Gapfruit for Avionics

Jim Podmore

Silver Fox Software

December 2020

Contents

1 Introduction

This report addresses the suitability of the Gapfruit operating system for use in commercial safety critical avionics embedded software, certified in accordance with the Do-178C standard. Do-178C is widely used by civil aviation authorities in the USA and Europe for approving software-based avionics systems and has become the de facto standard for developing safety critical and mission critical avionics software.

DO-178C defines a set of objectives that need to be achieved depending on the Development Assurance Level (DAL) of the software, which is derived from the safety analysis of the system. The DAL can range from A (high, failures can be catastrophic) to E (low, failures have no effect), and the effort required to meet the objectives for higher DAL systems can be considerable. Modern avionics software systems therefore allow for a mixed criticality implementation, where critical components are isolated from less critical ones, thus reducing the overall certification effort, while retaining the integrity of the critical components.

The Operating System therefore itself becomes a critical component and must meet the objectives for the highest DAL in the system, as well as assuring separation between components. Widely used OS's have been developed to the Arinc 653 standard, which defines how a software system can be partitioned into separate components that are isolated in time and space. It is this environment that Gapfruit will need to compete in if it is to be used in safety critical avionics. This report seeks to identify actions needed to allow Gapfruit to be used in such applications, and how it can provide advantages over the established competition.

2 About the Author

The author has over 35 years' experience in developing embedded real time software for the defence and aerospace industry. Projects have included the UK air traffic control system, a robot arm for the International Space Station, an Air-to-Air refuelling system and many others. Most recent work was on development and certification of software for the Pilatus PC-24 business jet, and an unmanned vertical take-off UAV. See www.linkedin.com/in/jim-podmore for detailed profile.



3 Background on Avionics Software Systems

3.1 DO-178C

'Software considerations in Airborne Systems and Equipment Certification' was developed by safety critical working groups at RTCA and EUROCAE and initially published as DO-178B in 1992. It provides guidance to determine if embedded software will perform reliably in an airborne environment. Although technically a guideline, it does embody best practices and consensus from the avionics community and has become the de facto standard for developing avionics software. It was updated to DO-178C in 2012 to refine the standard and reflect advances in software development techniques, such as model based development and formal methods.

As its title suggests, the standard is only part of an overall system certification process, the system that is certified being on an aircraft. Crucial to the certification is the safety assessment process and hazard analysis, which identifies potential failure conditions in the overall system, and categorizes their effects on the aircraft, crew, passengers, and environment. Failure conditions are categorized as follows:

> Catastrophic: Failure conditions which would result in multiple fatalities, usually with the loss of the aircraft.
>
> Hazardous: Failure conditions which would reduce the capability of the aircraft or the ability of the flight crew to cope with adverse operating conditions to the extent that there would be:
>
> - A large reduction in safety margins or functional capabilities.
> - Physical distress or excessive workload such that the flight crew cannot be relied upon to perform their tasks accurately or completely.
> - Serious or fatal injury to a relatively small number of the occupants other than the flight crew.
>
> Major: Failure conditions which would reduce the capability of the aircraft or the ability of the flight crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to the flight crew, or physical distress to passengers or cabin crew, possibly including injuries.
>
> Minor: Failure conditions which would not significantly reduce aircraft safety, and which involve crew actions that are well within their capabilities. Minor failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as routine flight plan changes, or some physical discomfort to passengers or cabin crew.
>
> No Safety Effect: Failure conditions that would have no effect on safety; for example, failure conditions that would not affect the operational capability of the aircraft or increase crew workload.

Safety analysis is clearly critical to the overall system design. Safety hazards may be mitigated in the design, but ultimately the functional, performance and reliability requirements of all components of the system will be affected by the safety analysis. When the component is implemented in software, DO-178C assigns a Development Assurance Level (DAL), which is derived from the most hazardous

failure condition that can be caused by the software. The DAL ranges from A to E, with A corresponding to catastrophic failures, E to failures with no safety effect.

DO-178C defines a set of objectives that must be satisfied in developing and verifying the software, with the number of objectives depending on the assigned DAL. The DAL also determines whether objectives need to be satisfied with independence, where for example the person verifying an item (such as a requirement or source code) must not be the person who authored the item, and this separation must be clearly documented. The combination of number of objectives and need for independence means that the workload for certifying software increases considerably for higher DAL levels, and the DAL for any component is determined by the most severe failure condition to which it can contribute.

Partitioning therefore becomes an important principle in avionics systems, to isolate software components from each other and thus avoid having to certify the entire system to the highest DAL. This can be achieved by allocating unique hardware resources to each component, but this becomes very expensive in a large and complex system, and itself introduces more complexity in the form of interfaces and physical wiring required. The PC-24 for example has over 20 separate sub-systems, and implementing each on separate hardware, with backups and the necessary cabling would have been impossible.

The concept therefore emerged in the 1990's of 'Integrated Modular Avionics', initially in fourth generation fighter jets such as the F-22, and more recently in commercial airliners such as the Airbus A380. Although not formally defined, this concept is an architecture that allows software components of differing criticality levels to be executed on the same hardware platform, this requiring partitioning to be performed by the underlying operating system.

DO-178C does not specify whether partitioning should be used, or how it is to be achieved, but does provide important guidance if it is used. From section 2.4.1:

a) A partitioned software component should not be allowed to contaminate another partitioned software component's code, input/output, or data storage areas.
b) A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.
c) Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components.
d) Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components.
e) Any hardware providing partitioning should be assessed by the system safety assessment process to ensure that it does not adversely affect safety.

Note the use of 'should' in all these guidelines. This is typical in DO-178C, as it notes in its introduction:

> This document recognizes that the guidance herein is not mandated by law but represents a consensus of the aviation community. It also recognizes that alternative methods to the methods described herein may be available to the applicant. For these reasons, the use of words such as 'shall' and 'must' is avoided.

