

Achieving Fault Tolerance via Robust Partitioning and N-Modular Redundancy

by

Brendan Anthony O'Connell
B.Sc. Computer Science
University of Ottawa, 1993

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

©2007 Brendan Anthony O'Connell. All rights reserved.

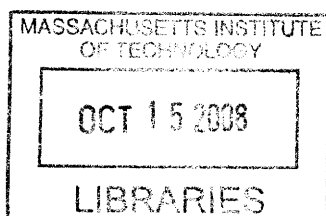
The author hereby grants to MIT permission to reproduce and to distribute publicly
paper and electronic copies of this thesis document in whole or in part
in any medium now known or hereafter created.

Signature of Author: _____
Department of Aeronautics and Astronautics
January 17, 2007

Certified by: _____
Dr. Joseph A. Kochocki
Thesis Supervisor, The Charles Stark Draper Laboratory, Inc.

Certified by: _____
Professor I. Kristina Lundqvist
Thesis Advisor, Department of Aeronautics and Astronautics

Accepted by: _____
Professor Jaime Peraire
Chair, Department Committee on Graduate Students



ARCHIVES

This page intentionally left blank

Achieving Fault Tolerance via Robust Partitioning and N-Modular Redundancy

by

Brendan Anthony O'Connell

Submitted to the Department of Aeronautics and Astronautics on
January 17, 2007 in partial fulfillment of the requirements for the
Degree of Master of Science in Aeronautics and Astronautics

ABSTRACT

This thesis describes the design and performance results for the P-NMR fault tolerant avionics system architecture being developed at Draper Laboratory. The two key principles of the architecture are robust software partitioning (P), as defined by the ARINC 653 open standard, and N-Modular Redundancy (NMR). The P-NMR architecture uses cross channel data exchange and voting to implement fault detection, isolation and recovery (FDIR). The FDIR function is implemented in software that executes on commercial-off-the-shelf (COTS) hardware components that are also based on open standards. The FDIR function and the user applications execute on the same processor. The robust partitioning is provided by a COTS real-time operating system that complies with the ARINC 653 standard.

A Triple Modular Redundant (TMR) prototype was developed and various performance metrics were collected. Evaluation of the TMR prototype indicates that the ARINC 653 standard is compatible with an NMR and FDIR architecture. Application partitions can be considered software fault containment regions which enhance the overall integrity of the system. The P-NMR performance metrics were compared with a previous Draper Laboratory design called the Fault Tolerant Parallel Processor (FTPP). This design did not make use of robust partitioning and it used proprietary hardware for implementing certain FDIR functions. The comparison demonstrated that the P-NMR system prototype could perform at an acceptable level and that the development of the system should continue. This research was done in the context of developing cost effective avionics systems for space exploration vehicles such as those being developed for NASA's Constellation program.

Technical Supervisor: Dr. Joseph A. Kochocki

Title: Principle Member of the Technical Staff, The Charles Stark Draper Laboratory

Thesis Advisor: Dr. I. Kristina Lundqvist

Title: Assistant Professor of Aeronautics and Astronautics, MIT

This page intentionally left blank


Acknowledgements

I would like to thank everyone at Draper Lab who helped me during my stay here. I would especially like to thank Joe Kochocki, Roger Racine, Piero Miotto and Rob Hammett for their time and support. I would also like to thank Professor Kristina Lundqvist at MIT for her sound advice.

Also, a big thanks goes out to Gareth and Ann for supplying all the laughs over the phone. Finally, I would like to thank my father for always providing words of encouragement.

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under contract NNM05AB50C, "CLV Task 6 Flight Software Engineering", sponsored by Jacobs Sverdrup Inc. and NASA Marshall Space Flight Center. Internal Draper support was also provided by project 21181, "GC Draper Fellows".

Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.


Brendan O'Connell
January 17, 2007

This page intentionally left blank

Table of Contents

1	Introduction	13
1.1	Cost Effective Avionics for Space Exploration	13
1.2	Research Objectives	17
1.3	Thesis Outline	20
2	Background	23
2.1	NASA Human-Rating Requirements	23
2.2	Safety Critical Software Development Standards	25
2.2.1	NASA Software Safety Standard	25
2.2.2	RTCA/DO-178B	26
2.3	Fault Tolerant Computing Concepts	27
2.3.1	Computer Channels	28
2.3.2	CCDL	29
2.3.3	Byzantine Faults	30
2.3.4	Synchronous vs. Asynchronous Channels	31
2.3.5	Channelized vs. Global Data Bus	33
2.4	Common Fault Tolerant Architectures	34
2.4.1	Dual and Triple Self Checking Pairs	34
2.4.2	N-Modular Redundancy with FDIR	36
2.5	Integrated Modular Avionics and Robust Partitioning	40
2.5.1	ARINC Background	41
2.5.2	ARINC 651 - Integrated Modular Avionics	41
2.5.3	ARINC 653 - Robust Partitioning	45
2.5.4	Example ARINC 653 Systems	48
2.6	Related Research	51
3	P-NMR System Architecture	55
3.1	Overview	55
3.2	A Frame Synchronous Approach	58
3.3	Simulated Channelized Data Bus Architecture	60
3.3.1	Sensor Input Processing by FTSS	62
3.3.2	Application Output Processing by FTSS	66
3.4	Channel Hardware	67
3.4.1	SBC750GX Single Board Computer	69
3.4.2	GbE CCDL	71
4	P-NMR Software Design	73
4.1	Channel Software Overview	73
4.1.1	Application Partitions and the APEX API	75
4.1.2	vThreads: The Partition OS	76
4.1.3	The core OS, FTSS and the BSP	77
4.1.4	XML System Configuration Files	78
4.2	FTSS Overview	79

4.2.1	Design Goals.....	80
4.2.2	Run-Time Implementation.....	82
4.2.3	Scheduling core OS FTSS Tasks.....	83
4.2.4	Rationale for placing FTSS in the core OS.....	84
4.3	GbE CCDL Device Driver.....	85
4.3.1	Core OS Driver Model.....	85
4.4	Frame Synchronization.....	86
4.4.1	The ARINC 653 Partition Schedule.....	87
4.4.2	PowerPC DEC Register and Core OS Kernel Ticks.....	89
4.4.3	PowerPC Time Base Register.....	90
4.4.4	Steady State Frame Synchronization Algorithm.....	91
4.4.5	Scheduling the FTSS SYNC Partition.....	95
4.4.6	The FTSS sync task.....	96
4.4.7	Frame Synchronization After System Cold Start.....	100
4.4.8	Impact of Frame Synchronization on Robust Time Partitioning.....	102
4.4.9	Summary.....	105
4.5	Scheduling FTSS Input/Output Processing.....	106
4.5.1	Scheduling Data Bus I/O.....	106
4.5.2	ARINC 653 Partition and Process Scheduling Overview.....	107
4.5.3	Implicit FTSS I/O Scheduling.....	109
4.5.4	Explicit FTSS I/O Scheduling.....	113
4.5.5	Channel Specific Schedules.....	123
4.5.6	Summary.....	124
4.6	Fault Tolerant ARINC 653 Ports.....	125
4.6.1	Communication Channel and Port Overview.....	125
4.6.2	Sampling and Queuing Ports.....	127
4.6.3	Pseudo Partitions and Ports.....	129
4.6.4	The P-NMR Port Design.....	130
4.6.5	Summary.....	141
4.7	FTSS Voter.....	142
5	P-NMR System Performance Results.....	143
5.1	Overview.....	143
5.2	Data Collection Methods.....	143
5.3	CCDL Gigabit Ethernet Performance.....	144
5.4	Time to Process One Kernel Tick.....	145
5.5	Frame Synchronization Performance.....	146
5.5.1	Observed Relative Clock Drift.....	146
5.5.2	Accuracy of 4000 DEC Interrupts Per Second.....	147
5.5.3	Frame Synchronization Performance.....	147
5.6	FTSS Input Exchange Performance.....	149
5.7	APEX Port System Call Performance.....	150
5.8	FTSS Output Exchange Performance.....	152
5.9	Comparison with X-38 FTTP.....	152
6	Conclusions.....	155

6.1	Use of ARINC 653 with an NMR Architecture.....	155
6.2	Use of COTS.....	157
6.2.1	COTS ARINC 653 RTOS and DO-178B Certification.....	157
6.2.2	COTS ARINC 653 RTOS Performance	158
6.2.3	COTS Hardware for Fault Tolerance.....	159
6.3	Future Work.....	160
6.3.1	FTSS Improvements	160
6.3.2	Software Fault Containment Regions	161
6.3.3	Scheduling.....	162
6.3.4	Configuration Files and Static Analysis Tools	163
References.....		165
Appendix A Acronyms		171
Appendix B XML System Configuration Files.....		173
B.1	Partition Schedule	173
B.2	Ports	174
B.3	Communication Channels.....	176
B.4	Port Groups	178
Appendix C GbE CCDL Driver Details		179
C.1	IEEE 802.3 MAC Frame Structure.....	179
C.1.1	Ethernet Frame Check Sequence	179
C.2	DMA, SDRAM and the PowerPC Cache	180
C.3	Polled Mode vs. Interrupt Mode	182
C.3.1	Sending Data.....	182
C.3.2	Receiving Data.....	183
C.3.3	Protection Against a Babbling Node.....	183
C.4	Static CCDL Configuration Table	184
C.5	Alternate Design: Partition Level CCDL Device Driver.....	184
Appendix D FAA Regulations and Guidelines.....		187
D.1	Airworthiness Regulations.....	187
D.2	IMA Guidelines	190

List of Figures

Figure 1.2-1 P-NMR Architectural Themes	20
Figure 2.4-1 X-38 FPHP (Courtesy Draper Laboratory)	38
Figure 2.5-1 Federated Avionics to Integrated Modular Avionics (IMA).....	43
Figure 3.1-1 P-NMR System Overview.....	57
Figure 3.1-2 P-NMR System Prototype in Lab	57
Figure 3.2-1 P-NMR Frame Synchronous Approach	59
Figure 3.3-1 P-NMR Simulated Channelized Data Bus Architecture	61
Figure 3.3-2 Sensor Input Processing by FTSS	63
Figure 3.3-3 GN&C FDIR partition added to schedule	65
Figure 3.3-4 Application Output Processing by FTSS	67
Figure 3.4-1 SBC750GX Board Components	70
Figure 4.1-1 Channel Software Overview	74
Figure 4.2-1 Application Output Detour to FTSS	81
Figure 4.4-1 ARINC 653 Partition Schedule.....	88
Figure 4.4-2 Steady State Frame Sync Algorithm	93
Figure 4.4-3 Clock Adjustment Algorithm.....	95
Figure 4.4-4 Major Frame Synchronization.....	98
Figure 4.4-5 Clock Synch Algorithm using DEC Register.....	101
Figure 4.5-1 I/O Processing added to partition schedule	107
Figure 4.5-2 Implicit Scheduling of FTSS I/O in Partition Schedule.....	112
Figure 4.5-3 Explicit Scheduling of FTSS I/O in Partition Schedule.....	116
Figure 4.5-4 Explicit Scheduling of FTSS with APEX Ports.....	119
Figure 4.5-5 First 20 ms minor frame	120
Figure 4.5-6 Timing within FTSS Input Task	122
Figure 4.5-7 Channel Specific Schedules	124
Figure 4.6-1 Different Partitions using FTSS services	135
Figure 4.6-2 Port Groups Assigned to Schedule Windows	138
Figure 4.6-3 CH1 GNC_01Hz_Input Voting	140

List of Tables

Table 2.2-1 DO-178B Software Levels and Objectives	26
Table 2.5-1 ARINC 653 Avionics Systems	51
Table 4.5-1 Process A1 System Call Durations	111
Table 4.5-2 GN&C Apex Processes	114
Table 4.6-1 GN&C Partition Input Ports	131
Table 4.6-2 FTSS_INPUT Pseudo Source Ports going to GN&C.....	132
Table 4.6-3 Sensor Input Channels Between FTSS_INPUT and GN&C.....	133
Table 4.6-4 GN&C Partition Output Ports	133
Table 4.6-5 FTSS_OUTPUT Destination Ports.....	134
Table 4.6-6 Command Output Channels Between GN&C and FTSS_OUTPUT	134
Table 4.6-7 FTSS_INPUT Sensor Port Groups For GN&C Partition	137
Table 4.6-8 Example of Asymmetric Inputs.....	141
Table 5.3-1 CCDL Gigabit Ethernet Latency (t _{del} for 40 bytes)	145
Table 5.5-1 Relative Clock Drift Between Channels.....	147
Table 5.5-2 CH1 Frame Synchronization Events	148
Table 5.6-1 FTSS Input Performance	150
Table 5.7-1 APEX Port System Call Performance	151
Table 5.8-1 FTSS Output Performance	152

This page intentionally left blank

Chapter 1

Introduction

1.1 Cost Effective Avionics for Space Exploration

NASA and its industrial partners have started to implement the Constellation program, an ambitious plan to extend human presence across the solar system, starting with a human return to the Moon by the year 2020, in preparation for human exploration of Mars and other destinations [1]. Constellation is a family of vehicle elements that must operate in concert with each other in order to achieve mission goals. Some of the elements will be mated together to form larger elements. Thus, Constellation follows a system-of-systems (SoS) architecture. In order to reduce costs, many of the vehicles will share common subsystems.

