Interner Bericht

DLR-IB-FT-BS-2022-198

Platform-Based Approach for Avionics Application Software

Hochschulschrift

Sven Friedrich

Deutsches Zentrum für Luft- und Raumfahrt

Institut für Flugsystemtechnik
Braunschweig

Deutsches Zentrum
DLR für Luft- und Raumfahrt

Institutsbericht
DLR-IB-FT-BS-2022-198

Platform-Based Approach for Avionics Application Software

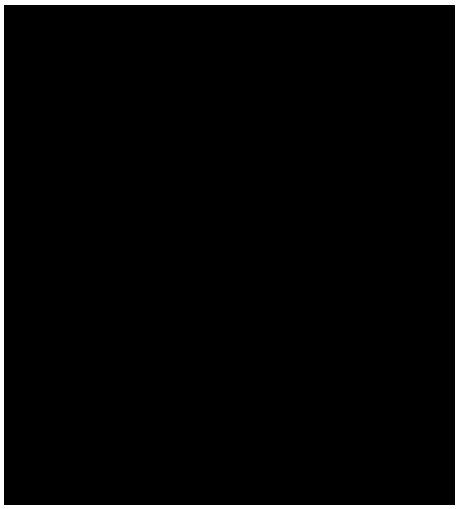
Sven Friedrich

Institut für Flugsystemtechnik
Braunschweig

82 Seiten
8 Abbildungen
1 Tabellen
49 Referenzen

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Institut für Flugsystemtechnik
Abteilung Sichere Systeme & Systems Engineering

Braunschweig, den 20.08.2024



Institutsleitung:	Prof. Dr.-Ing. S. Levedag
Abteilungsleitung:	Dipl.-Ing A. Bierig
Betreuer:in:	M. Sc. W. Zaeske
Verfasser:in:	M. Sc. S. Friedrich

TU Clausthal

PLATFORM-BASED APPROACH FOR AVIONICS APPLICATION SOFTWARE

MASTER'S THESIS

presented by

SVEN FRIEDRICH

Aeronautical Informatics Group
Department of Informatics
Technische Universität Clausthal

Sven Friedrich: *Platform-based approach for Avionics Application Software*

STUDENT ID

[redacted]

ASSESSORS

First assessor: Prof. Dr.-Ing. habil. Umut Durak
Second assessor: Prof. Dr. Günter Kemnitz

DATE OF SUBMISSION

22nd November 2022

EIDESSTATTLICHE VERSICHERUNG

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, wurden als solche kenntlich gemacht. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsstelle vorgelegt.

Clausthal-Zellerfeld, 22. November 2022

Sven Friedrich

PUBLICATION AGREEMENT

Ich erkläre mich nicht mit der öffentlichen Bereitstellung meiner Master's Thesis in der Instituts- und/oder Universitätsbibliothek einverstanden.

Clausthal-Zellerfeld, 22. November 2022

Sven Friedrich

ABSTRACT

In the avionic software domain, application development pace is comparatively slow while verification cost is high. Moreover, through reliance on expensive standards and software, entering the market is connected with a significant burden. This work explores the establishment of a practical APEX (ARINC 653) development platform. Improving the overall development experience, pace, safety, openness as well as code reusability are the goals in this endeavor implemented in the Rust programming language.

A Rust native APEX API was designed, assuring compatibility to C-language based hypervisors along with Rusts safety guarantees. Further, grouping APEX functionality into a set of traits, an APEX port extension library was developed for ascertaining powerful extensibility. For fast prototyping and the exploration of the APEX API, an already partially ARINC 653 part 4 compliant hypervisor was developed. It relies solely on Linux kernel features for uncomplicated usage and features built-in support for our Rust native APEX API. As a last step, partitions were developed utilizing our APEX API and extension library. By executing them on our as well as the proprietary XNG hypervisor, portability was demonstrated.

Demonstrating portability of generic extensions libraries and partitions, the potentiality of code-reuse is shown. Likewise, our hypervisor displayed functionality equaling SKE whereas requiring less setup. Enforcing memory safety related practices by default, Rust especially lends itself to a safety critical domain like avionics. These results suggest that a Rust based platform for avionic development may serve as the entry point for a flourishing ecosystem.

ZUSAMMENFASSUNG

Die Entwicklungsgeschwindigkeit ist in der Domäne von Avionik Software vergleichsweise gering, zeitgleich sind aber die Kosten für Verifizierung hoch. Hinzu kommt, dass der Zugang zum Markt durch teure Standards und Softwarelizenzen erschwert wird. Um diesen Hindernissen entgegenzuwirken untersucht diese Thesis Möglichkeiten, eine praktische, auf APEX (ARINC 653) ausgerichtete Entwicklungsumgebung zu schaffen. Ziel dieser ist, eine bequeme zu nutzende Entwicklungsumgebung anzubieten, mit der in kurzer Zeit Code von einer hohen Güte (erreichte Sicherheit) erzeugt werden kann. Des Weiteren soll die für und in Rust gemachte Entwicklungsumgebung das Erstellen von wiederverwendbarem Code vereinfachen und die Entstehung eines offenen Ökosystems begünstigen.

Für dieses Ziel wurde zuerst eine APEX API in Rust entworfen, die trotzdem auch Kompatibilität mit Hypervisoren welche in C geschrieben sind ermöglicht. Nicht nur wurde diese API so entworfen, dass sie möglichst viele von Rust's safety properties ausnutzt, auch wurden die APEX Methoden in Traits gruppiert. Dadurch werden mächtige Erweiterungen ermöglicht, was am Beispiel einer Kommunikationsbibliothek aufgezeigt wird. Um eine schnelle Feedbackschleife zu ermöglichen wurde des Weiteren ein Hypervisor entwickelt, der jetzt bereits große Teile von ARINC 653 Part 4 erfüllt. Dieser Hypervisor greift auf Primitive des Linux Kernel zurück um temporale und spatiale Isolation zu erreichen und dabei die oben erwähnte Rust APEX API zu implementieren. Zuletzt wurde ein Satz Beispielpartitionen entwickelt, welcher sowohl die APEX API als auch die eingangs benannte Kommunikationsbibliothek nutzen. Die Ausführung der Partitionen, sowohl auf unserem Linux Hypervisor wie auch auf dem proprietären XNG Hypervisor, zeigen die Portabilität unseres Ansatzes.

Nicht nur dass, sondern auch die Code-Wiederverwendbarkeit wird dadurch aufgezeigt. Gleichermaßen zeigt sich, dass unser Hypervisor ähnliche Fähigkeiten wie SKE besitzt. Durch das Erzwingen von memory-safety drängt sich Rust geradezu als Sprache für die Implementierung von Software in sicherheitskriti-

schen Domänen auf. Unsere Ergebnisse legen nahe, dass eine Rust basierte Plattform für Avionik Entwicklung ein guter Angelpunkt für ein aufblühendes Software-Ökosystem ist.

ACKNOWLEDGMENTS

While writing this Thesis, I have received numerous contributions in the form of textual review, help on technicalities and encouraging words when my motivation stagnated. It is only through these contributions that I could deliver this work in the present form. I want to express my deepest gratitude towards my friends, colleagues and supervisors for guiding me.

- ▷ Wanja Zaeske
- ▷ Umut Durak
- ▷ Tim Schubert
- ▷ Gustav Baier
- ▷ Hany Abdelmaksoud

To those who will never read this Thesis, let alone the title, I want to thank my family for supporting me throughout my studies, allowing me to go my own way.

- ▷ Stefan Friedrich
- ▷ Simone Friedrich
- ▷ Elfriede Friedrich

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

API	Application Programming Interface
APEX	APplication EXecutive
OCI	Open Container Initiative
ABI	Application Binary Interface
OS	Operating System
POS	Partition Operating System
IMA	Integrated Modular Avionics
HAL	Hardware Abstraction Layer
CAN	Controller Area Network
FFI	Foreign Function Interface
DAL	Design Assurance Level
CPU	Central Processing Unit
RAM	Random Access Memory
IPC	Inter-Process Communication
UTS	UNIX Time-Sharing
PCI	Peripheral Component Interconnect
NIS	Network Information Service
seccomp	Secure Computing Mode
POSIX	Portable Operating System Interface
ARINC	Aeronautical Radio, Incorporated
XNG	XtratuM Next Generation
ELF	Executable Linking Format
UART	Universal Asynchronous Receiver-Transmitter
maf	major frame
ptw	partition time window
SoC	System on a Chip
CI	Continuous Integration
XNG	XtratuM Next Generation
SKE	Separation Kernel Emulator

1 INTRODUCTION

Nowadays, new technologies are adopted at an exceptionally slow pace in avionics software[1]. What might be considered well established in IT, may only find usage decades after. While this makes sense for a domain where safety is the greatest good, new additions to the safety tool belt of developers get the same treatment, resulting in subpar incorporation of well researched technologies and methodologies.

Two aspects are of great importance for avionics software development here – Process and Platform. With the development process being strictly regulated, through for example DO-178C[2], the established Waterfall Model is usually employed, resulting in slower than necessary time to market[3]. Improving on this, more agile approaches like DevOps are sought after, which are already put to the test by the US Air Force[4]. As for the platform, with Integrated Modular Avionics (IMA), isolation needs to be guaranteed for allowing software of mixed criticality to run on the same system. For this, introduced in the broadly used ARINC 653 standard[5], the APplication EXecutive (APEX) is provided, an API for partitioning and isolation of software components.

In years to come, the most impactful addition to the platform related tool belt could be the Rust programming language. Rust is a promising candidate for the future of safe systems' software. Not only does it contribute by enforcing various safety related constraints through its linear type-system, one of its greatest values lays in enabling (near) zero-cost abstractions such as compatibility layers via its composable trait system. But how can Rust be connected to established software landscape of avionics? Is it possible to maintain compatibility with legacy software relying upon its proven dependability while also incorporating Rust's safety mechanisms? And when done, can the result remain manageable and approachable i.e. is a sustainable developing experience achievable?

In this work we introduce building-blocks for a platform that empowers developers to write avionic applications faster, without compromising on safety while also promoting code-

reuse within partition. For this, in Chapter 4, we explore existing software ecosystems which already achieved significant upscaling and community building. Following this, we identify problems with the APEX API, which ultimately hinder it in terms of hypervisor independence and safety. To address these problems, we then propose *apex.rs*, our own re-imagined APEX API in Chapter 5. It is implemented in Rust with a focus on ARINC 653 compatibility, safety and extendability. Complying with ARINC 653 part 4 and implementing *apex.rs*, we finally introduce our Rust based and Linux native hypervisor in Chapter 6. Demonstrating hypervisor independence and extensibility, in Chapter 7 we put our proposed platform to the test.

Therefore, in the following BACKGROUND chapter we start by providing an overview of the technologies used, such as the ARINC 653 standard and the Rust programming language.

2

BACKGROUND

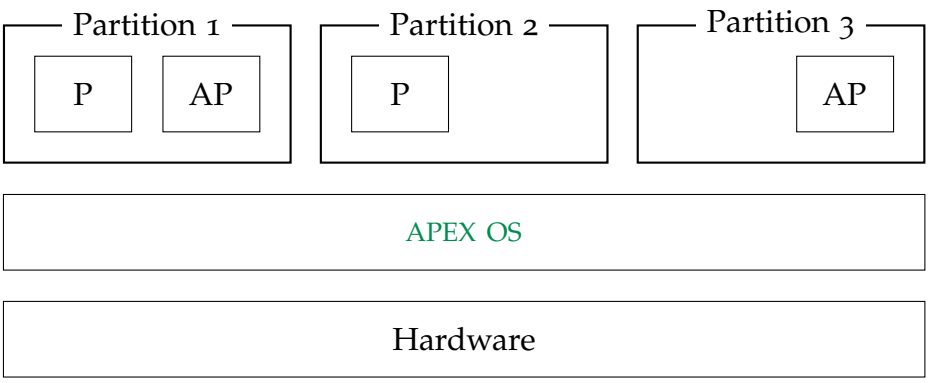


Figure 2.1: ARINC 653 Example Architecture
P: Periodic Process AP: Aperiodic Process

2.1 RUNTIME FOR AIRBORNE SOFTWARE: ARINC 653

ARINC 653 is an avionics standard for isolated partitioning of applications in a real-time context, published by Aeronautical Radio, Incorporated (ARINC)[5]. It introduces the APplication EXecutive (APEX), an Application Programming Interface (API) which specifies data types and functions (referred to as "services"). The ARINC 653 standard is split into multiple parts, with the required services being distributed into three of them:

- ARINC 653 Part 1: Standard Services [6]
- ARINC 653 Part 2: Extended Services [7]
- ARINC 653 Part 4: Subset Services [8]

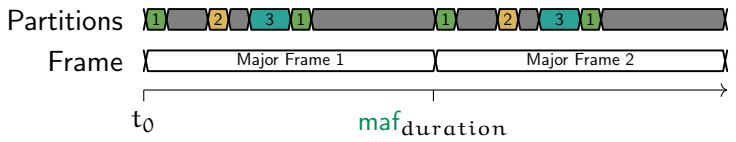
Here ARINC 653 part 1 marks the standard services required, with ARINC 653 part 4 being a subset and ARINC 653 part 2 being a superset of them. In theory, all of these feature sets are compatible with each other, as long as the APEX partition developer follows compatibility recommendations. This means that a partition written for ARINC 653 part 4 can run on a hypervisor implementing ARINC 653 part 1 or part 2 without any problems. As for the remaining parts of the ARINC 653

standard, part 0[5] gives a basic overview of the standard and part 3a[9] and 3b[10] specify conformity tests for hypervisors. In the following subsections, most information is taken from the ARINC 653 standard and hence references to it are omitted.

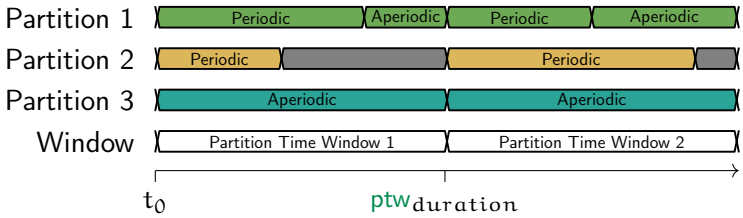
2.1.1 *Architecture*

A very basic exemplary structure of an ARINC 653 compliant system is shown in Fig. 2.1. An APEX API implementing and ARINC 653 compliant OS is running on top of some hardware. Hosted on top of the ARINC 653 OS are partitions, which together build an ARINC 653 module. Partitions are temporally and spatially isolated from each other. Spatial isolation means that partitions can not access nor corrupt the memory of another partition[5, 11]. Furthermore, temporal isolation guarantees that partitions can not consume more CPU resources than they are assigned. Through this an environment is constructed in which partitions can not harm the execution of each other[5, 11]. For what is executed inside a partition, the limited ARINC 653 part 4 allows for up to two processes, one of which is considered as the background process, running indefinitely and aperiodically. The other is the foreground process, which is started with each partition time window's (ptw) start.

2.1.2 *Scheduling*



(a) Major Frames



(b) Partition Time Window

Figure 2.2: ARINC 653 Scheduling

An ARINC 653 part 4 implementing hypervisor is only expected to utilize a single processor core. Moreover, allowing for only two processes inside of partitions, scheduling is kept simple.

MAJOR TIME FRAME In a schedule, to all partitions, a period, duration and offset is assigned. These values need to allow for a disjoint scheduling of partitions, which fits into the given major frame (maf) duration. Fig. 2.2a depicts APEX's largest scheduling primitive – the *major frame*. The period of a partition is for specifying the absolute time between consecutive starts of a partition's time window. This means that in this example, the set period for partition 1 is half the maf duration. This can be inferred since the time window of partition 1 appears twice in the maf. On the other hand, partition 2 and 3 both have a period equal to the maf duration, because they only have a single time window in each maf. The offset is for setting the start time of the ptw in relation to the start of the maf. In our example, this would mean an offset of zero for partition 1 and an offset larger than zero but smaller than half the maf duration for partition 2 and 3. For their duration that defines the length of the ptw, we can say that partition 1 and 2 are identical, with partition 3 running longer than both of them.