The document however is now widely accepted as the standard for airborne software certification, and suppliers need to give careful thought to justifying any deviations from its guidance.

The update to DO-178C recognized that the standard needed revision to add clarity to the guidelines in the document itself, and to reflect advances in software development techniques. Rather than make huge changes to the structure of the document itself, the authors decided to add supplemental documents to be used alongside the main core text as follows:

- DO-330 Adds guidance for the qualification of tools used in the development and verification process.
- DO-331 Adds guidance for the use of model-based development techniques.
- DO-332 Adds guidance for the use of Object-Oriented techniques in software.
- DO-333 Addresses the use of formal methods to complement the testing process.

The supplements are not stand-alone documents but define modifications to the core text to apply the guidance required for their specific technology.

The documents are available from the RTCA store at Store - Community Hub (rtca.org)

3.2 Arinc 653

First published in 1996, Arinc 653 is a software specification for space and time partitioning in safety-critical avionics real-time operating systems (RTOS). Its use is not mandatory, but it does satisfy the requirements in DO-178C for software partitioning and is a well understood standard with several established and certified implementations available. Hence if it is not used, then authorities may pay close attention to how DO-178C guidelines are achieved.

Each partition has its own memory space, thus code in one partition is unable to affect data in another. Within each partition, multi-tasking is allowed, but the processing has a dedicated time slot allocated to it by a schedule configured at build time. Communication between partitions is achieved via channels linking ports, which can be sampling or queuing ports, again configured at build time. Shared memory areas can also be defined, with partitions having read and / or write access.

Sampling ports provide a way of sharing a single message generated from a source partition, to one or more destination partitions. Messages written to the source port simply overwrite the old copy and are then copied by the channel to the destination ports. A timeout can be used to invalidate the data if it is not updated at a specified rate.

Queuing ports allow multiple messages to be transmitted between partitions, thus messages written to the source ports are queued in a FIFO, which is then copied to the destination port. The size of messages and length of FIFO queues is configured at build time.

Each partition can contain multiple threads running under a pre-emptive scheduler, but each partition can only execute in a limited time slice defined in a time schedule.

Multiple time schedules can be defined in the build configuration, and the application code can command a switch between schedules, thus allowing some control over the run-time configuration. Partitions can also be stopped and restarted if necessary.

Device drivers are built into the underlying Module Operating System (MOS) and can be accessed from the application partitions via system level function calls.

Configuration is defined in xml files that are processed by the build system.

Calls to the underlying RTOS from the application software are via an API called APEX (Application Executive).

The standard is available on the Arinc store at [600 Series | SAE ITC (aviation-ia.com)]

4 Example of Avionics Software

Figure 1 shows the high-level software architecture of an example avionics system. It is based on a flight control computer, which provides control, navigation, and mission management functions for an Unmanned Aerial Vehicle (UAV). Although unmanned, the system is considered safety critical in the sense that the aircraft must not be allowed to lose control and crash on populated areas, so must be certified to DO-178 standards, with the safety critical aspects at DAL A.

The system is divided into the following sub-systems:

1. Interface Processing

   Handles cyclic processing of critical interfaces with other systems on the aircraft. DAL A.

2. Control Processing

   Functionality for controlling the aircraft, generating steering commands in response to attitude, position, flight plan etc, handling Command and Control (C2) commands and providing telemetry to ground station. DAL B.

3. Monitor Processing

   Monitors the status of the control function, checking that the system is still under control from the ground station, and has not strayed outside permitted airspace. Maintains watchdog command when all is well, otherwise stops the watchdog and issues kill signals to bring the aircraft down. DAL A.

4. Support Processing

   Provides handling of mass storage and networking functions. System config data and geometry of the Permitted Airspace are loaded from the network into mass storage pre-flight and loaded into shared memory on system reboot. DAL C.

The system is implemented using an Arinc 653 based Real Time Operating System (RTOS), with each subsystem implemented as a partition. The scheduling scheme defines a 20ms major frame, within which the Interface Processing runs every 2ms to provide the necessary polling rate of the interfaces. Control and Monitoring processing is then scheduled to run in the intervening gaps. Assuming this is a 2-core processor, the Support processing is scheduled to run continuously on core 2 to provide asynchronous functions.