One area in which the Constellation vehicles will differ substantially from their Apollo era counterparts is the avionics subsystem [2]. This is due to the digital technology revolution that has occurred since the 1960's. The throughput of today's processors and the capacity of memory far exceeds what was available when Apollo was designed. This allows Constellation avionics designers to implement powerful new functionality in software such as autonomous operations and integrated vehicle health management (IVHM). This will enable some of the Constellation vehicles to operate with or without a human controller [2].

In addition to having increased functionality, the avionics systems will also have to be safe and reliable. This is especially true for the flight control function on a crewed vehicle where a fault could result in the loss of the vehicle and the crew. NASA requires that any system used on a human-rated vehicle be two-fault tolerant [3]. That is to say, the system must continue to operate correctly even after experiencing two faults.

A common avionics architecture that can provide robust fault tolerance is N-Modular Redundancy (NMR) with fault detection, isolation and recovery (FDIR). In this kind of system, identical software executes on N identical redundant computers (called channels). The computers communicate with each other over a cross channel data

link (CCDL). Sensor inputs are distributed to each computer and actuator command outputs from each computer are collected and voted to mask out any potential faults in one or more of the computers. Since voting is used, there needs to be at least three computers ($N \geq 3$). A three computer system ($N = 3$) is called a triple modular redundant (TMR) system.

A significant advantage of an NMR system with cross channel data exchange and voting is that it allows for the detection of a faulty computer. This computer can then be removed from the operational group of computers, reset and then readmitted back into the group. However, the exchange of data between channels also means that this kind of system is susceptible to so called *Byzantine faults* [4]. A Byzantine fault is an exceptionally difficult kind of fault to tolerate because it is not symmetrical. Different parts of the redundant computer system will see *different* results that are due to the same single fault. Thus, a Byzantine fault is classified as being *asymmetrical*. This asymmetrical nature can lead to disagreement among the redundant computers, especially during voting activities. However, a properly designed NMR system can correctly detect and isolate components that are generating Byzantine faults.

Since NMR with FDIR systems have been used extensively in previous flight control applications for both aircraft and spacecraft [5], it is a leading candidate architecture for ensuring avionics reliability on both manned and unmanned Constellation vehicles.

Another aspect of the avionics system that is related to reliability is dynamic reconfiguration. Reconfiguration could occur after a permanent fault in one of the computers. The software that was executing on the failed computer may need to be migrated to a healthy computer. This is especially applicable to long duration missions lasting several years where permanent equipment failures are possible. Reconfiguration of the avionics may also occur due to a reconfiguration of the vehicle itself. An example of this is when two orbiting vehicles dock together. It may be advantageous to have the computers on both vehicles form a single fault tolerant group.

The combination of functionality, reliability and reconfiguration requirements can result in an avionics system that is complex and difficult to verify and validate. A significant portion of the verification effort could be spent integrating hardware and

software from many different internal and external suppliers. These individual components may not use common standards resulting in a difficult system integration and test phase.

One way in which NASA could significantly reduce the complexity and cost of avionics system development is to minimize the use of custom or proprietary hardware and software components. Instead, NASA should maximize the use of proven commercial off-the-shelf (COTS) components that are based on open standards and that have been widely adopted within one or more industries. An open standards approach promotes reusability, extensibility and maintainability of the avionics system and it tends to reduce design and development costs [6]. It also minimizes the chance of obsolescence issues associated with proprietary products from a single vendor. This approach of using open standards and COTS has already been adopted by the commercial and military avionics industry [7, 8, 9].

One commercial avionics standard that may be applicable to Constellation projects is ARINC 651 which provides design guidance for integrated modular avionics (IMA) [10]. Over the last decade, aircraft manufacturers have moved away from traditional distributed federated avionics systems towards IMA [11, 12]. In a federated system, each individual avionics function, such as collision avoidance, is implemented as a separate hardware box on the aircraft. Each box has its own chassis, backplane, power supply, processor, memory and I/O hardware. All the individual avionics boxes are then connected together with point-to-point data buses. However, in an IMA system, multiple functions are hosted within a single, standardized IMA cabinet.

Inside the IMA cabinet are standardized “modules”. Examples include power supply modules, processing modules and I/O modules. The modules may communicate with each other over a backplane within the cabinet. In most cases, multiple avionics functions will share the same processor and memory on one module. In effect, individual avionics functions *share common resources*. One function can communicate with another using software instead of physical wiring. The advantages of an IMA architecture include a reduction in avionics size, weight, power consumption and wiring complexity [10]. Size and weight savings are critical for any Constellation vehicle being launched into space.

In order for an IMA architecture to be viable in a safety critical environment such as transport aircraft avionics, a mechanism must be provided that greatly reduces the risk of one function interfering with another function within the same IMA platform. This is especially true when the two functions have different criticality levels. For example, one function could be responsible for automatically landing the aircraft in bad weather whereas the other function could be responsible for maintaining the passenger cabin temperature. Obviously, the cabin temperature function must not interfere in any way with the autoland function.

This isolation of different functions is provided by a concept called *robust partitioning*. Robust partitioning is implemented by the real time operating system (RTOS) that is running on each processor within the IMA cabinet. Each avionics function is called an *application* and each application is placed inside a *partition*. A partition can be thought of as a protective container for the application. The RTOS provides three main types of partition protection: time, space and I/O. These are explained more in section 2.5.3. Rather than have each avionics supplier develop its own version of an RTOS that supports robust partitioning, the commercial avionics industry has moved towards a *single open standard for robust partitioning*. The ARINC 653 standard defines the robust partitioning requirements for an RTOS kernel as well as the application executive (APEX) application programming interface (API) [13]. The APEX API defines a standard set of robust partitioning services that an application programmer can use. There are currently at least five COTS RTOS products that support the standard.

A significant advantage of using the ARINC 653 standard and the APEX API instead of a custom solution is that avionics applications are *portable and reusable*. They are not tied to a single RTOS product from a single vendor. An application that follows the APEX API can be moved from one COTS ARINC 653 RTOS to another without having to edit and retest the application code. All that is required is a recompile. This advantage is significant for a large and long duration program like Constellation where several RTOS vendors may be used over the lifetime of the program.

Another important standard that is closely related to ARINC 653 compliant COTS products is RTCA/DO-178B, “Software Considerations in Airborne Systems and Equipment Certification” [14]. This standard provides guidelines for the production of

software for aircraft avionics systems such that there is a reasonable expectation that the software will perform its intended function and will be free of serious defects. The guidelines illustrate how development processes can be setup for each phase of the software development lifecycle (e.g. requirements, design, code, verification) and it specifies objectives that should be met for each process. It also specifies what kinds of evidence can be used to prove that the objectives were satisfied during the development process. This evidence can then be used when the certification process begins for the aircraft.

Given the expense of developing an ARINC 653 RTOS using DO-178B guidelines, RTOS vendors have started to create a *certification package*. This package is an optional product that can be purchased in addition to the RTOS itself. The package contains all the required evidence and documentation that shows how each DO-178B process objective was met. Thus, the cost of using DO-178B software development processes can be spread among multiple RTOS customers. An avionics system supplier can simply purchase the COTS RTOS product and the DO-178B certification package off-the-shelf. This allows the avionics supplier to focus on their core business instead of spending time and resources collecting DO-178B evidence required for using the RTOS in the final certified system.

If NASA decided to accept DO-178B certification packages from commercial RTOS vendors then a significant cost saving could be had on Constellation avionics projects. However, if there is an intent to use the certification package then the COTS product should not be altered or customized by the avionics system developer.

The challenge for NASA and its industrial partners is applying open standards, COTS and commercial avionics technology in a way that still provides an avionics system that is safe, reliable and cost effective.

1.2 Research Objectives

The following issues are associated with the described approach to cost effective avionics for space exploration.

First, there are few examples in the literature of an ARINC 653 compliant RTOS with the APEX API being used for NMR flight control computers (FCCs). Most previous

applications of the ARINC 653 standard have been for IMA based mission management computers (MMCs) that are not responsible for flight control and that do not use an NMR with FDIR architecture. Instead, only two MMCs are used; one is the primary and the other is a backup. Thus, a typical configuration on an aircraft may be a pair of IMA based MMCs that use ARINC 653 and another group of voting NMR FCCs ($N \geq 3$) that perform the flight control function and that do not use ARINC 653. Given that Constellation vehicles have severe size and weight constraints, they cannot afford two different sets of computers. The FCCs and MMCs must be integrated into a single group of vehicle management computers (VMCs) that host both the flight control and mission management functions.

Second, it has not been established that a COTS ARINC 653 RTOS can be used in an NMR with FDIR architecture without substantial modifications. It is also not clear if a COTS RTOS has sufficient performance for such an application. The impact of the RTOS on processor throughput and latency needs to be evaluated. Any modification of the COTS kernel to change behavior or to improve performance could render significant portions of the COTS DO-178B certification package invalid.

Third, most previous fault tolerant NMR avionics systems used for flight control have used some proprietary hardware to implement portions of the FDIR functionality. Few NMR systems implement the fault tolerance and redundancy management functions in software executing on COTS hardware. An often stated rationale for this approach is concerns over the performance of COTS hardware.

Given the above issues, this research attempts to answer the following questions:

1. Is the ARINC 653 standard for robust partitioning, specifically the APEX API, compatible with an NMR/FDIR architecture? The focus is on two main areas:
 - a. How can the ARINC 653 two-level scheduler be used to schedule fault tolerance processing such as sensor input distribution and output voting?
 - b. How can APEX communication ports be adapted to fault tolerant I/O processing?

2. Can a COTS ARINC 653 RTOS be used in a fault tolerant NMR voting architecture without substantial modifications?
 - a. How can the fault tolerance functions be integrated with the COTS RTOS product?
 - b. What is the performance of the COTS RTOS product? This includes evaluating the ARINC 653 scheduler and the APEX port I/O facilities.
3. Can widely adopted COTS computer components be used to implement a fault tolerant NMR voting architecture? Is custom proprietary hardware necessary? Can the fault tolerance functions be implemented in software executing on a COTS processor? What is the performance of the system?

In order to answer these questions, a system prototype was developed. The system design uses a *partitioned, NMR* approach, and is thus referred to as a ***P-NMR architecture***. It uses COTS hardware and software and open standards such as ARINC 653 and IEEE Standard 802.3 (Ethernet). Various performance metrics were collected and compared to a previous NMR system built by Draper Laboratory, the X-38 Fault Tolerant Parallel Processor (FTPP). This system was developed for NASA in the late 1990's and it does not use ARINC 653 robust partitioning and it does use custom hardware to implement certain FDIR functions.

In summary, this research will use the P-NMR prototype to investigate the concurrent use of three main architectural themes to develop a cost effective, fault tolerant avionics architecture for space vehicles such as those being developed for the Constellation program. These three main architectural themes, as shown in Figure 1.2-1, are N-modular redundancy with FDIR, ARINC 653 robust partitioning and COTS hardware and software based on open standards.