PARTITION TIME WINDOW As already mentioned, up to two processes are allowed in an ARINC 653 part 4 compliant hypervisor. The foreground process (periodic) is scheduled for the start of every ptw and once it stops, the background process (aperiodic) takes over. Contrary to the scheduling of partitions, the scheduling inside is neither static nor does it grant temporal isolation. As a result, the periodic process is allowed to run as long as it wants, potentially not leaving time for the background process. Because the deadline for the periodic process should always be equal to the ptw, failure to finish will result in a health monitor event being emitted. Fig. 2.2b shows the potential scheduling of processes within 3 different partitions. The first one represents a partition with both a periodic and an aperiodic process. Once the periodic process declares that it wants to wait for the next ptw, the aperiodic process is executed. The example's second partition only contains one periodic process. When it finishes its work, no process is scheduled for the rest

of the ptw. The periodic process in partition 1 and 2 run for different durations in each ptw, displaying the arbitrariness of intra-partition scheduling. Lastly, partition 3 contains a sole aperiodic process, always running during the entire ptw.

2.1.3 *Channel*

For inter-partition communication, ARINC 653 part 4 specifies two types of unidirectional channels - Sampling and Queuing ports. Due to them overstepping partition boundaries in a sense (one partition can now influence another one), they need to be statically defined at system design-time, allowing solely for predefined interaction between partitions. Moreover, to decouple the partitions, they may not know the destination nor source of a port – only the boundaries and name are known.

SAMPLING PORT Realizing one-to-many communication, the sampling port allows a single message to be multicasted to any number of partitions. At system design-time, the boundaries, namely source, destinations as well as name and maximum message size for each sampling port need to be defined. Additionally, during runtime destination-partitions may set an individual *refresh period* with the initialization of their sampling destination port. This refresh period is then used for providing a validity flag with every read message. Should the age of the message be within this defined period, the message is considered valid and vice versa. Due to this, the sampling port is especially suitable for sensor data. A partition, responsible for periodically reading a sensor can read its data and then write it to the sampling port. During design-time, all partitions relying on the data of the sensor can then be declared as destinations of the port. Setting a tight refresh period, reliant partitions may quickly notice if the sending partition did not send new data. This information may then be used for failover to a redundant data source.

QUEUING PORT Queuing ports allow for one-to-one communication only, but with the added capability of sending multiple messages at once. Similarly to sampling ports, source, destination, name and maximum message size need to be specified at design-time, with the addition of the maximum number of

queued messages. Further on, due to the existence of an upper limit for the number of queued messages the send operation can fail if the queue is already full. This means that with queuing ports, the destination partition can affect the source partition by not reading from the port. Therefore, the source partition should be able to deal with this situation.

2.1.4 *Health Monitor*

ARINC 653 considers three error levels:

- ▷ Process: May affect any number of processes in a single partition
- ▷ Partition: May affect a single partition
- ▷ Module: May affect all partitions

Sources of **process level errors** include the explicit raising of an error through the APEX `raise_appliation_error` service, execution errors like memory access violation and deadline misses. **Partition level errors** are mainly raised due to faulty partition configurations or erroneous system functionality. Lastly, **module level errors** can also originate from faulty configurations and system functionality as well as for example power failures. All of these error sources are examples and are ultimately hypervisor implementation dependent.

For recovery actions, ARINC 653 part 4 handles process level errors as partition level errors. While available recovery actions are hypervisor dependent as well, popular recovery actions involve *ignoring* the error, *stopping* or *restarting* the partition. These recovery actions are then set for the available error types and are accordingly applied. This means the recovery action of a partition initialization error may restart the partition, and a module initialization error may restart the module.

2.1.5 *Intra-Partition Execution Flow*

Partitions can be in one of four modes - `COLD_START`, `WARM_START`, `NORMAL` and `IDLE`. When a partition is scheduled for the first time, it is in the `COLD_START` mode. While the condition for starting into the `WARM_START` mode is hypervisor dependent, both start modes are for initializing the partition, processes and channel. Even though the start modes are for initialization of the partition,

all APEX services may be used during either start mode. But since during both start modes no process may be eligible for scheduling, the mode should be changed to `NORMAL` through the `set_partition_mode` service after initialization.

2.2 MODERN SYSTEMS PROGRAMMING LANGUAGE: RUST

Rust is a young and promising safety focused systems programming language, which enjoys great popularity among developers and organizations alike[12]. Because of its safety benefits, the Rust programming language is already considered for safety critical systems[13, 14]. But lacking a certified toolchain, the usage of Rust code in safety critical systems is aggravated[15]. Confirming Rust's potential for safety critical systems, the renowned AdaCore team joined the endeavour of the Ferrous Systems GmbH in developing a qualified Rust toolchain which adheres to current standards[13].

Rust provides safety through multiple ways. Extensive compile time checks combined with a linear type system cause many errors to be caught at compile-time. A borrow checker uses lifetimes to ensure no unsafe memory operations occur. Finally, a correctness driven API design is implemented with a rich mixture of compile-time and run-time checks to catch errors early, often already during compile-time. While there is much more to say about these approaches regarding safety, we focus on Rust's handling of pointers only for the scope of this section.

2.2.1 *Pointer*

Rust utilizes the concept of ownership for its data, effectively constructing a powerful abstraction over the concept of pointers[16, 17]. With ownership the owner of data is always clear to the compiler, effectively allowing for only two ways of passing data - surrendering ownership and borrowing[16, 18]. For enforcing this concept, Rust employs the borrow checker at compile-time, a tool for static analysis of data lifetimes[16, 18]. Through this, the compiler knows exactly how long pointers to data live compared to how long the data itself lives, raising a compile-time error in case of incompatibility[16, 18]. Not only does this relieve Rust of requiring garbage-collection for automatic memory man-

agement, many error classes related to referencing data can be identified at compile-time as well[17]. Use-after-free errors for example are made unattainable through this concept, preventing dangling pointer bugs[17]. Another facet of ownership is the differentiation between mutable and non-mutable data[16]. This allows the compiler to also prevent race-conditions, as the borrow checker allows either for a single mutable borrow to some data or multiple non-mutable borrows to exist at any given time, which are always safe to access simultaniously[17].

Compared to pointer usage in the C programming language, ownership as an abstraction over pointers is a lot more restrictive. But in C, pointers are also used for contiguous sequences of data, like for example arrays, where pointer arithmetic is used to access the data. In Rust, this is done via the *slice* primitive, which internally holds a pointer and a length, representing a reference to a contiguous sequence of variables[16, 17]. Among other already mentioned errors, this abstraction over an indexing pointer further prevents i.e. index out of bounds errors which could lead to writing data to unintended addresses[17].

2.2.2 *Extension Traits*

Rust as a programming language mandates the separation of data and functionality, as it urges "Composition over Inheritance"[19]. This separation manifests especially in Rust's means of realizing interfaces – *Traits*. Like with interfaces, traits define a set of functions which need to be provided by the implementor[16, 19]. Moreover, similar to templates in C++, these traits can be implemented for generic types as well. As Rust allows to constrain these generic types to implement specific traits (trait bounds), contrary to templates, generics are fully type-checked[20]. This means that a successfully compiling implementation of a trait for a constrained generic type is guaranteed to work[20]. Through this, the concept of *extension traits* is made possible. As a simple example for extension traits, Listing 2.1 may be used. Here we define the `Shout`-trait, defining the `shout`-function, which returns an all-uppercase string. Then in line 4 we implement this trait for all generics `T` that implement the `ToString`-trait. In line 6 we then call the `to_string`-function on `self`, followed by the `String` implemented function `to_uppercase`. Only because we constrained the generic `T` to implement the

Listing 2.1: `ToString` extension

```
pub trait Shout {
    fn shout(&self) -> String;
}
impl<T: std::string::ToString> Shout for T{
    fn shout(&self) -> String{
        self.to_string().to_uppercase()
    }
}
```

`ToString`-trait, we are allowed to call the `to_string`-function on `self` here. Otherwise, the compiler would complain that `T` is not guaranteed to provide a function with this name. By doing this, every type implementing the `ToString`-trait now also offers the `shout`-function.

3 RELATED WORK

3.1 SELECTIVE MIDDLEWARE

The approach introduced in 'Selective Middleware'[21] describes an alternative to current IMA architectures which focuses on control algorithms, called – the law. A platform is provided that given a distributed computer system and a set of laws (where the law is described as simplex) can configure a system to deploy the law on the hardware while fulfilling various desired redundancy patterns. The patterns (such as N-modular redundancy) are implemented using set of platform management tools and a middleware allowing to develop the law independent of redundancy without having to opt out of it. In a nutshell we understand this work to be an (ARINC 653 part 4 compatible) redundancy library combined with a set of tools to configure a system of systems. While this work touches many of the same aspects as ours, it focuses on control engineering (the law) and redundancy patterns. In the meantime we consider the development experience of applications running in the system, as opposed to the orchestrations of them.

3.2 ARINC 653 HYPERVISOR

In this section we want to introduce the current state-of-the-art surrounding ARINC 653 compliant hypervisors. For this, we take a look at existing type-1 and type-2 hypervisors, with type-1 hypervisors directly running on bare-metal and type-2 hypervisors requiring an underlying OS[22, 23]. Since we aim to provide our own ARINC 653 compliant type-2 hypervisor, the main focus lies on type-2 hypervisors. For the sake of completeness, prominent type-1 hypervisors are mentioned here briefly.

Offering memory isolation along fixed scheduling for partitions, fentISS's XtratuM Next Generation (XNG) is the most basic type-1 hypervisor we want to mention[3, 24]. XNG can run on System on a Chip (SoC) devices like the Xilinx Zynq Ul-

traScale and will be used by us to exercise our avionic platform. While XNG itself is not ARINC 653 compliant, fentISS also provides LithOS, a guest OS extending XNG with ARINC 653 APEX functionalities[11]. Moreover, fentISS offers an emulator product called SKE[3, 25], which will be further discussed in Sec. 3.2.2.1.

Other noteworthy ARINC 653 compliant type-1 hypervisors are SYSGO's PikeOS[26], Wind Rivers VxWorks 653[27], Green Hills' INTEGRITY[1], DDC-I's Deos[2] and Lynx's LynxOS[3]. Both are commercial hypervisors, offering ARINC 653 and POSIX compliance[28]. The latter of which enables them to be used as the underlying Operating System (OS) in POSIX based ARINC 653 solutions.

3.2.1 *ARINC 653 on POSIX*

POSIX	**ARINC 653**
Event driven execution	Cyclic based execution
Pre-emptive priority scheduling	Fixed time slices for partitions & Pre-emptive priority scheduling for processes
No temporal partitioning	Temporal partitioning
Sockets & IPC	Sampling and queuing ports

Table 3.1: Comparison of POSIX and ARINC 653[29]

In 2004, Airbus already explored the possibility of Linux in civil avionics, starting with a comparison between POSIX and ARINC 653[29]. Part of this comparison can be seen in Table 3.1, highlighting differences. The most severe difference is that POSIX does not offer any way of enforcing temporal isolation[29]. While sampling & queuing ports can be mapped to sockets & IPC, temporal isolation, as it is required in ARINC 653, can not be reproduced. This restricts POSIX based solutions to be used solely on top of Partition Operating Systems (POSs), where the scheduling strategy also matches ARINC 653[29]. The POSIX interface providing OS must then enforce temporal isolations for its partitions.

1 https://ghs.com/products/rtos/integrity.html
2 https://www.ddci.com/products_deos_do_178c_arinc_653/
3 https://www.lynx.com/products/lynxos-posix-real-time-operating-system-rtos

One POSIX based solution is realized through the *Portable APEX*[30]. The *Portable APEX* is an interface which implements the APEX API via the POSIX API[30]. Through this, any POSIX compliant OS can attain APEX compliance. When the *Portable APEX* was developed, it was specifically made with the AMOBA simulator in mind[30, 31]. With the main goal of providing means for testing and verification of ARINC 653 partitions, the AMOBA simulator claims to do so on any POSIX compliant system[31]. Because actual partitions scheduling is not possible with only the POSIX API, the *Portable APEX* may only be used inside of partitions.

3.2.2 *ARINC 653 on Linux*

As previously mentioned, Airbus already considered to use Linux in civil avionics[29, 32]. Reasons named for Linux are readily available packages and drivers, freeing developers from the need to reinvent the wheel as well as a simple development process[32]. But also acknowledging issues with Linux, usage in safety critical systems is complicated through the absence of real-time capabilities[32–34], and the enormous effort required for certification.

Next, we outline some solutions from the lots of research available regarding ARINC 653 compliance in Linux.

3.2.2.1 *Without Isolation*

Missing isolation forbids full ARINC 653 compliance, thus non-isolating hypervisors may be considered emulators[5, 6]. These still can be used for simple functional testing of ARINC 653 applications[3, 31]. SKE is one example of this; it provides an API similar to that of XNG, but it executes the partitions as normal process in Linux[3]. While its temporal behaviour is not at all accurate the semantics of inter-partition communication are[25]. Therefore, its use-case is primarily testing of partition code for functional correctness. Since the temporal behaviour is not accurate SKE allows to emulate a hypervisor configuration faster than real-time[25]. This is used for example to test for overflows and other errors which only occur after extended runtime of an application[3]. In this context, emulation of the XNG hypervisor entails that fentISS's ARINC 653 compliant LithOS also runs on SKE, allowing it to be used for testing of ARINC 653 applications.

3.2.2.2 *With Isolation*

With most research surrounding the usage of Linux in avionics, targeting full ARINC 653 compliance, isolation must be guaranteed[5, 6]. While all approaches we investigated differ in terms of final architectural design, the means for attaining compliance always involved writing Linux kernel modules or rewriting of the kernel itself[29, 35–37]. As a representative of these approach we want to briefly mention Han and Jins 'Full virtualization based ARINC 653 partitioning'[35]. For their approach an ARINC 653 kernel module was developed for the Linux kernel[35]. Through this, APEX services requests can be done via Linux system calls provided by the new kernel module[35]. Partitions are then executed in type-2 virtualization environments i.e. VMware or VirtualBox, providing temporal and spatial isolation[35]. Although their approach does not allow for inter-partition communication yet, partitioning and APEX service requests were implemented[35].

3.3 ASSURANCE AT THE LANGUAGE LEVEL

3.3.1 *MISRA-C*

Through MISRA-C, guidelines are specified for usage of the C programming language in safety critical systems[38]. Because of its provided safety benefits, the 1998 first published MISRA-C guidelines have found adoption in many industries, including automotives and avionics[14]. With Rust already providing safety through its compiler, it is to no surprise that Rusts compliance with these guidelines was investigated already[14, 39]. While we do not intend to go into too much detail about Rusts compliance to established coding standards like MISRA-C, we would like to give an example at least. In Pinho, Couto and Oliveira's 'Towards rust for critical systems'[14] some MISRA-C rules, which are already guaranteed by Rust, are listed. One of which is part of Rule 21.1, where minimization of runtime failure is to be attained through i.e. static code analysis, which Rust already provides through the borrow checker[14]. Newton's interactive Rust project *MISRA-Rust*[39] also investigates which rules are already satisfied by Rust. For this, Rust tests were written, for explicitly breaking MISRA-C rules. As a result of

these tests, Newton found that 108 of 143 MISRA-C rules can be enforced through Rust's compiler[39].

3.3.2 *Tightness driven development*

Regarding safe API design with the Rust programming language, the blog article *Tightness Driven Development in Rust*[40] by Mansanet, explores methods that use Rust type system to enforce sound usage of data types. Defining "tightness" as "the proportion of invariants that are upheld in the type definition as opposed to its methods"[40], the idea of *invariant driven development*[41] is improved upon. By passing as many invariant checks as possible to the type definition, the compiler guarantees inexpressibility of invalid states, reducing uncertainties at runtime[40]. As an extreme example, the `Shape` type in Listing 3.1

Listing 3.1: non-tight example: `Shape`

```
pub struct Shape {
   is_circle: bool,
   is_square: bool,
   dimension: i32,
}
```

illustrates what it means for a type to be tight. For the type's value to be valid, it must either be a circle or a square and the dimension may only be zero or positive, as it is used for representing either radius or side length. Evaluating how many expressible states of this type are valid, we are left with 25%, as only half of the boolean combinations and half of the values for dimension are valid[40]. The `Shape` type in Listing 3.2 on the

Listing 3.2: 100% tight example: `Shape`

```
pub enum Shape { Square(u32), Circle(u32) }
```

other hand is 100% tight, with no possibility for it to express an invalid state[40]. With the help of Rusts enum primitive, the type may either represent a circle or a square, and through using `u32` instead of `i32`, the dimension is restricted to zero or positive values. The blog goes as far as declaring that "unless your type's sole responsibility is to restrict the values of its fields, it

can always be 100% tight"[40] which one should always strive to achieve. Since complete tightness of a type restricting the value of its field may not always be achievable, it is acceptable for these types to be runtime checked. For demonstration, with

Listing 3.3: Value restricting `BatteryPercent`

```
1 pub struct BatteryPercent(u8);
```

Listing 3.3 a type is shown, representing battery level in percent. Considering the maximum value (255) for the internal `u8` and the maximum value for the battery level (100), `BatteryPercent` only achieves a tightness of about 39%, requiring constructor methods for runtime invariant checks. What we gain from this is that higher order types, for example `MobilePhone` or *ElectricalCar*, can now utilize this type for attaining 100% tightness themselves[40]. In that regard, tightness is more about delegating responsibility for runtime invariant checks to lower level types[40].

