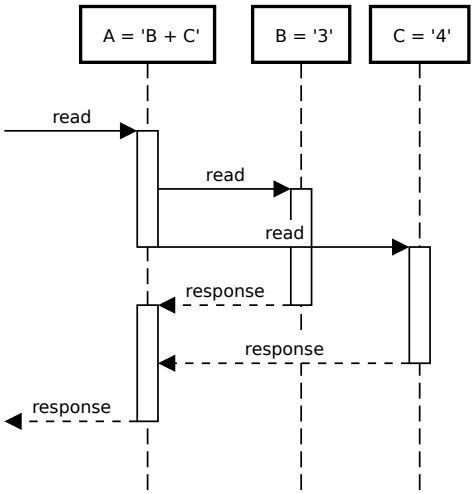

Figure 12: Parallel evaluation of subexpressions

Figure 12 shows an example of execution of a user defined parameter that depends on other parameters. Once the set of parameters is collected, reads are executed in parallel using the clustered I/O subsystem, detailed in section 5.10.

5.11.4 Handling automatic updates

Having established the dependencies, the parameter can then observe its requisite parameter's values for updates. In principle, once a change to any requisite parameter's value is detected, the expression must be reevaluated. As a consequence, if this evaluation results in a new value, each parameter that is dependent on that value is then updated as well.

This leads to a technically minor but very noticeable problem. If multiple requisite parameters are updated in rapid succession, but not quite at the same time due to for instance hardware delays, the expression will be evaluated multiple times, which is undesirable for two reasons.

1. The expression is evaluated multiple times but only the last result is actually used. Similarly, the dependencies are updated multiple times for no apparent benefit.
2. Because the value of the user defined parameter can change for each update to its requisite parameters, this can result costly updates to the GUI if this parameter is directly bound to a visible element. In addition, rapidly changing numbers can be hard to read.

We can avoid this problem by limiting the update frequency. Instead of immediately reevaluating the expression for each update, we schedule the evaluation to run at a later time. If a requisite parameter's value changes before the timer is finished, we can safely ignore that change. The disadvantage of this approach is that updates are delayed by a small amount of time. In testing, a 100ms delay was found to be reasonable compromise for user visible parameters.

It should be noted that control processes are unaffected by this delay. Controllers issue a `read()` call on the input parameters, which pulls the latest value from the hardware, completely ignoring the automatically updated and potentially cached value.

5.11.5 Polling

In theory, a user (in all likelihood the GUI) of the framework can avoid polling a user defined parameter if all its requisite parameters are already polled, and rely on automatic updates. Explicitly polling a user defined parameter is counter productive in this case, as it can lead to duplicate reads from the underlying hardware.

However, a user shouldn't have to handle user defined parameters differently form regular hardware parameters to be efficient. There are multiple potential solutions to this problem.

1. The polling subsystem's `Scheduler` could know about the set of requisite parameters for each user defined parameter. It can then ensure that each hardware parameter is only read once. This implies a tight coupling between the two subsystems, which we would like to avoid to keep everything self contained.
2. Cache the result of each read during the polling cycle. Multiple reads from the same parameter in the same cycle return the same result. To implement this, `read()` must grow support for returning a cached value. This cache can exist for the duration of the poll cycle, or it can be part of a generic caching mechanism that could expire entries based on their lifetime. The latter approach has the advantage that we can also eliminate duplicate reads between different polling processes, at the cost of increased complexity and loss of deterministic hardware access patterns. Caching is further explored in section 5.12.

5.11.6 Handling recursion

Given the following user defined parameters, the system will quickly run out of memory trying to evaluate either paramA or B.

```
1 paramA = paramB + 1
2 paramB = paramA
```

Because each expression is a function with no input, such a loop will never terminate. Although situations like this are unlikely to be intentional, they can arise by accident while editing multiple user defined parameters.

The framework must detect and abort evaluations of expressions with infinite recursion before they crash the system.