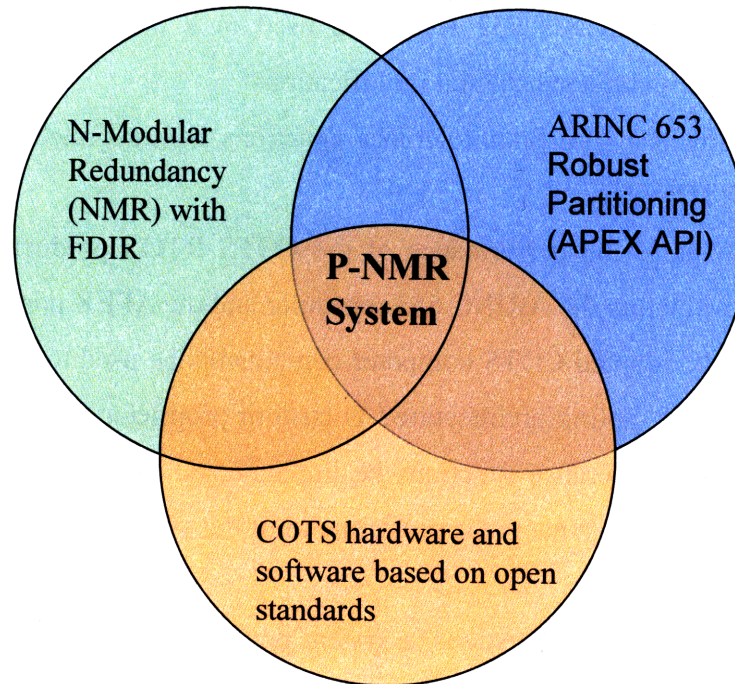


Figure 1.2-1 P-NMR Architectural Themes

1.3 Thesis Outline

Chapter 2 provides background and related research on reliability requirements, fault tolerant avionics and robust partitioning. Chapter 3 describes the P-NMR system architecture at a high level and includes a description of the COTS hardware components being used for each redundant computer in the TMR prototype. Chapter 4 is a detailed description of the software executing on each redundant computer. It discusses in depth how the fault tolerance functions have been implemented in software and how they have been integrated with the COTS ARINC 653 RTOS. Chapter 5 presents the performance results for the P-NMR prototype and compares these results with the Draper Laboratory X-38 FPHP system. Chapter 6 presents our conclusions including whether or not the ARINC 653 standard and the APEX API are compatible with an NMR and FDIR architecture. Chapter 6 also suggests areas for future research using the P-NMR prototype. Appendix A is a list of acronyms, Appendix B provides some examples of the P-NMR Extensible Markup Language (XML) system configuration files, Appendix C provides details on the implementation of the CCDL using COTS Ethernet hardware and

Appendix D contains a sample of reliability requirements and guidelines from the commercial aircraft industry including guidelines for IMA systems.

This page intentionally left blank

Chapter 2

Background

This chapter provides some background and related research on reliability requirements, fault tolerant avionics and robust partitioning. Section 2.1 describes certain NASA human-rating requirements that are relevant to any avionics system architecture being proposed for crewed vehicles. Section 2.2 is a brief description of two safety critical software development standards, one used by NASA and the other used by the aircraft industry. Section 2.3 discusses some fault tolerant computing concepts often used when describing avionics systems. Section 2.4 discusses two common fault tolerant avionics architectures: self checking pairs and N-modular redundancy. Section 2.5 talks about IMA, robust partitioning via ARINC 653 and some existing systems that currently use an ARINC 653 compliant RTOS. Finally, in section 2.6, there is a literature review of research related to robust partitioning and fault tolerance.

2.1 NASA Human-Rating Requirements

This section discusses certain NASA reliability requirements that will affect the design of any avionics system being proposed for use on a crewed NASA space vehicle. For comparison purposes, similar requirements for transport aircraft are described in Appendix D.

Compared to aircraft, spacecraft and their avionics have to operate in a harsh environment. There is severe vibration during launch. While in space, the avionics may be exposed to vacuum conditions and there may be large temperature extremes. Avionics components may be exposed to solar and cosmic radiation. This radiation can lead to single event effects (SEE) such as single event latchups (SEL) and single event upsets (SEU) that may induce a computer reset. Thus, avionics reliability for human-rated vehicles must be carefully considered.

The NASA procedural requirements for the human-rating of space systems is contained in [3]. *Reliability* is defined as the probability that a system of hardware,

software, and human elements will function as intended over a specified period of time under specified environmental conditions. A *fault* is a defect, imperfection, mistake, or flaw of varying severity that occurs within some hardware or software component or system. Fault is a general term and can range from a minor defect to a failure. The term *failure* is not defined. *Fail-safe* is defined as the ability to sustain a failure and retain the capability to safely terminate or control the operation.

The human-rating standard requires that all space systems that will support humans be designed such that no two non-simultaneous failures result in crew or passenger fatality or permanent disability. In other words, systems must be “fail-op, fail-op”. In systems with relatively short periods of operation, or where dynamic flight modes (such as powered ascent) are involved, NASA suggests that *installed redundancy* be the principal means of ensuring the system's reliability. The system must also provide an FDIR mechanism for faults that affect critical functions.

The standard also has specific requirements for flight control systems that will be used on a human rated vehicle. Specifically, the system design shall prevent or mitigate the effects of *common cause failures* in time-critical software (e.g., flight control software during dynamic phases of flight such as ascent). An example of a common cause failure is where the same microprocessor is used in redundant flight control computers. If the microprocessor has a design flaw (e.g. in the floating point unit), it may affect all redundant computers at the exact same time since the computers may all be executing the same software. Such a flaw may be difficult to detect since all computers will produce the same answer. Common cause failures can be avoided by using dissimilar hardware and/or software. Some of the methods suggested by NASA for meeting the intent of this requirement include:

1. Redundant independent software running on a redundant identical flight computer.
2. Use of an alternate guidance platform, computer and software (e.g., using the space craft guidance to control a booster).
3. Use of nearly identical source code uniquely compiled for different dissimilar processors.

2.2 Safety Critical Software Development Standards

2.2.1 NASA Software Safety Standard

The NASA standard for the development of software for space vehicles and their associated ground equipment is contained in [15]. There are only two main classes of software: *safety critical* and *mission critical*. Software is safety-critical if it meets at least one of the following criteria:

1. Resides in a safety-critical system (as determined by a hazard analysis) AND at least one of the following:
 - a. Causes or contributes to a hazard.
 - b. Provides control or mitigation for hazards.
 - c. Controls safety-critical functions.
 - d. Processes safety-critical commands or data.
 - e. Detects and reports, or takes corrective action, if the system reaches a specific hazardous state.
 - f. Mitigates damage if a hazard occurs.
 - g. Resides on the same system (processor) as safety-critical software.
2. Processes data or analyzes trends that lead directly to safety decisions.
3. Provides full or partial verification or validation of safety-critical systems, including hardware or software subsystems.

Mission critical software is defined as software which must retain its operational capability to assure mission success. An example would be software associated with the collection of science data on the surface of a planet.

Of note is that any software that resides on the same processor as safety-critical software is also deemed safety critical. However, the standard goes on to indicate in a note that methods to separate the code, such as partitioning, can be used to limit the software defined as safety-critical. If such methods are used, then the isolation method is safety-critical, but the isolated non-critical code is not.

Regarding COTS software, the NASA software standard indicates that including software in a safety-critical system when the software was not developed specifically for that system “can be risky”. The requirements go on to state that COTS used in safety-

critical systems shall undergo safety analysis that considers its ability to meet required safety functions. This analysis must include the evaluation of functions not planned to be used during a mission but present in the COTS product.

2.2.2 RTCA/DO-178B

In terms of aircraft, the Federal Aviation Administration (FAA) has indicated that avionics developers may use RTCA/DO-178B as a means to secure FAA approval of digital computer software. It should be noted that the FAA does not mandate that avionics software developers use DO-178B. They may use other methods that satisfy the intent of DO-178B. It should also be noted that DO-178B is not a set of requirements that must be followed exactly. It is more a set of guidelines that can be customized to some degree.

DO-178B defines five failure condition categories based on their impact on continued safe flight of the aircraft. These failure categories are catastrophic, hazardous, major, minor and no effect.¹ The document also defines five software levels, A through E, based on the contribution of software to potential failure conditions. The software level determines how rigorous the software development process must be and it determines how many objective must be satisfied during the development process. Table 2.2-1 shows the failure conditions, software levels and the number of development process objectives that must be satisfied. As can be seen, Level A software has the most objectives and is thus very expensive to develop and verify.

Failure Condition	Software Level	Number of Objectives
Catastrophic	A	66
Hazardous/Severe-Major	B	65
Major	C	57
Minor	D	28
No Effect	E	0

Table 2.2-1 DO-178B Software Levels and Objectives

¹ For more details on these categories, see Appendix D.

In terms of robust partitioning, DO-178B states that the technique can be used to provide isolation between functionally independent software components to contain and/or isolate faults and potentially reduce the effort of the software verification process. The standard goes on to state that if protection by partitioning is provided, the software level for each partitioned component may be determined using the most severe failure condition category associated with that component. Also, if some of the partitioning protection is provided by software as opposed to hardware, then that software must be assigned the software level corresponding to the highest level of any of the application partitions using the protection mechanisms.

COTS software must be assigned the appropriate level and the vendor must use appropriate software development processes that satisfy the intent of the objectives in DO-178B. The vendor must produce all evidence that is required to prove that appropriate processes were followed during the software development lifecycle.

2.3 Fault Tolerant Computing Concepts

This section discusses some common concepts used in the context of safety critical, fault tolerant computer architectures.

The primary method used for achieving fault tolerance in safety critical applications is redundant software executing on redundant hardware. The main reason for using this kind of redundancy is that it provides fail-operational capability. After experiencing a single failure in one hardware component, the system can continue to operate *without interruption*. This type of redundancy uses “hot” redundant hardware components meaning that they are always powered on and operational. One computer may be the primary but all the standby computers are processing all the same inputs as the primary. A “cold spare” approach uses redundant computers and components that can be powered off – they are only powered on after a failure occurs. A hot redundancy approach is essential for time critical functions such as flight control.

Another main advantage of redundancy is that it aids in fault detection. For example, the outputs from redundant processors executing identical software using identical sensor inputs can be compared. A miscompare of the outputs indicates that one of the processors has experienced a fault.

Using redundancy to identify faulty hardware is especially important if the built-in test (BIT) functionality native to the hardware components (e.g. circuit boards) cannot guarantee that 100% of all possible hardware faults will be detected. BIT functionality will execute at power up and it will execute continuously (CBIT) during normal operation. An example of hardware BIT is memory integrity tests. In general, BIT functions cannot detect 100% of all possible hardware faults in a current generation single board computer [5].

The main disadvantages of active redundancy is an increase in avionics cost, size, weight, power consumption and complexity. Although redundancy will increase operational system reliability, there is an increase in the arrival rate of faults simply because there are more components on the vehicle that can fail. This can lead to increased maintenance costs, increased spares inventory and a decreased dispatch rate. Depending on the avionics architecture, increasing the number of redundant computers can also decrease overall system throughput and bandwidth. Thus, there is a trade that needs to be made between increasing reliability using more redundancy versus all of the mentioned drawbacks.

2.3.1 Computer Channels

A *hardware channel* is a fault containment region (FCR). An FCR is a set of hardware elements that provides physical separation, independent power, electrical isolation and independent clocking. A fault that occurs within one channel cannot escape to (or pollute) another healthy channel. This includes incorrect data as well as physical threats such as an electrical power surge.

A channel is usually a single computer chassis since it contains some simplex elements that all the modules within the chassis share such as a single power supply or a single backplane bus.¹ Thus, a single event such as a power surge could damage all the modules within one chassis. This is why the chassis is considered the channel and not the individual boards within the chassis. All safety critical, fault tolerant avionics designs

¹ A module can be a printed circuit board.

will contain at least two physically separated channels that use independent power supplies.

Note that the components within each redundant channel do not necessarily have to be homogeneous. For example, a different type of processor could be used in each channel. This would avoid common mode failures due to a design flaw in a particular type of processor. However, heterogeneous components in each channel can make timing analysis more difficult as the processors will take different amounts of time to execute the same source code. We also note that there may be redundancy *within* a channel. For example, a single printed circuit board within a single channel may contain two lockstep processors whose outputs are compared. Combining these two ideas, an architecture could consist of N identical channels with each channel containing two different types of processors.