4

ARINC 653: APEX API

The APplication EXecutive (APEX) was implemented with one thing in mind: creating a standard platform for the development of airborne software[42]. Through defining a ubiquitous OS interface but leaving the implementation of the OS open, the foundation of a software ecosystem was build. Nowadays, there are multiple commercial vendors[24, 27] as well as a handful of open source projects[43, 44] that offer a (to varying degree) APEX compatible OS.

4.1 Case Study: Flourishing Software Ecosystems

4.1.1 *OCI*

A notable software ecosystem from a totally different domain is the world of containerization, which is also established around a ubiquitous software interface – Open Container Initiative (OCI).

The OCI defines several specifications regarding containers and their runtime environment. One of which is the OCI runtime-specification[45]. Adopting it, many pre-existing as well as newly developed container runtime environments could join the grand landscape of containerization, competing in features rather than vendor-locked interfaces[33].

On the other side of the interfaces are the containers, defined through the image-specification[46]. Following it, virtually anyone can build containers which can be executed inside any compliant runtime environment. Furthermore, the specification defines a layered layout for containers, promoting reuse as the layers of a pre-existing container may be used as the base layers of a newly created one. While this only allows for one-dimensional dependencies, well-established and tested groundwork may be build upon instead of being reinvented.

The creation of a container offering a generic static website may be used for illustrating this kind of dependency-layers. While a working Linux and Web server could be added by the

container architect manually, a premade container like nginx[4] could be used as the base-line instead, already setting up a working Web server. The official nginx container in turn depends on a Debian container, building upon already made efforts again.

4.1.2 *embedded-hal*

Another noteworthy software ecosystem is centred around the embedded-hal[5] – a Rust native Hardware Abstraction Layer (HAL) library which acts as the cornerstone for inter-compatible software design for Rust on embedded devices. It offers interfaces for various embedded-native interfaces i.e. I/O-pins, I^2C, SPI, Serial and even CAN bus. With this it exactly defines which functionalities are expected by these interfaces, composing a standard platform allowing for inter-compatible application software and high-level driver development.

4.2 SHORTCOMINGS OF THE APEX API

Enabling hypervisor agnostic usage of partitions has always been a major goal of APEX as the certification process of avionic software makes up for a majority of the development cost[47]. It would not be unthinkable for some application/partition to be verified and used with different APEX compliant hypervisors in different hardware-environments. In that regard, APEX already attempts to become the pivot for the interchangeable usage of hypervisors and partitions alike. Unfortunately, due to some problematic parts of the ARINC 653 standard, the APEX API may not be used in that manner without reservations.

4.2.1 *Implementation Dependant Types*

The first problem we want to address are hypervisor dependent types. ARINC 658's sole constraint towards identifier types used throughout the APEX API is that they are expected to be integer types[6]. Both 32 and 64-bit integer types may be used for identifiers, the choice is up the implementer of a hypervisor[6]. Furthermore, the identifier type used is not expected to be

4 https://hub.docker.com/_/nginx
5 https://github.com/rust-embedded/embedded-hal

consistent across interfaces. The ID of a sampling port may use a 32-bit integer, while queuing ports expect 64-bit ones to be used. This implementation dependant detail alone results in the loss of inter-compatibility of most APEX defined functions.

4.2.2 *Unconstrained Attribute Order*

Listing 4.1: Rust C-binding `MUTEX_STATUS_TYPE`

```
#[repr(C)]
pub struct MUTEX_STATUS_TYPE {
    pub MUTEX_OWNER: PROCESS_ID_TYPE,
    pub MUTEX_STATE: MUTEX_STATE_TYPE,
    pub MUTEX_PRIORITY: PRIORITY_TYPE,
    pub LOCK_COUNT: LOCK_COUNT_TYPE,
    pub WAITING_PROCESSES: WAITING_RANGE_TYPE,
}
```

Another hurdle for inter-compatibility is the implementation dependent ordering of parameters in APEX structs[6]. This means that there is no guarantee for any type used with the APEX API to adhere to any form of structure other than what attributes need to be included[6]. The `MUTEX_STATUS_TYPE` struct shown in Listing 4.1 may use any ordering, breaking Application Binary Interface (ABI) compatibility. There is also the fact that `MUTEX_STATUS_TYPE` includes a process ID which may either be 32 or 64-bit, further adding to the problem.

4.2.3 *Unchecked Buffer Length*

Listing 4.2: Rust C-binding `READ_BLACKBOARD` function

```
pub fn READ_BLACKBOARD(
    BLACKBOARD_ID: BLACKBOARD_ID_TYPE, // In
    TIME_OUT: SYSTEM_TIME_TYPE, // In
    MESSAGE_ADDR: *mut APEX_BYTE, // In
    LENGTH: *mut MESSAGE_SIZE_TYPE, // Out
    RETURN_CODE: *mut RETURN_CODE_TYPE, // Out
);
```

One last APEX design decision we want to discuss does not degrade portability but greatly affects memory safety. APEX functions reading or receiving messages only require a pointer to the starting address of the buffer to write into[6]. An example is the `READ_BLACKBOARD` function in Listing 4.2. While a `LENGTH` parameter is provided, it is not used for restricting the length of the buffer[6]. After this functions successfully returned, the `LENGTH` solely reports how many bytes were written to the provided buffer[6]. It is entirely possible for this number to be larger than the actual length of the provided buffer[6], resulting in uncertain consequences as well as potentially undefined behaviour.

4.3 OPPORTUNITIES

As we now established, the APEX API specified in the ARINC 653 standard does not allow for hypervisor independent usage of partitions because of its inherent flexibility and lack of tightness. Although portability of partition implementations can be achieved through hypervisor dependent C header files, declaring identifier types and parameter ordering of structures, the need for re-certification may not always be averted completely[47]. Due to this, one of APEXs main objectives is harmed. In the next chapter, we are going to introduce our Rust based APEX library, addressing some of the aforementioned issues as well as allowing for hypervisor independent code reuse and extensibility.

6 Can never be larger than the max message size of the blackboard

5 APEX ABSTRACTION LIBRARY

Solving some of APEXs issues, *apex.rs* aims to improve on memory safety as well as hypervisor independence to some degree. This is expected to be achieved through providing an easily implementable API for any APEX compliant hypervisor. As with the aforementioned embedded-hal ecosystem, a set of portable *traits* should build the baseline for an extendable ecosystem, promoting re-use of driver and feature-libraries.

5.1 ADVANTAGES OF USING RUST

For attaining the goal of simple integration with APEX compliant hypervisors, Rust functions with signatures compatible to ARINC 653 were defined. While this would have been sufficient for a usable system, we opted for wrapping most functions in more Rust befitting equivalents.

5.1.1 *Uncertain Results*

Listing 5.1: Rust C-binding `GET_SAMPLING_PORT_STATUS` function

```
pub fn GET_SAMPLING_PORT_STATUS(
    SAMPLING_PORT_ID: i64, // In
    SAMPLING_PORT_STATUS: *mut STATUS_TYPE, // Out
    RETURN_CODE: *mut RETURN_CODE_TYPE, // Out
);
```

The `GET_SAMPLING_PORT_STATUS` function, shown in Listing 5.1, may be used for illustrating one advantage of slightly changing the ARINC 653 defined functions. As the function is defined here, it would be perfectly compatible with any hypervisor following the reference header files provided with the ARINC 653 standard. Hence, it would reduce the implementation effort for any compliant hypervisor to just map from a Rust C-binding

to this function. The disadvantage lays in the lack of tightness and lifetime guarantees[40].

Dependent on the value of `RETURN_CODE` we are allowed to use the value of `SAMPLING_PORT_STATUS`. This leaves us in a situation where we either care for a non-zero `RETURN_CODE` or the `SAMPLING_PORT_STATUS`.

Listing 5.2: Rust `get_sampling_port_status` function

```
pub fn get_sampling_port_status(
    sampling_port_id: i64,
) -> Result<ApexSamplingPortStatus, ErrorReturnCode>;
```

As shown in Listing 5.2 this uncertainty is easily solved by using the Rust built-in `Result` type. This way, the `ApexSamplingPortStatus` is returned on success and a non-zero `ErrorReturnCode` is returned on failure.

5.1.2 *Compiler Guarantees*

Listing 5.3: Rust C-binding `WRITE_SAMPLING_MESSAGE` function

```
pub fn WRITE_SAMPLING_MESSAGE(
    SAMPLING_PORT_ID: i64, // In
    MESSAGE_ADDR: *mut u8, // In
    LENGTH: i64, // In
    RETURN_CODE: *mut RETURN_CODE_TYPE, // Out
);
```

Another advantage of the decision to use more Rust-like functions can be demonstrated with the `WRITE_SAMPLING_MESSAGE` function, as it can be seen in Listing 5.3. The problem is that the function shown takes two mutable pointers as inputs, which are then changed during the function call. Usually Rust's compiler would guarantee that mutable references to objects live as long as the objects themselves, but here pointers are used which the compiler can make no guarantees for[16, 17]. This means that the mutable object whose pointer was handed to this function could already have been freed, resulting in undefined behaviour[17]. Additionally, there is no guarantee that the `MESSAGE_ADDR` parameter points to an array of `LENGTH` bytes[17].

Listing 5.4: Rust `write_sampling_message` function

```
pub fn write_sampling_message(
    sampling_port_id: i64,
    message: &[u8],
) -> Result<(), ErrorReturnCode>;
```

In Listing 5.4 Rust's solution to this problem is presented, the primitive `slice` type. Moreover, through the usage of a referenced slice, the implementer and caller of the function get all guarantees Rust's borrow checker can provide[17]. One such guarantee is that the content of the `slice` is not going to change throughout this function's execution, since a non-mutable reference is used[17].

5.1.3 *Unsafe*

The functions we looked at so far demonstrate how following Rust's philosophy can grant us certain guarantees as well as raise our overall code quality. But there is one type of functions, required by the ARINC 653 standard, which we can not declare as safe – Receive/Read functions. The `RECEIVE_QUEUING_MESSAGE`

Listing 5.5: Rust C-binding `RECEIVE_QUEUING_MESSAGE` function

```
pub fn RECEIVE_QUEUING_MESSAGE(
    QUEUING_PORT_ID: i64, // In
    TIME_OUT: i64, // In
    MESSAGE_ADDR: *mut u8, // In
    LENGTH: *mut i64, // Out
    RETURN_CODE: *mut RETURN_CODE_TYPE, // Out
);
```

function in Listing 5.5 serves as an example for this. Just like the `WRITE_SAMPLING_MESSAGE` function in Listing 5.3, the parameter `MESSAGE_ADDR` and `LENGTH` are defined, but the length is a mutable pointer instead[6]. Indeed, taking a look at the `RECEIVE_QUEUING_MESSAGE` service request definition in the ARINC 653 standard it becomes apparent that the defined `LENGTH` parameter is not intended as an input, but an output[6]. This means that there is no way of checking whether the received message will fit into

the provided buffer at the address defined in `MESSAGE_ADDR` or not. Because of this it is entirely possible for this function to

Listing 5.6: Rust `receive_queuing_message` function

```
unsafe fn receive_queuing_message(
    queuing_port_id: i64,
    time_out: i64,
    message: &mut [u8],
) -> Result<i64, ErrorReturnCode>;
```

write into memory locations which do not belong to the provided message buffer. Listing 5.6 presents our implementation of this function. Again we use a referenced `slice` for the message buffer, which grants more guarantees and information to the function implementer. It is declared as mutable since we intend for it to be written to, and because of it being a slice, information on the buffer length is also provided. Something which is different compared to the previously defined functions is the `unsafe` keyword which explicitly declares this function as not memory safe.

Not memory safe does not inherently mean that this function should not be used. According to Rust's documentation it can be used for declaring that there are API contracts which the compiler can not check, requesting additional care of the programmer[16]. In our case the user of this function is required to guarantee that the received message is not larger than the provided message buffer. This may be done by either knowing the maximum length of next incoming message or by providing a buffer of the maximum message size defined for the requested queuing port.

5.1.4 *Language Independence*

As we now established, using more Rust-like functions, instead of function signatures closely following the service requests defined in the ARINC 653 standard, yields additional compile-time guarantees as well as more certainty regarding return types.

Now, one could argue that we make implementing *apex.rs* for APEX compliant hypervisors harder. Our response to this would be that the Rust programming language was specifically designed for safe systems programming and compatibility with

the C programming language[16]. By maximizing the usage of idiomatic Rust types and patterns we ultimately make using the APEX interface more safe and secure. Furthermore, though types like `core::mem::MaybeUninit`[7] and the Foreign Function Interface (FFI)[8] it is fairly simple to interact with external C functions[16].

One disadvantage of doing so is that there is usually no way around using `unsafe` functions and code-blocks. While we can not prevent this, interfacing Rust functions is almost always possible without `unsafe`. Indeed, this is the desired design process in Rust; providing safe abstractions even over unsafe code. This means that the Rust native hypervisor, we are going to present in Chapter 6, can implement *apex.rs* with fewer `unsafe` code, relying on the Rust compiler's guarantees as much as possible.

5.1.5 *Opt-in Features*

For *apex.rs* we decided to adopt the grouping of functions in the ARINC 653 standard, where for example sampling port related functions are combined in the "Sampling Port Services"[6]. Each of these service groups was made into a Rust trait for *apex.rs* which an hypervisor can implement in its compatibility library, called "shim". While some service groups are defined in each part of the ARINC 653 standard, they are always arranged in a super/sub-set relation, as previously mentioned in Sec. 2.1. This means that even though the number of functions in a service group defined in different ARINC 653 parts (P1, P2 and P4) can be different, they are conflict-free composable.

Our defined sampling port traits in Listing 5.7 demonstrate this relation. As it can be seen, `ApexSamplingPortP4` builds the basis for sampling ports with only three required functions. `ApexSamplingPortP1` then requires two additional functions while also requiring that `ApexSamplingPortP4` is implemented. Lastly, `ApexSamplingPortP2` introduces three more required functions while entailing the existence of functions defined in `ApexSamplingPortP1` and `ApexSamplingPortP4`.

Sampling ports are an example of a service group which gets extended by every feature set level defined in the ARINC 653

7 https://doc.rust-lang.org/core/mem/union.MaybeUninit.html
8 https://doc.rust-lang.org/nomicon/ffi.html

Listing 5.7: Sampling Port traits

```
pub trait ApexSamplingPortP4 {
    fn create_sampling_port(..) -> ..
    fn write_sampling_message(..) -> ..
    unsafe fn read_sampling_message(..) -> ..
}
pub trait ApexSamplingPortP1: ApexSamplingPortP4 {
    fn get_sampling_port_id(..) -> ..
    fn get_sampling_port_status(..) -> ..
}
pub trait ApexSamplingPortP2: ApexSamplingPortP1 {
    unsafe fn read_updated_sampling_message(..) -> ..
    fn get_sampling_port_current_status(..) -> ..
    unsafe fn read_sampling_message_conditional(..) -> ..
}
```

standard[6–8]. But there are also examples of service groups whose corresponding trait only appears in a single feature set level, most noteworthy the `ApexPartition` trait which is introduced in part 4 and never extended[8].

By grouping the required functions into separate traits per service group/feature set level, we gain a few things. Foremost, hypervisors which offer a richer feature set for certain service groups can simply offer the additional functions to the partition developer. Feature deprived hypervisors, on the other hand, can only implement whatever service group they offer, basically allowing for a sub-set to even part 4. Our Rust native hypervisor presented in Chapter 6 may be used as an example of a feature deprived hypervisor which does not even satisfy part 4.

Another advantage we gain is that of extension libraries which are only tightly coupled to that part of the *apex.rs* API which they actually interact with.