Recursion in read() calls Because the set of dependencies for each parameter is dynamically adjusted based on the current values of the required parameters, the framework cannot reliably detect loops until the parameters are actually evaluated.

To that end, the expression evaluator must keep track of a call stack[15]. In conventional programming languages, the call stack is allocated in a linear fashion; there's only one function running at any time (the top of the stack). User defined parameters are slightly different, in the sense that all dependencies of a parameter are evaluated concurrently (fig. 12), resulting in multiple "child" functions that run at the same time. This results in a call tree.

Fortunately, we don't need to use an explicit tree representation, as each user defined parameter only needs to check its own "lineage" - a simple list of parent parameters - in the call tree for another instance of itself to detect a loop. Because this process only walks backwards towards the root of the tree, each node (frame) in the tree can be represented by a parameter and a pointer to its parent. This results in a number of singly linked lists that happen to share a prefix list.

Recursion in automatic (push) updates This scenario happens when one of the parameters involved in an infinite recursion scenario is reevaluated because one of its required parameters was updated.

```
1 paramA = paramB + flow1.measurement
2 paramB = paramA
```

In this example, whenever flow1.measurement changes, paramA is reevaluated. This causes paramB to be reevaluated in turn, which causes paramA to be reevaluated, ad infinitum, until the value of

[15] https://en.wikipedia.org/wiki/Call_stack

either paramA or paramB stabilizes. Once a value stabilizes, no further update events are generated, and the loop stops. Note that in the above example the loop never terminates if flow1.measurement is non-zero[16].

This recursion does not result in infinite memory allocation, for two reasons.

1. The stack is unwound between each evaluation, because the update signal isn't handled until the next iteration of the global event loop.
2. The evaluation of a user defined parameter during an automatic update does not recurse into each required parameter - instead it uses their latest known values.

It is possible to detect this situation by passing the "call stack" along the event - a history of each event that resulted in the generation of the update event. However, since failing to do so does not harm program operation, and the user is not expected to create such a situation intentionally, this is currently not implemented.

5.11.7 Discussion of the expression parser

The `evaluate()` function itself uses the existing QML javascript interpreter to evaluate the expression. The QML javascript interpreter was primarily chosen because it is convenient to use when the project already uses Qt/QML. The integration with QML comes with a number of benefits; it allows user defined parameters that are defined in QML code to access QML properties defined in that same code (not necessarily device parameters), and making user defined parameters available to the interpreter is as easy as adding them to a shared QML context.

One disadvantage of this approach is that it is hard to collect the set of parameters that are referenced inside the expression during execution. The currently implemented approach uses an overloaded `valueOf()` function on each parameter. That method is called by the interpreter to convert an object (parameter) to a number. It returns the latest value of the parameter and adds the parameter to the set of dependencies of the currently executed expression. This uses global state (thread local), which is undesirable.

Future research is necessary to find out if there's a more elegant solution available and/or if it is necessary to revisit the choice of interpreter.

[16] The loop in this example will eventually terminate when paramA = paramA + c is true, which is the case for very large paramA and very small c if calculated using IEEE-754 floating point math.

5.12 Transaction caching

Caching is often a poor band aid to fix performance problems, because it can easily lead to bugs caused by stale cache entries, cache stampedes[17], where an evicted entry is recreated by multiple concurrent users at the same time, as well as bugs resulting from the increased total complexity of the system.

For this project, we don't really need a generic caching solution. All we want is to eliminate duplicate reads from the same parameter within a single poll cycle.

It follows then that a cache for such a transaction only needs to live for the duration of that transaction. This completely insulates the user from errors resulting from incorrectly set cache expiration times. In fact, the user can be completely unaware that there is a cache - it is purely an implementation detail.

As an added benefit, by eliminating duplicate reads within a single transaction, a parameter's value is only determined once, and is therefore the same for every user of that value in that transaction. This helps the GUI maintain consistency between parameters that depend on the same value.