Related to this discussion is the notion of a single dissimilar backup channel. As an example, a primary flight control system may use four identical channels. It is possible that a common mode fault will cause all four primary channels to fail at the same time. Thus, a fifth channel that uses different hardware and software may be desirable. The hardware and software design and development for the backup system may be performed by a different organization using different personnel to reduce the use of common design assumptions. The introduction of a backup system that uses dissimilar hardware and dissimilar software developed by a separate team of engineers can significantly increase the cost and complexity of the avionics design. The backup systems ability to increase reliability is also controversial [16].

The P-NMR system prototype uses three channels, each with a single processor of the same type, as discussed in section 3.1.

2.3.2 CCDL

An important aspect of many fault tolerant computing designs is the notion of communication between channels. One reason for this communication would be to allow the comparison of the outputs of each channel in order to identify faults. This communication is implemented using a *cross channel data link (CCDL)*. The physical

CCDL medium and the methods used for exchanging data between channels is critically important because the CCDL connects all redundant channels together and thus it could be the source of a common mode failure. The CCDL functionality has the ability to corrupt or damage all redundant channels simultaneously, nullifying any benefits of redundant hardware.

The P-NMR architecture uses a point-to-point CCDL which is described in section 3.4.2 and Appendix C.

2.3.3 Byzantine Faults

In the landmark paper by Lamport et al [4], the authors introduce the notion of a *Byzantine fault*. This is the most difficult type of fault to be tolerated by a group of three or more channels that use some form of cross channel voting to identify a faulty channel. The difficulty lies in the fact that different channels will observe the fault in different ways. In essence, the fault is *asymmetrical*. In order to properly tolerate “f” *simultaneous* Byzantine faults, there needs to be $(3f + 1)$ channels and $(f + 1)$ rounds of data exchange. Simultaneous means that the first fault is not corrected before the second fault occurs.

A two round data exchange (to tolerate just one Byzantine fault) starts with each channel sending all other channels a message indicating “this is my data”. This is called round one. The second round involves all channels forwarding the round one data they received from other channels. In essence, the round two message indicates “here is the data I received from other channels during round one”. As long as there are enough channels, this method of data exchange can properly identify a faulty channel.

Thus, the cost of handling Byzantine faults is high, both in terms of required hardware redundancy and in terms of the amount of communication required. For example, in order to tolerate two simultaneous faults, 7 channels and 3 rounds of data exchange are required.

The authors also show that if message authentication is used then only $(f + 2)$ channels are required. Message authentication requires some form of secure digital signature and is computationally expensive [17]. Thus, redundant systems with a CCDL often use simpler forms of error detection and correction such as a cyclic redundancy

code (CRC) [18]. Since a CRC is not sufficient for message authentication, $(3f + 1)$ channels are still required to tolerate “f” Byzantine faults.

The authors in [19] indicate that there has been a reluctance on the part of avionics designers to accept that a Byzantine fault is possible in an actual avionics system. However, they present several examples of actual occurrences in production systems and they explain how real hardware can produce Byzantine faults including slight-off-specification (SOS) timing faults. We also note that the design for the Boeing 777 fly-by-wire system takes into account Byzantine faults [20].

Finally, since Byzantine faults are especially difficult to tolerate, many fault tolerant designs do not use three or more channels that use cross channel voting schemes. Instead, a popular alternative is a series of self checking pairs, as is discussed in section 2.4.1.

The P-NMR software architecture has been designed to tolerate one Byzantine fault when four channels are used. However, the initial prototype only has three channels ($N = 3$) with no message authentication and thus some aspects of the software design were altered to accommodate this limitation.

2.3.4 Synchronous vs. Asynchronous Channels

A major decision for some fault tolerant designs is whether the system should be *synchronous or asynchronous* or some combination of the two. A synchronous system means that all redundant computers execute the same software using the same inputs at approximately the same time. If no faults occur then all channels should produce the same outputs. An asynchronous system, on the other hand, means that the individual channels do not need to be in lockstep with one another and do not have to use the exact same inputs at any given instant in time [21].

The advantage of synchronous systems is that it allows simple bit-for-bit, majority voting of the channel outputs. This is sometimes referred to as *exact agreement*. Exact agreement is possible because in the fault free case, all channels should produce identical outputs (e.g. actuator commands). The disadvantage of this approach is that it requires some form of distributed clock synchronization among the channels. In order for the

clock synchronization to be fault tolerant, a single clock source is not used. Instead, all the channels have their own local clock but they will use a distributed clock synchronization algorithm (using the CCDL) to “agree” on a global time to be used by all channels. Unfortunately, these kinds of algorithms can be susceptible to Byzantine faults that can lead to disagreement and clock divergence [22]. An example of this is slightly-off-specification timing errors in a CCDL transmitter. In addition, the special cases of system startup and the introduction of a new channel into a group of channels that are already synchronized is non-trivial.

The advantage of asynchronous systems is that no clock synchronization is required. The channels can operate autonomously. Each channel reads sensor values, executes the control laws and produces actuator commands. However, the channels may read the sensor values at slightly different times. This is especially true for high rate control loops executing at 25 Hz or higher. The disadvantage of this approach is that the command outputs from each channel may be slightly different and so simple bit-for-bit majority voting (i.e. exact agreement) cannot be used. Instead, *approximate agreement* methods must be used. This kind of agreement involves voting schemes such as median value selection (MVS) for three values or, in the case of four values, the average of the two middle values after removing the smallest and largest values. Reasonableness thresholds may also need to be defined. Finally, integrators used in the control laws on each channel will diverge over time producing significantly different outputs and so some form of channel equalization of key control law variables is required on a periodic basis [20, 21].

Finally, there is the concept of a globally asynchronous, locally synchronous (GALS) architecture. In this case, the channels are allowed to operate asynchronously. However, within each channel, there will be two or more processors that execute synchronously in lockstep. The lockstep processors are used to verify that an individual channel has not experienced a fault. The output value from the synchronous processors is then sent to some other part of the avionics system that will make a determination as to which channel’s value will be used to command the actuators.

The P-NMR architecture uses a synchronous approach and is described more in section 3.2.

2.3.5 Channelized vs. Global Data Bus

Another important design decision is the topology for the data bus which connects the redundant computer channels to other components of the avionics system. Of primary importance is the method used to connect flight control sensors and actuators to the redundant channels. The data bus design is tightly coupled to the CCDL data exchange design and the timing design (i.e. synchronous or asynchronous). Two common data bus designs used in safety critical aerospace applications are *channelized* and *global*.

A channelized data bus topology dates back to the earliest fly-by-wire systems. Using this design, there is a set of sensors and actuators that are *only* connected to *one* channel using a single data bus. A single data bus with the sensors, single channel and actuators connected to it is sometimes referred to as a *string*. Each sensor and actuator only has one bus interface unit. Thus, if there are four channels then there will be four data buses and each data bus will be connected to only one channel and one set of sensors/actuators. Each channel may or may not be connected to the other channels using a CCDL. The advantage of this design is simplicity and isolation. One string cannot interfere or damage another string. A disadvantage is that the loss of the computer channel means the entire string is lost including the sensors and actuators attached to that channel. Another issue is that input congruency on each channel may necessitate sensor data exchanges across the CCDL.

A global topology connects all sensors and actuators to all channels. If there are four channels then there will be four data buses and each data bus will be connected to all four channels and all four sets of sensors. With this design there is no need to exchange sensor inputs using the CCDL between channels. The advantage of this design is that it is fault tolerant in the sense that if one channel fails, the remaining channels still have access to all sets of sensor devices. The disadvantage is complexity. Each sensor unit (or remote data concentrator) will have four bus interface units. The design is also susceptible to certain common mode faults. For example, since one bus connects all four channels together, a single power surge along the single bus could damage all the

channels. Another example would be when a “babbling node” or a “stuck on transmit” node on a single data bus interferes with all the channels.

The data bus design is also related to the channel timing design (synchronous or asynchronous). In a synchronous design, it may be desirable to have the bus controller on each channel request sensor data at the same time. Bus designs that use a time triggered approach instead of an asynchronous one are better suited for such applications [23].

The P-NMR architecture uses a channelized design and is described more in section 3.3.

2.4 Common Fault Tolerant Architectures

A thorough review of fault tolerant avionics architectures used in aerospace applications can be found in [5]. This section discusses two of those architectures which are relevant to this research.

2.4.1 Dual and Triple Self Checking Pairs

A self checking pair (SCP) is comprised of two processors executing the same software and comparing results. This is most commonly accomplished at the printed circuit board level where a single board has two processors that execute in lockstep using the same board level system clock. Usually, each processor has its own private main memory. Comparator hardware is used to compare the outputs of the two processors. The comparator may be on the same circuit board as the two processors or it may be an external device.

In general, exact agreement is used since the processors execute in lockstep. If the outputs agree then the output is sent to the actuator. If there is a disagreement, the output value is not used and the pair of processors is considered faulty. It is important to note that if there is only one SCP then it is difficult to determine which processor experienced a fault. BIT may be used to attempt to determine the faulty processor but this is not a guaranteed solution since BIT cannot obtain 100% fault coverage. Also, the

determination of the individual faulty processor is often not required since the entire circuit board is reset.

Normally, a single SCP board is contained within a single channel (i.e. computer chassis). A single SCP can prevent any malfunction but is twice as likely to experience a loss of function due to the fact that two processors are required to produce an output [5]. The probability of loss of function can be reduced if one processor is allowed to operate in simplex mode if BIT fails for one of the processors. The drawback of a single SCP board is that the two processors are in the same fault containment region (i.e. channel) and thus they are both susceptible to a single common area fault such as a fire. In some implementations, the two processors may also use the same power supply and clock.

In order to increase reliability, dual and triple SCPs are used. In the dual case, there are two channels in the avionics system. Each channel contains one board or module that has the SCP processors. The outputs from the SCPs are compared, either within each channel or at a location external to both channels. If the first SCP compare is good then its value is used otherwise the second SCP compare is performed. This can be extended to a three channel system. A dual SCP architecture is often used in IMA systems, as discussed in section 2.5.4.

An advantage of a dual or triple SCP architecture is that it does not require a distributed clock synchronization algorithm since clocking is only performed at the board level within a single channel. In general, these systems use a globally asynchronous, locally synchronous approach where the channels are not synchronized but the pair of processors within a channel are synchronized. Since the channels are asynchronous, some designs may require channel equalization of critical variables to prevent the gradual divergence of the channel outputs.

SCP designs also do not require majority voting or mid-value selection since a simple comparison of two values is used. This means that Byzantine faults are not possible since there needs to be at least three values being compared for an asymmetric fault to occur.

2.4.2 N-Modular Redundancy with FDIR

This section discusses *distributed* N-modular redundant (NMR) systems that use physically separated channels as opposed to NMR implementations that use three or more lockstep processors on a single printed circuit board.

A distributed NMR architecture involves using N channels with $N \geq 3$. Channels are connected using a CCDL. There is a dedicated, point-to-point CCDL between each pair of channels – a bus or network topology is not used for the CCDL. In order to accomplish FDIR, the channels exchange application outputs using the CCDL and then the outputs are voted within each channel. Exact or approximate voting methods can be used depending on whether the design is synchronous or asynchronous.

A design where $N = 3$ is called a *triple modular redundant (TMR)* design. A TMR system may be designed to use an SCP configuration after the first channel failure and a simplex channel after the second failure. If this is the case, the TMR design can prevent loss of function and malfunction after two consecutive symmetric faults. However, a TMR system cannot prevent a malfunction after a single Byzantine (asymmetric) fault. A *quadruplex* system ($N = 4$) can tolerate two simultaneous symmetric faults and one Byzantine fault.

NMR systems with FDIR often use some form of *group membership protocol*. This involves all channels exchanging system state information so that the channels can come to a consensus (using voting) on the health of each channel in the system. A channel that is deemed faulty by the majority will be removed from the *operational group* of channels and will be prevented from participating in any future votes. The faulty channel may be reset and then monitored by the operational group until the group deems that the reset channel is healthy again. At this point the channel will be readmitted back into the operational group and will be allowed to participate in voting.

In general, a TMR ($N = 3$) with FDIR system is more reliable than a single channel with an SCP [5, 24]. To provide the same fault protection as a quadruplex NMR system would require three SCPs [5]. However, NMR systems can be more complex than SCP systems, mainly due to the increased number of channels and the voting activities. Synchronous NMR systems that use a distributed clock synchronization algorithm can be especially challenging to develop [25].

