RUNTIME MONITORING ON HARD REAL-TIME OPERATING SYSTEMS

by

Kaveh Darafsheh

July, 2015

Director of Thesis: Nasseh Tabrizi

Major Department: Computer Science

This thesis will compare and evaluate different approaches in integrating runtime monitors into processes running on a hard real-time operating system. The host system is a single board computer (SBC) with a VxWorks 653 hard real-time operating system henceforth referred to as a flight control computer (FCC). The FCC is an integrated modular avionics (IMA) system representative of actual flight computers. VxWorks 653 is based on the ARINC 653 standard and provides time and space partitioning for IMA systems.

RUNTIME MONITORING ON HARD REAL-TIME OPERATING SYSTEMS

A Thesis

Presented To the Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Computer Science

by

Kaveh Darafsheh

July, 2015

RUNTIME MONITORING ON HARD REAL-TIME OPERATING SYSTEMS

by

Kaveh Darafsheh

APPROVED BY:

DIRECTOR OF THESIS: _____

M.H. Nassehzadeh Tabrizi, PhD

COMMITTEE MEMBER: _____

Krishnan Gopalakrishnan, PhD

COMMITTEE MEMBER: _____

Karl Abrahamson, PhD

COMMITTEE MEMBER: _____

Alwyn E. Goodloe, PhD

CHAIR OF THE DEPARTMENT
OF COMPUTER SCIENCE: _____

Venkat Naidu Gudivada, PhD

DEAN OF THE GRADUATE SCHOOL: _____

Paul J. Gemperline, PhD

ACKNOWLEDGEMENTS

The author would like to thank the members of the department of computer science at East Carolina University for their superb instructions. He would like to specially thank Prof. Nasseh Tabrizi without whose support this thesis would not have happened.

The author would also like to thank the members of the Safety Critical Avionics Systems Branch (D320) of NASA Langley Research Center, without whose input this research would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Early jets required a large crew to control the plane, in addition to the pilot and copilot, a navigator and flight engineer were present in the cockpit. Evolution in hardware and software has improved the ability of avionics systems to increase operational efficiencies by introducing automation, thus reducing crew required, and improving crew situational awareness while increasing safety margins [26].

Early avionics were federated. In federated avionics systems each avionics function has a dedicated computational resource. More modern avionics are integrated modular avionics (IMA). In IMA systems multiple applications are integrated on a single hardware platform. There has been a gradual shift in aviation industry from federated systems towards IMA systems. The move to IMA systems is in part to make avionics systems cheaper, lighter, and more serviceable [32]. In order to achieve this integration, the system must maintain complete separation between applications. The IMA operating system does this by providing space and time partitioning, so failures are localized and recoverable [17]. As avionics reduced the required crew, IMA systems aim to do the same for federated systems, reduce the number of computational resources while maintaining the same safety requirements.

One of the applications in an IMA is the flight control task. The flight control task is composed of sensing, control logic, and actuation subtasks [26]. The flight controller is a control system. The objective of a control system is to control its own output in some manner prescribed by the processing of inputs through the elements of the control system [9]. The purpose of a control task is to govern the behavior of the plane. It does so by taking sensor values, which include the current state of the plane, and pilot commands and computing the changes needed to control surface that place the plane in the desired state [27].

Some control logics are so complex that full verification and certification is not feasible, yet they provide benefits not available from simple controllers. Runtime monitors can be used to alleviate the verification and certification problem [20]. They can be used to implement a simplex architecture [25] where a simple controller, verified using traditional means, exists alongside complex and unverifiable controller in an IMA, but the monitor decides which will be operating. If the monitor predicts the aircraft will be entering an unsafe state it switches to the safe controller, while the complex controller is used only if the monitor deems it safe.

Neither formal verification nor testing can ensure system reliability. Runtime verification (RV), where monitors detect and respond to property violations at runtime, holds particular potential for ensuring that ultra-critical systems are in fact ultra-reliable, but there are challenges. In ultra-critical systems, RV must account for both hardware and software faults. Whereas software faults are design errors, hardware faults can also be a result of random failure [15]. CoPilot is a RV system targeted at hard real-time embedded systems such as civil aviation. CoPilot is an open source Haskell Embedded Domain Specific Language (EDSL) created by researchers at National Aeronautics and Space Administration (NASA), the National Institute of Aerospace (NIA), and Galois as part of an ongoing research effort in this area.

Given the benefit of runtime verification, we would like to evaluate and compare different approaches in integrating runtime monitors into a representative flight system. In order to achieve this, we created a simulation environment where multiple approaches to implementing runtime monitors are compared and evaluated side by side. This simulation environment consists of the Assessment Environment for Complex Systems (AECS) and two Linux hosts. AECS is representative flight system with similar hardware and software as operational units and is composed of four FCCs and a time-triggered Ethernet network to provide real-time

communications between them. Each of the four FCCs has a separate approach to integrating runtime monitors. The two Linux hosts are used to represent sensors and effectors, one is used for generating sensor data and transmitting it to the FCCs and the other is used for receiving and logging sensor data and FCC effector commands. This approach is based on the closed loop controllers where control logic receives sensor data and after performing computations it transmits commands to effectors.

Our contribution in this thesis is the evaluation and comparison of different approaches in integrating runtime monitors into a representative flight system. We encountered the limitations posed by each of the different FCC configurations. A configuration required more memory, some configurations had tasks miss deadlines which caused an operating cycle long delay, and most required great amount of reconfiguration. We also discovered shortcomings in CoPilot that makes usage inconvenient.

CHAPTER 2: RELATED WORKS

The general implementation of real-time software for ARINC 653 based systems has been covered is research such as [2] and [24]. The authors of [2] look at the benefits of the ARINC 653 avionics architecture and gives an overview of the application executive (APEX) application programming interface (API) by presenting a case study and examining the different API calls. The author of [24] also looks at the ARINC 653, APEX API, and the software hardware interface. And it uses a pitch control application as its case study, it covers the controls, timing requirements and the application structure. Both authors use VxWorks 653, Wind River's ARINC 653 compliant real-time operating system (RTOS), for their case studies, in addition the author of [24] uses the Avionics Full-Duplex Switched Ethernet (AFDX) for a communication bus and has to contend with the delays it introduces. The authors of [19] extend the work presented by the author of [24] by implementation functionality on another ARINC 653 compliant RTOS, PikeOS, in addition to VxWorks 653. It also provides test results for the implemented pitch controller through simulation. While there are areas of overlap between these papers and our research, mainly the use of ARINC 653 compliant RTOS, there are enough differences to justify our research. We look at the integration of runtime monitors into an ARINC 653 based system and we use time-triggered Ethernet as the communication bus. Also our implementation is on a hardware platform that is representative of actual flight hardware.

There has been research on scheduling in ARINC 653 based systems. A scheduling algorithm for fault-tolerant IMA systems is provided in [11]. The proposed algorithms is priority preemptive and it uses a deadline-monotonic priority assignment where priorities are inversely proportional to deadline lengths. A new scheduling scheme for APEX partitions is proposed in [10]. APEX partitions have a deterministic cyclic scheduling at the OS level and processes

within partitions have a fixed priority scheduling. This algorithm can be used to guarantee the schedulability of fixed priority algorithms. A compositional techniques for automated scheduling in an ARINC 653 based environment composed of partitions and processes is developed by [8]. This is to alleviate the time consuming approach where system designers manually schedule the system based on interaction with the application vendors. And [39] gives an overview of the rate monotonic scheduling which we used initially to schedule our partitions but with the introduction deterministic communication and requirements for static scheduling it was quickly rendered useless. While they provide valuable insight to scheduling, our scheduling was based on the physical requirements of an aircraft controller. Accordingly we scheduled the system around this deadline. Also since we use time-triggered Ethernet, which provides deterministic communications, the network could be scheduled to deliver the data required by applications before the activation of said applications. This meant that each process that required network data needed a single activation per period. Our scheduler was simplified due to lack of other active application partitions which will not be the case with real systems.

Two level health management strategies to assemblies in software components is applied in [12]. It is not however an actual health management tool that can be implemented in an ARINC 653 system. We applied the approaches presented by the authors and selected the in component and out of component health monitoring to integrate our runtime monitors.

The simplex architecture whose goal is to make software changes safer, easier, and cheaper is introduced by [18]. Safer change comes from allowing change of function without the need for shutdown. The ease of change come from the modular design that allows developers to modify and replace components. The simplex architecture can allow the safe use of an unverifiable complex controller by using a verified switching logic and verified safety controller.

They provide an example for a motion control that performs a safe upgrade of an unstable motion system. An alternate simplex architecture targeted at control systems is proposed by [6]. Simplex allows for safe use of a complex controller by allowing a switching logic to activate a safety controller before an unsafe act takes place. It must ensure that the system never enters an unsafe space and the use of complex controller should be maximized. The alternate design uses control-theoretic optimization and offline hybrid systems reachability computation to show that safety can be maintained while maximizing complex controller use.