The other problems associated with caching don't magically disappear however. The implementation must take care of the cache stampede problem, which means that it must track not only cached parameters, but also parameters that are in the process of being put into the cache. Users that `read()` from those parameters must wait until a `read()` initiated by the first reader completes before continuing.

Fortunately, this additional logic can be shared by all `read()` implementations that want to support caching.

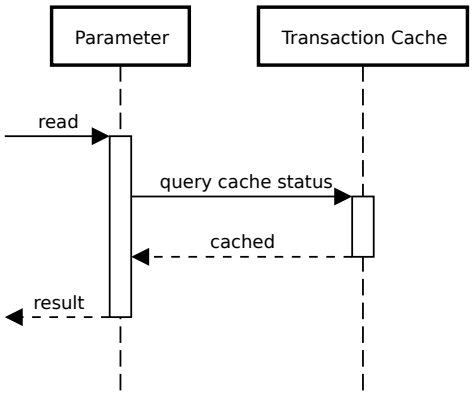

Figure 13: Cache hit

[17] In a naive caching system, if a cached object expires or is evicted, and multiple concurrent readers try to fetch that object, they all find out that at approximately the same time that the cache is stale. If there's no further cooperation between the readers, each reader will then attempt to recreate the requested object from scratch and insert that into the cache, only to be immediately be overwritten by the next reader. This can lead to the system overloading because of the (wasted) duplicate effort, if recreating the object is expensive compared to fetching that object from cache.

Figure 13 shows the program flow if the parameter was already cached inside this transaction. All that is necessary is a quick check with the transaction cache. Because cached entries never go stale the implementation of this is trivial. The “Transaction Cache” object does not store the actual data - the data is stored in the parameter objects themselves, as the last known value. The cache only keeps track of which parameter is up-to-date.

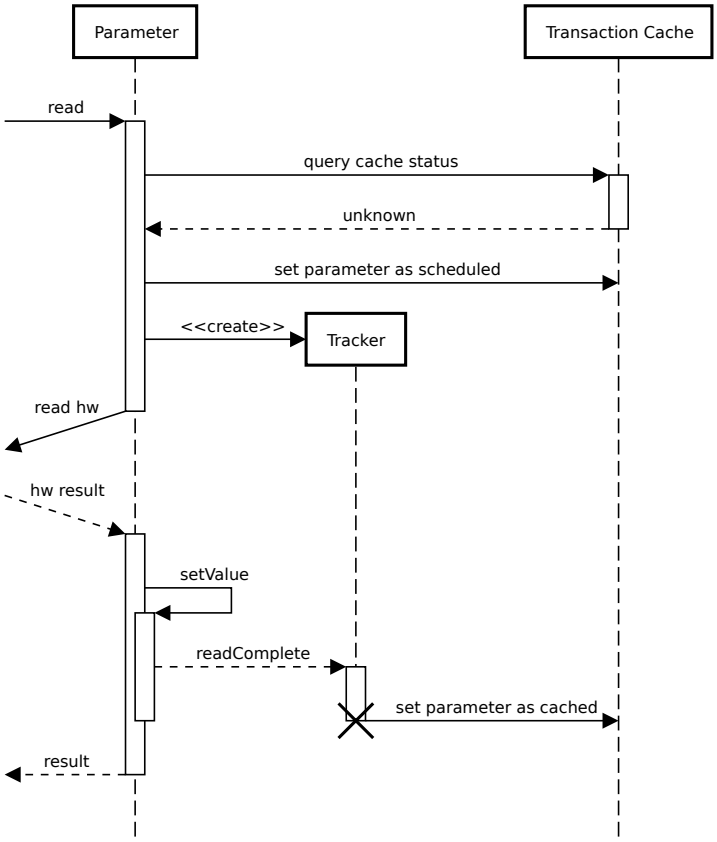

Figure 14: Cache miss

However, a cache miss as shown in fig. 14 is much more complex. The framework must track the outstanding I/O so that it can insert the parameter into the cache when the read completes. The tracker object shown here is implemented as a lambda function that is executed once the `readComplete` signal fires. The original `read()` implementation has no idea that this happens - all it needs to do is call `setValue()` on itself, which always raises the `readComplete` signal. Unlike the `valueChanged` signal described earlier, this signal is raised even if the value is the same, to ensure that the parameter is marked as fresh.