5.2 ABSTRACTIONS

Though Rust like functions were defined for all APEX service requests, most of the used types were kept as is. While this keeps implementation overhead low for hypervisor compatibility libraries, the partition developers are stuck with types which Rust offers more suitable ones for. One promise we made for

apex.rs is extendability, which we already build upon within apex.rs itself, implementing more suitable types and functions for interacting with the APEX API.

5.2.1 *Types*

Various APEX required types were abstracted over for enhanced usability of the APEX API. An example is the `ApexSystemTime` type.

Listing 5.8: ApexSystemTime

```
1 pub type ApexLongInteger = i64;
2 pub type ApexSystemTime = ApexLongInteger;
```

According to the ARINC 653 standard, the system time type is a 64-bit signed integer which corresponds to nanoseconds since a defined t_0. The -1 value identifies an infinite time value but for compatibility reasons all negative values should be interpreted as infinity.

Since 2015, Rust offers the `Duration` type in both the `core` and `std` library. The `Duration` type represents a zero or positive time and its smallest unit is nanoseconds. Because of this, it is especially fit as the system time type for APEX, since ARINC 653 smallest time unit is also that of a nanosecond[6]. The only thing missing is that `Duration` does not allow for an infinite time value. For that reason we introduced the enum presented in Listing 5.9, as the system time abstraction. It offers two variants, one for the infinite time and one for zero or positive time values. Other than the two functions directly implemented for `SystemTime` in line 7 and 10, various convenience `From` traits were implemented as well. Through them `into` and `from` functions are provided, converting for example from `Option<Duration>` to `SystemTime`. In this case, `None` is converted to `SystemTime::Infinite` and `Some(Duration(10s))` ends up as `SystemTime::Normal(Duration(10s))`.

Unfortunately, the introduced `SystemTime` comes with a cost in form of a higher memory consumption. The base APEX type equivalent `ApexSystemTime` requires exactly 8-bytes of memory. Rust's `Duration` type on the other hand requires 16-bytes of memory and through using an enum which requires a tag for identifying the variant, another 8-bytes are added. This nets us 24-bytes for `Systemtime` which is trice as much as the ARINC 653

Listing 5.9: SystemTime

```
pub enum SystemTime {
    Infinite,
    Normal(Duration),
}
impl SystemTime {
    // For getting a SystemTime from a ApexSystemTime
    pub fn new(time: ApexSystemTime) -> SystemTime { .. }
    // For getting the inner Duration easily
    // Panics if this is SystemTime::Infinite
    pub fn unwrap_duration(self) -> Duration { .. }
}
impl From<Duration> for SystemTime { .. }
impl From<SystemTime> for Option<Duration> { .. }
impl From<Option<Duration>> for SystemTime { .. }
impl From<ApexSystemTime> for SystemTime { .. }
impl From<SystemTime> for ApexSystemTime { .. }
```

defined system time type demands. While this may be a problem for some systems, the convenience, tightness and compatibility to other high level Rust types may outweigh the extra memory required[40]. This is an example of a non zero-cost abstraction.

In this fashion, types like `ApexName` and `ErrorReturnCode` received sensible abstractions and trait implementations which can be seen in Listing A.1.

5.2.2 *Services*

Using the newly abstracted types, higher level abstractions were introduced by us for all APEX required services. For illustrating the abstraction treatment all services received, our abstraction for blackboards is used. In Listing 5.10 the previously defined trait `ApexBlackboardP1` is partially shown. What becomes apparent is that the blackboard ID always appears either as an input or an output. This is because the ARINC 653 standard expects blackboards and other services to be used solely through IDs[6]. One disadvantage of doing this is that all the IDs used in the APEX API are type aliases on the same integer type[6]. This implies that there is no type safety between for example the ID of a blackboard and the ID of a sampling port. They could be

Listing 5.10: ApexBlackboardP1 trait

```
pub trait ApexBlackboardP1 {
    fn create_blackboard( Name, Size ) -> Result<Id, Error>;
    fn display_blackboard( Id, Msg ) -> ..;
    unsafe fn read_blackboard( Id, .. ) -> ..;
    fn clear_blackboard( Id ) -> ..;
    fn get_blackboard_id( Name ) -> Result<Id, Error>;
    fn get_blackboard_status( Id ) -> ..;
}
```

used interchangeably without any warning being emitted by the compiler at all. One way of circumventing this issue would be to wrap the ID types into their own distinctive structs as an abstraction. While this works, it is not in the sense of object-oriented programming where a type is initialized and then its member functions are called. Hence, we opted for introducing a `Blackboard` struct which can be initialized and called with all functions expected of an ARINC 653 blackboard.

Listing 5.11: Blackboard abstraction

```
pub struct Blackboard<const MSG_SIZE: usize,
  B: ApexBlackboardP1> {
    id: BlackboardId,
    _b: PhantomData<B>,
}
impl<const MSG_SIZE: usize, B: ApexBlackboardP1>
  Blackboard<MSG_SIZE, B> {
    pub fn new(..) -> ..
    pub fn from_name(Name) -> ..
    pub fn id(&self) -> ..
    pub const fn size(&self) -> ..
    pub fn display(&self, ..) -> ..
    pub fn read(&self, Buffer, ..) -> ..
    pub unsafe fn read_unchecked(&self, Buffer, ..) -> ..
    pub fn clear(&self) { .. }
    pub fn status(&self) -> ..
}
```

The required data stored inside a potential `Blackboard` struct includes the blackboard ID and the allowed max message size. Furthermore, some means of calling the APEX base functions

must be provided as well. Listing 5.11 depicts how the `Blackboard` struct was eventually designed. Internally two variables are used, the ID for the blackboard and some zero-sized `PhantomData` for the used hypervisor. Moreover, two generic parameters are utilized for constraining the blackboard at compile-time. The first one is for the max message size specified for this blackboard instance. We opted for a `const` generic here since blackboards are initialized during the cold/warm start of a partition and therefore their size should be fully known from the start. The other generic is for the used hypervisor, and it is required to implement at least `ApexBlackboardP1`. Through this, all functions expected in the `ApexBlackboardP1` trait can be used on the generic type `B` inside the struct.

Other than introducing member functions for all functions provided through the APEX API, we also offered some extra functions. One small addition are getter for the generic max message size and internally stored ID, but the more important addition is a safe read function. Revisiting the `ApexBlackboardP1` trait in Listing 5.10 we notice the `unsafe` keyword in front of the `read_blackboard` function. This is due to read/receive functions specified in the ARINC 653 standard not being memory safe because of the absence of a length parameter for the provided buffer. But as a result of `Blackboard`'s `const` generic specified max message size we can grant the required safety constraint for safely calling `read_blackboard`. Internally, the length of the provided buffer is checked against the fixed maximum message size and an error is returned, should the length be insufficient.

In this manner, all services are abstracted over, aiming to offer a more struct centred approach while also reducing the need to call `unsafe` functions.

5.3 EXTENDABILITY: (DE-)SERIALIZATION FOR apex.rs

The only data that ports in the APEX API are expected to transfer are byte sequences of predefined max length[6]. While this allows for sending basically anything over these ports, some form of parser logic is required to utilize them for sending custom types. Furthermore, not all data can be safely interpreted as byte data. Especially heap allocated data like strings may not simply be cast to bytes, as only the pointer may be cast accidentally. But since many data types are safely serializable

an extension library offering exactly this – sending and receiving only serializable data – was developed for apex.rs.

Through the importance of serialization in various software domains, a generic serialization/deserialization framework established itself in the Rust ecosystem - Serde[9]. Offering the traits `Serialize` and `Deserialize`, any implementing type may be serialized and deserialized from and to Rust native types. On the other side of the equation serializers exist for various target formats like JSON[10], TOML[11] and YAML[12]. But there are also compatible serializers for non-human-readable serialization. One of which is Postcard[13] which aims to, among other things, be `#[no_std]`[14] compatible as well as resource efficient with regard to memory usage and CPU time.

With the existence of an established serializer framework in addition to a serializer which is optimized for embedded use-cases, the idea to utilize both of them for sending advanced data types over APEX ports presents itself.

5.3.1 *Implementation*

To illustrate how apex.rs can be externally extended we will implement a set of functions for queuing and sampling ports which will allow for sending and receiving serializable data types. As the Rust programming language endorses the principle of composition over inheritance, extension libraries can extend the functionality of foreign types. This means that a trait introduced in our extension library can be implemented for the `SamplingPortSource` struct which is declared in apex.rs.

Listing 5.12 contains the declaration of the `SamplingPortSourceExt` trait. This extension trait features only a single function – `send_type`. That function takes a generically typed payload as an input and returns a Result. The payload type is bound to implement the `Serialize` trait, which enables us to serialize it using the Serde framework.

9 https://crates.io/crates/serde
10 https://crates.io/crates/serde_json
11 https://crates.io/crates/toml
12 https://crates.io/crates/serde_yaml
13 https://crates.io/crates/postcard
14 Rust attribute, pledging that only platform-agnostic parts of the standard library are used

Listing 5.12: `SamplingPortSourceExt` trait

```
pub trait SamplingPortSourceExt {
    fn send_type<T>(&self, payload: T)
        -> Result<(), SendError>
    where
        T: Serialize;
}
```

Our example implementation of `SamplingPortSourceExt` is shown in Listing 5.13. Here we first had to define a buffer which Postcard serializes the given input into. Through the usage of the `const` generic `MSG_SIZE` set for this `SamplingSource` instance, we are guaranteed to adhere to the message size limit set. This way we can successfully serialize the given input only if it fits into the used sampling port, otherwise an error is returned – `SerializeBufferFull`.

Listing 5.13: `SamplingPortSourceExt` implementation

```
impl<const MSG_SIZE: MessageSize, Q: ApexSamplingPortP4>
    SamplingPortSourceExt for SamplingPortSource<MSG_SIZE, Q>
where
    [u8; MSG_SIZE as usize]:,
{
    fn send_type<T>(&self, p: T)
        -> Result<(), SendError>
    where
        T: Serialize,
    {
        let buf = &mut [0u8; MSG_SIZE as usize];
        let buf = postcard::serialize_with_flavor
            ::<T, SerSlice, &mut [u8]>
                (&p, SerSlice::new(buf))?;
        self.send(buf).map_err(SendError::from)
    }
}
```

Other than the `SamplingSourceExt` trait, our Postcard extension for APEX ports also features a `SamplingPortDestinationExt` trait. This trait introduces a `recv_type` function which can then be used by the receiving end of the sampling port for deserializing incoming messages into advanced types. Should this function

fail during the deserialization process, the received byte array is returned instead, for the developer to attempt other type deserializations.

Equivalent traits and implementations are provided for queuing ports as well. All of this can be found in Appendix B.

5.3.2 *Shortcomings*

5.3.2.1 *Uncertain Message Size*

Because serialization generally allows for dynamically structured and sized types, the size required for a serialized variable can not always be deducted at compile-time. Due to this limitation, `send` function calls fail when the port's max message size is exceeded.

As an alternative extension, bytemuck[15] may be used for sending types over ports. Bytemuck is a crate for safely casting between byte types. This means that a bytemuck compatible type may be cast to a byte array of fixed size and back. Through this it would be possible to know at compile-time if a type fits into a port or not without unexpected side effects. The caveat is that a bytemuck compatible type is any type implementing bytemucks marker trait - `Pod`. This trait is explicitly marked as `unsafe` and annotated with a strict set of requirements for retaining safety. Only if these requirements are guaranteed to be upheld, the trait may be used on user defined types. Due to this restriction we opted for implementing a Postcard based port extension first, still mentioning bytemuck as a possible alternative to serialization.

5.3.2.2 *Nightly Toolchain*

This library introduces two functions – `send_type` and `recv_type` – for every possible sampling and queuing port in apex.rs. While doing so, it successfully uses the channel provided `const` generic `MSG_SIZE` for the buffer length used for serialization by Postcard. The caveat with this is that for generically specifying the size of an array from `const` generics, an unstable Rust feature is required - `generic_const_exprs`[16]. Unstable features generally

15 https://crates.io/crates/bytemuck
16 https://github.com/rust-lang/rust/issues/76560

require Rust's nightly toolchain which includes potentially unsafe features which are subject to breaking-changes.

5.4 PARTITION DEVELOPMENT

Because APEX partitions require somewhat large code segments for initialization, development may feel discomfiting. Likewise, through the requirement of a hypervisor shim library, which is used for all types in `apex.rs`, the code gets more bloated and harder to read. For providing a more ergonomic development experience Rusts *proc-macros* are employed.

Listing 5.14: Barebone APEX partition *proc-macro*

```
#[partition(HypervisorShim)]
mod partition {
    #[start(cold)]
    fn cold_start(ctx: start::Context) {
        ...
    }
    #[start(warm)]
    fn warm_start(ctx: start::Context) {
        ...
    }
    ...
}
```

Other than the C preprocessor which only performs text manipulations, Rust's macro system is based on compiler plugins. These plugins operate on the token stream of the compiler, allowing for compile-time reflection of types, arbitrarily rewriting of code sections all while being much more hygienic than the C preprocessor. With Rust macros offering means of code generation, we introduce convenience *proc-macros* hiding away bloated code segments while making the development process more accessible. Listing 5.14 depicts the minimum required code necessary for an APEX partition when using our *proc-macros*. The partition needs to be located inside a module (`mod`) that is annotated with the `partition`-attribute. Inside this attribute the hypervisor shim for the underlying hypervisor needs to be specified. Other than that, start functions for cold and warm start are necessary, which need to be annotated with `start`-attributes. As a provided para-

meter, `start::Context` is handed to each start function, offering the hypervisor's API, plus initialization functions for APEX primitives. All the required logic, like entry function for the partition and partition state based selection of either warm or cold start is done via code generation in the background.

5.4.1 *Channel*

Listing 5.15: APEX channel *proc-macro*

```
#[sampling_out(msg_size = "10KB")]
struct SamplingOutExample;
#[sampling_in(refresh_period = "110ms")]
#[sampling_in(msg_size = "1200B")]
struct SamplingInExample;
#[start(cold)]
fn cold_start(ctx: start::Context) {
    ctx.init_sampling_out_example().unwrap();
    ctx.init_sampling_in_example().unwrap();
}
```

Listing 5.15 illustrates how channels can be specified when using our *proc-macros*. Via either attaching a `sampling_out` or `sampling_in`-attribute to a struct, APEX channel can be defined. Further, sampling ports require a specified message size and the destination also requires a maximum age for received messages (`refresh_period`). Any missing fields result in a compiler error

Listing 5.16: apex.rs *proc-macro* misuse

```
error: Missing field ‘msg_size‘
  -> tests/partition.rs:13:5
13 |     struct SamplingInExample;
   |     ^^^^^^
```

explicitly stating what is missing (Listing 5.16). In this manner duplicate definitions and usage of wrong types are also prevented. As for the initialization of channels, `init`-functions are automatically provided with the `start::Context` parameter of the start functions for every defined channel. Because the ARINC 653 standard explicitly states the existence of an upper (per partition)

limit for initialized channels the `init`-functions may return an error. For individual handling of the `init`-result, the calling is for the developer to do.

5.4.2 *Processes*

Listing 5.17: APEX process *proc-macro*

```
#[aperiodic(
    time_capacity = "Infinite",
    stack_size = "100KB",
    base_priority = 1,
    deadline = "Soft"
)]
fn aperiodic_example(ctx: aperiodic_example::Context) {
    loop {
        ctx.report_message(b"Ping").unwrap();
        sleep(Duration::from_millis(1));
    }
}
#[start(cold)]
fn cold_start(ctx: start::Context) {
    ctx.init_aperiodic_example().unwrap();
}
```

Likewise, *proc-macros* exist for the creation and usage of APEX processes. As shown in Listing 5.17, the `aperiodic`-attribute may be used, but the `periodic`-attribute also exists. Like with APEX channels, processes also need to be manually initialized in the start functions and similar to the start functions, a `ctx`-parameter is provided. Again, this context offers functionalities of the underlying hypervisor, while also yielding defined channels. If the `SamplingInExample` sampling port from Listing 5.15 also exist here, the `aperiodic_example::Context` contains a field named `sampling_in_example`. The field would then be of type `Option<SamplingPortSource<_>>` containing the port if the `init`-function was successfully.