A survey of the field of runtime verification is provided by [7]. The runtime verification is intended for distributed hard real-time systems which can be used to increase the reliability of safety-critical systems. The survey is broadly related to monitoring distributed real-time systems. It provides an introduction to field by giving definitions, terminology, and concepts to newcomers. It gives examples of failures of safety-critical systems, and how runtime monitors could be used as a part integrated vehicle health management to provide system reliability. It provides three conceptual architectures for monitoring distributed hard real-time systems and it describes properties need to be monitored. The research in [7] was a preliminary step before creation of CoPilot introduced in [14]. Our goal is to study CoPilot and its integration into a representative safety-critical system and make improvements with the goal of implementing the simplex architecture and using a runtime monitor as the safe switching logic described in [6].

CHAPTER 3: EXPERIMENT SETUP

There are many variables in creating an IMA system and as such many approaches to integrating monitors, but we have settled on two major variables, one variable is monitor's resident partition, whether the monitor should be in the same partition or in a different partition as the task being monitored. The other variable is the granularity of the flight control partition, whether all flight control subtasks should be in the same partition or each in its own partition.

These two variables leave us with four approaches in integrating monitors. These four approaches are as follows, all subtasks and monitor in the same partition, leading to a single partition system. All subtask in the same partition with the monitor in a different partition, leading to a two partition system. Each subtask and its monitor is in its own partition, leading to a three partition system. Each subtask and each monitor is in its own partition, leading to a six partition system.

Figure 1 represent the structure of the experiment's computers and the network. In the coming sections we will explain each component, including the arrows between the boxes. The initial set up only included a single FCC which had the design and timings shown in Figure 2. All FCCs were derived from it. The Sensor/Control/Effector/Monitor partition is composed of 13 tasks, there are four sensing tasks, four control tasks, four actuation tasks, and one runtime monitoring task. The four sensing tasks are for the inertial measurement unit (IMU) sensor, global positioning system (GPS) sensor, airspeed sensor, and the pilot stick. Each task receives sensor data and prepares it for use by control tasks. The four control tasks control the pitch, roll, yaw, and throttle. Each task receives corrected sensor reading and creates an effector demand based on the pilot demand. The four effector tasks control the elevator, aileron, rudder, and engines. Each task converts control commands for use by effectors and transmits it to the

appropriate effector. The monitoring task executes as the last task in the partition and perform runtime monitoring. These tasks were placed in a base FCC and were instrumented to measure their runtime. And 5ms was found to be sufficient amount of time for all tasks to execute successfully.

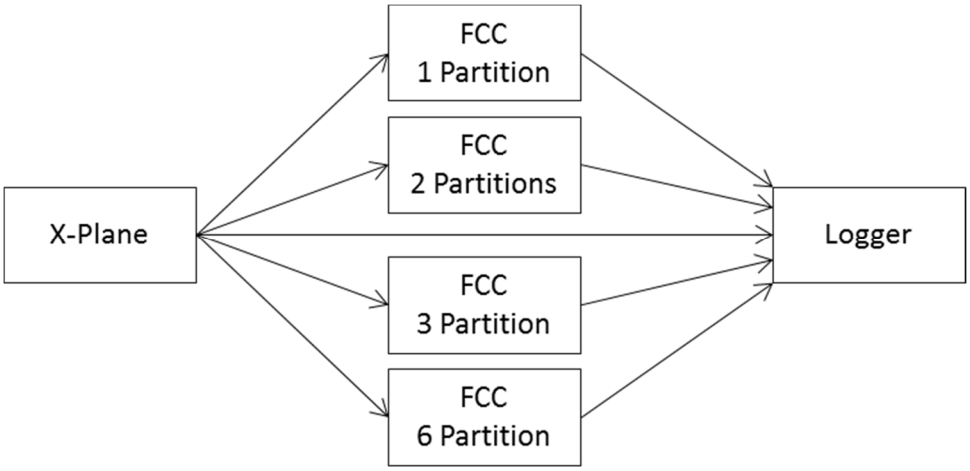

Figure 1: Experiment Setup

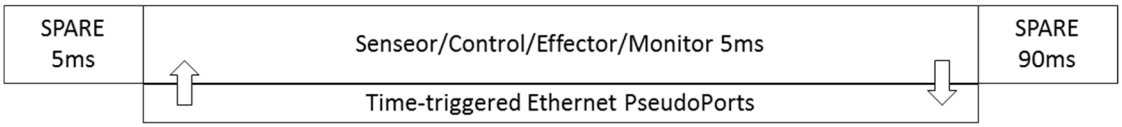

Figure 2: Partitioning of the Base FCC

3.1 VxWorks 653

VxWorks 653 is a certified, DO178C Level A, hard real-time operating system that complies with the ARINC 653P1-3 Specification. ARINC specification 653 part 1 defines an APEX (Application/Executive) interface between an operating system and hosted applications. They provide facilities required for multiple functions to reside on the same hardware, without effecting one another, negatively or positively [35].

This concept of non-interference is generally referred to as robust partitioning. In particular robust partitioning demands that no application function in one partition can under any circumstances: Impact the temporal processing resource allocation of another partition; access the memory assigned to another partition; adversely affect the I/O resources of another partition [13].

The architecture of an ARINC 653 module [1] is shown in Figure 3.

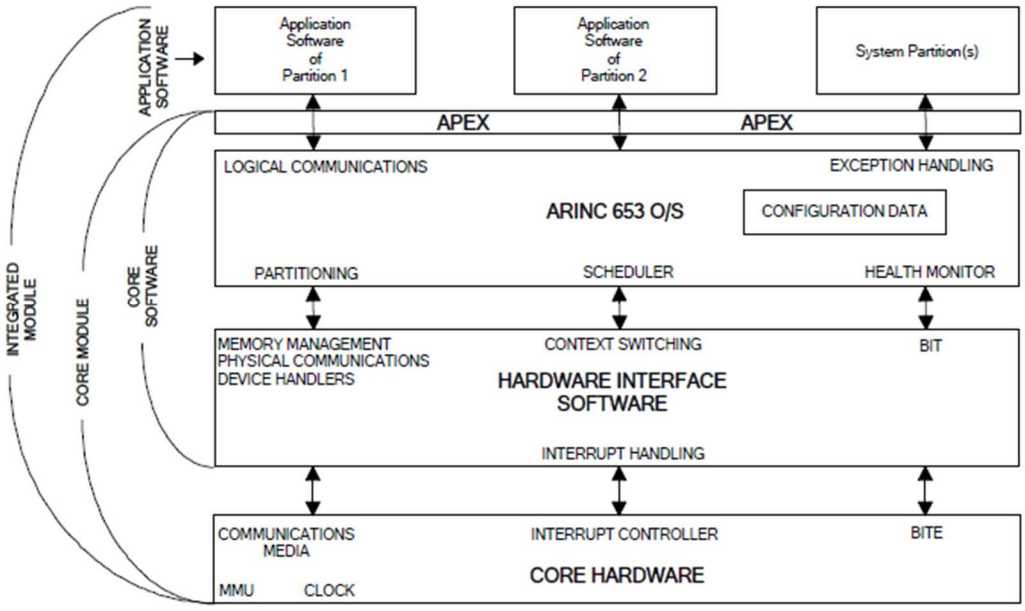

Figure 3: ARINC 653 Module

In the context of FCCs the Core Hardware is typically a single board computer (SBC) which has timers, interrupts, memory management unit (MMU), and possible I/O to the outside of the integrated module. The Hardware Interface Software role is handled by the board support package (BSP) that initializes the hardware and provides drivers for the available hardware [33]. The ARINC 653 OS is a safety critical, real-time, deterministic software that provides the resources required by the ARINC SPECIFICARION 653P1-3.

Multiple roles are associated with a 653 design. The Platform Provider provides the Core Hardware and Hardware Interface Software. This could involve selecting an existing SBC or designing one for the system at hand. And the BSP could be made available by the OS or board vendor or it may be written from scratch for custom hardware [34].

The System Integrator configures the ARINC 653 OS by defining the architecture of the system which includes creating the system schedule, creating the required number of partitions, providing each partition with the resources it requires, time, space, hardware services, and restricting partition behavior.

The Application Developer uses the time and memory resources provided by the System Integrator to the assigned partition to implement the required functionality. Each partition's resources differ based on functionality and criticality of the application it hosts. DO-178C Level D or E software may allocate dynamic memory but Level A software may not.

This is not intended to be a comprehensive treatment on the topic of system configuration, therefore we will only cover configuration elements relevant to the project. In our setup an integrated VxWorks 653 module with 4 partitions has architecture shown in Figure 4. This is only presented to make explanation of the system components easier.