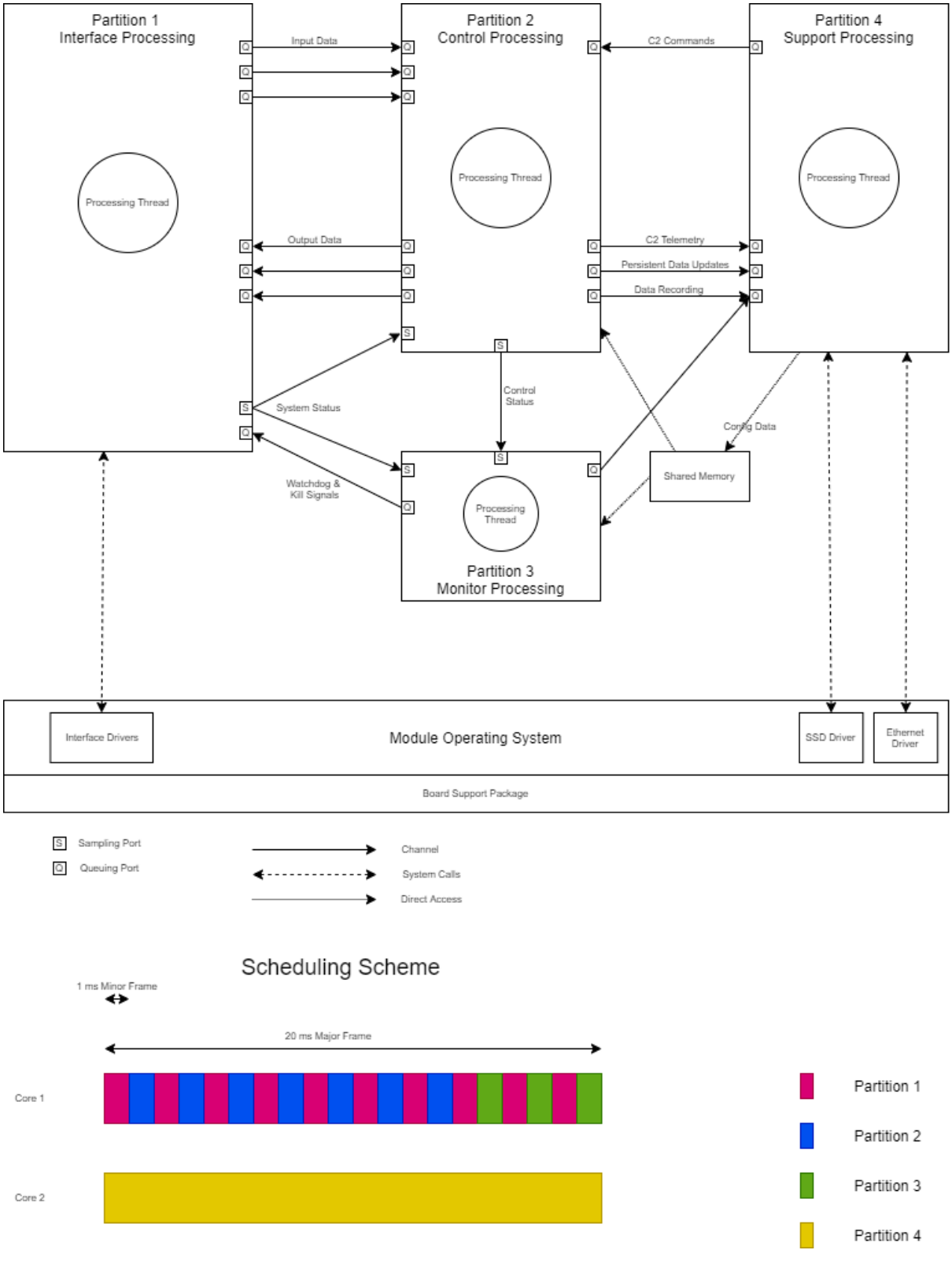

Figure 1. Example Avionics Software

5 Implementation in Gapfruit

Clearly Gapfruit must provide the ability to reproduce the Arinc 653 architecture, in particular the isolation of partitions in time and space, and reliable communication between partitions.

5.1 Partitioning

Spatial isolation is a strong feature of Gapfruit, as provided by the Genode protection domains, and a variety of options available for inter component communication, so there are many options for designing a system. Each partition may be implemented as a Gapfruit slice, with communication ports implemented as servers that provide access capabilities to clients in other slices.

For Queuing ports, the server is provided by the destination slice, which has read capability. Write capability can be delegated to many other client components.

For sampling ports, the source component is the server, with a write capability in its slice. Multiple destination components can exist in many client components.

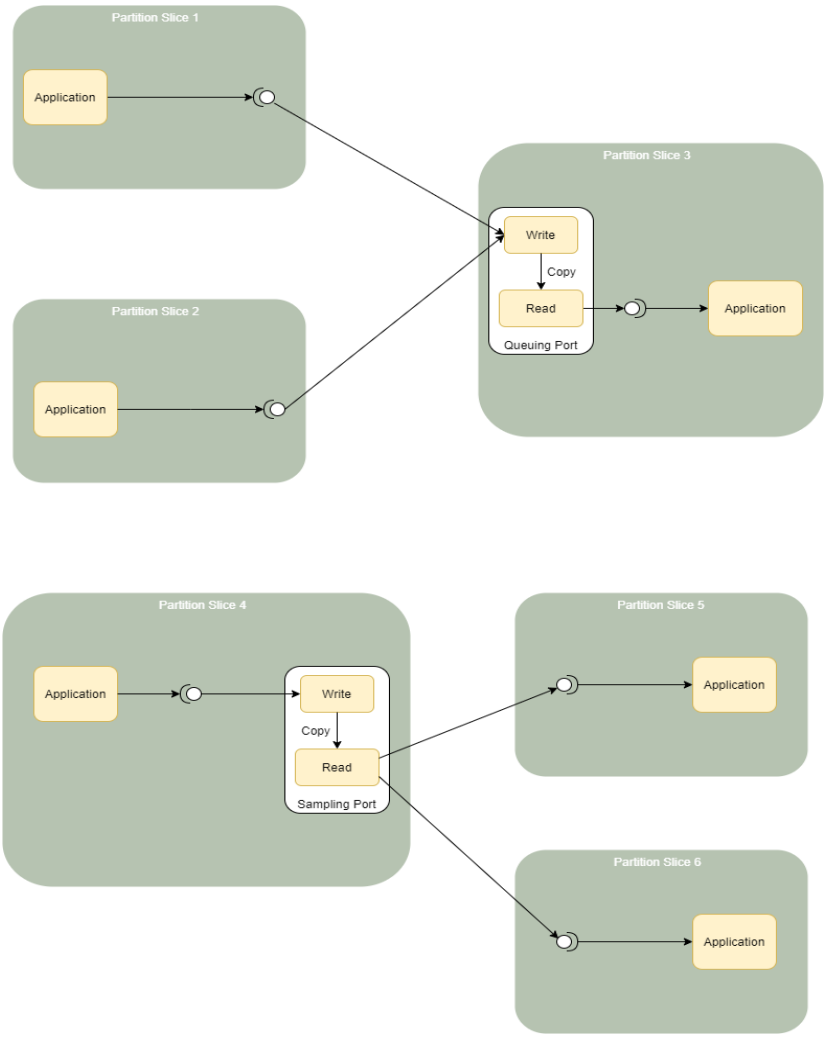

Figure 2. Ports and Channels in Gapfruit

Ports will need to be Gapfruit generic components, so certified with the OS. Instantiated with message types and queue sizes so resources deterministic and correctly allocated at run time.