6 LINUX HYPERVISOR

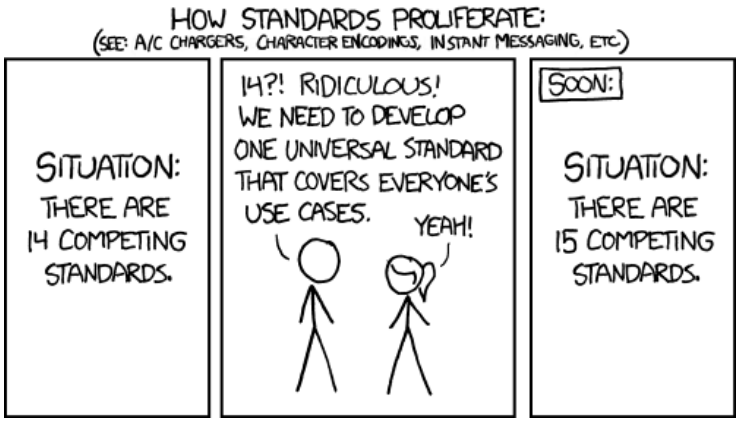


Figure 6.1: Standards[48]

Most compilation artifacts are not actually released but only tested inside a development environment. Therefore, one can infer that the usability and accuracy of a development execution environment is highly relevant. Over the past, approaches towards achieving this kind of development environment have been attempted. In Sec. 3.2 we already talked about SKE[25] and AMOBA[31] whose purpose is to assist during the development process.

In this chapter we are now going to present our approach towards advancing the state of the art (Fig. 6.1).

6.1 OBJECTIVES

Building a Linux native hypervisor, we aim to provide a tool for fast, simple and dependency free prototyping of modules and partitions in the sense of ARINC 653. Through relying solely on Linux kernel features, the hypervisor should work with any Linux running a recent kernel version, relinquishing any further dependencies. With these objectives already being plenty, we also want the hypervisor to be fully functional, even when executed by an unprivileged user.

6.2 TEMPORAL AND SPATIAL ISOLATION PRIMITIVES IN LINUX

Since the Linux kernel already offers various means of isolation, we attempt to use them for providing spatial as well as temporal isolation capabilities to our hypervisor[33, 49].

6.2.1 *Temporal Isolation*

The first isolation requirement towards our hypervisor is that partitions are isolated temporally. This means that a partition should never be able to monopolize on CPU resources more than configured during design-time[5, 11]. In the ARINC 653 standard, execution of partitions is stated to be done through a fixed schedule in which partitions can freely utilize assigned CPU cores during their respective time windows[6]. Outside their declared time windows, partitions are not allowed any CPU resources at all[5]. As a result, defective and malicious partitions can not obstruct the execution of well-behaving ones[5, 11].

In our hypervisor we are going to use Linux's *cgroups* to achieve this. They grant us the ability to group together processes or threads into a *cgroup* and suspending the execution of them with a single syscall[49]. Further, *cgroups* cpuset controller allows allocating a *cgroup* exclusive access to a CPU core. This not only constraints the *cgroup* to only run on a selected group of CPU cores, but also prevents processes from other *cgroups*[17] to utilize said CPU cores, i.e. the allocation is truly exclusive. Because of these capabilities, *cgroups* are fit for implementing ARINC 653s scheduling, enforcing temporal isolation, even against the rest of the system[49].

6.2.2 *Spatial Isolation*

At its core, spatial isolation boils down to that a partition can not affect the memory of other partitions[5, 11]. Also, a partition may only use memory allocated to during design-time[5]. This is a very important property: running out of memory is a typical cause of system failure which in cases like

17 unless those other *cgroups* are children of the *cgroup* exclusively allocated with cores

small memory leaks may not be found easily by testing. The temporal isolation causes these kinds of failure to be contained inside the originating partition. The ARINC 653 standard also speaks of "Robust Resource Partitioning" which is thought of as a superset of spatial partitioning[5]. It further requires that failures of hardware used only by a single partition, may not affect other partitions[5]. With spatial isolation being complex compared to temporal isolation, multiple Linux feature need to be utilized for an attempt at covering it fully.

6.2.2.1 *Resource Limits*

For limiting memory as well as enforcing obedience of partitions, *cgroups* and *tmpfs* file systems together with namespaces can be used.

CGROUPS The *cgroup* memory controller can grant us the ability to specify an upper limit for RAM consumption of an entire *cgroup* subtree[49]. Should the limit be exceeded, all processes inside the *cgroup* are killed immediately. While this behaviour is not exactly desired, there is currently no other way of limiting RAM consumption for a process or group of processes.

TMPFS FILE SYSTEM The Linux kernel allows for the creation of *tmpfs* file systems which reside entirely in the RAM. Due to that fact, files in these file systems are volatile and lost when closing the *tmpfs*. The advantage is that on partition error related restarts, all used memory is completely wiped offering a fresh start. Additionally, created from inside a partitions *cgroup*, the same RAM limits specified through the *cgroups* memory controller apply, limiting file system usage too.

MOUNT NAMESPACE One of Linux's namespaces is the mount namespace, which offers the possibility to mount and unmount file systems inside of mount namespace separate from the rest of the system[49]. Hence, adding a partition to its own mount namespace, all file systems can be unmounted, leaving only the *tmpfs* file system of the partition. Even the root file system may be unmounted as long as no file from the root file system is opened inside the mount namespace. Prior to unmounting the root file system, the root of the partition needs to be replaced with a new

one, for example the *tmpfs* created for the partition. Doing this, all memory accessible by a partition is in the RAM and therefore limited by the *cgroups* memory controller, offering strict resource limits for partitions.

6.2.2.2 *Resource Isolation*

Isolation of memory, as in restricting access to specified resources, can be realized through namespaces and *memfd*.

NAMESPACES As mentioned before, mount namespaces can be used to unmount all file systems that a partition should not access. While this prevents a partition from browsing these file systems from inside its mount namespace, a partition can still open some files opened by other processes. In `/proc/{pid}/fd` all open file descriptor of every process are listed. Using the `fopen` syscall on them opens the file, this especially holds for volatile files which are not located on any filesystem, but are located in the RAM. Because this can be used to break isolation between partition, the process namespace is used as well.

Processes inside their own process namespace are isolated from outer processes – outer processes are not listed in Linux's `/proc` folder. This way partitions inside their individual process namespaces are further isolated from each other, especially preventing access to namespace external files.

MEMFD Not only requiring inaccessibility to external resources, another facet of resource isolation involves restricting write permission to shared memory. APEX sampling ports for example provide means of inter-partition communication, where source and destination are clearly defined. While a source needs to be able to write to a sampling port, the destination partition may only read the port. The recommended Linux feature to employ for realizing shared memory is the `memfd_create` syscall, which creates an anonymous volatile file in RAM. The most important part about files created with `create_memfd` is that they can be sealed against future write-enabled memory mappings.

This allows us to create a *memfd* and send a write-enabled memory mapping of it to partitions, granted write access. Afterwards we can seal it against further write-enabled memory mappings and send it to all partitions with read-only access.

This sealing is necessary because like with other files, they can be reopened through their file descriptors in `/proc`. Process namespaces isolate from external processes files, but under `/proc/self/fd` the processes own file descriptors are still accessible. As a result of sealing the *memfd* against future write-enabled memory mappings, we prevent a process from reopening the *memfd* file descriptors with write-flag set.

6.2.3 *Unprivileged Isolation*

Another noteworthy Linux namespace is the user namespace. Even though an unprivileged user can not gain root user level access to the system, a process can still gain access to privileged operations inside user namespaces[49]. While these privileges can not breach out of the user namespace, not allowing for example reading of protected files, they allow for creation of other namespaces like process and mount namespaces, which usually requires privileges.

In that regard, user namespaces synergize exceptionally well with mount namespaces. Because privileged operations exerted inside user namespaces may not reach out of it, mount operations are not permitted as they would affect the whole system. A mount namespace created from within the user namespace on the other hand becomes part of its domain allowing for mount and unmount syscalls as they solely affect internals of the user namespaces. Besides, using mount namespaces without backing user namespace requires privileges for creating the namespace, mounting and unmounting, rendering it completely unusable in an unprivileged context.

cgroups are already usable without privileges, while not without reservations. Through their hierarchical structuring the root level *cgroups* are accessible by privileged users only, with branches also existing for user level *cgroups*. It is by utilization of these user accessible branches that unprivileged user can also create *cgroups*.

The caveat is that for a *cgroup* controller to be available, it must be activated by every branch from the root to the target *cgroup*. The cpuset controller, used for exclusively assigning CPU cores to *cgroups*, is a controller which is usually not propagated to unprivileged *cgroups*. Using this feature, a privileged user has to either activate the cpuset controller for all *cgroups* from the

root to the target group or create a user accessible group directly on top of the root level.

6.2.4 *Isolation Hardening*

Although not directly required by the ARINC 653 standard, it is desirable to limit access to Linux-syscalls, -features and other information in order to improve isolation. Furthermore, it is important to disable means of communication not foreseen by ARINC 653. This way, adherence to the communication primitives provided by APEX is enforced.

6.2.4.1 *Other Namespaces*

To achieve this, the remaining unused namespaces may be utilized. The network and IPC namespace for example can help further isolating a partition from the outside[49]. As freshly created network namespaces always start out empty, not a single networking interface of the system is usable by the processes inside it[49]. Also, the IPC namespace isolates in terms of IPC resources such as POSIX message queues.

Besides these two, there are also the time and UTS namespaces which can restrict access to information[49]. The UTS namespace allows for changing the hostname and NIS domain name. Moreover, the time namespace provides the means for altering monotonic and boot time clock[49]. Hence, both allow for hiding information of the host system, they could however be used for providing relevant information to each partition like partition name or partition start time.

The last namespace we want to address is the *cgroup* namespace. Changing the root of visible *cgroups* to the *cgroup* of the namespace creating process, it restricts both, information of the host system and Linux feature accessibility.

6.2.4.2 *Seccomp*

Probably being the most powerful tool in terms of restricting access to Linux functionalities, Secure Computing Mode (seccomp) can be used for disallowing (or better specifically allowing) syscalls[49]. Should a process, restricted by seccomp, attempt to perform a forbidden syscall anyway, a predefined action is per-

formed. The most important one are immediate termination of the offending process and the return of a preset error code[49]. Using seccomp, the processes of a partition can be denied usage of all unnecessary syscalls, ultimately benefitting isolation.

6.3 LINUX NATIVE APEX SERVICES

Since our hypervisor is supposed to be executable on Linux with minimal effort, offering a custom kernel is out of the question. Because of this as well as the constraint of unprivileged execution, we are stuck with execution in userspace. Where a normal hypervisor would introduce kernel level primitives for the realization of APEX services, we need to make use of Linux primitives from within userland.

6.3.1 *Ports*

For inter-partition communication ARINC 653 part 4 defines the two port types, sampling and queuing ports[8].

SAMPLING PORT Providing one-to-many communication and retaining only the last sent message, sampling ports are a prime candidate for appliance of shared memory. Unfortunately, sampling ports additionally require information about the age of the contained message. Due to this requirement, the same shared memory may not be handed to source and destination, as the source could lie about the age of the message. For solving this, two stages are used. The source writes some data to the shared memory and the hypervisor recognizes this as a sampling port send operation. Afterwards, the trustworthy hypervisor writes the data to the destinations shared memory, adding the current time for age calculation by the receiver.

QUEUING PORT Queuing ports are offering unidirectional one-to-one communication of a set maximum number of buffered messages. Because of the max buffered messages limitation, Linux communication channels such as POSIX message queues and Unix sockets can not be used here. As an alternative, we propose the use of a ring buffer on shared memory. Similar to our solution for sampling ports, two stages are used, with

the hypervisor passing the data to the shared memory of the destination port. This prevents the sender from corrupting the ring buffer, since the hypervisor can enforce conformity before copying the data. What is different is that the receiver is handed a write-enabled shared memory too. Considering the receiver needs to indicate which messages were read already, the same shared memory is taken advantage of. Again, the two stage approach prevents corruption propagation here as well.

6.3.2 *Processes*

For processes, Unix processes in child *cgroups* of the partitions *cgroup* are used. Because ARINC 653 part 4 only allows for the creation of a single periodic and single aperiodic process, a child *cgroup* is created by the hypervisor for each one. Within a partitions time window, the hypervisor then unfreezes the periodic process first, should it exist. Through accessibility of the partitions *cgroups* by processes, the periodic process can then freeze itself should it be done for the current time window. The hypervisor can detect this and unfreeze the aperiodic process until reaching the end of the ptw.

6.3.3 *System calls*

For providing access to APEX services, a hypervisor would typically offer them as system calls which are directly usable by partitions. Utilizing `apex.rs` from Chapter 5, our Rust native hypervisor implements traits for the ARINC 653 part 4 features. Internally, service requests from partitions in our hypervisor are conducted using Unix Sockets, offloading work to each partition.

OFFLOADING Most services offered through the APEX API do not require involvement from the hypervisor. While we propose that sampling and queuing ports are realized by two staged shared memory, immediate involvement of the hypervisor is not required. According to ARINC 653 part 1, synchronous behaviour of ports is not required, allowing us to handle inter-partition communication in between ptws[6]. Processes do not require direct communication with the hypervisor either. As previously mentioned, *cgroups* with the ability for a process

to freeze itself are enough for providing everything which is expected. Many of the remaining APEX services are solely for requesting information from the hypervisor. Through shared memory, information for service requests like `GET_PARTITION_STATUS` and `GET_TIME` can always be provided. `CREATE_SAMPLING_PORT` and `CREATE_QUEUING_PORT` only succeed if they exactly match the configuration table. Because of this, all ports along with their shared memory can always be provided to partitions, which can then utilize them on their own.

UNIX SOCKETS For service requests, requiring immediate acting of the hypervisor, Unix sockets are utilized. Through them the requests for changing the partition mode, reporting a message and raising an error are made. During the partitions time window, the hypervisor waits for messages and acts on them if any arrive. Partition mode change requests start the transition progress, message reports are written to stdout and errors are passed to the healthmonitor.

6.4 EVALUATION

With this effort a first attempt at developing a prototype of a Linux native hypervisor is made. Aiming to be fully compliant to the ARINC 653 part 4 standard, we set a bold goal. To get an initial feedback regarding the achievement of said goal, we conduct some work towards evaluating our prototype.

6.4.1 *Is it up to the promise?*

TEMPORAL ISOLATION The partition binaries are started directly in their own *cgroup* and *cgroup* namespace. Since the capability to freeze exists within every *cgroup*, partition time frames are enforced. Further, we employ additional child *cgroups* inside each partition's *cgroup* to realize intra-partition scheduling of APEX processes. By also trapping partitions in individual *cgroup* namespaces, escape from their *cgroup* should not be possible as long as the implementation of *cgroups* in the Linux kernel is sound. Considering that the ARINC 653 part 4 standard expects only a single core, the *cgroups* cpuset controller was not utilized in the hypervisor itself. Besides, setting up a *cgroup* with

cpuset controller usually requires privileges and hence should be done prior to running the hypervisor if needed.

SPATIAL ISOLATION For a first prototype we opted for primarily focusing on scheduling, because it marks the bare minimum required for running and testing partitions and modules. As a consequence, spatial isolation is still lacking. While we place each partition in an isolated mount namespace and attempt to unmount all file systems but an individually mounted *tmpfs*, some issues remain. After the applied `pivot_root` syscall, which is used for swapping the old root file system for the *tmpfs*, we were unable to unmount the old root file system completely. We mainly attribute this to bind mounts used for getting access to `/dev/null` and the partition binary, but given the relative insignificance for a first prototype we did not pursue resolution of this further. With an already broken spatial isolation we did not attempt to limit resource consumption just yet either.

DEPENDENCY AND PRIVILEGE-FREE Considering the previously mentioned capabilities were all provided by the Linux kernel itself, no runtime dependencies are required to implement said behaviour. Further on, an unmodified Linux Kernel is perfectly sufficient to execute our hypervisor, but the idea of using the RT PREEMPT patches lends itself if low jitter real-time behaviour is desired. With only the *cgroups* cpuset controller requiring privileges at the initial start of the hypervisor, running the hypervisor does not require escalated privileges. This ensures a frictionless experience during development. Should exclusive allocation of a CPU core to the hypervisor be desired, a privileged user can manually create a *cgroup* limited to one core and pass it to the hypervisor config.

ARINC 653 PART 4 The most important goal of the hypervisor is implementation of the APEX API services defined in ARINC 653 part 4. Again, serving as a first prototype, our hypervisor does not fully satisfy this goal either. The two service groups missing are queuing ports and health monitoring. While we propose a concept for implementing queuing ports with Linux primitives, for demonstrating inter-partition communication, sampling ports suffice for now. Likewise, we come up with means of sending intra partition errors via Unix sockets to the

hypervisor. For other errors, the hypervisor could observe the partition processes execution for termination, deadline misses and non-compliance with the proposed handling of ports. Due to architectural design flaws in the proof of concept, integration of the health monitor for handling errors proves complicated at the moment, therefore it is postponed for the time being. Lastly, the lack of a C ABI for APEX API services prevents our hypervisor from being interfaced by C-based partitions.