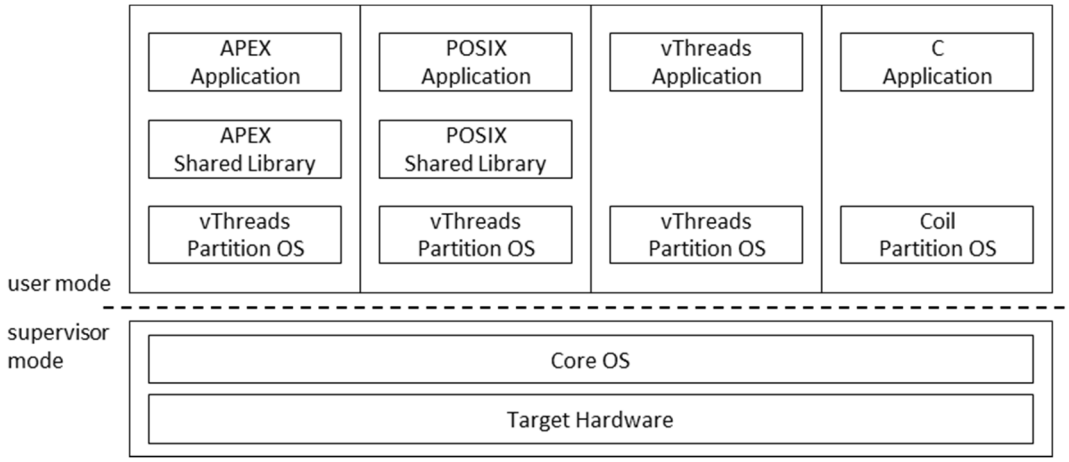

Figure 4: 4 Partition VxWorks 653 Module

The mode, user or supervisor, alludes to the roles involved in the making of a VxWorks 653 module. They are platform provider, system integrator, and application developer. The platform provider provides the target hardware and the BSP that will be used by the core OS. The system integrator configures the system by assigning resources, time and memory, to partitions. The application developer used the assigned resources to develop an application that performs the required function.

The target hardware is a Curtiss-Wright VME-183 board and the core OS the VxWorks 653 operating system. Each partition requires a Partition OS, vThreads provides the default services for task execution. The COIL Partition OS (core OS Interface Library) provides a set of functions that allows the implementation of a non vThreads based Partition OS. A subset of COIL can be used in systems that require DO-178B Level A certification. Each partition could have a library that allows it to provide services to the application it hosts. The APEX shared library provides access the APplication EXecutive functions. The POSIX shared library allows implementation of POSIX compliant applications. And the absence of a shared library allows implementation of vThreads applications. These application are executed on Wind River's non-

653 RTOS offerings such as VxWorks 6.x or 7.x which allows porting of software from other systems.

This choice of partition OS and shared libraries allows for implementation of mixed criticality applications on a single module. It also allows for porting of existing applications.

The amount configuration of the system should be as follows. The module OS should not require configuration since the build structure matches the module hardware and the module hardware is fixed. Change is only necessary when there is a hardware change or driver change. The configuration for the partition OS should be minimal as it only requires indicating which OS or shared libraries are being used. Partitions should require more configuration. Its configuration is composed of Application, Shared Library, Shared Data Region, and Shared I/O Region configuration as well as individual settings such as partition health monitoring. Shared library is present in all partitions since the partition OS is a shared library. An example configuration is shown in Figure 5.

```
<PartitionDescription  xmlns=" "  xmlns:xsi=" "  xsi:schemaLocation=" ">
  <Application NameRef="exampleApplication"/>
  <SharedLibraryRegion NameRef="fccPos"/>
  <SharedDataRegion NameRef="exampleSharedDataRegion" UserAccess="READ_WRITE"/>
  <Settings
    RequiredMemorySize="0x100000"
    PartitionHMTable="fccAppHm"
    maxGlobalFDs="10"
    numFiles="0xFFFFFFFF"
    numDrivers="0xFFFFFFFF"
    isrStackSize="0xFFFFFFFF"
    syscallPermissions="0xFFFFFFFF"
    maxEventQStallDuration="INFINITE_TIME"
  />
</PartitionDescription>
```

Figure 5: Example Partition Configuration

Applications configuration include memory allocation and port definitions for interpartition communication. Other configurations should be easy to understand based on their names. An example configuration is shown in Figure 6.

```
<ApplicationDescription   xmlns=" "   xmlns:xsi=" "   xsi:schemaLocation=" ">
  <MemorySize
    MemorySizeBss="0x10000"
    MemorySizeText="0x10000"
    MemorySizeData="0x10000"
    MemorySizeRoData="0x10000" />
  <Ports>
    <QueuingPort
      Name="exampleQueuingPort”
      Direction="DESTINATION"
      MessageSize="1000"
      QueueLength="10"
      Protocol=" NOT_ APPLICABLE" />
    <SamplingPort
      Name="exampleSamplingPort"
      Direction="SOURCE"
      MessageSize="1000"
      RefreshRate="10000" />
  </Ports>
</ApplicationDescription>
```

Figure 6: Example Application Configuration

The shared data region has the following attributes, this could have been included the application configuration but since it’s shared between partitions it was placed in a separate file. An example configuration is shown in Figure 7.

```
<SharedDataDescription
   xmlns=" "    xmlns:xsi=" "    xsi:schemaLocation=" "
   CachePolicy="DEFAULT"
   DataType="VIRTUAL"
   Size="0x1000"
   SystemAccess="READ_WRITE"
   >
</SharedDataDescription>
```

Figure 7: Example Shared Data Region Configuration

Communication within APEX partitions can be done using the following APEX objects, buffers, blackboards, semaphore, and event. Communication between partitions can be done using messages, ports, and channels.

Ports could be queuing or sampling, a queuing ports can accommodate multiple messages, up to their queue length, while sampling ports contain a single message whose data will be overwritten on each on each write. All ports could have the following properties, Name, Direction, Attribute, and DriverName. The first two are required while the last two are optional. Attribute can be used to indicate port type while DriverName indicates which driver should be used, we use them in our PseudoPort configurations.

Queuing port have these attributes in addition to the base properties, MessageSize, QueueLength, Protocol, and RefreshRate. The first three are required while the last is optional. Protocol corresponds to the port direction and could be RECEIVER_DISCARD, SENDER_BLOCK, or NOT_APPLICABLE. And RefreshRate is the maximum age for a message before it's considered stale.

Sampling port have these attribute in addition to the base properties, MessageSize, and RefreshRate. Both are required and RefreshRate sets the expected refresh rate for the port.

PseudoPorts and PseudoPartitions are used to provide communication to the outside world through APEX compliant methods. The configuration shown in Figure 8 creates a queuing port that receives messages on the Time-Triggered Ethernet network. Since the Time-Triggered Ethernet network has a separate configuration process each port's configurations should have similar properties on sender, transmitter, and network switches.

```
<PseudoPartitionDescription   xmlns=" "   xmlns:xsi=" "   xsi:schemaLocation=" " >
  <Ports>
    <QueuingPort
      DriverName="ttePseudoPortDrv"
      Attribute="DIRECT_ACCESS_PORT"
      Name="pseudo_examplePort"
      Direction="SOURCE"
      MessageSize="1000"
      QueueLength="10"
      Protocol="RECEIVER_DISCARD" />
  </Ports>
</PseudoPartitionDescription>
```

Figure 8: Example PseudoPort Configuration

The item with most configuration is the Module since it integrates the system and contains every other configuration. It include the core OS configuration, all application configurations, all shared data region configurations, all shared library configurations, all partition configurations, module schedules, connection information, system health monitor configuration, and the payload configuration. An example configuration is shown in Figure 9.

Module schedules indicated the schedule of the module. The partitions are scheduled in a round robin fashion. Each schedule, indicated by the id, is referred to as the Major Frame and each item in a schedule is called a Minor Frame. Each minor frame's duration is set in seconds so Duration="0.1" indicates a duration of 100ms. The allowable minimum duration of minor frames is set by the Core OS. Connection information indicates how the ports created in

application configuration or pseudo partition configuration are connected to one another. System health monitor configuration include the system, module, and partition health monitor information. This information indicates how each component will react to the faults it catches. Payload configuration indicates where in memory each item resides.

```
<Module  xmlns=" "  xmlns:xi=" "  xmlns:xsi=" "  xsi:schemaLocation=" ">
  <CoreOS>  <xi:include href="bsp.xml"/>
  </CoreOS>
  <Applications>
    <Application Name="applicationOne">
      <xi:include href="applicationOne.xml"/>
    </Application>
  </Applications>
  <SharedDataRegions>
    <SharedData Name=" exampleSharedDataRegion ">
      <xi:include href=" exampleSharedDataRegion.xml"/>
    </SharedData>
  </SharedDataRegions>
  <SharedLibraryRegions>
    <SharedLibrary Name="fccPos">
      <xi:include href="fccPos-shlib.xml"/>
    </SharedLibrary>
  </SharedLibraryRegions>
  <Partitions>
    <Partition Name="partitionOne" Id="1">
      <xi:include href="examplePartitionOne.xml"/>
    </Partition>
    <PseudoPartition Name="pseudo" Id="2">
      <xi:include href="examplePseudoPartition.xml"/>
    </PseudoPartition>
  </Partitions>
  <Schedules>
  <Schedule Id="0">
    <PartitionWindow PartitionNameRef="partitionOne" Duration="0.1" ReleasePoint="true"/>
  </Schedule>
  </Schedules>
  <Connections>
    <Channel Id="0">
      <Source PartitionNameRef="pseudo" PortNameRef="pseudo_examplePort"/>
      <Destination PartitionNameRef="partitionOne" PortNameRef="examplePort"/>
    </Channel>
  </Connections>
  <HealthMonitor/>
  <Payloads>
    <CoreOSPayload/>
    <SharedLibraryPayload NameRef="fccPos"/>
    <ConfigRecordPayload NameRef="configRecord"/>
    <PartitionPayload NameRef="applicationOne" Base_Address="0x04800000" Online="true"/>
  </Payloads>
</Module>
```

Figure 9: Example Module Configuration

Using these resources we implemented four different configurations on the FFCs. These four configurations are: all subtasks and monitor in the same partition, leading to a single partition system. All subtask in the same partition with the monitor in a different partition, leading to a two partition system. Each subtask and its monitor is in its own partition, leading to a three partition system. Each subtask and each monitor is in its own partition, leading to a six partition system.