The P-NMR architecture uses an NMR with FDIR design that is described more in Chapter 3.

2.4.2.1 NMR Example: X-38 FTPP

The performance metrics for the P-NMR prototype described in this thesis were compared with another Draper Laboratory avionics system called the Fault Tolerant Parallel Processor (FTPP). This system was developed by Draper Laboratory under contract to NASA Johnson Space Center for use on the now cancelled X-38 Crew Return Vehicle [26]. This system is an NMR design with FDIR. The example configuration shown in Figure 2.4-1 has four flight critical channels and one network element fifth unit (NEFU). Each channel is an FCR with its own power supply and internal clock source. There is a point-to-point CCDL between every pair of channels. The CCDL uses optical fiber for electrical isolation. The four channels and the NEFU operate synchronously using a distributed clock synch algorithm and a channelized data bus architecture is used for the sensors and actuators. The FTPP system is estimated to have an availability of 99.999%.

Each channel (except the NEFU) consists of a chassis with a Versa Module Eurocard (VME) backplane bus, a Flight Critical Processor (FCP) board, an Instrumentation Control Processor (ICP) board, a Multi-Protocol Communications Controller (MPCC) board, various I/O boards and a Network Element (NE) board. All components are COTS except for the NE board. The NE is at the heart of the fault tolerant architecture and performs such functions such as clock synchronization, group membership, input congruency and output voting. These functions are implemented in hardware using a field programmable gate array (FPGA).

The FCP hosts the flight critical guidance, navigation and control (GN&C) software application and it hosts some non-critical software applications. ARINC 653 partitioning is not used to isolate software applications. The ICP is responsible for performing I/O processing using the I/O boards. The four FCPs form a quadruplex *virtual group*. A virtual group is a novel way of grouping individual processor boards in each of the four channels into a fault tolerant group of processors. The ICP boards do not form a virtual

group and operate in simplex mode. The NE in each channel contains a configuration table that has the number of boards in the other channels as well as a description of the virtual groups.

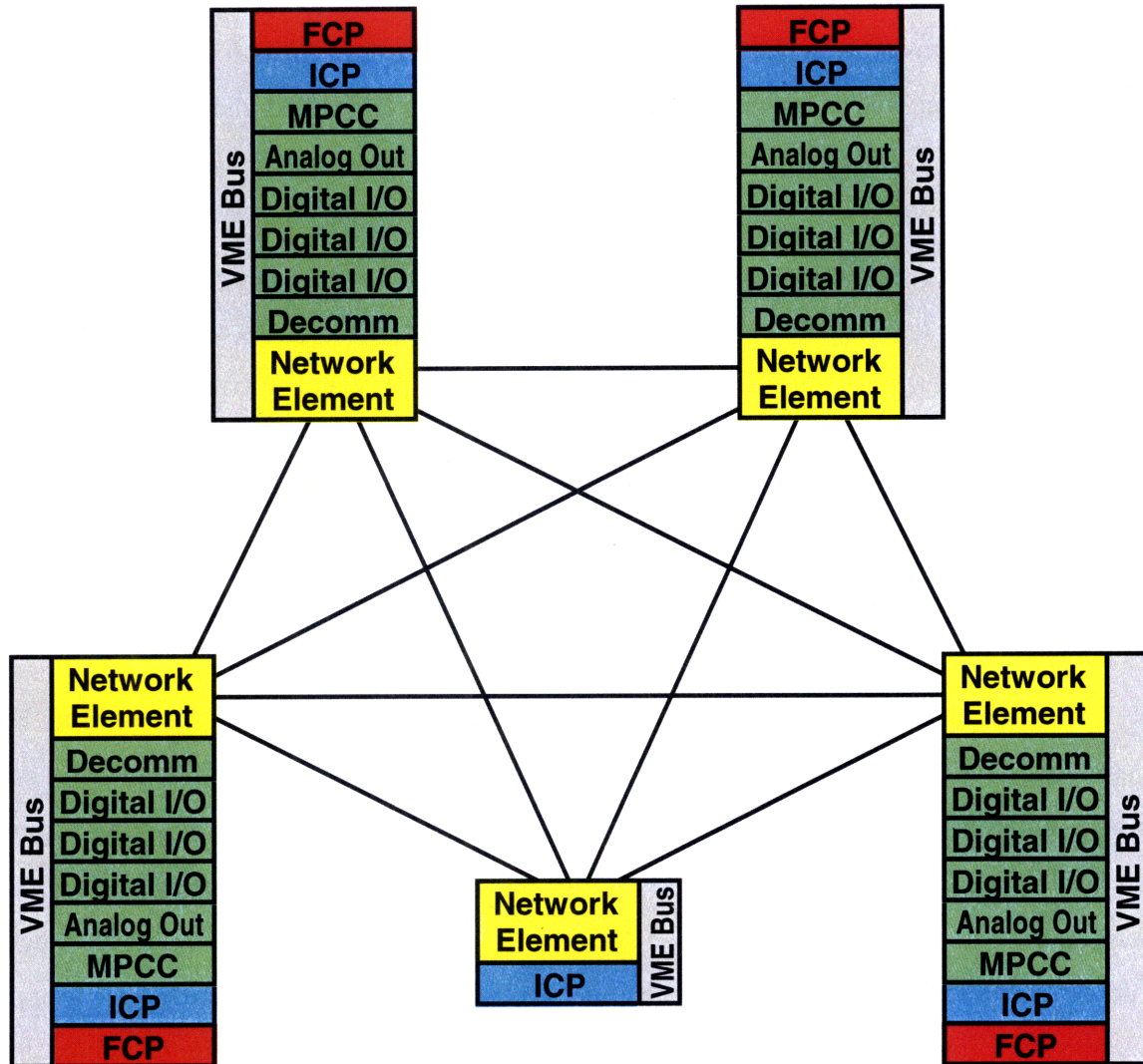


Figure 2.4-1 X-38 FTTP (Courtesy Draper Laboratory)

The X-38 avionics architecture uses a channelized data bus design where a set of sensors is only connected to a single channel. The ICP on each channel will send a request to the NE to have local sensor data distributed to all other channels using the CCDL. The NE will use a two round exchange for this sensor data distribution. Once the two round exchange is complete, the NE on each channel votes the sensor data and writes it to memory accessible by the FCP in each channel.

The FCP then executes the flight critical GN&C code which uses the sensor values to compute actuator commands. The FCP then sends a request to the NE to have the actuator command data voted by the four channels. The NE uses a one round exchange with the other channels. After the vote, each NE placed the voted actuator command in memory accessible by the ICP. The ICP reads the voted command and sends it to the actuators using the appropriate I/O card.

2.4.2.2 Other NMR Examples in Brief

The Boeing C-17 Globemaster is a large military transport aircraft that was designed in the late 1980's and uses a digital fly-by-wire flight control system that is fail-op, fail-op, fail-passive [27]. The probability of aircraft loss due to flight control system failures must be "extremely remote" which is defined as 5×10^{-7} occurrences per mission for military transports. The fly-by-wire (FBW) system uses a quadruplex NMR design with a channelized data bus (Mil-Std-1553b) topology. The four channels are frame synchronized and exchange data using a CCDL. What is interesting about this design is that fault tolerance and redundancy management functions such as voting are implemented in software.

The Boeing 777 transport aircraft was designed in the early 1990's and uses a digital fly-by-wire flight control system [16, 20]. This system must satisfy FAA reliability requirements for catastrophic failures, namely a 1×10^{-9} probability per hour or event. The system is certified for CAT IIb automatic landings (see Appendix D). The FBW system uses a TMR design that includes three flight control computers that are connected to three ARINC 629 data buses. The system uses a global data bus architecture where sensors are connected to all three channels.

A globally asynchronous, locally synchronous approach is utilized. There are three channels that operate asynchronously. Within each channel are three “lanes” where each lane has a different type of processor. The three lanes within a channel are frame synchronized to within a few microseconds using a CCDL that is internal to a channel. One lane is the command lane and the other two are monitor lanes. The outputs from the command lane within each of the three channels are voted using MVS. Before the MVS value is sent out on the ARINC 629 data bus, the values from the monitor lanes are compared with the local channel command lane as well as the monitor lanes on the other channels. A description of a recent in-flight malfunction of the flight control system can be found in [28].

The Space Shuttle avionics consist of four channels and one backup flight computer [29]. The system uses a global data bus architecture where all redundant sensors are connected to all four channels. All four channels are synchronized. The four computers exchange data and use bit-for-bit majority voting. A channel is declared failed if it is in the minority for two consecutive votes.

The Multi-Path Redundant Avionics Suite (MPRAS) architecture was developed by General Dynamics for space launch vehicles in the early 1990’s [21]. It uses redundant channels that are frame synchronized and it uses a channelized data bus approach. This solution is interesting because it uses both one round and two round data exchanges over the CCDL and it uses both MVS and majority voting schemes.

The X-33 is a single-stage-to-orbit, unmanned demonstrator vehicle developed by Lockheed Martin for NASA. It was cancelled in 2001 before its first flight. The flight control system uses a TMR design with three channels that operate in a frame synchronous fashion [25, 30]. Each channel includes a processor board for executing application software and a redundancy management system (RMS) board which implements the fault tolerance functions in hardware.

2.5 Integrated Modular Avionics and Robust Partitioning

This section discusses two open standards that are closely related to one another and that are being widely adopted within the civil and military avionics industry. These two standards are ARINC 651 which provides design guidance for integrated modular

avionics and ARINC 653 which defines a standard application interface to an RTOS which provides services for the robust partitioning of user applications. As will be shown, these standards are having a significant impact on the way new avionics systems are being designed.

2.5.1 ARINC Background

Aeronautical Radio Inc. (ARINC) is a private corporation whose main shareholders are the U.S. airlines and a smaller number of foreign airlines. Within ARINC is the Airlines Electronic Engineering Committee (AEEC) which is an international body of airline representatives, avionics developers and RTOS vendors. The AEEC leads the development of technical standards for airborne electronic equipment including avionics used in commercial, military, and business aviation. The AEEC created the ARINC 651 and ARINC 653 standards and it is responsible for their upkeep. It has been estimated that the use of ARINC standards to foster a competitive avionics marketplace saves the airline industry nearly \$300 million annually [6].

In the past, ARINC avionics standards have tended not to use other standards that are ubiquitous in the commercial computer industry. However, this is starting to change. An example of this is ARINC Specification 664P7, Avionics Full Duplex Switched Ethernet (AFDX) Network, which adapts IEEE Standard 802.3 (Ethernet) for use in safety critical applications on aircraft. This standard is being used, in conjunction with an IMA computing platform, on the most recent transport aircraft programs.

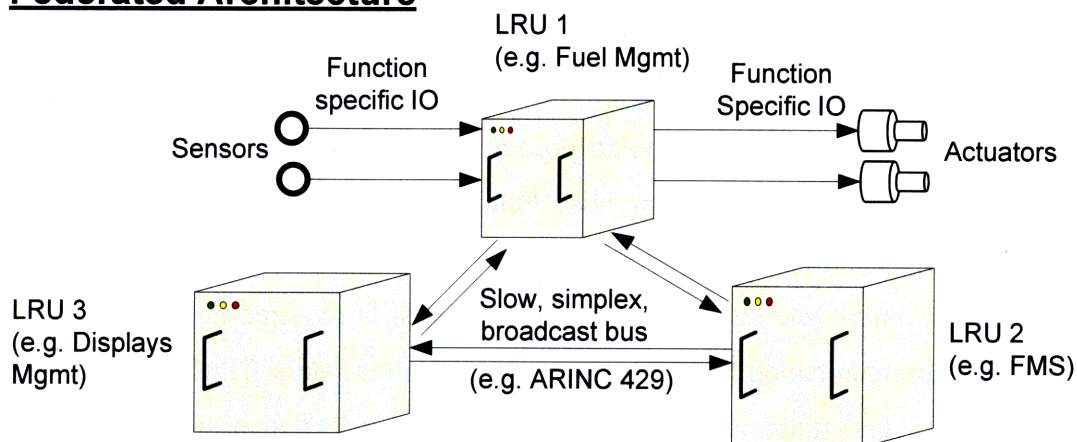
2.5.2 ARINC 651 - Integrated Modular Avionics

A recent trend in the civil and military aircraft industry is a move away from federated avionics architectures towards integrated modular avionics (IMA). The ARINC 651 standard provides guidance for using IMA architectures including guidelines for implementing fault tolerance and for using BIT to identify hardware failures [10]. The following is a brief comparison of federated and IMA architectures. Example architectures are illustrated in Figure 2.5-1.