Implementation needs to ensure access to a port by one component has minimal blocking on other components.

5.2 Scheduling

Arinc 653 style time slicing is currently not available in Gapfruit, and this will need addressing.

The main scheduling requirement is derived from the DO-178C statement in section 2.4.1 on components only being allowed to consume shared processor resources during a scheduled period. The OS therefore needs to be able to suspend and resume threads in response to a time slice. Server threads for Queuing and Sampling ports should always be active to prevent excessive blocking of application threads. Components must only interact via the ports, thus ensuring that one application component cannot block another.

The seL4 microkernel does support domain scheduling, where threads are assigned to domains that are statically configured at compile time and are non-preemptible hence assuring completely deterministic scheduling. Scheduling contexts also allow budgets and periods to be defined, so the necessary functions are available, but more investigation is needed.

6 Certification of Gapfruit

Gapfruit will itself need to be certifiable to the highest DAL used by its customers, which means DAL A if it is to compete. Thus, a complete set of plans, requirements, design, reviews, test results etc being generated as defined in the standard, and evidence being available for the customer to include in their presentation to the authorities.

The level of effort required to produce DO-178C certification evidence for Gapfruit is unknown and should not be underestimated. It is important to note though that an RTOS is not certified in isolation, it is just one part of a larger system, which is itself certified. Hence the RTOS needs to be *certifiable* before it can really be of use to a customer. Established suppliers such as Wind River, Greenhills etc have already been through this process, with their products and their documentation being known to certification authorities. Customers will need confidence that Gapfruit will be able to conform to DO-178C and will probably perform their own audits before submitting to inspection by EASA or the FAA.

The seL4 microkernel has already been proven using formal methods and the DO-333 supplement does provide guidance as to how this can fit into the process. Does the seL4 verification evidence conform to DO-178C and DO-333?

Genode is a problem, as it is open source, and presumably requirements, design, etc are not available. A snapshot of the current baseline will need to be taken and brought under config control, so the audit trail can be established, but the hard part will be defining requirements. DO-178C does recognize that reverse engineering may be used in section 12.1.4 (d), but states '… additional activities may need to be performed to satisfy the software verification process objectives.' No further guidance is given, but the plans will need to reflect the reality of the process, and detail what other actions, e.g., independent reviews are to be performed.

Note that DO-178C does not allow 'dead' code (as opposed to 'deactivated' code), see section 6.4.4.3 (c), so it may be necessary to define a small sub-set of Gapfruit for use in safety critical embedded systems, removing any unused functionality.

Planning is critical to DO-178C. Section 4.0 of the standard requires the following plans, which define how the various objectives are to be achieved:

a. Plan for Software Aspects of Certification (PSAC). Serves as the primary means for communicating the proposed development methods to the certification authority and defines the means of compliance with the standard.
b. Software Development Plan (SDP). Defines the software life cycle, development environment and the means by which development objectives will be satisfied.
c. Software Verification Plan (SVP). Defines the means by which the software verification process objectives will be satisfied.
d. Software Configuration Management Plan (SCMP). Defines the means by which the software configuration management process objectives will be satisfied.
e. Software Quality Assurance Plan (SQAP). Defines the means by which the software quality assurance process objectives will be satisfied.

During the certification process for a system, the authority will participate in reviews of the plans and progress against them called ‘Stages of Involvement’ (SOI) as follows:

1. SOI 1 reviews the software plans to ensure they comply with the standard.
2. SOI 2 reviews actions from SOI 1, and development of requirements, design and code once 75% of that is complete.
3. SOI 3 reviews actions from SOI2, and verification and test data once 75% of that is complete.
4. SOI 4 is the final conformance review once all activities are complete.

A software supplier such as Gapfruit will need to be prepared to participate in these reviews when they are performed as part of the certification process for a system. This presents a ‘chicken and egg’ type problem for a supplier since it is hard to demonstrate certifiability without going through the complete process with the certification authority.

Knowledge of the standard and how it is applied in practice by the authorities is crucial here, and there are no short cuts. Gapfruit could consider engaging a specialist company such as PMV or AFuzion to provide training and consultancy services to ensure it will be able to meet the objectives of DO-178C. Independent reviews of plans and accomplishments can then be used as evidence to present to potential customers.

Alternatively, Gapfruit could seek to partner with an established avionics software supplier who can provide DO-178C experience, plus established processes, an experienced development and QA team and a suitable project to be certified with the new OS. As well as the technical advantages offered by Gapfruit, there could be commercial benefits in saving the cost of RTOS licensing as well as a share of future business.

7 Advantages of Gapfruit

Arinc 653 RTOS's are well established, understood by developers and certification authorities, so Gapfruit will need to provide considerable advantages if it is to compete.

7.1 User Level Device Drivers

In an Arinc 653 system, any device drivers that need to be accessed by multiple partitions are commonly installed as Module Operating System (MOS) components. They could be implemented in their own partition, and dedicated channels used to communicate to the application partitions, but this brings performance penalties, since the driver partition would need to be scheduled in its own time slice and may need to poll a device at a very high rate to be effective. But if all drivers are implemented in the MOS, then they would need to be certified to the highest DAL in the system.

7.2 Dynamic Configuration

A major feature of Gapfruit is the dynamic delegation of capabilities at run time, thus enabling inter component communications to be reconfigured as required. This can be exploited to provide facilities unavailable in 653.

As discussed earlier, inter partition communication in Arinc 653 is statically defined at build time, and this is fine for operational flight software where the config does have to be strictly controlled. However, flight software does not only get used in the operational flight domain, but there are also other use cases where components or the complete system may be executed.

7.2.1 Software Test Environment

Testing of flight software as part of the DO-178C process is clearly critical. This is where the user generates the evidence that the software satisfies all its requirements, so a systematic approach to defining tests that trace to requirements is essential. This comes down to being able to control the inputs to a component under test and have visibility of the outputs generated by its execution.