6.4.2 *Comparison to Existing Solutions*

Attempting to be the jack of all trades for ARINC 653 applications, the Linux hypervisor tries to be a development tool as well as a type-2 hypervisor. Considering spatial isolation marks a hard requirement for ARINC 653 compliance, with our Linux hypervisor not providing said requirement, usage for safety critical systems would be ill-advised[5]. Because of this, comparison to fully ARINC 653 compliant hypervisors also becomes unnecessary. Leaving us with the comparison to development focused solutions - SKE and AMOBA[31].

In terms of completeness regarding provided APEX API functionalities, the Linux hypervisor is also inferior to SKE and AMOBA. Both SKE and AMOBA offer most ARINC 653 part 1 services and the Linux hypervisor does not even implement all services required by ARINC 653 part 4. With this being all disadvantages we know of, the Linux hypervisor has one strong advantage over both of its competitors, it realizes temporal isolation. The SKE simulator for example demands cooperation of its partitions to pause themselves. Here a partition would be thinkable which never pauses, effectively stalling the simulation. This makes our solution particularly interesting to simulate situations in which some partitions misbehave while other partitions are expected to keep operational (degraded operation). AMOBA on the other hand only allows for testing and verification of single partitions, since the POSIX API which it depends on does not provide scheduling. This means that with AMOBA we do not even gain the capability to test multiple partitions in a module simultaneously[31].

Out of these two development tools, SKE is the closest to our Linux hypervisor. Both do not require any dependencies other

than a Linux OS to run on and both can be executed with little effort in Continuous Integration (CI) systems like GitHub actions.

7 AVIONIC PLATFORM

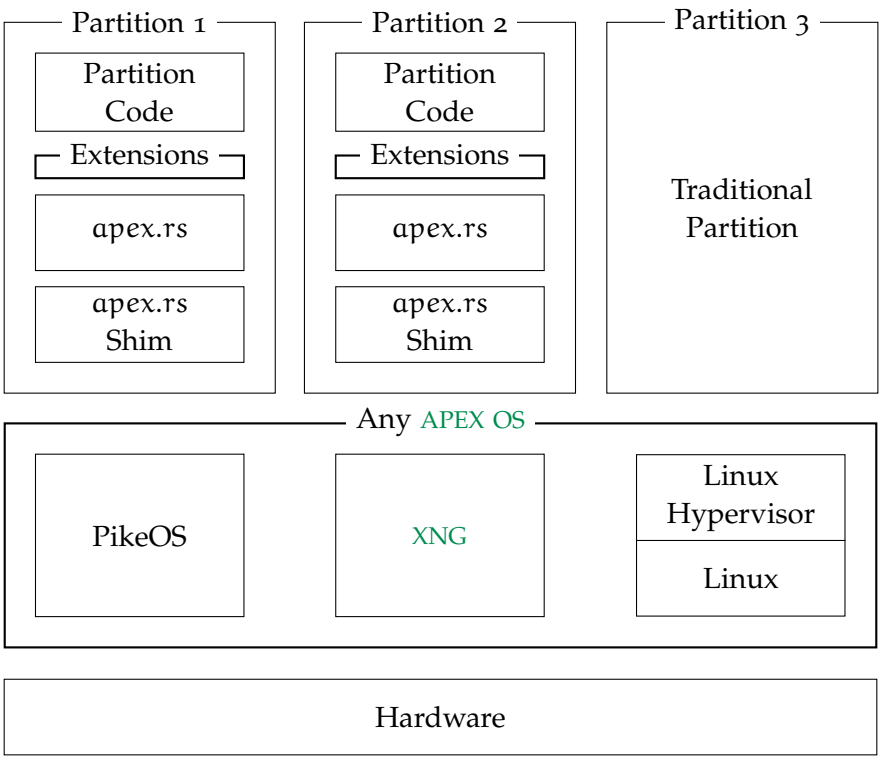
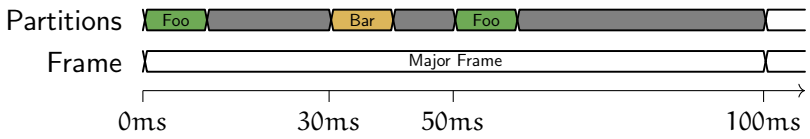


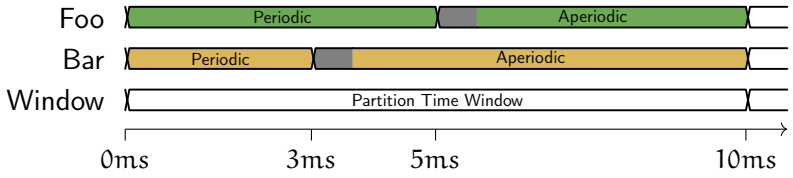
Figure 7.1: System Architecture using apex.rs

With the extendable and safety hardened APEX API developed in Chapter 5 as well as the Linux native hypervisor introduced in Chapter 6, we acquired all necessary pieces of our new avionic platform. For demonstrating its flexibility we will now develop an APEX module with two partitions. The partitions will utilize our sample extension, which adds functions to all ports for sending and receiving *postcard* serialized data. Afterwards we will compile the partitions for our Linux hypervisor and the XNG hypervisor. By executing them on both hypervisors we exercise the flexibility of this platform. Lastly, we will evaluate the usability along with further benefits presented by the platform.

7.1 DEMONSTRATOR MODULE



(a) Major Frame



(b) Partition Time Window

Figure 7.2: Demonstrator Scheduling
Foo:= offset: 0ms, duration: 10ms, period: 50ms
Bar:= offset: 30ms, duration: 10ms, period: 100ms

The two partitions *Foo* and *Bar* are created for our demonstrator module. Additionally, the sampling port *Hello* is declared, with *Foo* as the source and *Bar* as the destination. The static schedule for our demonstrator module is defined analogous to the schedule shown in Fig. 7.2a. For reasons of clarity, the Rust code for the partitions is not shown in this chapter but in the appendix (Listing C.1 and Listing C.2).

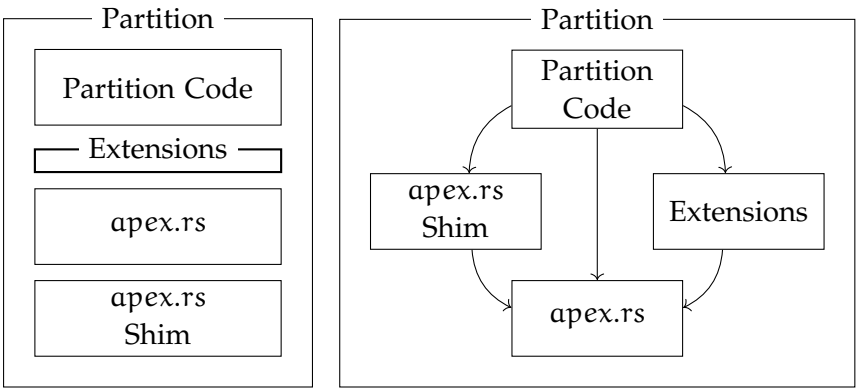
PARTITION FOO As it can be seen in Fig. 7.2b, the *Foo* partition introduces both a foreground (periodic) and a background (aperiodic) process. These processes report a message through APEXs `REPORT_APPLICATION_MESSAGE` service every millisecond, in which a number is incremented. After five reported messages (> 5ms), the foreground process writes a struct to the sampling port, utilizing our *postcard* extension for APEX channels. When the message is written, the foreground process waits for the next maf and hence allows the background process to be scheduled.

PARTITION BAR Just like the *Foo* partition, the processes of the *Bar* partition also report messages every millisecond, incrementing a number. After only three reported messages (> 3ms), the foreground process attempts to read and deserialize a message from the *Hello* sampling port, using our *postcard*

extension. Also reporting this message to the hypervisor via the `REPORT_APPLICATION_MESSAGE` service, the foreground process then waits for the next maf.

Because temporal isolation is not expected between processes of a partition, the actual start of the background process (aperiodic) is marked as uncertain in Fig. 7.2b.

7.2 HYPERVISOR INDEPENDENCE



(a) Architecture (b) Library Dependencies

Figure 7.3: Comparison of Partition Architecture and Dependencies

Effortless interchangeability of the hypervisor along with completely hypervisor independent extensibility are requirements towards our platform. Through them, a flourishing ecosystem surrounding ARINC 653 may be constructed, lowering the burden for newcomers to the avionic market. To demonstrate the existence of these qualities in our proposed platform, we will execute our example module from Sec. 7.1 on our Linux hypervisor as well as the proprietary XNG hypervisor running on a Xilinx SoC. Considering the architectural structure of partitions in Fig. 7.3a, everything in a partition is build on top of the apex.rs shim. However, contrary to the architectural structure, only the partition code truly depends on the apex.rs shim. Hence, compiling a partition for various hypervisors should be as easy as swapping the shim for another one, as long as the partition does not depend on hypervisor specific services.

7.2.1 *Linux Hypervisor*

Because apex.rs only allows for hypervisor independent partition development, the hypervisor themselves must be individually configured for partitions, channels and schedule. As a result of the simplicity of the Linux hypervisor, the configuration is kept short and can be seen in Fig. 7.4a.

```
major_frame: 100ms
partitions:
  - id: 0
    name: Foo
    duration: 10ms
    offset: 0ms
    period: 50ms
    image: foo_part
  - id: 1
    name: Bar
    offset: 30ms
    duration: 10ms
    image: bar_part
    period: 100ms
channel:
  - !Sampling
    name: Hello
    msg_size: 10KB
    source: Foo
    destination:
      - Bar
```

(a) Config

```
Partition: Foo > "200ms": P MSG 16
Partition: Foo > "201ms": P MSG 17
Partition: Foo > "202ms": P MSG 18
Partition: Foo > "203ms": P MSG 19
Partition: Foo > "204ms": P MSG 20
Partition: Foo > "205ms": AP MSG 16
Partition: Foo > "206ms": AP MSG 17
Partition: Foo > "207ms": AP MSG 18
Partition: Foo > "208ms": AP MSG 19
Partition: Foo > "209ms": AP MSG 20
Partition: Bar > "230ms": P MSG 6
Partition: Bar > "231ms": P MSG 7
Partition: Bar > "232ms": P MSG 8
Partition: Bar > "233ms": P MSG 9
Partition: Bar > "234ms": P MSG 10
Partition: Bar > "235ms":
    Received via Sampling Port:
        CustomMessage {
            msg: "Sampling MSG 4",
            when: 205.708074ms },
        valid: Valid }
Partition: Bar > "235ms": AP MSG 4
Partition: Bar > "236ms": AP MSG 5
```

(b) Log

Figure 7.4: Demonstrator on the Linux Hypervisor
Foo Partition Bar Partition

Fig. 7.4b shows part of the output from the Linux hypervisor when running our demonstrator module. Here the aperiodic and periodic processes report messages as expected. Furthermore, the *Bar* partition successfully read and deserialized the custom message from the sampling port using our extension library.

CONTINUES INTEGRATION With testing being an anticipated use-case of our Linux hypervisor, we also attempted to run the demonstrator module inside GitHub's CI system - GitHub Actions. For this we designed a workflow, which builds both partitions, in addition to the hypervisor and then runs the Linux hypervisor with the aforementioned configuration. Running on a GitHub hosted Ubuntu, the workflow successfully executed

from the first try. Throughout this, the only dependency was a Rust toolchain that was solely used for compiling partitions and hypervisor. Afterwards the CI run hypervisor executed the partitions as expected with an output similar to Fig. 7.4b.

7.2.2 *XNG*

```
"200ms": P MSG 16
"201ms": P MSG 17
"202ms": P MSG 18
"203ms": P MSG 19
"204ms": P MSG 20
"205ms": AP MSG 16
"206ms": AP MSG 17
"207ms": AP MSG 18
"208ms": AP MSG 19
"209ms": AP MSG 20
"251ms": P MSG 21
"252ms": P MSG 22
"253ms": P MSG 23
```

```
"230ms": P MSG 6
"231ms": P MSG 7
"232ms": P MSG 8
"233ms": P MSG 9
"234ms": P MSG 10
"234ms":
    Received via Sampling Port:
       CustomMessage {
          msg: "Sampling MSG 4",
          when: 205.328572ms },
       valid: Valid }
"235ms": AP MSG 4
"236ms": AP MSG 5
```

Figure 7.5: XNG Log Snippet running the Demonstrator
Foo Partition Bar Partition

For running our demonstrator partitions on the XNG hypervisor, a Xilinx SoC was used as the host device. After compiling the partitions for the target platform, an ELF file including the hypervisor itself was build and executed. The output at the UART of the SoC can be seen in Fig. 7.5. Because the name of the message reporting partition is not printed here, the output was coloured for indicating the message source. A small evidence for out claim of hypervisor independence is the output of said execution; except for little timing differences, the output is equivalent to Fig. 7.4b.

7.3 EVALUATION

With the execution of our demonstrator module on both the Linux hypervisor and XNG we showed that hypervisor independence was achieved to some degree. Still, the caveat remains that partitions need to be compiled for every target hypervisor with the corresponding *apex.rs* shim. An issue we did not encounter

with our simple example is that ARINC 653 compliant hypervisors are allowed to provide additional hypervisor specific functions to partitions and processes[6]. Because of this, partitions can be further bound to a particular hypervisor, should these functions be provided by a shim and be used by the partition.

Nevertheless, even if a partition may be bound to a hypervisor due to exclusive functionality, `apex.rs` and its trait remain extendable, since extensions do not depend on the shim at all (Fig. 7.3b). The two examples demonstrate that the serialization extension actually works; the custom message type could be serialized, send and deserialized in the second partition.

Another takeaway of our demonstrator is that compilation along with execution of the Linux hypervisor and partitions works with Continuous Integration (CI) systems. While we mentioned in Sec. 6.4.1 that the Linux hypervisor is still lacking in safety, it can be used for testing of modules and partitions. Solely requiring a Linux OS with no reliance on proprietary hardware and software, multiple instances of a module can run in parallel on a powerful system. Doing this, partitions could be automatically tested in various configurations, ultimately enabling DevOps for ARINC 653 partition/module development[3].

8

CONCLUSION AND FUTURE WORK

Starting with the implicit question of how to improve the developer experience in avionics, we choose to propose a modern development platform for ARINC 653 in this thesis. Establishing a baseline of essential knowledge, we first introduced ARINC 653 and the Rust programming language in Chapter 2. In consideration of our platforms primary goal of assisting with the development process and us designing our own hypervisor for testing purposed too, we mention various existing hypervisors in Chapter 3. With this, we especially focus on hypervisors that run under Linux and are designed for development of ARINC 653 applications as well. In Chapter 4, we then looked at already established software ecosystems to understand factors involved in a succeeding ecosystem surrounding ARINC 653. Following this, issues potentially preventing the APEX API from growing more as an ecosystem including safety issues are enumerated. With *apex.rs* our Rust native APEX API is then introduced in Chapter 5, distinguishing itself through extendability and improved safety. Built upon this API an example extension is also developed, extending APEX channels by allowing to send and receiving custom data types. Chapter 6 then shows our Linux hypervisor, which was intended not only as a tool for development but also as a full-fledged hypervisor for low-DAL systems. As a result of inadequate isolation, the hypervisor can not be used as a hypervisor for live systems just yet, but already achieves the primary goal of improving the development and testing process. In Chapter 7, the platform, with *apex.rs*, extensions and the Linux hypervisor was then put to the test. Successfully execution of an *apex.rs* based ARINC 653 module, featuring two partitions communicating with each other by using our APEX channel extension, proofed applicability. Especially, portability of the APEX API has been demonstrated by execution of the same ARINC 653 module on our Linux hypervisor as well as the proprietary XNG hypervisor with minimal changes.

8.1 FUTURE WORK

LINUX HYPERVISOR Running APEX partitions, which only require a subset of ARINC 653 part 4, is already possible with our hypervisor. As a next step, the missing ARINC 653 part 4 services for queuing ports and health monitoring should be implemented.