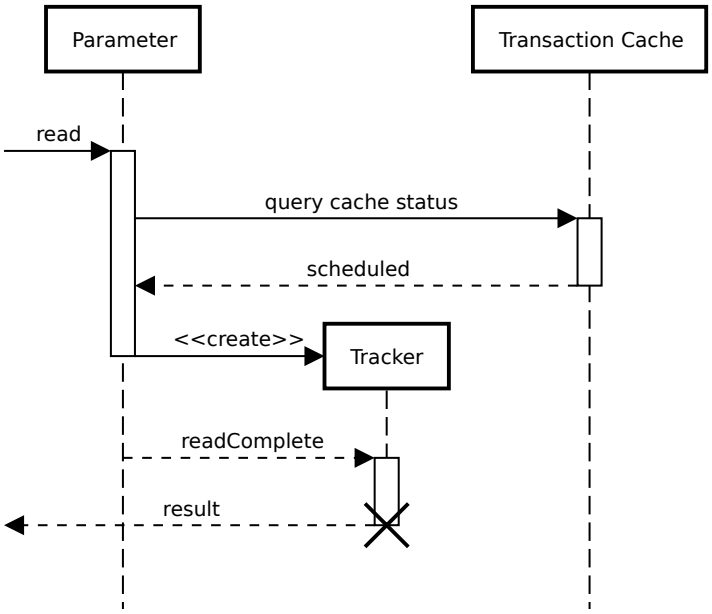

Figure 15: Cache hit, but incomplete

Finally, to solve the cache stampede problem, if another reader tries to access the cache (fig. 15) while the parameter is in the progress of being read (fig. 14), it must wait until that parameter is inserted into the cache. The tracker gets created with the context (parameter, handler) necessary to signal the handler that was passed to the initial `read()` call. When the cache is filled, it can then return the result by emitting the `handler.finished()` signal with the latest value of the tracked parameter.

6 Testing

6.1 Testing API design

Learning a new framework can be a difficult process, and it is therefore of the utmost importance that use of this particular framework is as frictionless as possible. To that end, a demo application based on this framework was developed to verify the framework's usability. This application simultaneously acts as a system test, a learning tool, and a way to "eat our own dog food" - testing API design by actual use.