In a traditional federated system, each avionic function (such as the fuel management, flight management and displays management) is implemented by a separate box or line replaceable unit (LRU). Each LRU box has its own chassis, power supply, backplane bus (e.g. CompactPCI) and processor board which hosts the software for the single function. The LRU may also contain custom I/O boards that interface directly to various sensors and actuators used by the function. Thus, on a large transport aircraft, there will be many LRUs, each implementing a single function. If two LRUs need to communicate with each other, they use dedicated point-to-point connections.

In an IMA architecture, several functions are hosted within a single LRU called an IMA *cabinet*. A single cabinet contains several boards called line replaceable modules or LRMs. Example module types include a core processing module (CPM), I/O module (IOM) and a data bus interface module (or gateway). All the modules may share the same power supply and backplane. In a typical configuration, several avionics functions will execute as software applications on the same processor that is on a CPM within the IMA cabinet. The cabinet will be connected to a high speed data bus that is connected to several remote data concentrator (RDC) units. The RDC units will interface with all the sensors and actuators on the aircraft that are used by the different functions hosted within the IMA cabinet. In essence, the different functions within an IMA cabinet *share common resources* such as the processor, memory, backplane and data bus I/O hardware. In a federated architecture, each function has dedicated resources.

Federated Architecture



Individual hardware boxes communicating over slow bus become software partitions running on a fast CPM communicating through software ports

IMA Architecture

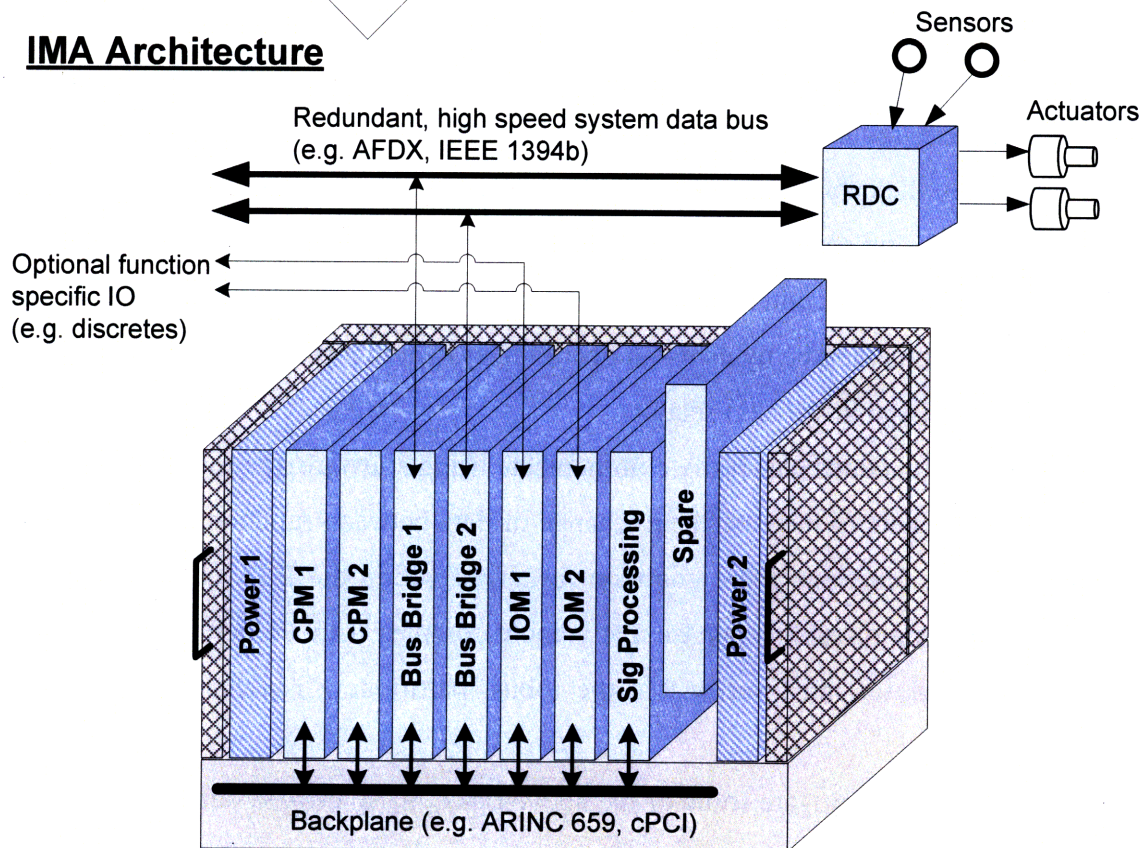


Figure 2.5-1 Federated Avionics to Integrated Modular Avionics (IMA)

There are several advantages to the IMA approach compared to a federated design. An IMA design reduces the size, weight, power consumption and wiring complexity of an avionics system [10]. Instead of many different LRU boxes from many different suppliers all connected with point-to-point wiring, there is just one IMA cabinet connected to a high-speed data bus. Functions hosted on a single processor can communicate with each other simply by using software and shared memory. No new wiring is required when a new communication path is required between two functions. This fast communication between different applications on a CPM has also led to new and sophisticated functionality not possible with a federated design. Finally, modules (processing, I/O etc) can be standardized so that several vendors can produce them for different IMA customers, and thus reducing per unit cost.

There are also several disadvantages to the IMA approach. Disparate functions that used to be physically distributed on the aircraft are now co-located on one LRU. This means the different functions are more susceptible to common mode and common area faults. For example, even though an IMA cabinet may have several redundant power supplies and redundant I/O modules, a single fire or power surge could cause all functions in the cabinet to be lost. Additionally, there is the potential for a low criticality function (e.g. passenger cabin temperature control) to interfere with a high criticality function (e.g. flight control) executing on the same processor. This risk of interference is reduced to some degree by using an operating system that provides robust partitioning, as will be discussed in the next section.

Since functions can easily communicate using software, there is also a tendency for system designers to increase the exchange of data between functions. This, coupled with the fact that functions share common resources within the IMA cabinet, can lead to a system integration and test phase that is more complex than a federated architecture. The system integrator must verify that the robust partitioning is reliable and implemented correctly.

In order to mitigate some of the above risks, many IMA systems use a dual cabinet approach where there is a pilot cabinet and a copilot cabinet with a CCDL. These dual systems usually host safety critical functions such as the flight management system

(FMS). Some systems even host the flight control function, as will be shown in section 2.5.4.

2.5.3 ARINC 653 - Robust Partitioning

2.5.3.1 Motivation

The development of IMA systems would not have been possible without robust partitioning techniques for the different avionics software functions (called applications). The goal of IMA is to be able to host several different software applications on the same processor and to allow the applications to share common resources within the IMA cabinet such as memory, the backplane bus and I/O hardware. The hosted software applications could be from different vendors and they could be completely unrelated. They could also be at different criticality levels. For example, one application could be designated as DO-178B Level A whereas another application could be designated as Level C. In order for IMA systems to be certifiable, a mechanism was required that minimized the risk of one application interfering with another application.

Robust partitioning was also essential in terms of reducing the cost of system development and test. A system without partitioning would mean that if one application on the processor was designated DO-178B Level A and nine other applications were designated Level C then all ten applications would have to be developed and tested using Level A criteria and objectives. This would be prohibitively expensive. Partitioning would allow each application to use a development process suitable for the assigned criticality.

Thus, the first robust partitioning techniques were developed in-house by the initial IMA pioneers [31]. The partitioning is provided by the RTOS that is executing on the CPM within the IMA cabinet. The airline and aircraft industry recognized the benefits that IMA and robust partitioning could provide and they also realized that a cost advantage could be had if different IMA/RTOS products provided the same set of partitioning services that could be used by application software developers.

A standard partitioning interface to the RTOS would mean that different software applications from different suppliers could be integrated on the same IMA platform

without requiring code changes. It would also mean that software applications would be portable and reusable; they could be moved from one IMA/RTOS platform to another without extensive code changes. The applications would also be hardware independent.

An RTOS standard for safety critical avionics would also mean that the major commercial RTOS vendors who had not previously been in the avionics marketplace could now provide an open, COTS product that could be used by several avionics system developers. It is out of this need for an open standard for safety critical RTOS products that ARINC 653 was born.

The ARINC 653 standard is titled “Avionics Application Software Standard Interface” [13]. It describes the robust partitioning responsibilities of the RTOS kernel and it specifies the application executive (APEX) API. The APEX API specifies the standard set of RTOS services that a programmer can use when developing an application. All RTOS products must provide the APEX API in order to be compliant with the ARINC 653 standard. It should be noted that although IMA was the genesis for ARINC 653, the standard can be, and has been, used in traditional federated systems as well.

2.5.3.2 ARINC 653 Overview

This section provides a brief overview of the ARINC 653 standard. More details are provided in Chapter 4 when discussing specific aspects of the P-NMR system design.

Each software application is contained in a *partition*. A partition is the software equivalent of a hardware channel described in section 2.3.1. In essence, it is a *software fault containment region*. A fault that occurs within one partition cannot escape to another partition. Incorrect data in one partition cannot pollute the data in another partition. One partition cannot perform an illegal write in another partition’s memory space.

A partition can also be thought of as a virtual computer. It is guaranteed a certain amount of processor time and a certain amount of protected memory. This is often referred to as time and space partitioning. A partition is also guaranteed access to I/O devices such as the backplane bus and the data bus. To the application programmer, it appears as if their application is the only application executing on the CPM within the

IMA cabinet. It is the system integrator's responsibility to configure the system such that all application partitions are granted the appropriate access to hardware resources.

At the CPM level, the RTOS is responsible for managing the different partitions. Partition management includes starting and stopping partitions (mode switches), scheduling, interpartition communication using ports and partition health monitoring.

Partitions are scheduled on a fixed, cyclic basis. The sequence of partitions is called the *major frame* and it is repeated continuously while the module is operational. There can be more than one major frame defined for the module. User partitions are allocated partition *windows* within the major frame. A single partition can have several windows within the major frame. A hardware interrupt handled by the RTOS may be used to guarantee that a partition does not exceed its current schedule window. The RTOS is responsible for saving the appropriate state (e.g. processor registers) when a partition switch occurs.

Partition communication is accomplished using ports. An ARINC 653 partition can communicate with another partition that is on the same module or on a different module in another cabinet. A partition can also communicate with non-ARINC 653 software as well as external hardware devices such as smart actuators. A communication channel defines the path from one partition to another. A communication channel must have a source port and a destination port.

An application partition is comprised of one or more processes. The processes can be periodic or aperiodic. The processes within a partition are scheduled using a priority preemptive scheme. Processes within a partition can also communicate with each other using buffers and blackboards. All the processes share the same memory space of the parent partition. Processes within a single partition may be managed by a local *partition level OS*. Each partition may contain a partition OS. All the instances of a partition level OS are completely distinct from the single core module OS that manages all the partitions.

The standard also defines a health monitoring facility. Errors can be defined at the process, partition or module level. Examples of errors include divide by zero, memory access violations and missed deadlines. Various response mechanisms can be defined for each kind of error and different responses can be used at different levels. If a response is

not defined at a particular level then the error is propagated to the next higher level. For example, if a process has no response defined for a missed deadline then the error will be passed up to the partition. A cold or warm restart is possible at the process, partition and module level.

The partition schedule, communication channels, ports and health monitoring can all be defined in system configuration files at design time. The ARINC 653 standard provides examples of how XML can be used to define the configuration data. The configuration data can also include the memory layout of the module including the memory allocated to each partition. The configuration files can be validated using offline static analysis tools. Once validated, the text data can be converted to binary tables that are loaded into ROM on the module.

The standard recognizes that some hardware support may be required to implement the partitioning services. This support may include the use of a memory management unit (MMU) to protect partition memory segments and it may include various hardware interrupt mechanisms used to implement robust time partitioning.

Finally, the standard does not specifically address fault tolerance or redundancy management design issues. The specification of health monitoring facilities only defines how faults can be handled by the RTOS at the process, partition or module level. It does not discuss how these fault handling techniques can be integrated into a larger system wide redundancy management architecture. For example, it does not discuss the notion of replicated partitions that are hosted on redundant IMA cabinets.