Figures 10 through 13 will better show the different configurations. They represent the partitioning, schedules, as well as the data flow across FCCs. The top layer is the partitioning scheme as well as the scheduling, the lower layers represent communications methods and arrows represent the data flow. The SPARE partition is predefined in the VxWorks 653 OS. It is used by the core OS to perform system tasks, it can also be used as a place holder for nonexistent partitions to emulate elapsed time. The numbers used to describe the number of partitions in a FCC refer to functional partitions and not SPARE partitions.

In the single partition system all subtasks and monitor in the same partition.

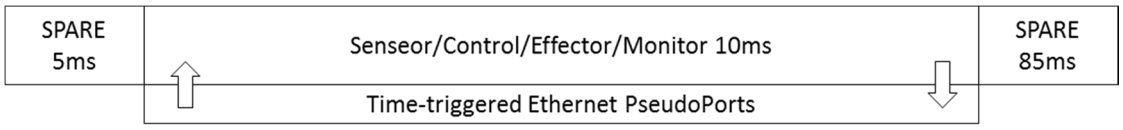

Figure 10: Single Partition System

In the two partition system all subtask in the same partition with the monitor in a different partition.

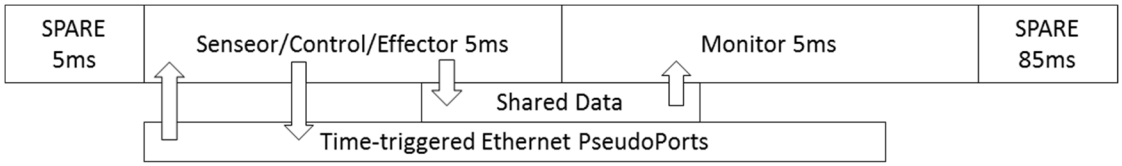

Figure 11: Two Partition System

In the three partition system each subtask and its monitor is in its own partition.

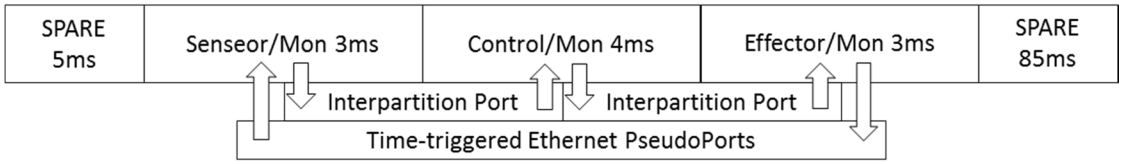

Figure 12: Three Partition System

In the six partition system each subtask and each monitor is in its own partition.

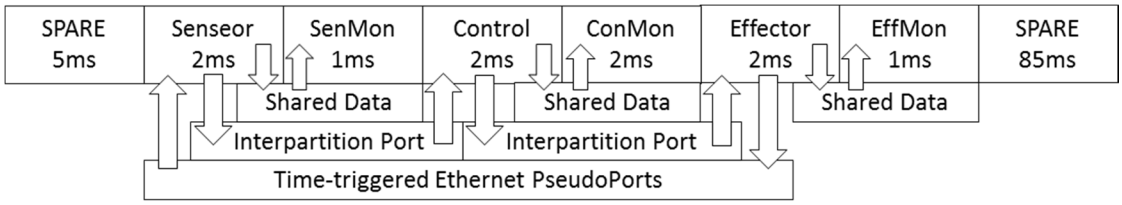

Figure 13: Six Partition System

3.2 Time-Triggered Ethernet

Time-Triggered Ethernet standard is a Layer 2 Quality-of-Service (QoS) that defines time-triggered services for Ethernet, IEEE 802.3, networks [21]. It provides deterministic communications, predictable real-time behavior, minimized jitter, and built-in fault tolerance over an Ethernet network. It can accommodate time-triggered, rate-constrained, and best-effort Ethernet traffic on the same network. The fault tolerance protects against benign, transmissive symmetric, omissive symmetric, and strictly omissive asymmetric faults, it does not protect against transmissive asymmetric faults however. More details about the fault model can be found in [5].

Time-triggered Ethernet uses a Time Division Multiple Access (TDMA) scheme to avoid collisions on the network. There is a global notion of time, which allows for a single global schedule, which gives each node access to the network. Each nodes on the network knows when it can transmit messages or when it should expect a message. This require that all devices, including the switches, on the network execute a synchronization protocol

Time-Triggered Ethernet usage has two phases, configuration and communication. The configuration phase involves creating a network description containing all nodes, links, and behaviors, building binaries based on the network description and loading the binary to the network devices. The communication phase involves using the time-triggered Ethernet's application programming interface (TTE-API) to send and receive messages over the network [28]. These message can be time-triggered, rate-constrained, or best-effort Ethernet messages.

For rate-constrained traffic, each Ethernet link has a bandwidth limit per time defined. The time period is referred to as bandwidth allocation gap (BAG). Rate-constrained only ensure bandwidth limitation, not timing, jitter, or transmit order.

Each node has a schedule, transmitters have a transmit window, switches have accept and forward windows and receivers have a receive window. Each link in each direction has a schedule of its own, but all schedules are synchronized to one global clock per sync domain.

Time-triggered Ethernet has two periods at minimum, integration cycle and communication cycle. Integration cycle is for clock synchronization. Communication cycle is for communications and is an integer multiple of an integration cycle.

Configuration phase involves creating an XML based network description. This file defines the network topology by specifying switches, nodes, and physical links. It contains traffic information, time-triggered, rate-constrained, and best-effort virtual links. It includes scheduling constraints [30]. Below is an overview of some of the elements in a configuration file.

The highest level is a *NetworkDescription*, it includes all schemas used and their location. An example description is shown in Figure 14. The schemas differ based on the version of the TTE-Plan used. All other items are nested within its tags. They were removed from the textbox below in favor of space.

The number of channels in the network is defined by *redundancy*. The tool creates extra identical switch configurations, once per channel. With the exception of management interface and best-effort routes. *ctMarker* is a marker used to identify critical traffic, time-triggered and rate-constrained, on the network. *enableDynamicRouting* is used for switch address learning, a false value requires best-effort routing to be configured statically. Since we are not interested in

best-effort traffic on our system and the extent of its usage is loading switch configurations it is set to false. *createUnknownDefaultRoutes* could be used in conjunction with *enableDynamicRouting* for best-effort traffics, but we have it set to false.

```
<nd:NetworkDescription
  xmlns:xsi=" " xmlns:buf=" " xmlns:logical=" " xmlns:nd=" " xmlns:topo=" " xmlns:vl=" " xmlns:c=" "
  xsi:schemaLocation=" "
  name="fccNetwork"
  interfaceVersionNumber="1"
  transmissionSpeed="1000Mbps"
  redundancy="2"
  ctMarker="AB:AD:BA:BE"
  enableDynamicRouting="false"
  createUnknownDefaultRoutes="false">
…
…
…
</nd:NetworkDescription>
```

Figure 14: Example High Level Network Description

The subset of *NetworkDescription* that contains the scheduler options is the *processingInstructions*. An example is shown in Figure 15. *seed* allows the tool to generate a new schedule without any modifications having to be made to the rest of the configuration file. *minimum_delta_r* defines the time between two consecutive Time-triggered frames. The value 2 indicates there will be an empty raster tick between two messages. This allows rate-constrained and best-effort messages to be sent on the network in addition to time-triggered messages. *raster_granularity_ns* defines the granularity of each raster tick. A unitless number is in nanoseconds.

```
<processingInstructions>
 <schedOption
    Key="seed"
    Value="12345"/>
 <schedOption
    Key="minimum_delta_r"
    Value="2"/>
 <schedOption
    Key="raster_granularity_ns"
    Value="20000"/>
</processingInstructions>
```

Figure 15: Example Processing Instruction

All periods used in the network, for integration cycles or virtual links, should be defined and included. Example periods are shown in Figure 16.

```
<period name="COMMUNICATION_PERIOD" time="100 ms"/>
<period name="IMU_PERIOD" time="100 ms"/>
<period name="AIR_DATA_PERIOD" time="100 ms"/>
<period name="PILOT_STICK_PERIOD" time="100 ms"/>
<period name="GPS_PERIOD" time="100 ms"/>
<period name="ACTUATOR_PERIOD" time="100 ms"/>
```