Further, we mentioned in Sec. 6.1 that we intend for our Linux hypervisor to feature actual temporal and spatial isolation. For achieving this, our encountered problems regarding spatial isolation need to be addressed as well. Moreover, scheduling of intra-partition processes may not be safe yet, because the *cgroups* are partially exposed to the internals of the partitions. Aiming for applicability in low DAL systems, lessons should be learned from other Linux based isolation domains, for example from OCI container.

Another future goal is allowing access to devices and the network. In ARINC 653 part 2, the memory block service is introduced as an extended service offering access to named write and read-enabled memory ranges[7]. While this service should be made available for memory mappings to devices (including PCI devices like network cards), advantages gained from Linux should be incorporated as well. Running on top of Linux, the operating system can already grant us device drivers as well as a working network stack. Offering device and network access via hypervisor implementation dependent services could be a valid solution to accomplish this.

One last goal is automatic ARINC 653 compliance testing. Previously mentioned, the Linux hypervisor is still missing essential services for ARINC 653 part 4, but we aspire part 1 compliance. As part of the ARINC 653 standard, part 3a[9] and 3b[10] respectively specify conformity tests for ARINC 653 part 1 and 2. Due to our Linux hypervisor being compilable and executable with CI tools, we fancy the idea of automatic conformity testing of our hypervisor. All tests defined in ARINC 653 part 3a and 3b could be automatically run after commits to our version control system, allowing for live updates on attained ARINC 653 conformity. This would also form a nice demonstration of applied DevOps for avionics software engineering.

PLATFORM Considering the focus of our platform lies on apex.rs, as the pivot of an extendable ecosystem for ARINC 653 compliant application development, we now plan to promote its usage through the open-source community. Moreover, complementary to apex.rs, the Linux hypervisor can serve as a shortcut into ARINC 653 application development, rewarding adoption of our platform. First, improving on the documentation for partition and shim development, we want to make apex.rs more accessible. Also, we now intend to put the platform to the test with an ongoing project that requires partitioning in the sense of ARINC 653. Through the time working on this project, we hope to mature the proposed platform while also adding to the apex.rs ecosystem by developing extensions for it.

BIBLIOGRAPHY

[1] C. J. Guerra, N. E. Carmichael and J. T. Nielson. 'Aircraft avionics strategic fleet update using optimal methods'. In: *2016 IEEE Aerospace Conference*. IEEE. 2016, pp. 1–5.

[2] J. Marsden, A. Windisch, R. Mayo, J. Grossi, J. Villermin, L. Fabre and C. Aventini. 'ED-12C/DO-178C vs. Agile Manifesto–A Solution to Agile Development of Certifiable Avionics Systems'. In: *ERTS 2018*. 2018.

[3] W. Zaeske and U. Durak. *DevOps for Airborne Software: Exploring Modern Approaches*. Springer Nature, 2022.

[4] Air Force Life Cycle Management Center. *Software innovations makes F-16 more capable*. 2020. URL: https://www.eglin.af.mil/News/Article-Display/Article/2169523/software-innovations-makes-f-16-more-capable/.

[5] ARINC. *Avionics Application Software Standard Interface Part 0 Overview of Arinc 653*. Standard. Bowie, MD, USA: Aeronautical Radio Incorporated, Aug. 2019.

[6] ARINC. *Avionics Application Software Standard Interface Part 1 Required Services*. Standard. Bowie, MD, USA: Aeronautical Radio Incorporated, Dec. 2019.

[7] ARINC. *Avionics Application Software Standard Interface Part 2 Extended Services*. Standard. Bowie, MD, USA: Aeronautical Radio Incorporated, Dec. 2019.

[8] ARINC. *Avionics Application Software Standard Interface Part 4 Subset Services*. Standard. Bowie, MD, USA: Aeronautical Radio Incorporated, Dec. 2012.

[9] ARINC. *Avionics Application Software Standard Interface Part 3A Conformity Test Specification for ARINC 653 Required Services*. Standard. Bowie, MD, USA: Aeronautical Radio Incorporated, July 2019.

[10] ARINC. *Avionics Application Software Standard Interface Part 3B Conformity Test Specification for ARINC 653 Extended Services*. Standard. Bowie, MD, USA: Aeronautical Radio Incorporated, July 2019.

[11] N.-J. Wessman, F. Malatesta, J. Andersson, P. Gomez, M. Masmano, V. Nicolau, J. Le Rhun, G. Cabo, F. Bas, R. Lorenzo et al. 'De-RISC: the first RISC-V space-grade platform for safety-critical systems'. In: *2021 IEEE Space Computing Conference (SCC)*. IEEE. 2021, pp. 17–26.

[12] B. Game, P. Computing and S. Lyu. 'Practical Rust Projects'. In: (2020).

[13] Q. Ochem and F. Gilcher. *AdaCore and Ferrous Systems Joining Forces to Support Rust*. 2022. URL: https://blog.adacore.com/adacore-and-ferrous-systems-joining-forces-to-support-rust.

[14] A. Pinho, L. Couto and J. Oliveira. 'Towards rust for critical systems'. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. 2019, pp. 19–24.

[15] EUROCAE. *Software Considerations in Ariborne Systems and Equipment Certifications*. Standard. Malakoff, France: European Organisation for Civil Aviation Equipment, Jan. 2012.

[16] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[17] A. K. Beingessner. 'You Can't Spell Trust Without Rust'. PhD thesis. Carleton University, 2016.

[18] W. Crichton. 'The usability of ownership'. In: *arXiv preprint arXiv:2011.06171* (2020).

[19] J. A. Vagedes. *A Study of Execution Performance for Rust-Based Object vs Data Oriented Architectures*. Tech. rep. Air Force Institute Of Technology Wright-Patterson AFB United States, 2020.

[20] W. Crichton. 'Type-Driven API Design in Rust'. Stange Loop Conference. 2021. URL: https://thestrangeloop.com/2021/type-driven-api-design-in-rust.html.

[21] B. Lüttig. 'Selective Middleware - an Approach Towards a Stepwise Integration of a Fault-Tolerant, Distributed Avionics-Platform for Applications in Large Aircraft Using Integrated Modular Avionics'. PhD thesis. University of Stuttgart, 2022.

[22] D. T. Vojnak, B. S. Đorđević, V. V. Timčenko and S. M. Štrbac. 'Performance Comparison of the type-2 hypervisor VirtualBox and VMWare Workstation'. In: *2019 27th Telecommunications Forum (TELFOR)*. IEEE. 2019, pp. 1–4.

[23] W. Steiner and S. Poledna. 'Fog computing as enabler for the Industrial Internet of Things'. In: *e & i Elektrotechnik und Informationstechnik* 133.7 (2016), pp. 310–314.

[24] M. Masmano, I. Ripoll, A. Crespo and J Metge. 'Xtratum: a hypervisor for safety critical embedded systems'. In: *11th Real-Time Linux Workshop*. Citeseer. 2009, pp. 263–272.

[25] *Software User Manual: SKE*. Fent Innovative Software Solutions. 2020.

[26] SYSGO GmbH. *PikeOS 5.1 - Certified RTOS with Hypervisor Functionality*. Rel. 1.3. Nov. 2022.

[27] Wind River. *VxWorks 653: Multi-core Edition*. Rev 11/2022. Nov. 2022.

[28] R. Zhou, Q. Zhou, Y. Sheng and K.-C. Li. 'XtratuM/PPC: a hypervisor for partitioned system on PowerPC processors'. In: *The Journal of Supercomputing* 63.2 (2013), pp. 593–610.

[29] S. Goiffon and P. Gaufillet. 'Linux: A multi-purpose executive support for civil avionics applications?' In: *Building the Information Society*. Springer, 2004, pp. 719–724.

[30] S. Santos, J. Rufino, T. Schoofs, C. Tatibana and J. Windsor. 'A portable ARINC 653 standard interface'. In: *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE. 2008, 1–E.

[31] E. Pascoal, J. Rufino, T. Schoofs and J. Windsor. 'AMOBA-ARINC 653 simulator for modular based space applications'. In: *emergency* 10 (2008), p. 2.

[32] A. Certain. 'Enabling Linux Usage in Space Applications'. Embedded Linux Conference. 2019. URL: https://elinux.org/Applications_Presentations.

[33] C. Rebischke. 'From the Cloud to the Clouds: Taking Integrated Modular Avionics on a New Level with Cloud-Native Technologies'. MA thesis. TU Clausthal, 2022.

[34] A. Farrukh and R. West. 'FLYOS: Integrated Modular Avionics for Autonomous Multicopters'. In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2022, pp. 68–81.

[35] S. Han and H.-W. Jin. 'Full virtualization based ARINC 653 partitioning'. In: *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*. IEEE. 2011, 7E1–1.

[36] S. Han and H.-W. Jin. 'Kernel-level ARINC 653 partitioning for Linux'. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. 2012, pp. 1632–1637.

[37] W. Ruan and Z. Zhai. 'Kernel-level design to support partitioning and hierarchical real-time scheduling of ARINC 653 for VxWorks'. In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. IEEE. 2014, pp. 388–393.

[38] *MISRA C:2012 Guidlines for the use of the C language in critical systems*. The Motor Industry Software Reliability Association. 2013.

[39] S. Newton. *MISRA-Rust*. Version caa5e5b. July 2018. URL: https://github.com/PolySync/misra-rust.

[40] P. Mansanet. *Tightness Driven Development in Rust*. 2021. URL: https://www.ecorax.net/tightness/.

[41] C. Koster. *Invariant Driven Development*. 2018. URL: https://medium.com/statuscode/invariant-driven-development-8231add95e33.

[42] A Cook. 'ARINC 653—challenges of the present and future'. In: *Microprocessors and Microsystems* 19.10 (1995), pp. 575–579.

[43] A. Poulsen, C. McCall, G. Vasluianu, J. Jensen and J. Cordeiro. *An OS implementation based on the ARINC 653 Standard*. Tech. rep. Aalborg University, 2016.

[44] A. Dubey, G. Karsal and N. Mahadevan. *ARINC 653 simulator*. Version 9928113. July 2019. URL: https://github.com/adubey14/arinc653emulator.

[45] opencontainers. *Image Format Specification*. Version 494a5a6aca. July 2022. URL: https://github.com/opencontainers/runtime-spec/blob/main/spec.md.

[46] opencontainers. *Image Format Specification*. Version a7ac485f4c. Sept. 2022. URL: https://github.com/opencontainers/image-spec/blob/main/spec.md.

[47] A. Cook and K. Hunt. 'ARINC 653—Achieving software re-use'. In: *Microprocessors and Microsystems* 20.8 (1997), pp. 479–483.

[48] R. Munroe. *Standards*. 2011. URL: https://xkcd.com/927/.

[49] I. Borate and R. Chavan. 'Sandboxing in linux: From smartphone to cloud'. In: *International Journal of Computer Applications* 148.8 (2016).