2.5.4 Example ARINC 653 Systems

This section discusses some avionics systems that use or will use ARINC 653 robust partitioning. Of particular interest are systems that have used the technology to host the flight control (i.e. control laws) function. This is the ultimate validation of the technology. An analysis of when it may make sense to host the flight control function on an IMA platform as opposed to a traditional system comprised of dedicated flight control computers can be found in [32]. Before discussing example IMA architectures, we first describe the fault tolerance approach used by many of these systems.

This fault tolerant IMA architecture uses a dual cabinet approach. Normally, there is a left (or pilot) cabinet and a right (or copilot) cabinet. Each cabinet is a fault containment region (or channel). The CPM within each cabinet uses two lockstep processors in an SCP configuration. The SCP executes identical software and compares outputs. The two cabinets communicate using a redundant CCDL. Often, a data bus architecture that is similar to a channelized design is used. Each cabinet is connected to its own dedicated avionics data bus (pilot bus and copilot bus). Each data bus has its own set of sensors and actuators connected to it (i.e. the sensors and actuators are dual redundant). However, the copilot cabinet can listen to the pilot data bus and the pilot cabinet can listen to the copilot bus.

One cabinet is called the *master*, and the other cabinet is called a *shadow*. Safety critical partitions such as the FMS and flight control are hosted on both the master and the shadow. The master calculates outputs using the sensors connected to its data bus and the shadow calculates outputs using the sensors connected to its data bus. However, only the master partition provides outputs onto its data bus. The shadow cabinet monitors this data. The master and shadow also exchange data from over the CCDL. If the master cabinet experiences a fault, the shadow will notice this either by a lack of data bus traffic or a lack of CCDL traffic. The shadow will then takeover and become the master. It will output data on its data bus. The old master can be reset and then once operational, it can become the shadow. An example of a fault may be a miscompare by the SCP processors on the CPM.

It should be noted that this dual SCP approach is immune from Byzantine faults since there are only two channels and cross channel majority voting is not used. Also, the two channels (master and shadow) usually operate asynchronously and so there is no need for a clock synchronization algorithm. We now discuss some actual IMA systems.

The Boeing 777 Airplane Information Management System (AIMS) was the first instance of IMA and it was the genesis for the ARINC 653 APEX API standard for robust partitioning [31, 33]. The first flight of the 777 was in 1994. The AIMS platform with the APEX operating system was developed by Honeywell. It uses the two cabinet approach described above with SCP processors. Modules within a cabinet communicate with each other using the ARINC 659 bus which uses a table driven, time division

multiplexing (TDM) approach. The two cabinets communicate using an ARINC 629 CCDL (later AIMS versions use an Ethernet CCDL). Safety critical applications such as the flight management system are hosted on both the master and the shadow cabinets. However, AIMS does not host the flight control function which is part of the aircraft's digital fly-by-wire system. The flight control function is still hosted on a dedicated TMR system with more fault tolerance than the two AIMS cabinet (the flight control computers are described in section 2.4.2.2).

Honeywell proceeded to develop the second generation of AIMS, called Versatile Integrated Avionics (VIA). VIA uses a smaller form factor and more powerful processors. It still uses the dual cabinet approach with SCP processors in each cabinet. The VIA is used on the Boeing 737 Next Generation (NG), the C-5 military transport and the KC-10 tanker. What is interesting about the C-5 application is that the flight control function is a software partition that shares the processor with other partitions [32]. The C-5 is also being certified for CAT IIIa automatic landings. Flight testing of the C-5 with new VIA based avionics was completed in 2006.

Honeywell has developed a third generation IMA system called Primus Epic™ that has been mainly used on large business aircraft. Primus Epic uses an even smaller form factor than VIA and it uses more COTS components. Primus has been used on aircraft with digital fly-by-wire systems and it has hosted the flight control function. However, the function is hosted on its own dedicated module within the cabinet. It does not share the module with other software partitions. Primus uses the DEOS™ operating system that does not follow ARINC 653 and it allows for the dynamic creation of processes [34].

Table 2.5-1 lists the above systems as well as some other systems that use an ARINC 653 compliant RTOS. The “IMA” column indicates if the avionics platform is an IMA architecture where different functions share the same resources. The “Flt Cntrl” column indicates if the flight control function is part of the avionics platform.

In summary, the ARINC 653 standard has been used for dual SCP architectures as well as for TMR architectures. Details of the TMR examples have not been published. It has not yet been used for hosting the flight control function on a large transport aircraft but it has hosted the flight control function in other applications such as military aircraft, business jets and unmanned aerial vehicles.

Vehicle	In Svc Year	Avionics Platform	ARINC 653 RTOS	IMA	Flt Cntrl	Comment	Ref
Boeing 777	1995	AIMS	Honeywell APEX	Yes	No	The first IMA system, origin of 653, dual SCPs	[31]
C-5 upgrade	2005	VIA	Honeywell APEX	Yes	Yes	Flt cntrl software partition shares processor, dual SCPs	[32]
Airbus A380	2007	IMA	Thales	Yes	No	Modified APEX API, Ethernet (AFDX) bus	[12]
Boeing 787	2008	Flight Cntrl Computers	Greennhills Integrity	No	Yes	FBW, NMR with SCPs	[7]
Boeing 787	2008	Common Core System	Wind River VxWorks	Yes	No	Ethernet (ARINC 664) data bus	[11]
S-92 Helicopter	2009	Vehicle Mgmt Computers	BAE CsLEOS	No	Yes	FBW, TMR with SCPs, frame synchronous	[35]
C-17 upgrade	2009	Flight Cntrl Computers	BAE CsLEOS	No	Yes	FBW, frame synchronous	[36]
X-47B UAV	Canceled	Vehicle Mgmt Computers	Smiths	Yes	Yes	TMR with voting, approximate agreement	[37]

Table 2.5-1 ARINC 653 Avionics Systems

2.6 Related Research

The authors in [38] describe an approach for integrating redundancy management services with an IMA based system. A proprietary scheme is used for time and space partitioning instead of the ARINC 653 standard with the APEX API. The authors use the MAFT architecture [39] and the X-33 vehicle management computers [25] as the starting point for the system architecture. However, these systems use a custom hardware board to implement the redundancy management system. The authors implement a software only version of RMS that executes on the same processor as the user applications. The software version of RMS implements clock synchronization and voting of data. Performance metrics were collected using a TMR prototype. The authors show that integrating RMS with applications on the same processor increased voting throughput and provided acceptable performance. They also indicated that strong partitioning was maintained even with the integration of RMS with applications and that the internal design of the RTOS has a large impact on the complexity experienced during system integration.

Reference [40] discusses using an IMA architecture with ARINC 653 for the avionics systems on Constellation vehicles. The authors propose using a master/shadow approach (or master/shadow/shadow) with SCPs in each IMA cabinet such that Byzantine faults can be avoided. In other words, an NMR architecture with cross channel voting would not be used. Details are not provided on how the shadow channels would take over. Also, it appears that the master and shadows would operate asynchronously but details are not provided on how divergence between the channels would be prevented. The authors also discuss using a COTS backplane such as CompactPCI within an IMA cabinet and a COTS high speed serial bus such as IEEE 1394b between IMA cabinets.

Reference [41] discusses methods and tools for scheduling user application partitions and I/O processing within a single IMA cabinet that uses a shared backplane. The paper uses a backplane model similar to ARINC 659 time division multiplexing. The IMA cabinet contains several modules and each module may contain one or more processors. All processors are using an ARINC 653 compliant RTOS. The ARINC 653 two level scheduling method is used on each processor (i.e. partitions are scheduled and then processes within a partition are scheduled). The authors discuss how user partitions can be scheduled along with the I/O processing that is required between the modules within the IMA cabinet. Fault tolerance is not addressed.

Reference [42] describes the Generic Avionics Scaleable Computing Architecture (GASCA). A TMR system prototype is used with some fault tolerance functionality. Each channel consists of a VME chassis with several PowerPC boards. Every board within a VME chassis is also connected to all other processing boards using an SCI link. The API used by applications is based on ARINC 653 but has been modified and extended. It does not appear that the POSIX based RTOS supports true robust partitioning. There is no mention of how synchronization and voting is performed.

The authors in [43] discuss the notion of hardware and software fault containment regions for an NMR fault tolerant system architecture model that uses IMA channels with software partitions. The data bus model is similar to the one used by the time triggered protocol (TTP/C).

Reference [44] proposes a common system software framework that uses the Real Time Publish/Subscribe protocol for framework-to-framework communication to extend

ARINC 653. The authors show how such a framework would be well suited for the Constellation program system-of-systems approach where vehicles can be reconfigured (e.g. by docking in orbit). Fault tolerance is not explicitly addressed.

References [45], [46] and [47] also discuss IMA and fault tolerance.

The P-NMR architecture discussed in this thesis is unique from all of the above works in that it uses an ARINC 653 compliant RTOS and the APEX API for an NMR with FDIR system that could be used for flight control applications. It specifically addresses how a software implementation of FDIR functions can be integrated with flight critical user application partitions on the same processor. This work also describes how user applications that require fault tolerant services can still use the APEX API contained in the ARINC 653 standard.

This page intentionally left blank

Chapter 3

P-NMR System Architecture

3.1 Overview

The P-NMR system uses a partitioned, distributed N-modular redundancy approach that utilizes COTS hardware and software based on open standards. The *Fault Tolerant System Services (FTSS)* component developed by Draper Laboratory provides the FDIR functionality¹. FTSS is implemented entirely in software that utilizes COTS hardware based on open standards such as IEEE-Std-802.3. Unlike the X-38 FTTP, no custom or proprietary hardware is utilized. The performance of the FTSS software has been optimized to reduce the fault tolerance overhead and thus reduce overall system response times.

The distributed NMR aspect of the design uses a classical frame synchronous approach. Each channel is an FCR and is connected to all the other channels using dedicated CCDLs. All channels synchronize their local clocks to an agreed upon global clock and all channels start a schedule major frame at (approximately) the same time. This allows for source input congruency (i.e. input exchange and voting) followed by application execution followed by voting of application outputs (i.e. computer fault masking).

A distributed NMR topology was selected for the fault tolerance strategy over other strategies such as dual self-checking pairs mainly for two reasons. First, Draper Laboratory wanted to compare the performance results of the new design with a specific instance of the earlier X-38 FTTP design which is a four channel NMR architecture. Second, as discussed in section 2.3.3, it has been well documented that a properly designed four channel NMR system with two-round exchanges can tolerate two simultaneous symmetric faults or one asymmetric (Byzantine) fault. Since a production P-NMR system could be used in a space environment where radiation induced single

¹ At the time of this writing, not all FDIR functions had been implemented in FTSS.

event effects can cause both symmetric and asymmetric faults, this was an important consideration.

The P-NMR system prototype used for the results in this thesis is a TMR system (NMR design with $N = 3$) simply because it allows more debug and test connections from each channel to the PC used for development and testing. However, the actual TMR performance results can be accurately extrapolated to a four channel system.

The other main aspect of the P-NMR architecture is the use of robust partitioning as used in IMA. The architecture integrates the flight control computers and the mission management computers into a single vehicle management computer. Both flight critical and mission critical application partitions use the same computing resources. The flight critical partitions are replicated on all channels. This IMA approach facilitates a reduction in size, weight and power requirements and it reduces the amount of wiring on the vehicle. Robust partitioning allows applications of different criticalities to be safely hosted on the same processor. A COTS RTOS that complies with the industry accepted ARINC 653 standard for robust partitioning is used on each channel. Each partition executing on a processor can be thought of as a software fault containment region.

Figure 3.1-1 depicts an overview of the current P-NMR system prototype. There are three identical, redundant channels: CH1, CH2 and CH3. Each channel is a single board computer (SBC750GX). The channels are connected to each other using point-to-point, Gigabit Ethernet CCDLs. Each channel is running an ARINC 653 compliant RTOS (VxWorks 653). Two software partitions are shown executing on each channel but in reality many more partitions would be present. The flight critical GN&C partition is replicated on all three channels whereas the non-flight critical partitions A, B and C are specific to a particular channel. Figure 3.1-2 shows the actual hardware setup in the lab. The red cables are the Gigabit Ethernet CCDLs, the blue cables are Ethernet connections to the development and test PC and the black cables are RS-232 connections to the development and test PC.