Components can be tested in isolation from the rest of the system, a process commonly referred to as unit testing. This can be effective in testing the component itself, but usually requires a unique software build, possibly in a host-based environment, and integrated with tools that are not usually included in the flight software. DO-178C (section 6.4.1) states:

> "A preferred test environment includes the software loaded into the target computer and tested in an environment that closely resembles the behaviour of the target computer environment.
>
> *Note: In many cases, the requirements-based coverage and structural coverage necessary can be achieved only with more precise control and monitoring of the test inputs and code execution than generally possible in a fully integrated environment. Such testing may need to be performed on a small software component that is functionally isolated from other software components.*

Certification credit may be given for testing done using a target computer emulator or a host computer simulator. “

As usual, the standard does not explicitly mandate how testing is to be performed, but the first and last paragraphs here are key.

Regulators do prefer to see testing performed in the target computer environment, using the software build that will ultimately be used in the system being certified. This provides unambiguous proof that the software will meet its requirements, whereas tests performed in any other environment will need to be supported by other tests or analysis that provide corroborating evidence.

(Certification to level A is analogous to a legal case that must be proven beyond any possible doubt. The jury much prefers a simple argument that is beyond dispute by the defence).

This presents challenges however in the development of a complex software system, since it may have many input and output channels, all dependent on external hardware that may or may not be available at any given time. The target hardware itself may not even be available in the early stages of development, and in a large system that needs many developers and testers, access to the system can become a serious bottleneck.

Consider the flow of data into a system from a typical interface, let us say a GPS system providing latitude, longitude, altitude etc. The raw data would be read from a device driver provided by the hardware supplier, say as a stream of bytes from a UART device. That stream would need to be buffered as it arrives and parsed to identify frame start and end points. The resultant frame could then be validated and decoded to extract the position data, and then this passed on to the application software components, or faults logged when the data is in error.

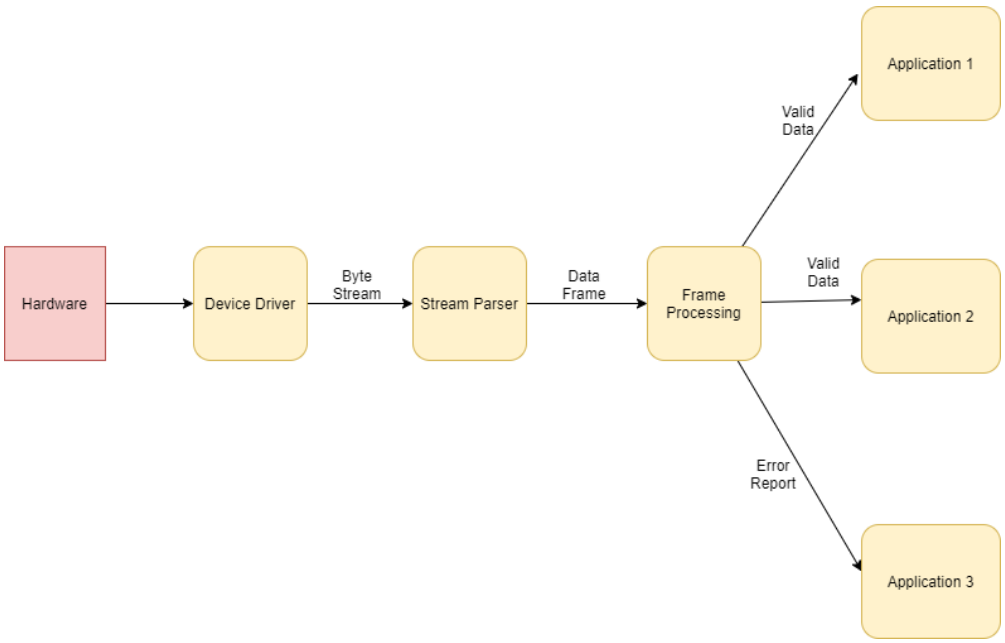

Figure 3. Example input data flow.

A complete end-to-end test could be performed on the target hardware if a real GPS were available, but this would not be a practical way of fully testing the software components. Testing requires faults to be injected, full data ranges exercised, all under control from a test script of some kind so that the test can be automated.

This can be achieved by building in access points to the software components, to allow the test environment to inject or read data at certain points as required. For example:

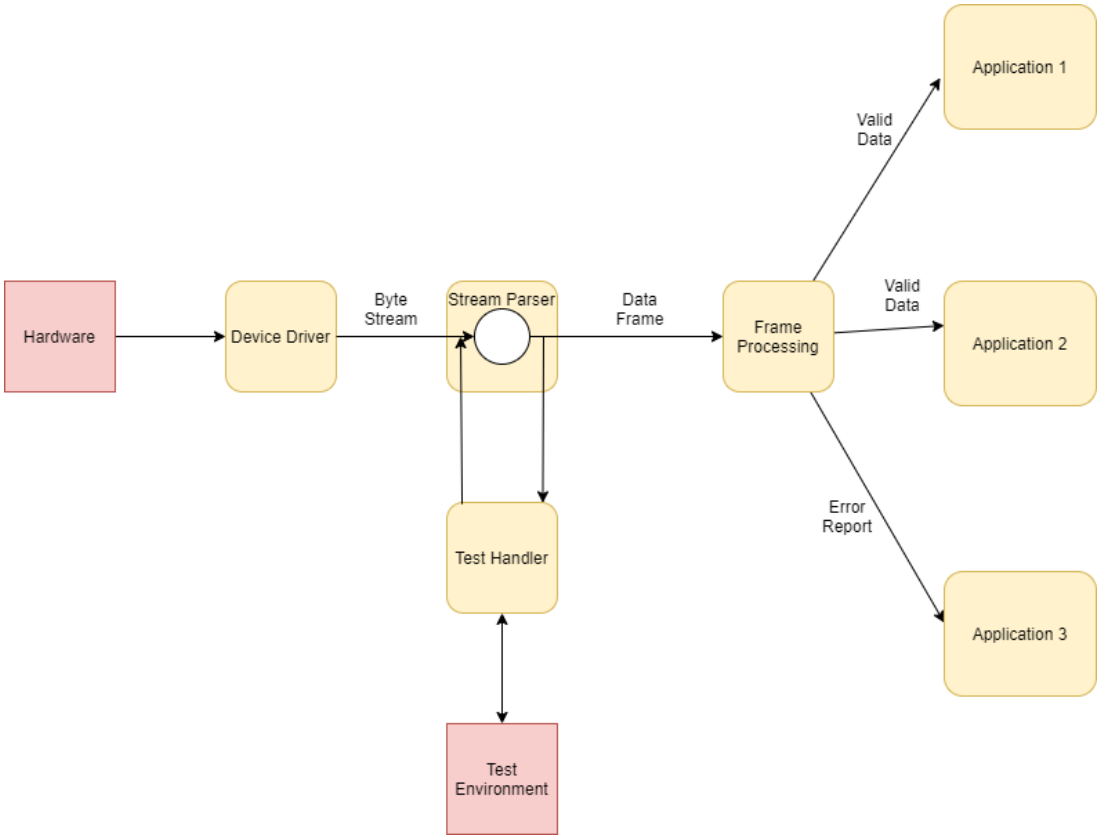

Figure 4. Data flow with test points.

In this case, test access points are added to the input and output of the Stream Parser. These enable a test handler to inject test data in place of the real byte stream and read the resulting data frame. The handler communicates with the test environment, typically via an ethernet connection. The test environment processes scripts written by a tester to generate the inputs and check outputs. The test handler can simply route messages to and from the component under test or contain its own processing that responds to tester commands. This mechanism can be extended to all other components, and the test access points can be implemented in generics or templates used for inter-component communication.

This architecture works well for testing the component but has some philosophical problems for certification. The access points need to be activated in some way, so the input stream is read from the test handler rather than driver, and outputs routed to the test handler. This adds extra complexity and processing overhead even when not testing. The activation mechanism would also

need to be certified to the highest DAL in the system, as it would be necessary to prove that it could not activate the test points inadvertently in normal operation. This can be done but adds more software that must be certified. In an Arinc 653 system it also means adding more ports and channels which in a large system could consume memory resources.

Gapfruit provides an alternative solution, in the form of a reconfiguration of capabilities at run time. Figure x shows how for testing purposes, capabilities can be re-assigned to isolate the component under test and allow access by the test handler. In this scenario, the operational software is completely unchanged, with no test specific code embedded in the components, and the test handler disconnected and not even scheduled for execution. For testing, the system is reconfigured to run the test handler, and delegate capabilities to it as needed for the specific test, in this case for the Stream parser tests. Here, the test handler becomes the write client for the raw byte stream and has an instance of the Data frame server (Q2) which delegates its write capability to the Stream Parser in place of the Frame Processing, so the test handler receives output data and can test them.

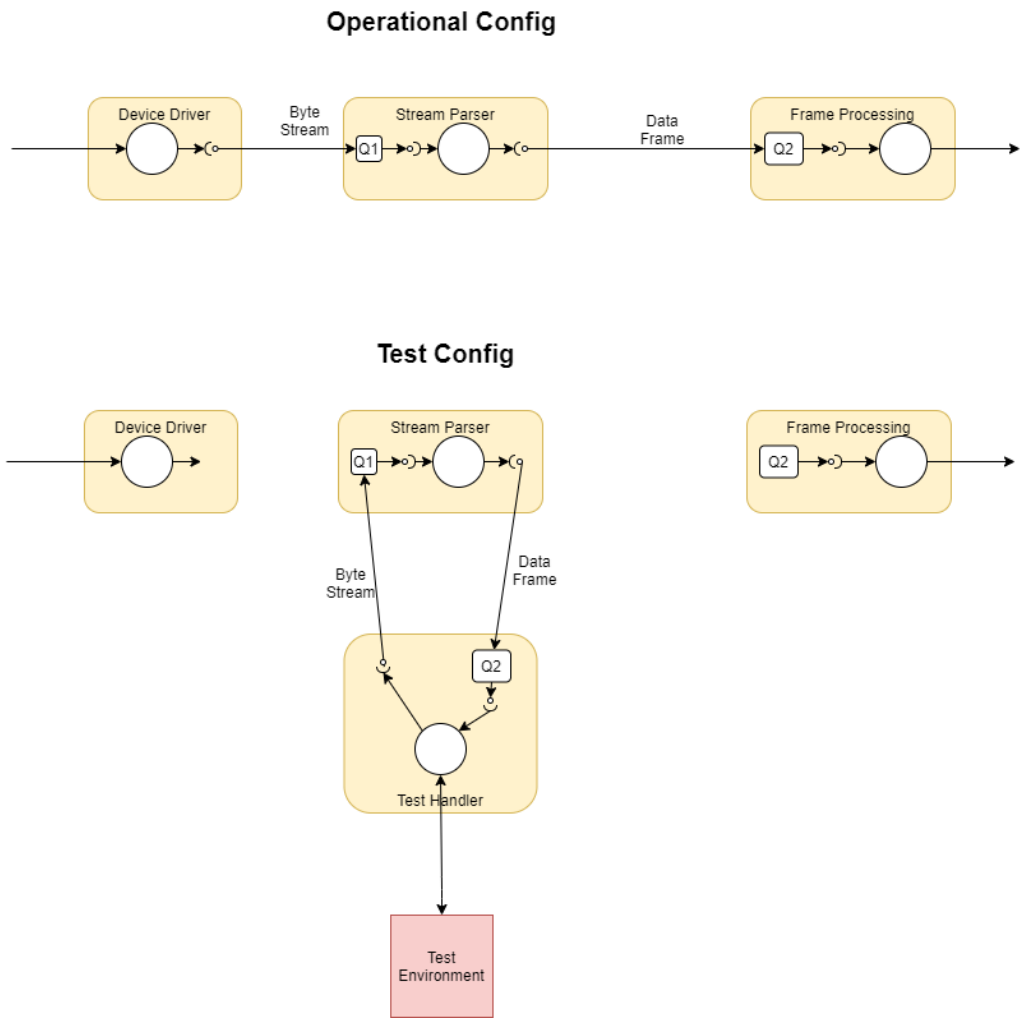

Figure 5. Data flow with reconfigured capabilities.