A

apex.rs

Listing A.1: apex-rs types.rs

```rust
pub mod basic {
    /// Max Length for Name Types
    ///
    /// According to ARINC653-P1, the maximum name length is always 32
    pub const MAX_NAME_LENGTH: usize = 32;
    /// C compatible function type
    pub type SystemAddress = extern "C" fn();
    /// Apex internal ReturnCode Type
    pub type ReturnCode = u32;
    /// Apex Name type using [MAX_NAME_LENGTH]
    pub type ApexName = [u8; MAX_NAME_LENGTH];

    // Base Types
    /// Apex Byte Type: 8-bit, 0..255
    pub type ApexByte = u8;
    /// Apex Integer Type: 32-bit, -2^31..2^31-1
    pub type ApexInteger = i32;
    /// Apex Unsigned Type: 32-bit, 0..4_294_967_295
    pub type ApexUnsigned = u32;
    /// Apex Long Integer Type: 64-bit: -2^63..2^63-1
    pub type ApexLongInteger = i64;

    /// Apex Message Size type: [ApexUnsigned]
    pub type MessageSize = ApexUnsigned;
    /// Apex Message Range type: [ApexUnsigned]
    pub type MessageRange = ApexUnsigned;

    /// APEX Error Return Code
    ///
    /// Basically the normal APEX Return Codes without the non-error variant
    #[repr(u32)]
    #[derive(Copy, Clone, Debug, PartialEq, Eq)]
```

```rust
    #[cfg_attr(feature = "serde", derive(serde::Serialize,
        serde::Deserialize))]
    #[cfg_attr(feature = "strum", derive(strum::FromRepr))]
    pub enum ErrorReturnCode {
        /// status of system unaffected by request
        NoAction = 1,
        /// resource required by request unavailable
        NotAvailable = 2,
        /// invalid parameter specified in request
        InvalidParam = 3,
        /// parameter incompatible with configuration
        InvalidConfig = 4,
        /// request incompatible with current mode
        InvalidMode = 5,
        /// time-out tied up with request has expired
        TimedOut = 6,
    }

    impl ErrorReturnCode {
        /// Convenience function for gaining a Result from a
            given [ReturnCode]
        ///
        /// # Return Values for given [ReturnCode]
        ///
        /// - `0` => `Ok(())`
        /// - `1..=6` => `Err(Self)`
        /// - `7..` => `panic`
        pub fn from(from: ReturnCode) -> Result<(), Self> {
            use ErrorReturnCode::*;
            match from {
                1 => Ok(()),
                2 => Err(NoAction),
                3 => Err(NotAvailable),
                4 => Err(InvalidParam),
                5 => Err(InvalidConfig),
                6 => Err(InvalidMode),
                unexpected => panic!("{unexpected}"),
            }
        }
    }

    /// Port Directions
    #[repr(u32)]
    #[derive(Debug, Copy, Clone, PartialEq, Eq)]
```

```rust
#[cfg_attr(feature = "serde", derive(serde::Serialize, serde::Deserialize))]
#[cfg_attr(feature = "strum", derive(strum::FromRepr))]
pub enum PortDirection {
    /// Source/Sender Port
    Source = 0,
    /// Destination/Receiver Port
    Destination = 1,
}

impl TryFrom<ApexUnsigned> for PortDirection {
    type Error = ApexUnsigned;

    fn try_from(value: ApexUnsigned) -> Result<Self, Self::Error> {
        match value {
            0 => Ok(PortDirection::Source),
            1 => Ok(PortDirection::Destination),
            _ => Err(value),
        }
    }
}

/// Queuing Disciplines
#[repr(u32)]
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
#[cfg_attr(feature = "serde", derive(serde::Serialize, serde::Deserialize))]
#[cfg_attr(feature = "strum", derive(strum::FromRepr))]
pub enum QueuingDiscipline {
    /// First in/first out queue
    FIFO = 0,
    /// Priority queue
    Priority = 1,
}

impl TryFrom<ApexUnsigned> for QueuingDiscipline {
    type Error = ApexUnsigned;

    fn try_from(value: ApexUnsigned) -> Result<Self, Self::Error> {
        match value {
            0 => Ok(QueuingDiscipline::FIFO),
            1 => Ok(QueuingDiscipline::Priority),
```

```
                _ => Err(value),
            }
        }
    }

    /// Apex SystemTime Type: [ApexLongInteger]
    pub type ApexSystemTime = ApexLongInteger;
    /// [ApexSystemTime] value indicating infinite time
    pub const INFINITE_TIME_VALUE: ApexSystemTime = -1;

    /// ProcessorCore Id Type: [ApexInteger]
    pub type ProcessorCoreId = ApexInteger;
    /// [ProcessorCoreId] value indicating no preference
    pub const CORE_AFFINITY_NO_PREFERENCE: ProcessorCoreId =
        -1;
}

pub mod abstraction {
    use core::panic;
    use core::str::{FromStr, Utf8Error};
    use core::time::Duration;

    // Reexport important basic-types for downstream-user
    pub use super::basic::{ApexByte, ApexUnsigned,
        MessageSize, MAX_NAME_LENGTH};
    use crate::bindings::*;

    /// Error Type used by abstracted functions.
    /// Includes all Variants of [ErrorReturnCode] plus a [
        WriteError] and [ReadError] variant
    #[derive(Clone, Debug, PartialEq, Eq)]
    #[cfg_attr(feature = "serde", derive(serde::Serialize,
        serde::Deserialize))]
    pub enum Error {
        /// status of system unaffected by request
        NoAction,
        /// resource required by request unavailable
        NotAvailable,
        /// invalid parameter specified in request
        InvalidParam,
        /// parameter incompatible with configuration
        InvalidConfig,
        /// request incompatible with current mode
        InvalidMode,
```

```
155         /// time-out tied up with request has expired
156         TimedOut,
157         /// buffer got zero length or is to long
158         WriteError(WriteError),
159         /// buffer is to small
160         ReadError(ReadError),
161     }
162
163     impl From<ErrorReturnCode> for Error {
164         fn from(rc: ErrorReturnCode) -> Self {
165             use Error::*;
166             match rc {
167                 ErrorReturnCode::NoAction => NoAction,
168                 ErrorReturnCode::NotAvailable => NotAvailable
                        ,
169                 ErrorReturnCode::InvalidParam => InvalidParam
                        ,
170                 ErrorReturnCode::InvalidConfig =>
                        InvalidConfig,
171                 ErrorReturnCode::InvalidMode => InvalidMode,
172                 ErrorReturnCode::TimedOut => TimedOut,
173             }
174         }
175     }
176
177     /// Abstracted SystemTime Variant making use of Rusts [
            Duration]
178     /// Includes Infinite-variant since [Duration] does not
            allow for negative values
179     ///
180     /// # Size
181     ///
182     /// [ApexSystemTime] => 8-Byte
183     /// [Duration] => 16-Byte
184     /// [SystemTime] => 24-Byte
185     #[repr(C)]
186     #[derive(Clone, Debug, PartialEq, Eq)]
187     #[cfg_attr(feature = "serde", derive(serde::Serialize,
            serde::Deserialize))]
188     pub enum SystemTime {
189         /// Infinite Time value
190         Infinite,
191         /// Normal positive Time value
192         Normal(Duration),
```

```
193     }
194
195     impl SystemTime {
196         /// Create new SystemTime from given [ApexSystemTime]
197         pub fn new(time: ApexSystemTime) -> Self {
198             time.into()
199         }
200
201         /// Returns Durations if this SystemTime is
                SystemTime::Normal
202         ///
203         /// Otherwise panics
204         pub fn unwrap_duration(self) -> Duration {
205             if let SystemTime::Normal(time) = self {
206                 return time;
207             }
208             panic!("Was Infinite")
209         }
210     }
211
212     impl From<Duration> for SystemTime {
213         fn from(time: Duration) -> Self {
214             Self::Normal(time)
215         }
216     }
217
218     impl From<SystemTime> for Option<Duration> {
219         fn from(time: SystemTime) -> Self {
220             match time {
221                 SystemTime::Infinite => None,
222                 SystemTime::Normal(time) => Some(time),
223             }
224         }
225     }
226
227     impl From<Option<Duration>> for SystemTime {
228         fn from(time: Option<Duration>) -> Self {
229             use SystemTime::*;
230             match time {
231                 Some(time) => Normal(time),
232                 None => Infinite,
233             }
234         }
235     }
```

```rust

impl From<ApexSystemTime> for SystemTime {
    /// Converts ApexSystemTime to a [SystemTime]
    /// Should ApexSystemTime be less than 0, its considered to be infinite.
    /// As stated in ARINC653P1-5 3.4.1 all negative values should be treated as ‘INFINITE_TIME_VALUE‘
    fn from(time: ApexSystemTime) -> Self {
        use SystemTime::*;
        // This conversion can only fail, if ApexSystemTime is negative.
        match u64::try_from(time) {
            Ok(time) => Normal(Duration::from_nanos(time)),
            Err(_) => Infinite,
        }
    }
}

impl From<SystemTime> for ApexSystemTime {
    /// Converts [SystemTime] into [ApexSystemTime]
    fn from(time: SystemTime) -> Self {
        if let SystemTime::Normal(time) = time {
            if let Ok(time) = ApexSystemTime::try_from(time.as_nanos()) {
                return time;
            }
        }
        INFINITE_TIME_VALUE
    }
}

/// Convenient Abstraction Name Type
/// Uses [ApexName] internally
#[derive(Clone, Debug, PartialEq, Eq)]
#[cfg_attr(feature = "serde", derive(serde::Serialize, serde::Deserialize))]
pub struct Name(ApexName);

impl Name {
    /// Create new [Name] from [ApexName]
    pub fn new(name: ApexName) -> Self {
        Name(name)
    }
```

```
274
275         /// Get [str] from this
276         pub fn to_str(&self) -> Result<&str, Utf8Error> {
277             let nul_range_end = self
278                 .0
279                 .iter()
280                 .position(|&c| c == b'\0')
281                 .unwrap_or(self.0.len());
282             core::str::from_utf8(&self.0[0..nul_range_end])
283         }
284
285         /// Dismantle this to its inner [ApexName] type
286         pub fn into_inner(self) -> ApexName {
287             self.0
288         }
289     }
290
291     impl From<Name> for ApexName {
292         fn from(val: Name) -> Self {
293             val.0
294         }
295     }
296
297     impl FromStr for Name {
298         type Err = ApexUnsigned;
299
300         fn from_str(s: &str) -> Result<Self, Self::Err> {
301             if s.len() > MAX_NAME_LENGTH {
302                 return Err(s.len() as ApexUnsigned);
303             }
304             let mut array_name = [0; MAX_NAME_LENGTH as usize
                    ];
305             array_name[..s.len()].copy_from_slice(s.as_bytes
                    ());
306             Ok(Self(array_name))
307         }
308     }
309
310     /// Read Error indicating that the buffer was not
            guaranteed to fit a payload on a given read operation.
311     #[derive(Clone, Debug, PartialEq, Eq)]
312     #[cfg_attr(feature = "serde", derive(serde::Serialize,
            serde::Deserialize))]
313     pub struct ReadError(usize);
```

```
314
315     impl ReadError {
316         /// Validate a buffer to fit at least a given size.
317         /// If not returns [Self] with the length of the
                passed buffer
318         pub fn validate(
319             size: MessageSize,
320             buffer: &mut [ApexByte],
321         ) -> Result<&mut [ApexByte], Self> {
322             if usize::try_from(size)
323                 .map(|ss| buffer.len() < ss)
324                 .unwrap_or(true)
325             {
326                 return Err(Self(buffer.len()));
327             }
328             Ok(buffer)
329         }
330
331         /// Returns the length of the buffer which was to
                small
332         pub fn found_buffer_size(&self) -> usize {
333             self.0
334         }
335     }
336
337     impl From<ReadError> for Error {
338         fn from(re: ReadError) -> Self {
339             Error::ReadError(re)
340         }
341     }
342
343     /// Write Error indicating that the buffer was to large
            for the given write operation.
344     #[derive(Clone, Debug, PartialEq, Eq)]
345     #[cfg_attr(feature = "serde", derive(serde::Serialize,
            serde::Deserialize))]
346     pub struct WriteError(usize);
347
348     impl WriteError {
349         /// Validate a buffer to be at most as long as the
                given usize.
350         /// If not returns [Self] with the length of the
                passed buffer
```

```
351         pub fn validate(size: MessageSize, buffer: &[ApexByte
                ]) -> Result<&[ApexByte], Self> {
352             if usize::try_from(size)
353                 .map(|ss| buffer.len() > ss)
354                 .unwrap_or(false)
355                 || buffer.is_empty()
356             {
357                 return Err(Self(buffer.len()));
358             }
359             Ok(buffer)
360         }
361 
362         /// Returns the length of the buffer which was to
                long
363         pub fn found_buffer_size(&self) -> usize {
364             self.0
365         }
366     }
367 
368     impl From<WriteError> for Error {
369         fn from(we: WriteError) -> Self {
370             Error::WriteError(we)
371         }
372     }
373 }
```

B

POSTCARD FOR apex.rs

Listing B.1: apex-rs-postcard sampling.rs

```
use apex_rs::bindings::*;
use apex_rs::prelude::*;
use arrayvec::ArrayVec;
use postcard::de_flavors::Slice as DeSlice;
use postcard::ser_flavors::Slice as SerSlice;
use serde::{Deserialize, Serialize};

use crate::error::*;

pub trait SamplingPortSourceExt {
    fn send_type<T>(&self, p: T) -> Result<(), SendError>
    where
        T: Serialize;
}

pub trait SamplingPortDestinationExt<const MSG_SIZE:
    MessageSize> {
    fn recv_type<T>(&self) -> Result<T, SamplingRecvError<
        MSG_SIZE>>
    where
        T: for<'a> Deserialize<'a>,
        [u8; MSG_SIZE as usize]:;
}

impl<const MSG_SIZE: MessageSize, Q: ApexSamplingPortP4>
    SamplingPortSourceExt
    for SamplingPortSource<MSG_SIZE, Q>
where
    [u8; MSG_SIZE as usize]:,
{
    fn send_type<T>(&self, p: T) -> Result<(), SendError>
    where
        T: Serialize,
    {
        let buf = &mut [0u8; MSG_SIZE as usize];
```

```
33          let buf =
34              postcard::serialize_with_flavor::<T, SerSlice, &
                    mut [u8]>(&p, SerSlice::new(buf))?;
35          self.send(buf).map_err(SendError::from)
36      }
37  }
38
39  impl<const MSG_SIZE: MessageSize, Q: ApexSamplingPortP4>
        SamplingPortDestinationExt<MSG_SIZE>
40      for SamplingPortDestination<MSG_SIZE, Q>
41  where
42      [u8; MSG_SIZE as usize]:,
43  {
44      fn recv_type<T>(&self) -> Result<T, SamplingRecvError<
            MSG_SIZE>>
45      where
46          T: for<'a> Deserialize<'a>,
47      {
48          let mut msg_buf = [0u8; MSG_SIZE as usize];
49          let (val, msg) = self.receive(&mut msg_buf)?;
50          let msg_slice = DeSlice::new(msg);
51          let mut deserializer = postcard::Deserializer::
                from_flavor(msg_slice);
52          match T::deserialize(&mut deserializer) {
53              Ok(t) => Ok(t),
54              Err(e) => {
55                  let mut msg = ArrayVec::from(msg_buf);
56                  msg.truncate(msg.len());
57                  Err(SamplingRecvError::Postcard(e, val, msg))
58              }
59          }
60      }
61  }
```

Listing B.2: apex-rs-postcard queuing.rs

```
1  use apex_rs::bindings::*;
2  use apex_rs::prelude::*;
3  use arrayvec::ArrayVec;
4  use postcard::de_flavors::Slice as DeSlice;
5  use postcard::ser_flavors::Slice as SerSlice;
6  use serde::{Deserialize, Serialize};
7
8  use crate::error::*;
9
```

```
10 pub trait QueuingPortSenderExt {
11     fn send_type<T>(&self, p: T, timeout: SystemTime) ->
           Result<(), SendError>
12     where
13         T: Serialize;
14 }
15
16 pub trait QueuingPortReceiverExt<const MSG_SIZE: MessageSize>
       {
17     fn recv_type<T>(&self, timeout: SystemTime) -> Result<T,
           QueuingRecvError<MSG_SIZE>>
18     where
19         T: for<'a> Deserialize<'a>,
20         [u8; MSG_SIZE as usize]:;
21 }
22
23 impl<const MSG_SIZE: MessageSize, const NB_MSGS: MessageRange
      , Q: ApexQueuingPortP4>
24     QueuingPortSenderExt for QueuingPortSender<MSG_SIZE,
           NB_MSGS, Q>
25 where
26     [u8; MSG_SIZE as usize]:,
27 {
28     fn send_type<T>(&self, p: T, timeout: SystemTime) ->
           Result<(), SendError>
29     where
30         T: Serialize,
31     {
32         let buf = &mut [0u8; MSG_SIZE as usize];
33         let buf =
34             postcard::serialize_with_flavor::<T, SerSlice, &
                    mut [u8]>(&p, SerSlice::new(buf))?;
35         self.send(buf, timeout).map_err(SendError::from)
36     }
37 }
38
39 impl<const MSG_SIZE: MessageSize, const NB_MSGS: MessageRange
      , Q: ApexQueuingPortP4>
40     QueuingPortReceiverExt<MSG_SIZE> for QueuingPortReceiver<
           MSG_SIZE, NB_MSGS, Q>
41 where
42     [u8; MSG_SIZE as usize]:,
43 {
```

```
44      fn recv_type<T>(&self, timeout: SystemTime) -> Result<T,
            QueuingRecvError<MSG_SIZE>>
45      where
46          T: for<'a> Deserialize<'a>,
47      {
48          let mut msg_buf = [0u8; MSG_SIZE as usize];
49          let msg = self.receive(&mut msg_buf, timeout)?;
50          let msg_slice = DeSlice::new(msg);
51          let mut deserializer = postcard::Deserializer::
                from_flavor(msg_slice);
52          match T::deserialize(&mut deserializer) {
53              Ok(t) => Ok(t),
54              Err(e) => {
55                  let mut msg = ArrayVec::from(msg_buf);
56                  msg.truncate(msg.len());
57                  Err(QueuingRecvError::Postcard(e, msg))
58              }
59          }
60      }
61  }
```

C

PLATFORM DEMONSTRATOR

Listing C.1: *Foo* Partition main.rs

```rust
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct CustomMessage {
    msg: String,
    when: Duration,
}

#[partition(linux_apex_partition::partition::
    ApexLinuxPartition)]
mod foo {
    #[sampling_out(msg_size = "10KB")]
    struct Hello;

    #[start(cold)]
    fn cold_start(ctx: start::Context) {
        ctx.init_hello().unwrap();
        ctx.init_periodic_foo().unwrap();
        ctx.init_aperiodic_foo().unwrap();
    }

    #[start(warm)]
    fn warm_start(ctx: start::Context) {
        cold_start(ctx);
    }

    #[aperiodic(
        time_capacity = "Infinite",
        stack_size = "100KB",
        base_priority = 1,
        deadline = "Soft"
    )]
    fn aperiodic_foo(ctx: aperiodic_foo::Context) {
        for i in 0..i32::MAX {
            let time = ctx.get_time().unwrap_duration();
            info!("{:?}: APMSG {i}", format_duration(time).
                to_string());
```

```
            sleep(Duration::from_millis(1))
        }
    }

    #[periodic(
        period = "0ms",
        time_capacity = "Infinite",
        stack_size = "100KB",
        base_priority = 1,
        deadline = "Soft"
    )]
    fn periodic_foo(ctx: periodic_foo::Context) {
        for i in 1..i32::MAX {
            let time = ctx.get_time().unwrap_duration();
            info!("{:?}: P MSG {i}", format_duration(time).
                to_string());
            sleep(Duration::from_millis(1));
            if i % 5 == 0 {
                ctx.hello
                    .unwrap()
                    .send_type(CustomMessage {
                        msg: format!("Sampling MSG {}", i /
                            5),
                        when: time,
                    })
                    .unwrap();

                ctx.periodic_wait().unwrap();
            }
        }
    }
}
```

Listing C.2: *Bar* Partition main.rs

```
#[derive(Clone, Debug, Serialize, Deserialize)]
pub struct CustomMessage {
    msg: String,
    when: Duration,
}

#[partition(linux_apex_partition::partition::
    ApexLinuxPartition)]
mod bar {
    #[sampling_in(refresh_period = "40ms")]
```

```
10      #[sampling_in(msg_size = "10KB")]
11      struct Hello;
12
13      #[start(cold)]
14      fn cold_start(ctx: start::Context) {
15          ctx.init_hello().unwrap();
16          ctx.init_periodic_bar().unwrap();
17          ctx.init_aperiodic_bar().unwrap();
18      }
19
20      #[start(warm)]
21      fn warm_start(ctx: start::Context) {
22          cold_start(ctx);
23      }
24
25      #[aperiodic(
26          time_capacity = "Infinite",
27          stack_size = "100KB",
28          base_priority = 1,
29          deadline = "Soft"
30      )]
31      fn aperiodic_bar(ctx: aperiodic_bar::Context) {
32          for i in 0..i32::MAX {
33              let time = ctx.get_time().unwrap_duration();
34              info!("{:?}: APMSG {i}", format_duration(time).
                    to_string());
35              sleep(Duration::from_millis(1))
36          }
37      }
38
39      #[periodic(
40          period = "0ms",
41          time_capacity = "Infinite",
42          stack_size = "100KB",
43          base_priority = 1,
44          deadline = "Soft"
45      )]
46      fn periodic_bar(ctx: periodic_bar::Context) {
47          for i in 1..i32::MAX {
48              let time = ctx.get_time().unwrap_duration();
49              info!("{:?}: PMSG {i}", format_duration(time).
                    to_string());
50              sleep(Duration::from_millis(1));
51              if i % 3 == 0 {
```

```
52                 let (valid, data) = ctx.hello
53                     .unwrap()
54                     .recv_type::<CustomMessage>()
55                     .unwrap();
56                 info!("Received via Sampling Port: {:?},
                       valid: {valid:?}", data)
57
58                 ctx.periodic_wait().unwrap();
59             }
60         }
61     }
62 }
```