Figure 16: Example Periods

Synchronization is only required for time-triggered traffic. Each network can have one synchronization domain and up to *n* synchronization priorities. An example synchronization configuration is shown in Figure 17. *clusterPeriod* is the period of the communication cycle. It should be a multiple integer of other periods in the network. *integrationCycleDuration* in the duration of one integration cycle. *faultTolerance* could be set to protect against 0, 1, or 2 faults with either standard or high integrity. For our setup we selected 0FTSI_2SM, which is 0 fault Tolerance Standard Integrity and it allows synchronization with only two synchronization masters, i.e. end systems. *precision* is the precision of the clock synchronization.

```
<syncDomain
   name="syncDomain_1"
   clusterPeriod="#//@period[name='COMMUNICATION_PERIOD']"
   integrationCycleDuration="1000000 ns"
   faultTolerance="0FTSI_2SM"
   precision="5008 ns"
   fullCBG="true"
   value="0">
  <syncPriority
     name="syncPriority_1"
     value="1"/>
</syncDomain>
```

Figure 17: Example Synchronization Options

Device configuration for switches and nodes is different. Both require basic information like *names* and *syncRoles* but most other parts are different. For switches we define physical ports, best-effort routes, and management interface information. An example switch configuration is shown in Figure 18.

```
<device
   xsi:type="topo:Switch"
   name="sw0"
   syncRole="syncCompressionMaster"
   syncPriority="#//@syncDomain/@syncPriority[name='syncPriority_1']"
   deviceTargets="platform:/plugin/com.tttech.tte.targetdevices/data/ND_Target_Devices.targets
#//@targetDeviceDescriptors[name='TTE_Dev_Switch_12port_1G']" >
   <port name="sw0_P1" type="P1"/>
   <port name="sw0_P2" type="P2"/>
   <port name="sw0_P3" type="P3"/>
   <port name="sw0_P4" type="P4"/>
   <port name="sw0_P5" type="P5"/>
   <port name="sw0_P6" type="P6"/>
   <port name="sw0_P7" type="P7"/>
   <port name="sw0_P8" type="P8"/>
   <port name="sw0_P9" type="P9"/>
   <port name="sw0_P10" type="P10"/>
   <port name="sw0_P11" type="P11"/>
   <port name="sw0_P12" type="P12"/>
   <port name="sw0_PMGMT" type="PMGMT"/>
   <port name="sw0_PSYNC" type="PSYNC"/>
   <bestEffortRoute
      …
   />
   <managementInterface>
      …
   </managementInterface>
  </device>
```

Figure 18: Example Switch Description

The best-effort routes are used for best-effort traffic. At a basic level every switch needs a best-effort route for reprogramming. This route is needed so the outside world can access the management interface. In operational systems this route should be removed. In our experimental setup we restricted the routes to the port dedicated to switch programming. An example best-effort route description is shown in Figure 19.

Note the channel property of each route in Figure 19. Since the program automatically creates a duplicate per the redundancy option we need to create as many sets of best-effort routes

as our redundancy option. The extra routes used by the duplicate channels should have different values for *destinationMacAddress* since that address connects to the management interface. And the management interface will be different per switch.

```
<bestEffortRoute
  destinationMacAddress="02:02:02:02:04:2F"
  addrMask="ff:ff:ff:ff:ff:ff"
  srcPorts="#//@device[name='sw0']/@port[name='sw0_P8']"
  dstPorts="#//@device[name='sw0']/@port[name='sw0_PMGMT']"
  channel="1"/>
<bestEffortRoute
  destinationMacAddress="02:02:02:02:08:2F"
  addrMask="ff:ff:ff:ff:ff:ff"
  srcPorts="#//@device[name='sw0']/@port[name='sw0_PMGMT']"
  dstPorts="#//@device[name='sw0']/@port[name='sw0_P8']"
  channel="1"/>
<bestEffortRoute
  destinationMacAddress="02:02:02:02:00:28"
  addrMask="ff:ff:ff:ff:ff:ff"
  srcPorts="#//@device[name='sw0']/@port[name='sw0_P8']"
  dstPorts="#//@device[name='sw0']/@port[name='sw0_PMGMT']"
  channel="1"/>
```

Figure 19: Example Best-Effort Route Description

The management interface allows the switch to be reprogrammed. If a switch's current configuration lacks this, the switch must be put into bootstrap mode. The bootstrap mode has a management interface with predefined addresses. On our experimental set up the switches can be put into the bootstrap mode by holding the reset button while powering the unit [29]. This requires physical access to the switch. Due to the duplication performed by the redundancy option, as many management interfaces should be included as the redundancy value. Each extra management interface should have a different address set, this address set should match the best-effort routes set earlier for extra channels. An example *managementInterface* description is shown in Figure 20.

```
<managementInterface
  sourceAddress="02:02:02:02:04:2F"
  unlockDestAddress="02:02:02:02:08:2F"
  channel="1">
 <macAcceptanceEntry
   acceptanceMacAddress="02:02:02:02:00:28"
   addressType="nonCriticalTraffic"
   unlockEnabled="true"
   resetEnabled="true"
   responseDestMacAddress="02:02:02:02:08:2F">
  <accessControl
    page="0"
    writeEnable="true"/>
  <accessControl
    page="1"
    writeEnable="true"/>
  <accessControl
    page="2"
    writeEnable="true"/>
  <accessControl
    page="3"
    writeEnable="true"/>
 </macAcceptanceEntry>
</managementInterface>
```

Figure 20: Example Management Interface

For end systems we define physical ports, best-effort routes, and partitions. Best-effort routes are required for best-effort traffic, we didn't include any since we do not use best-effort traffic. Partitions are used to group data ports. Each device must have at least one partition, and can have an arbitrary number of partitions. Each partition must have a unique name and be assigned to a physical port. Data ports represent the termination points of virtual links. They can be of multiple types, COM (for critical traffic) or SAP (for traffic between service access ports) on MAC, IP, or UDP layers. We are interested in time-triggered traffic on MAC layer. COM-UDP layer is used for ARINC 664p7, Avionics Full-Duplex Switched Ethernet (AFDX), critical traffic. Max payload size depends on the transport layer and network interface card, for MAC layer it's limited to 1500 bytes but IP and UDP layers can accommodate 8200 bytes. Each port

can have a buffer associated of either sampling or queuing type. An example end system description is shown in Figure 21.

```
<device
  xsi:type="topo:EndSystem"
  name="fcc"
  syncRole="syncMaster"
  syncPriority="#//@syncDomain/@syncPriority[name='syncPriority_1']"
  deviceTargets="platform:/plugin/com.tttech.tte.targetdevices/data/ND_Target_Devices.targets
#//@targetDeviceDescriptors[name='TTE_PMC_ESys_1G']">
 <port name="fcc_P1" type="P1"/>
 <port name="fcc_P2" type="P2"/>
 <port name="fcc_P3" type="P3"/>
 <port name="fcc_PMGMT" type="PMGMT"/>
 <port name="fcc_PSYNC" type="PSYNC"/>
 <port name="fcc_PHOST" type="PHOST"/>
 <partition
  name="fccPart"
  physicalPort="#//@device[name='fcc']/@port[name='fcc_PHOST']">
  <dataPort
    xsi:type="logical:RxComMacPort"
    name="fccIMU"
    maxPayloadSize="100"
    macType="0">
   <buffer
     xsi:type="buf:QueueingBuffer"
     bufferDepth="10"/>
  </dataPort>
  <dataPort
    xsi:type="logical:TxComMacPort"
    name="fccAileron"
    maxPayloadSize="100"
    macType="0">
    <buffer
     xsi:type="buf:QueueingBuffer"
     bufferDepth="1"/>
  </dataPort>
  …
 </partition>
 …
</device>
```

Figure 21: Example End System Description

The physical topology of the network is created using *physicalLink*. It is used to create a connection between defined end systems and switches, or in the event best-effort traffic, any connected device and switches. Each physical connection, plugged in cable, requires one. *transmissionSpeed* can be 10Mbps, 100Mbps, or 1000Mbps. And *mediaType* can be copper or fiber. An example *physicalLink* description is shown in Figure 22.

```
<physicalLink
   name="fcc_link"
   transmissionSpeed="1000Mbps"
   mediaType="copper"
   port="#//@device[name='fcc']/@port[name='fcc_P1']
#//@device[name='sw0']/@port[name='sw0_P1']"
   cableLength="0 m"/>
```

Figure 22: Example Physical Link Description

The logical topology of the network is created using *ethernetLink*. It can be time-triggered, rate-constrained, or best-effort. It connects one sender to one or more receivers. The connection is between the sender and receivers' *dataPorts*. The *vlid* is used during end system programming to send or receive data from *dataPorts*. The *redundancyCheck* could be *integrity_check*, *rc_redundancy*, *tt_redundancy*, and *no_check*. The first two are required for compliance with the AFDX standard, and *no_check* means not redundant copies of messages are removed. *tt_redundancy* uses time of arrival to pass the first copy and discard redundant copies. An example *ethernetLink* description is shown in Figure 23.

```
<ethernetLink
   xsi:type="vl:TTVirtualLink"
   name="IMU_LINK"
   senders="#//@device[name='xp']/@partition[name='xpPart']/@dataPort[name='xpIMU']"
   receivers="#//@device[name='logger']/@partition[name='lgrPart']/@dataPort[name='lgrIMU']
   #//@device[name='fcc']/@partition[name='fccPart']/@dataPort[name='fccIMU']"
   maxFrameSize="118"
   vlid="501"
   redundancyMgmt="tt_redundancy"
   period="#//@period[name='IMU_PERIOD']"/>
```

Figure 23: Example Ethernet Link Description

Although the schedule is generated automatically constraints allow for a degree of fine tuning. TTE-Plan ensures periods and not the causal order of virtual links. For some applications transmit and receive order are important. In our application the effector commands cannot be transmitted before sensor messages have been transmitted. Also there must be a delay between sensor data transmission and effector command transmission to account for the computation time required for the control system.