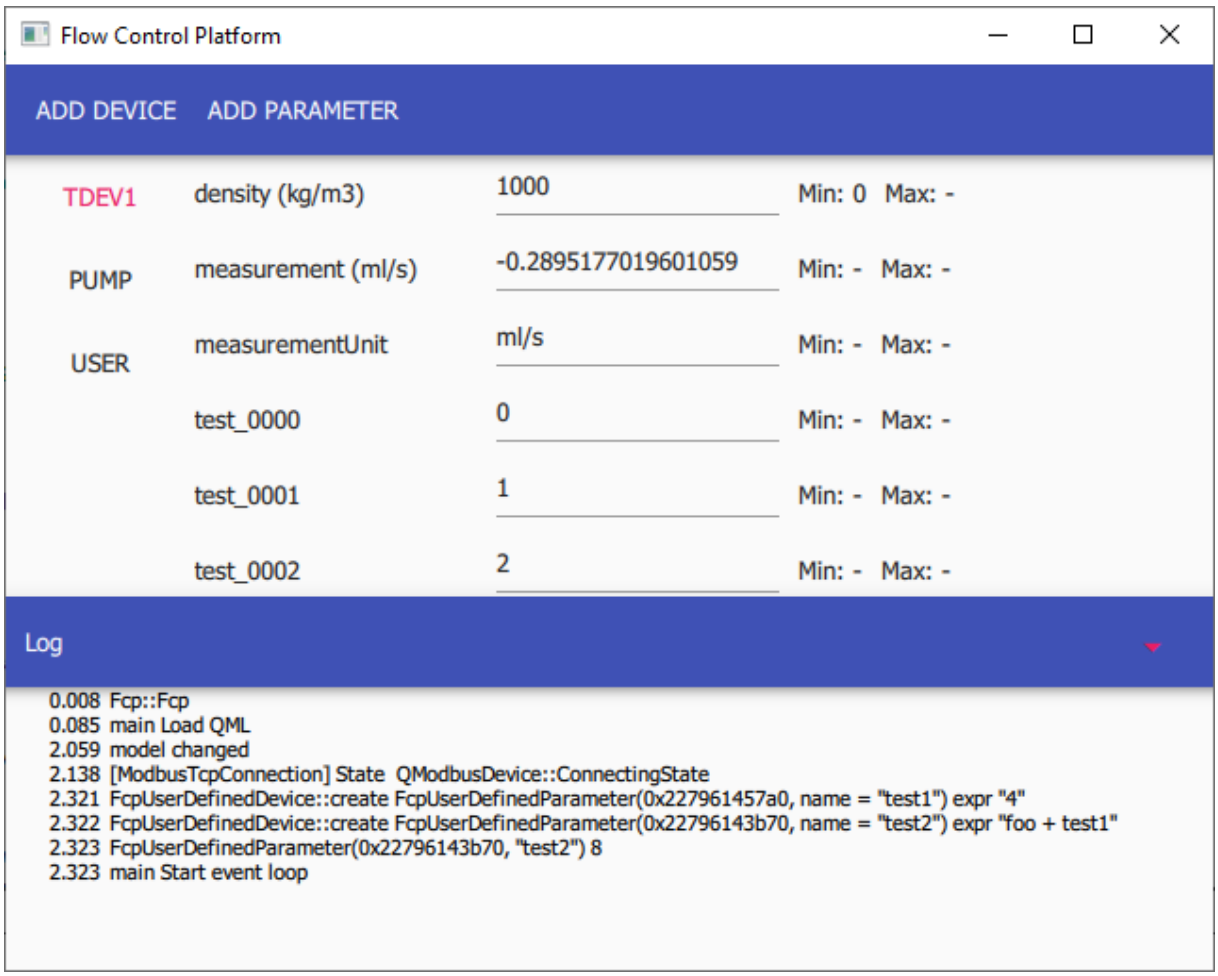

Figure 16: Demo Application

fig. 16 shows the main screen of the demo application. On the left side a list of devices is displayed, and on the right side the list of parameters for the selected device is displayed. Parameters can be edited in place to adjust their values. The “User” device is not a real device but holds all the user defined parameters. All parameter values are updated in real time.

This application allows the user to:

- Inspect parameters from devices.
- Change any value.
- Show limits and units, and alter those if they’re sourced from a hardware parameter.
- Add user defined parameters.
- Test the automatic polling support.

An application like this acts as a smoke test; the real test comes when the framework is used in pro-

duction.

6.2 Unit testing

During development, the framework was tested against simulated hardware, coarsely modeled on existing hardware drivers. This allowed functional testing, and some performance testing of the framework itself.

Complex parts of the framework, like the user defined parameters, are subjected to unit tests to ensure that they remain functional during development. Basic performance tests of user defined parameters were also performed, mostly to ensure that the performance of these features is reasonable. Some examples:

Expression	Wall Time
param1[18]	< 1 µs
udp0 = 52	2 µs
udp1 = param1 + 123	14 µs
udp2 = param1 + udp1[19]	17 µs

As can be clearly seen from this table, read performance of user defined parameters is an order of magnitude worse than reading from a hardware parameter directly. Some possible reasons:

- At least two evaluations are necessary to gather the set of required parameters.
- The javascript interpreter has to cross the javascript/C++ boundary once for each referenced parameter. These calls can't be inlined or optimized and incur datatype conversion overhead.
- In the current implementation, each evaluation issues a clustered I/O request, even if only one parameter is involved for simplicity. This results in unnecessary overhead in these simple cases, because the clustered I/O subsystem is optimized for parallel I/O throughput, not for CPU usage.