The actual mechanism for controlling reconfiguration will be discussed below, but clearly this approach offers tremendous flexibility in the testing process. Individual components can be tested in isolation as here, or integration tests performed on multiple components depending on the config and which capabilities are taken over by the test environment. It also makes it possible to reconfigure the system to allow testing on hardware where not all interfaces are available, or for host-based testing to allow testers to develop their tests on desktop computers prior to formal testing on real hardware. The test config becomes part of the test plan and can be developed and reviewed as part of the verification process.

7.2.2 Non-Operational Modes

Avionics software is not only used in its operational flight environment. Aircraft need maintenance, systems need to have stored data downloaded, sensors must be calibrated etc, all of which can require the system to be activated and used in ways not envisaged for flight and having a completely different safety case.

For example, a flight control system may be activated during an aircraft maintenance operation, one example from the author's experience being for calibration of sensors used for measuring the positions of flight surfaces on the aircraft.

In this case, the aircraft would be powered up, and flight surfaces moved to known positions before commands being sent to the FCS to save the measured positions for later use in flight. To move flaps for example, the FCS needed certain input data parameters set to certain values, e.g., landing gear, engine speed, airspeed etc, which is clearly not possible when the aircraft is in the hanger. Some way therefore must be found to override input signals with controlled values needed just for the specific maintenance activity.

This ability can be built into the flight software, and indeed was in this case. But this added further functionality to the system that needed certifying to DAL A standards, and it was a highly inflexible mechanism that was hard and expensive to modify as the maintenance requirements evolved. Much better, would have been a more flexible system that allowed any input signal to be overridden by set values under control of an external system, say a maintenance laptop.

Here again, the ability of Gapfruit to dynamically reconfigure the system at run time can be an advantage, by moving the maintenance functionality to a configuration that cannot be executed in flight. Like the test environment case described above, the system could be reconfigured to re-assign write capabilities for a component to a maintenance handler that communicates to a maintenance laptop, allowing full control over the scenario.

7.2.3 Multiple Variants and Optional Functions

Avionics systems can interface to many other systems on the aircraft, from sensors like GPS and air data to complete systems like engines or communications. During the lifetime of an aircraft, different versions of some systems may be added, some functions may not be available or need to be deliberately disabled for specific customers. The number of possible combinations of software components can soon become prohibitive, and generating a new software build for each combination a logistical headache.

Given the above use cases, it could be argued that the ability of Gapfruit to reconfigure the systems capabilities has great advantages in the safety critical avionics domain. Reconfiguration allows the designer to add functionality that can be activated in a controlled way without making any modifications to the certified flight code. It could be argued that the static configuration of the Arinc 653 architecture imposes great limitations on a system and increases complexity when non-operational functions are added.

DO-178C recognizes the potential for loading 'Parameter Data Items' that can affect the execution of the software and provides guidance as to how to certify this in section 2.5.1. Most importantly, the design must ensure that inadvertent selections of non-approved or non-operational configurations cannot occur. This is partly covered by the fact that Gapfruit only configures the system during initialization, so the config can be checked for validity and compatibility with the software build before starting the system. A distinction could be made between operational configurations that can be loaded into ROM and used by default to run the system, and temporary configurations which could be loaded into RAM, and used following a warm restart of the system. These would be used for testing and maintenance operations as described above, with a cold restart restoring the operational config.

Gapfruit would therefore need to provide the ability for the user to add some pre-processing to the init function, to select the source of the config file and to perform validation functions.

Config files clearly become a vital part of the system, and their development would have to come under the control of DO-178C. Possibly a model-based approach could be used for defining the architecture of a system, identifying components, and their connections so config files could be generated automatically. Framework components like the queuing and sampling ports, test router etc could also be auto generated from the model.

Changing the configuration of the system at run time using Gapfruit can provide the supplier with a way of managing multiple system variants with one software build. Different configurations could be loaded to the target hardware on installation, or multiple configurations built into the software and selected at initialization depending on hardware connector pin settings.

7.3 Scheduling Flexibility

Arinc 653's scheduling scheme allocates distinct time slices to each partition in a system, but only one partition is allowed for each time slice. Since the slice needs to be long enough to allow for the partition's worst case processing requirements, this can mean considerable unused CPU time in the time slice. In addition, the granularity of the scheduling is determined by the system clock, which may be quite coarse (say 1ms intervals), again leading to unused processing time. Although the problem is somewhat mitigated in modern multi core processors, this can lead to the number of partitions being limited by the need to fit them into the schedule scheme.

DO-178C does not itself impose this restriction, the only relevant guideline being its section 2.4.1 para b quoted previously. But this only states that a component's processing be confined to a scheduled execution period, thus preventing a failing component from completely monopolising the processor. If multiple partitions could be assigned to a time slice, the software designer would have much more flexibility in how to partition the system. Gapfruit can provide this flexibility by treating processing time as a resource that can be delegated from parent to child components.

For example, the chain of components in the previous discussion i.e., Driver, Parser, Frame Processing, Application would be very wasteful of processing time if implemented as separate partitions in an Arinc 653 system, as each would need its own time slice. But partitioning to this level of granularity does have great advantages as discussed. Under Gapfruit, the components could all execute in the same time slice.