To TTE-Plan this is a list of independent virtual links with set periods while there may be a causal relationship between some of the virtual links. Using these constraint we can ensure that the network schedule is causal. This requires knowledge of system's timing requirements. *SendTimeConstraint*, constrains the transmitting end system's access window to the network for message transmission. *TransmissionDurationConstraint*, constrains the duration of message's time on network before it reaches the receiving end system. *ReceiveTimeConstraint*, constrains the message's arrival time at the receiving end system. An example description of constraints is shown in Figure 24.

```
<constraint
   xsi:type="c:SendTimeConstraint"
   name="IMU_LINK_constraint"
   virtualLinks="#//@ethernetLink[name='IMU_LINK']"
   windowStart="0ms"
   windowEnd="5ms"
   devices="#//@device[name='xplane']" />
<constraint
   xsi:type="c:TransmissionDurationConstraint"
   name="IMU_LINK_max_trans_dur_constraint"
   maxTimeSpan="1ms"
   virtualLinks="#//@ethernetLink[name='IMU_LINK']" />
<constraint
   xsi:type="c:ReceiveTimeConstraint"
   name="FCC5_AILERON_LINK_constraint"
   virtualLinks="#//@ethernetLink[name='FCC5_AILERON_LINK']"
   windowStart="10ms"
   windowEnd="15ms"
   devices="#//@device[name='xplane'] #//@device[name='logger']" />
```

Figure 24: Example Constraint Descriptions

Using the above tools we can configure a system of any topology or timing. Real systems are more complex than our setup. Having configured the network we should focus on programming the end systems.

The TTEthernet API is designed so that access to the TTEthernet protocol is platform independent. It can be accessed either through an OS with a TTEthernet driver or directly by an application in the absence of an OS. The basic TTEthernet API is a set of functions, data types, and constants that are always provided to the TTEthernet application, regardless of the underlying hardware [31].

End system programming involves three phases, initialization, communication, and termination. The initialization phase requires the tree calls shown in Figure 25. They initialize the API, configure the controller, and executes the synchronization protocol. Initialization is

required since API functions cannot be called unless the API has been initialized. The controller configuration step is provided by the application and it involves reading the Intel HEX format configuration file generated by TTE-Build and loading it into the controller. Synchronization is required before any time-triggered messages can be transmitted or received.

```
tte_init ()
configure (ctrl_id, configuration)
tte_start (ctrl_id)
```

Figure 25: Calls Necessary for Initialization of TTE API

The communication phase has calls for transmitting or receiving time-triggered, rate-constrained, and best-effort traffic. The calls range from simple and low performance to difficult and high performance.

The first method is direct copy. The function shown in Figure 26 writes the message to controller memory, which will be transmitted according to the schedule. This method is not recommended for real-time code since it may be non-deterministic.

```
tte_write_tt_msg (ctrl_id, frame)
```

Figure 26: Call Necessary for Direct Copy

Second method is use of buffers and direct copy. The first function call in Figure 27 retrieves the buffer handle and second function call writes the message to the buffer, which will then be transmitted according to the schedule. The first function call prevent non-determinism at runtime since there is no need to look up the critical traffic id at runtime. The first method use this method internally.

```
tte_get_tt_output_buf (ctrl_id, ct_id, buffer)
tte_write_output_buf (buffer, frame)
```

Figure 27: Calls Necessary for Buffer and Direct Copy

Third method is use of buffers and direct memory access (DMA). This method resembles the second method but has the added steps for DMA operations. The steps necessary are shown in Figure 28 and they involve allocating memory for DMA and setting its status to active, initializing the output buffer, setting it to DMA mode, and sending messages using the buffer. Although they may appear burdensome, this is the only way to utilize the full bandwidth of the network.

```
tte_set_dma_size (ctrl_id, size)
tte_set_dma_active (ctrl_id, ACTIVE)
tte_get_tt_output_buf (ctrl_id, ct_id, buffer)
tte_set_dma_buf_mode (buffer, DMA_ACTIVE)
tte_write_output_buf (buffer, frame)
```

Figure 28: Calls Necessary for Direct Memory Access

The termination phase requires two calls, they are shown in Figure 29. They stop the execution of the synchronization protocol and free all resources allocated by the API. Optionally a configuration function, implemented by the application, could be called to erase the old configuration by loading a dummy configuration file into the memory of controller.

```
tte_stop (ctrl_id)
configure (ctrl_id, configuration)
tte_exit ()
```

Figure 29: Calls Necessary for Termination of the TTE API

3.3 X-Plane

X-Plane is a software flight simulator. It is high fidelity, it can be part of a flight simulator certified as training devices by the Federal Aviation Administration (FAA) [36]. This requires additional certified hardware. It is extensible and the provided SDK can be used to develop plugins and extend functionality [37]. It is also used by other projects at the NASA such as [22] and [23]. This provides both access to experienced developers and it also allow for compatibility with existing research projects. We use it as a source for data (input and sensor data), flight dynamics, and flight visualization.

X-Plane provides data output and input. The data can be from a wide range of sources onboard a craft. But the variables of interest are the g-loads (normal, axial, side) on the aircraft, angular velocities (Q, P, and R representing pitch, roll, and yaw), pitch, roll, and true heading, latitude, longitude, altitude, and airspeed. These values represent the data collected from a set of sensors and input devices, which are IMU/GPS, pitot tube, and pilot stick. As for the output, it can be over network, to a file on disk, or on screen display.

This data set is of interest because it's used by ArduPilot to provide software or hardware in the loop (SIL or HIL) simulation. SIL and HIL are two simulation methods that allow testing of the system without flight testing. SIL allows testing of the software, while HIL allows testing of hardware, to a degree, as well. These methods allow for online or offline flights, where a pilot or autopilot controls the plane in a live flight while in the other the plane is flown with a set of recorded data to see how the simulation pairs with the actual flight.

In order to use time-triggered Ethernet we must use the TTE API which means employing the X-Plane SDK. The SDK allows for writing of plugins that provide additional

features and functionality to X-Plane. The same data sources available through the GUI are also available through the SDK and are called datarefs [38]. Datarefs can be used to send the same data over time-triggered Ethernet. They can also be used to give commands to the system by overwriting them. This requires that their access be blocked to the system and be only available through the plugin.

Using this we wrote a plugin to transmit sensor data and receive actuator commands, although the latter part was not used. We also included an error injection capability in our plugin. The plugin is aware of what the acceptable ranges for sensors are and it can transmit values which fall outside those ranges. For our tests he error injector looks at the round numbers and injects error on a predetermined schedule, this is to ease testing.

3.4 ArduPilot

ArduPilot is a project that aims to provide software that can controls Planes, Multi-Copters, and Rovers [3]. The software is intended to be executed on low cost Arduino boards hence the name. The project's growth has allowed it to expand beyond Arduino boards and to other hardware platforms. This software allows hobbyist to control and fly remotely piloted vehicles. This platform provide for simple, rapid, and economical flights. And due to this nature it is used by research projects at NASA such as [22] and [23] that need to fly experiments without needing to focus on the platform. This provides both access to experienced developers and it also allow for compatibility with existing research projects.

In order to use ArduPilot three components are needed, a vehicle specific firmware, a hardware to execute the firmware, and the ground based Mission Planner software to observe the status of the plane, update flight plan, and view the flown path [4]. This software also allows for programming the firmware into the hardware. All of this would be used with a remotely controlled vehicle that has a set of sensors and actuators.

Our interest is in the firmware. The firmware includes many libraries, such as barometer, camera, geo-fencing, and navigation libraries but our interest is in the control algorithm. In general the purpose of a control algorithms is to govern the behavior of a physical process by producing a set of outputs that place the process in the desired state. For ArduPilot the control algorithm governs the behavior of the vehicle by sending values to actuators, steering components such as control surfaces, based on the current state of the vehicle and the operator demand. We extracted the subset of classes that provide the control laws for a fixed wing craft and build the architecture of each FCC around them.

3.5 CoPilot

CoPilot is an embedded domain-specific language implemented in Haskell that is intended for creating runtime monitors for hard real-time, distributed, reactive systems. A hard real-time system is a system that has a statically bounded execution time and memory usage. A distributed system is a system which is layered out on multiple pieces of hardware. A reactive system is a system that responds continuously to its environment. Runtime monitors are programs that execute alongside the program under observation and verify whether program under observation is acting as defined or whether it's deviating from the predefined behavior. Therefore runtime monitors are specifications for correct system behavior [16].

The steps required to include runtime monitoring to a project are writing and reifying a specification, including it into an existing code and placing the necessary monitoring code, and compiling the system into a binary. Most focus should be on the specification writing since reifying, including, and compiling are mechanical processes.