7 Project Evaluation

The intended planning initially set out for this project was mostly followed. The planned bi-weekly evaluations by the project supervisor turned to monthly evaluations, and testing against real hardware was postponed due to circumstances beyond our control (COVID-19 pandemic). Instead, more

[18] A simulated hardware parameter, directly read for comparison purposes.
[19] This test uses caching to read param1 only once.

time was spent on the services provided on top of the framework. In addition, more time was spent on documenting design than was initially estimated; about 50% of the total effective time, after project startup, was spent on this aspect.

During development, each biweekly sprint was started by gathering tasks that were scheduled for that sprint, based on the use-cases. Standup meetings were held every morning to discuss the project's progress. The initial three sprints focused on development and task velocity was high, completing most tasks that were planned for those sprints. Later on as the focus shifted more towards documentation, development slowed down and documentation tasks regularly spilled over to the next sprint. The lack of proper time estimates for documentation tasks can be attributed to the lack of experience writing technical documentation for extended amounts of time. In addition, writing for a poorly defined audience, consisting of individuals of wildly different experience levels and backgrounds, even though they're all in "IT", turned out to be harder than expected.

Documentation is particularly important in this case, because ease of use and increased understanding of this framework increases the productivity of the user; a direct goal of this project. In the author's opinion, more effort should have been spent towards documenting the actual use of the framework. Unfortunately, due to the scope of this project, this has not yet come to fruition and remains future work. Creating end-user documentation while the design of the framework was still in flux meant that this documentation was pushed backwards towards the end of the project.

The design presented in this document was strongly guided by previous work done on the same subject within Convergence; this has helped give direction to what would otherwise be a fairly abstract work. As this project was a solo effort and Convergence lacks experienced developers in this area, input from other people concerning implementation details or design decisions was unfortunately limited, but not unexpected.

8 Conclusions

The most important contribution of this framework is the way it unifies access to sensor data and control registers. Some facets of this framework like I/O clustering and transaction caching will immediately benefit users by providing improved performance over a naive implementation. Others, like optional automatic polling of visible parameters will greatly ease development of interactive user interfaces.

As currently implemented, the framework is a suitable building block for applications and software components that need to be portable between devices and platforms, as long as those platforms run Qt 5. This is demonstrated by the demo application.

It is the author's hope that with additional services built on top of this platform, applications for a

wide variety of equipment can be rapidly developed. This framework is just one part of the equation, but an important one, as it has a significant impact on the way an application interacts with the outside world. The APIs as documented here are the culmination of multiple prototype efforts, so there is some confidence that the framework presented here is suitable for purpose. At the very least, by strictly defining the boundary between implementation (driver details) and API user, the framework allows rapid iteration on its implementation without breaking dependent software.

9 Future Work

Ultimately, this framework is a building block required for a fully automated and configurable experiment editor and runner. Using such software, an end user could implement an automated experiment based on simple commands like “set setpoint to 5 kg/h”, “wait until measurement > 4 kg/h”. When combined with automatic data logging and report generation, this allows a user to conduct a wide variety of experiments in an automated fashion without the need for specialized software.

In the near future however, the following ideas are worth exploring to increase the framework’s usability.