Obviously, the software designer needs to be aware of the performance penalties of partitioning the system to such a fine granularity, as every RPC call from client to server results in a context switch. Performance is less of an issue with modern processors however than the need for a deterministic real-time behaviour.

7.4 Alternative Inter-Process Communications

Arinc 653 style ports and channels are convenient methods of inter-processor communication when the messages are relatively small but can incur processing overheads for larger messages due to the number of copies and context switches involved. The only alternative available in Arinc 653 is shared memory, but this must be carefully managed to prevent uncontrolled access. The user can of course design generic components that incorporate shared memory and communication ports, but Genode already has such facilities built in, e.g., asynchronous bulk transfer of packet streams.

7.5 Host-Based Testing Environment

Host-based testing is an essential tool in avionics software development. As discussed previously, access to the real target hardware can be very limited, especially during early system development, and full system test rigs can be enormously expensive to develop and maintain. Large and complex systems need a test environment that can be scaled up to allow many developers and testers to work in parallel. Test development is a lengthy, iterative process, and testers need to be able to execute their tests on their own desktop equipment before the tests are formally run ‘for score’, i.e., to gain credit for certification.

The host test environment therefore needs to closely resemble the environment used for the formal test runs, usually the target hardware on a dedicated test rig. This can be hard to achieve on a typical desktop running Windows or Linux, connected to a network and running multiple development tools, browsers, anti-virus software etc as the hard real time scheduling of components cannot be performed in a reliable way. Target emulators and Arinc 653 RTOS simulators are available, but they still need to run under the control of the host OS, so real time performance is always a problem.

Gapfruit can run as a hypervisor, so can itself be used as a host OS, with Linux or windows running in its own slice under a Virtual machine. Components for the avionics system can therefore be run under the same software architecture as used on target, with deterministic real time behaviour preserved. The host test environment is therefore much more reliable, and highly representative of the target architecture. This can enable certification credit to be claimed for formal test runs performed in the host environment, helping to achieve coverage necessary for DAL A and B software for example without taking up valuable rig time.

8 Conclusions and future work

Gapfruit can potentially address a major limitation of Arinc 653 based RTOS's, which is their static configuration based on channels connecting ports defined at build time. The ability to reconfigure a system dynamically, albeit in a carefully controlled manner provides great flexibility in a system while limiting the complexity of individual components. Reconfiguration of capabilities can have advantages in the operational use of a system as well as the development and certification process.

Arinc 653 is now widely used, and the avionics industry is very conservative and resistant to change. It may be best therefore to promote Gapfruit for Avionics as an evolutionary product, aiming to reproduce the existing Arinc 653 features but to add the benefits of dynamic reconfiguration and other features identified above. The APEX API provides a good starting point and abstracting the implementation below this layer would make it easier for customers to migrate their existing Arinc 653 based systems to Gapfruit. Aircraft systems have long lifespans and typically evolve considerably, often with software being ported to updated hardware platforms. This may be an easier way for Gapfruit to enter the market than with a customer developing a new system from scratch.

With this goal in mind, the following activities are recommended:

- Consider training courses in DO-178C to improve understanding of objectives and processes. Will need to consider how development to DO-178C will fit in with existing processes, and possibly other domains to be targeted for Gapfruit (medical industry, automotive?).
- Review current state of Gapfruit against DO-178C objectives for DAL A software and estimate level of effort needed to develop necessary evidence for certification.
- Review design and verification evidence available for seL4 and / or other microkernels to determine suitability for certification.
- Consult with potential customers to evaluate market for Gapfruit given the advantages identified in this report.
- Prototype technical solutions to issues identified above, i.e., framework components for implementation of IPC ports, domain scheduling, reconfiguration etc.
- Develop the Software Plans identified by DO-178C. Assume all software in the OS will be developed to DAL A, so objectives will need to reflect this.
- Consider using independent consultancy firm to review plans and accomplishments.
- Identify high level software requirements for Gapfruit for Avionics. These will define *what* the software does, i.e., what functionality it provides. Can base these on Arinc 653 standard functionality, the APEX API and dynamic reconfiguration. HLRs need to provide a level of detail sufficient for the software to be tested.
- Bring existing baselined code into the development process. Review against HLRs, identify components that need modification, new development, or removal. Define discrete tasks for software development after baseline that can be tracked through the software lifecycle process.
- Identify Low level software requirements. These define *how* the software will fulfil its requirements and embody the software design. All LLRs must be traceable to HLRs. LLRs can be reverse engineered from existing code but plans will need to reflect how this will be done. Review code and LLRs for consistency.
- Verify software in accordance with the SVP.
- Consider IDE requirements to allow customer to develop system, configuration files etc.
- Develop User guides and training courses for potential customers.

9 Abbreviations

APEX	Application Executive (Arinc 653 API)
API	Application Programming Interface
Arinc	Aeronautical Radio Incorporated
DAL	Development Assurance Level
EASA	European Aviation Safety Agency
EUROCAE	European Organisation for Civil Aviation Equipment
FAA	Federal Aviation Administration
FCS	Flight Control System
FIFO	First-In First-Out queue
GPS	Global Positioning System
HLR	High Level Requirement
IDE	Integrated Development Environment
IMA	Integrated Modular Avionics
IPC	Inter Process Communication
LLR	Low Level Requirement
MOS	Module Operating System (Arinc 653 component)
OS	Operating System
POS	Partition Operating System (Arinc 653 component)
PSAC	Plan for Software Aspects of Certification
RAM	Random Access Memory
ROM	Read Only Memory
RTCA	Radio Technical Commission for Aeronautics
RTOS	Real-Time Operating System
SCMP	Software Configuration Management Plan
SDP	Software Development Plan
SOI	Stage of Involvement
SQAP	Software Quality Assurance Plan
SVP	Software Verification Plan
UART	Universal Asynchronous Receiver Transmitter

UAV	Unmanned Aerial Vehicle