The general anatomy of a module is shown in Figure 30. Three sections are of interest, the declaration of external variables, creation of necessary functions, guard or utility, and creation of at least one specifications with a trigger. Each module could contain multiple specifications but only one can be reified. And each specification can have multiple triggers but each trigger requires a unique external function and guard.

The external variables are values of interest to the monitor. They can be strings or any of the primitive types, signed or unsigned integrals, floating points, doubles, etc.

The functions can act as guards for triggers or be used in the specification to compute some value of interest. Guard function are determine when each trigger is pulled.

Triggers are how CoPilot effects outside world and are composed of three parts; they are an external function which is called when its trigger in the specification is pulled, a guard which determines when a trigger should be pulled, and a list of arguments to pass to the external function when the trigger is pulled. While the triggers are written by the specification writer the external functions does not have to be, they can be an existing function in the system under observation.

```
module ExampleMonitors where

import Language.Copilot
import Copilot.Compile.C99
import qualified Prelude as P

airspeed :: Stream Float
airspeed = extern "airspeed" Nothing

outOfBounds value low up = up < val || val < low || up < low
airspeedRangeCheck airspeed min max = outOfBounds airspeed min max

rangeCheck :: Spec
rangeCheck = do
   trigger "airspeedTrigger" (airspeedRangeCheck airspeed 0 761) [ airspeed ]
```

Figure 30: General Anatomy of a Runtime Monitor Module

To reify a spec the command in Figure 31 should be executed in a Glasgow Haskell Compiler interactive window where name is the *FileName.hs* is the Haskell file where the monitor specifications reside and *SpecificationName* is the name of the specification name we would like to reify to C code.

```
GHCI> :load FileName.hs
GHCI> reify SpecificationName >>= compile defaultParams
```

Figure 31: Commands Necessary to Reify a Specification to C Code

To integrate the generated code based on the specification include the generated header file in the file where the main executable is and add the function *step ()* inside the function where the main thread of execution resides. It is recommended to place it as the last call or after all the properties it monitors have been set. After this step has been completed the monitored system's code can be compiled into a binary. Figure 32 shows an overview of this integration process.

```
#include "copilot.h"
…
main () {
  …
  step ();
}
```

Figure 32: Overview of Runtime Monitor Integration into a File

Using this knowledge we wrote a specification to monitors the values of sensor data received from X-Plane and actuator commands transmitted from the FCC. The monitors perform range checks on received, computed, and transmitted values. The minimum and maximum for all the values were determined. The determination was based on the nature of the value measured, heading must be between 0 and 360 degrees, or the vehicle's characteristics, airspeed confined between 0 and 761 mph, or operational limitations, class A airspace confined between 18,000 and 60,000 feet mean sea level.

Another set of triggers were added to the specification for the messages received by FCCs. Each message contains a serial number and a round number. The serial numbers are assigned by the X-Plane plugin before transmission and should be unique, while the round number is shared amongst all the messages belonging to the same round of computation. Each round of computation is the same as the time-triggered Ethernet's communication cycle and VxWorks 653's major frame time.

The serial number trigger checks that the received messages' serial numbers are different. This check is performed on each FCC and it only compare the message serial numbers for that round. The check is not across all messages received by all FCCs for all rounds.

The round number trigger checks that the received messages' round number are the same in each round of computation.

While the functional components of the specification, triggers, were the same, each FCC received a tailored module. This was due to the internal architecture and organization of the partitions.

The single partition FCC was the first to receive a runtime monitor since it was the simplest to integrate. This specification was inside a single file. A new process was created inside the functional partition that would execute the monitor's step function. This process received the same period as all other processes but it ran at a lower priority to ensure that other processes, which generated values of interest, had completed their task.

The two partition FCC received the same monitoring specification as the single partition FCC. And although the monitoring partition had a single process, this process had to access the shared memory, read its contents, and save it locally for the monitor's access in addition to executing the step function.

The same pattern was repeated with the three and six partition FCCs. The single partition FCC's monitor specification was divided into three specifications. The first contained the fragments necessary for monitoring sensor values, the second for monitoring control system's computed values, and the third for monitoring the actuator commands. These three specifications were reified and were incorporated into the three and six partition FCCs.

For the three partition FCC each partition hosted the monitor that was relevant to its function. Each partition had a new process, with the same period but lower priority than other processes, created to execute the monitor's step function.

For the six partition FCC each monitoring partition received the monitor that was relevant to the functions it was monitoring. Similar to the two partition design the monitoring partition had a single process and this process had to access the shared memory, read its contents, and save it locally for the monitor's access in addition to executing the step function.

3.6 Logger

The logger is program written for a Linux based computer. This program's task is to record all the time-triggered messages transmitted by other end systems on the network. It can do so because in the TTE network description this client was configured as a recipient in every virtual link. The software uses principals similar to the X-Plane plugin since it is based on the receiving half of the X-Plane plugin. It uses the TTE-API to access the time-triggered Ethernet network to receive messages.

Since the network schedule is known a priori, this program records received messages in a round based format. First it expect messages from the X-Plane plugin, then it expects replies from the FCCs. Using this approach we can quickly verify the transmission/reception of messages and save them to a text file for later processing. This logged data represents the results of our experiments.

This program acts as our measurement instrument and we use the logs to assess the performance and behavior of the FCCs. Since the logs contain messages transmitted by all nodes we can look at the sensor values, and compute actuator command using an offline controller, and compare those to received values to see whether the FCCs produced correct results. We can also compare the commands transmitted by different FCCs to identify any faulty FCCs.

CHAPTER 4: RESULTS

The Linux host responsible for generating and transmitting sensor data to the FCCs has been programmed. The source of sensor data is X-Plane flight simulator. This involved creating an addon for X-Plane that uses time-triggered Ethernet to transmit flight data. The addon also has the capability to transmit incorrect data to simulate sensor faults. The faults were predefined out of bound values.

All four FCC configurations have been built, they are the single partition (all subtasks and monitor in the same partition), a double partition (All subtask in the same partition with the monitor in a different partition), a three partition (each subtask and its monitor is in its own partition), and a six partition (each subtask and each monitor is in its own partition) systems. In these systems the control logic remains the same but there have been modification to accommodate the partitioned nature of some FCC systems.

The time-triggered network has been configured. The network has to be configured before operation and it cannot be reconfigured during operations. This configuration involves creating communication links between the end systems where each link has a set message size and a period.

The Linux based logging system has been programmed. This host's task is to receive time-triggered messages and save them to a text file for later processing. This logged data represents the results of our experiments.

We conducted a set of experiments. The goal of the experiment was to send the same sensor data to all four FCCs and receive the same effector commands. The format of the experiments involved flying a plane in X-Plane and performing computations on FCCs. The

computations were done in rounds. X-Plane would initiate each round based on the communication schedule and the FCCs would respond to the received message. Before the beginning of the next round the Logger would record all time-triggered traffic on the network.

After conducting the experiment we observed that on the FCCs CoPilot monitor behaved as it should have. It caught all intentional faulty message transmitted from the X-Plane host. On the logger we observed that certain actuator command messages transmitted from three and six partition FCCs were consistently delayed by a single period. On examination it was discovered that this delay is caused by the additional partitions pushing the FCCs message write window past the transmit window of the TTE network.

Comparing the four approaches the single partition design is the easiest to strap on an existing system since there is less work involved, it will also have an easier recertifiablity process since the changes made were to the partition and not the module. All other approaches (two, three, and six partition designs) are more difficult to implement and recertify since the changes alter the module as a whole and that impacts every other partition and application hosted on the system.

From a timing perspective the single and two partition designs were on schedule. There were no deadline misses. But the three and six partition designs were hampered by their additional partitions and consistently missed transmission deadlines. The additional partitions pushed the effector partition's message transmit window after the network's transmit window which caused all messages to be late by a single period.

The single partition approach might appear to be the best approach but it suffered a problem not encountered by other designs, it ran out of space. The initial single partition design

had enough space and time allocated for the control task, and it could accommodate a simple runtime monitoring specification with a few external variables and triggers. But as triggers and external variables were added to the runtime monitoring specification the system would fail to build, an inspection revealed that the application's size was larger than the space assigned to the partition.

While the multi-partition is more robust and is ultimately is easier and cheaper to recertify, the configuration process is complex and requires initial set up for many components. This complexity increase the chances for introduction of errors.

During our implementation we observed problems that transcended the FCCs and were with CoPilot. CoPilot is limited to primitive types and can't access external abstract data types, structs, or pointers. This means the code had to be extensively instrumented, one of the goals of CoPilot is to be non-intrusive. Since structs can be used to bundle data for quick communication, as used in interpartition ports or time-triggered Ethernet message payloads. Lack of support for structs requires unpacking of data inside them into local variables accessible by CoPilot.

Availability of pointers eases access to the shared data region contents. Shared data regions are an area of memory reserved for use by assigned partitions, this allocation occurs at configuration step. Access to this area is up to the application developer. At runtime the applications in assigned partitions can access this area by acquiring the pointer address and follow the steps ordered by the application developer. A lack of pointer support in CoPilot means that when the address of a shared data region is acquired the content of that memory location must be written to a local variable before CoPilot can access it. This leads to further instrumentation of the code

Each CoPilot triggers must have a unique external function. This prevents a diagnostics or error logging function to be shared by multiple triggers. For example an external function for engine shutdown can be triggered by engine fire, engine overheat, and debris damage. This can be alleviated by having all-encompassing guard functions, but that raises the probability of error. There is positive effect which is that this prevents over actuation. Current design prevents an engine shutdown to be called multiple times in the event of debris damage, due to the added overheat from the possible fire.