- Various existing drivers need to be ported to the new framework. This mainly consists of updating their APIs to confirm to the HAL as described in this document.
- Persistent configuration of a hardware platform. One way of doing this would be to leverage the QML engine to load a device tree expressed as a QML file. At the very least this would allow the user to edit the hardware configuration outside of the application. If the framework grows support for generating QML files, then it can store the current device tree as such a file.
- User defined parameters should support user defined units. It isn’t exactly clear how the user is supposed to set these. Support could be added to the existing Units subsystem to bind a parameter as a unit to another parameter, or the user defined parameter API could gain a call to set these directly. Either way, this needs to be investigated.
- Hardware often requires a specific initialization sequence. Currently these are implemented with explicitly coded finite state machines. If we can wrap the read/write calls in Promises, these FSMs can be implemented as Promise chains and later using `async`/`await` when the javascript engine grows support for these keywords. This would greatly enhance the readability of these state machines.
- Experiment with multi-dimensional data types, such as light spectra. This should already work, but the interaction with user defined parameters should be investigated and best practices documented.
- Support for time dependent properties of a parameter. For instance, a function like `stable(flow1, 10)` could return true if a certain flow has been stable for 10 seconds.

- Devices implemented in this framework are (potentially) visual QML components. It should be possible to position them in a *flowsheet*, with appropriate graphics. This allows a user to easily map the visual representation to the physical reality of the equipment the software is controlling.

10 Acknowledgments

I am extremely grateful to the following people for helping me through the process of writing this thesis, without whom this would have never happened. Chielant de Wit, for his friendly reminder that I still needed to finish my degree every year, and sponsoring this project. Paul Goolkate, who's gently guidance led me safely through the administrative work required to let me start this project, and his yearly prodding to actually get me started. Willem Prakken, for putting up with my nonsense and be my project supervisor. And finally, Felix Broens for supervising this project and giving me almost total freedom to shape this project the way I envisioned.

11 Appendix

11.1 Version History

Version	Date	Changelog
1	2020-05-30	Initial concept
2	2020-06-14	Final version

11.2 Usage Examples

These examples show the intended use of the framework inside a QML application.

11.2.1 Bind a Parameter to a GUI element

```
PhSensor { id: phSensor }
Label { text: phSensor.measurement.value }
Label { text: Units.unit(phSensor.measurement).value }
```

Note that the `.value` suffix is necessary to bind to the actual value, not the `Parameter` object. `Unit.units()` also returns a `Parameter`. This code assumes that measurement has a unit.

11.2.2 Polling a parameter

Setup a shared polling scheduler:

```
FcpScheduler { id: sched }
Timer {
    interval: 1000
    running: true
    repeat: true
    onTriggered: sched.schedule()
}
```

The FcpPoller object registers the parameter with the scheduler, ensuring that `phSensor.measurement` is only updated as long as the label is in scope (visible in the user interface).

```
Label {
    text: phSensor.measurement.value
    FcpPoller {
        scheduler: sched
        parameter: phSensor.measurement
    }
}
```

The code can easily be abstracted with a new QML item that replaces the `Label` to make this less verbose.

11.2.3 Using an FSM to control sequencing

The `FcpAsync::onFinished()` signal has three parameters: `result`, `userData` and `error`. This example uses `userData` to track the state, making the implementation reentrant.

```
FcpAsync {
  id: fsm
  onFinished: {
    switch(userData) {
    case 'start':
      // setpoint = 5
      flow1.setpoint.write(5, fsm, 'read')
      break
    case 'read':
      flow1.measurement.read(fsm, 'log_data')
      break
    case 'log_data':
      console.log('Measurement:', result)
      break
    }
  }
}
```

```
18
19  Button {
20    text: 'Start!'
21    clicked: fsm.finished(undefined, 'start')
22  }
```

11.3 Glossary

Fieldbus Modular industrial I/O system. Bridges the gap between analog hardware and a digital computer.

Modbus Protocol for industrial automation. Often used in combination with RS485.

RS232 A full-duplex serial peer-to-peer connection.

RS485 A half-duplex serial master-slave bus, where each device has a unique address.

RTTI Runtime type information.

Signal Observable event. A signal can have multiple observers.

Slot Signal observer.

QML A user interface specification and programming language built on top of Qt.