CoPilot monitors have only a single function to perform monitoring, the step function. This may ease integration into existing projects but it introduces a new set of problems. The most obvious being the case of variables under observation being updated at different frequencies. This causes the step function to either sample a stale value, too fast, or miss some updates, too slow. This can be remedied by writing specifications that leverage the stream nature of CoPilot and writing complex guard functions, but it could lead to errors. Also in the event of a failure due to a fault in an external function the entire step function halts and not just the offending external function. Due to these reasons multiple step functions each dedicated to a trigger or specification would provide more options to the end user.

CoPilot's backends are limited to C99, it should provide more options for different C standards. Compilers for certified systems are costly and VxWorks 653 supports ANSI C. Since CoPilot requires access to stdint.h, a C99 header, for access to different types and that was not available to us we had to use vxWorks.h, a vxWorks 653 header, instead.

CHAPTER 5: CONCLUSIONS

Our contribution in this thesis was the evaluation and comparison of different approaches in integrating runtime monitors into a representative flight system. We encountered the limitations posed by each of the different FCC configurations. A configuration required more memory, some configurations had tasks miss deadlines which caused an operating cycle long delay, and most required great amount of reconfiguration. We also discovered shortcomings in CoPilot that makes usage inconvenient. These findings were discussed in detail in the Results section.

Considering what we have presented in regards to CoPilot, it is easier to implement CoPilot runtime monitors in the same partition as the functions being monitored than it is to implement it in a partition separate from the functions being monitored. The same partition approach is eased by the fact that the data can be shared by use of global variables or intra-partition communication methods, buffers or blackboards, which only require partition level configuration. While the multi-partition approach is hindered by the fact that sharing data between two partitions is done by inter-partition communication methods, ports or shared data, which requires module level configuration.

Taking the certification or recertification costs into account the multi-partition implementation fares better. Because while the single partition design eases initial implementation, the multi-partition design is easier to change and recertify since changes made to CoPilot monitors do not require a recertification of the monitored functions as well. If the system is designed from the ground up the multi-partition design is better since it's more modular and after each change only that module needs recertification. For example in the six partition design any changes made to the effector monitor are localized to that partition and its

recertification will be less costly that a complete system recertification. If the runtime monitors are being integrated into an existing project, the same partition design is better, since only a single partition needs to be recertified. This is assuming the single partition can accommodate memory and time requirements of the runtime monitors.

While we have demonstrated the benefits and shortcomings of the CoPilot runtime verification language we would like to further study multi-partition designs. A subject of interest is how to insert monitoring partitions between existing partitions and intercept the data in the ports to verify its correctness.

REFERENCES

[1] Aeronautical Radio Inc. (2010). *Avionics Application Software Standard Interface Part 1 – Required Services* (ARINC Specification 653P1-3).

[2] Ananda, C. M., Nair, S., & Mainak, G. H. (2013). ARINC 653 API and its application–An insight into Avionics System Case Study. *Defence Science Journal*, *63*(2), 223-229.

[3] ArduPilot WWW site. http://ardupilot.com.

[4] ArduPilot source code WWW site. https://github.com/diydrones/ardupilot.

[5] Azadmanesh, M. H., & Kieckhafer, R. M. (2000). Exploiting omissive faults in synchronous approximate agreement. *Computers, IEEE Transactions on*, *49*(10), 1031-1042.

[6] Bak, S., Johnson, T. T., Caccamo, M., & Sha, L. (2014, December). Real-time reachability for verified simplex design. In *Real-Time Systems Symposium (RTSS), 2014 IEEE* (pp. 138-148). IEEE.

[7] Goodloe, A. E., & Pike, L. (2010). *Monitoring distributed real-time systems: A survey and future directions*. National Aeronautics and Space Administration, Langley Research Center.

[8] Easwaran, A., Lee, I., Sokolsky, O., & Vestal, S. (2009, August). A compositional scheduling framework for digital avionics systems. *In Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on* (pp. 371-380). IEEE.

[9] Kuo, B. C., & Golnaraghi, F. (2003). *Automatic control systems* (8th ed.). Prentice Hall PTR.

[10] Lee, Y. H., Kim, D., Younis, M., & Zhou, J. (1998, October). Partition scheduling in APEX runtime environment for embedded avionics software. In*Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on* (pp. 103-109). IEEE.

[11] Lee, Y. H., Younis, M., & Zhou, J. (1998, March). An integrated scheduling mechanism for fault-tolerant modular avionics systems. In *Aerospace Conference, 1998 IEEE* (Vol. 4, pp. 21-29). IEEE.

[12] Mahadevan, N., Dubey, A., & Karsai, G. (2012, April). Architecting Health Management into Software Component Assemblies: Lessons Learned from the ARINC-653 Component Mode. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on* (pp. 79-86). IEEE.

[13] Moir, I., Seabridge, A., & Jukes, M. (2013). *Civil avionics systems* (2nd ed.). John Wiley & Sons.

[14] Pike, L., Goodloe, A., Morisset, R., & Niller, S. (2010, January). Copilot: a hard real-time runtime monitor. In *Runtime Verification* (pp. 345-359). Springer Berlin Heidelberg.

[15] Pike, L., Niller, S., & Wegmann, N. (2012, January). Runtime verification for ultra-critical systems. In *Runtime Verification* (pp. 310-324). Springer Berlin Heidelberg.

[16] Pike, L., Wegmann, N., Niller, S., & Goodloe, A. (2013). Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, *9*(4), 235-255.

[17] Prisaznuk, P. J. (2008, October). ARINC 653 role in integrated modular avionics (IMA). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th* (pp. 1-E). IEEE.

[18] Rivera, J. G., Danylyszyn, A. A., Weinstock, C. B., Sha, L. R., & Gagliardi, M. J. (1996). *An Architectural Description of the Simplex Architecture* (No. CMU/SEI-96-TR-006). Carnegie-Mellon University, Software Engineering Institute.

[19] Rogalski, T., Samolej, S., & Tomczyk, A. (2011, August). ARINC 653 Based Time-Critical Application for European SCARLETT Project. In *AIAA Guidance, Navigation, and Control Conference* (pp. 08-11).

[20] Rushby, J. (2009, April). A safety-case approach for certifying adaptive systems. In *AIAA Infotech@ Aerospace Conference* (pp. 2009-1992).

[21] SAE Aerospace. (2011). *Time-Triggered Ethernet* (SAE6802).

[22] Saha, B., Koshimoto, E., Quach, C., Hogge, E., Strom, T., Hill, B., & Goebel, K. (2011). Predicting Battery Life for Electric UAVs. *AIAA Infotech@ Aerospace*.

[23] Saha, B., Koshimoto, E., Quach, C. C., Hogge, E. F., Strom, T. H., Hill, B. L., & Goebel, K. (2011, March). Battery health management system for electric UAVs. In *Aerospace Conference, 2011 IEEE* (pp. 1-9). IEEE.

[24] Samolej, S. (2011). ARINC Specification 653 Based Real-Time Software Engineering. *e-Informatica*, *5*(1), 39-49.

[25] Sha, L. (2001, July/August). Using simplicity to control complexity. *IEEE Software*, vol.18, no. 4, pp. 20-28. IEEE.

[26] Spitzer, C., Ferrell, U., & Ferrell, T. (Eds.). (2014). *Digital avionics handbook* (3rd ed.). CRC Press.

[27] Tewari, A. (2007). *Atmospheric and space flight dynamics*. Birkhäuser Boston.

[28] TTTech. (2013). *TTEBuild the Node Design Tool for TTEthernet Networks* (No. D-TTE-G-01-002).

[29] TTTech. (2013). *TTELoad the TTEthernet Download Tool* (No. D-TTE-G-01-006).

[30] TTTech. (2013). *TTEPlan the TTEthernet Scheduler* (No. D-TTE-G-01-010).

[31] TTTech. (2012). *TTEthernet API Reference Manual* (No. D-TTE-G-01-007).

[32] Watkins, C. B., & Walter, R. (2007, October). Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC'07. IEEE/AIAA 26th* (pp. 2-A). IEEE.

[33] Wind River. (2010). *VxWorks 653 BSP Developer's Guide 2.3*.

[34] Wind River. (2010). *VxWorks 653 Configuration and Build Guide 2.3* (2nd ed.).

[35] Wind River. (2010). *VxWorks 653 Programmer's Guide 2.3* (2nd ed.).

[36] X-Plane WWW site. http://www.x-plane.com/desktop/home.

[37] X-Plane SDK WWW site. http://www.xsquawkbox.net/xpsdk/mediawiki.

[38] X-Plane Datarefs WWW site. http://www.xsquawkbox.net/xpsdk/docs/DataRefs.html.

[39] Zalewski, J. (1995). What every engineer needs to know about rate-monotonic scheduling: a tutorial. *Real-Time Magazine*, *1*, 95.