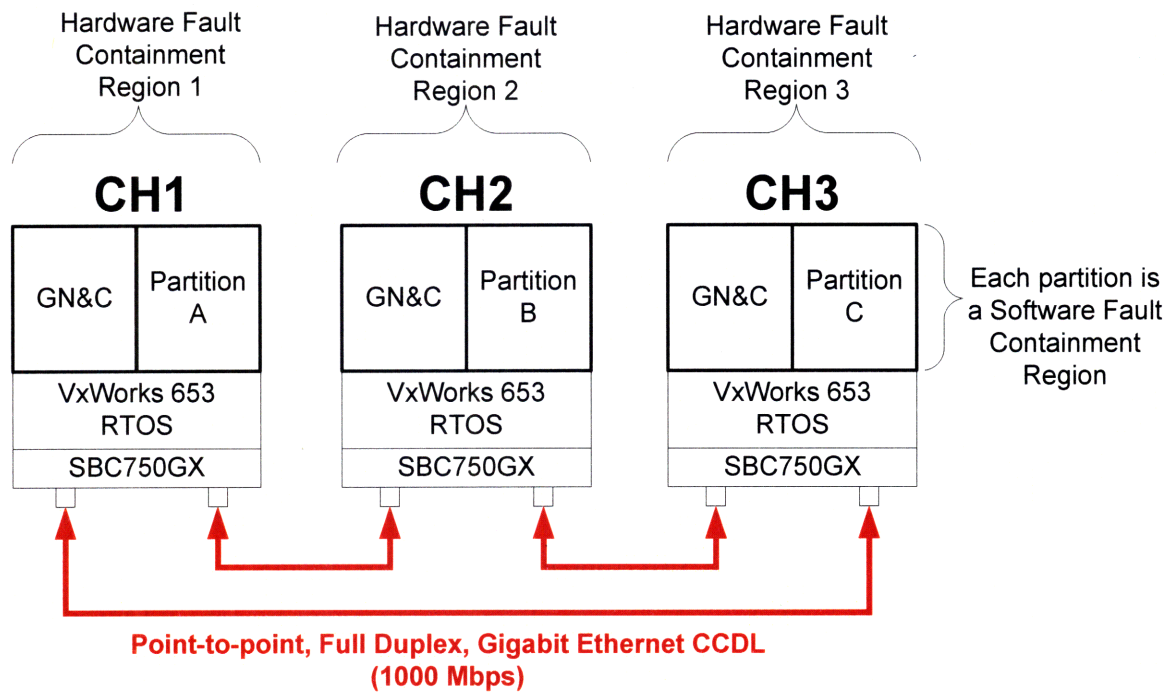


Figure 3.1-1 P-NMR System Overview

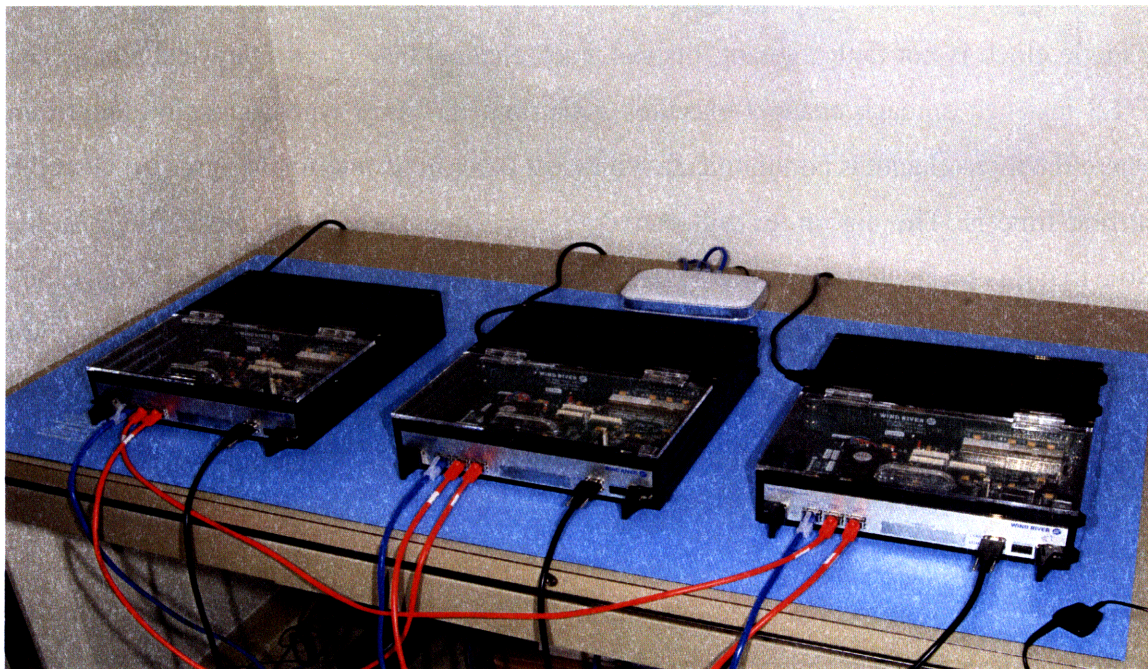


Figure 3.1-2 P-NMR System Prototype in Lab

3.2 A Frame Synchronous Approach

The P-NMR system uses a frame synchronous approach to facilitate fault tolerance and redundancy management functionality. This means that all three channels in the TMR architecture are (loosely) synchronized to the same “global clock” and that all three channels execute the same fixed partition schedule at the same time (there is an exception to this rule dealing with non-flight critical partitions which will be discussed later). This approach is required so that FTSS can guarantee the following:

- input congruency – all channels have the SAME sensor inputs at the SAME time,
- synchronous application execution – all channels execute the SAME application code at the SAME time using the SAME inputs,
- output voting – all channels have application outputs ready at the SAME time so that they can be distributed and VOTED by FTSS.

A synchronous fault tolerant system allows for simple bit-for-bit voting of input exchanges and application outputs. As discussed in section 2.3.4, asynchronous systems require more sophisticated voting and channel equalization.

The global clock is not a single clock source that is used by all three channels since a single clock is not fault tolerant. Instead, each channel has its own physical clock. The FTSS instance on each channel executes a distributed, frame synchronization algorithm where the three channels communicate with each other in an attempt to agree on what the current time is. The algorithm will converge over time even in the presence of faults. Note that this thesis will occasionally use the term *frame synchronization* instead of clock synchronization. The goal of frame synchronization is similar to clock synchronization and frame synchronization can often use the same algorithms. However, the primary goal of frame synchronization is to align the execution schedules on all three channels which is not always the same goal for clock synchronization algorithms presented in the literature.

Figure 3.2-1 shows the basic operation of FTSS and a 50Hz GN&C application that is replicated on all three channels. An execution rate of 50 Hz means that each execution period will be 20 ms. This is called the *minor frame*. Before the GN&C partition executes, FTSS input processing takes place. The individual inputs such as sensor data

that go into each channel are reliably distributed to all other channels by FTSS. Then identical GN&C application code executes on each channel. The GN&C code uses the inputs from FTSS. Finally, the GN&C outputs from each channel (such as actuator commands) are distributed to all channels and voted by FTSS output processing. The result of the vote could be sent directly to an I/O driver or to an I/O Processing partition that manages outputs destined for actuators or other subsystems on the vehicle. Other applications may execute after the FTSS output processing in the minor frame. This sequence will repeat continuously.

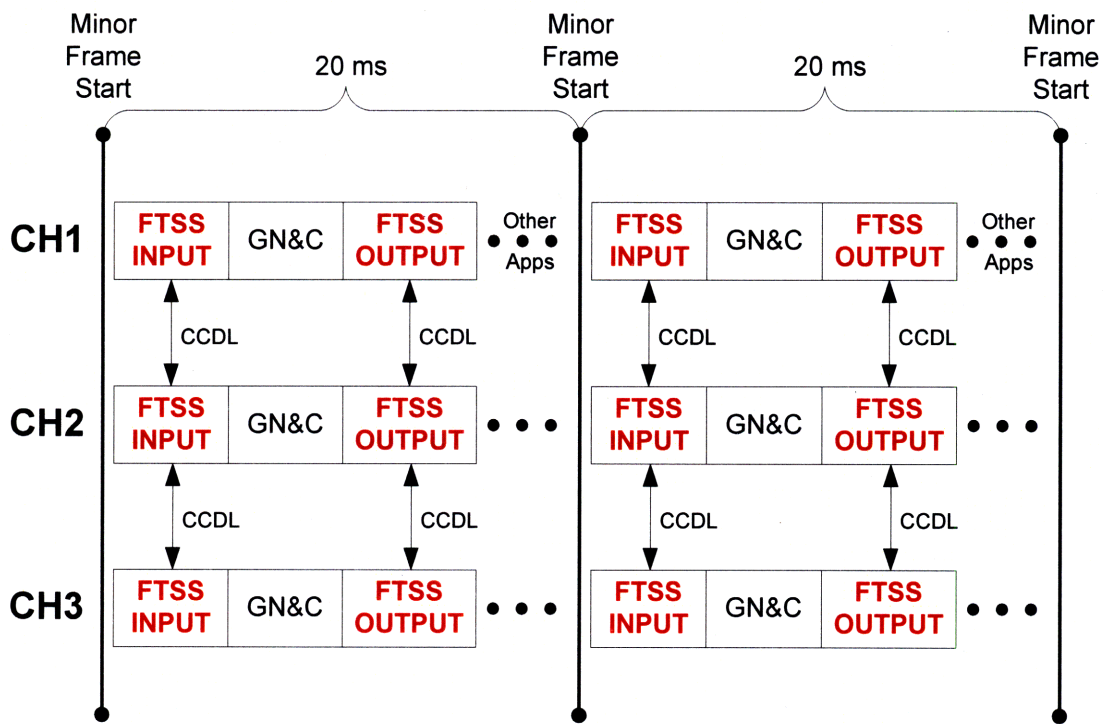


Figure 3.2-1 P-NMR Frame Synchronous Approach

The FTSS and GN&C processing must happen at approximately the same time on each channel. It would be unacceptable if one instance of GN&C used sensor data from the previous minor frame (20 ms ago) while the other two channels used fresh data acquired only 1 ms ago. This would result in the channels operating asynchronously, making bit for bit voting of the outputs impossible. The processing is aligned on a frame boundary. In other words, each channel starts a frame at approximately the same time.

Finally, the FTSS operations before and after an application executes have been made as efficient as possible to reduce system response times. A goal for the P-NMR system is that the total time from sensor reading to actuator activation must be less than 10 ms. This duration comes from the assumption that the GN&C software may be used for flight control during dynamic phases of the mission such as ascent and reentry.

3.3 Simulated Channelized Data Bus Architecture

In an avionics system, the data bus connects RDCs, sensors, actuators and other avionics subsystems to the TMR channels. It is a completely separate bus from the CCDLs which link channels together. The current P-NMR prototype does not have a real high speed data bus such as IEEE 1394b or AFDX. Instead, inputs from sensors are simulated using test software running on each channel.

For initial prototyping and performance evaluation, a channelized data bus architecture is simulated. As discussed in section 2.3.5, a channelized architecture means that a set of sensors and a set of actuators are only connected to a single channel. The set of sensors, the channel and the set of actuators is called a string. The channelized approach promotes electrical isolation and fault containment and it only requires one bus interface unit per device. It is also easier to verify in terms of bus timing analysis since the channels can only communicate with each other using the CCDLs – they cannot use the data buses to communicate.

Figure 3.3-1 shows the simulated data bus architecture used for initial P-NMR performance evaluation. Connected to each channel is a single data bus. Connected to this data bus are three sensor types: miniature inertial measurement unit (MIMU), GPS and barometric altimeter (ALTIM). All three sensor types are triple redundant so that each channel has its own set of three sensors. A single sensor is only connected to one channel using one data bus. On the output side, there are three redundant reaction control system (RCS) thrusters. A single thruster is only connected to a single channel using the same data bus as the sensors.

It should be noted that the sensors and actuators for a single channel may feed into an RDC and then the RDC would be connected to the single channel using the data bus. As will be discussed later, the FTSS software does not have to change if a switch is made

from a channelized data bus architecture to a global data bus architecture where all sensors and actuators are connected to all channels using the data buses.

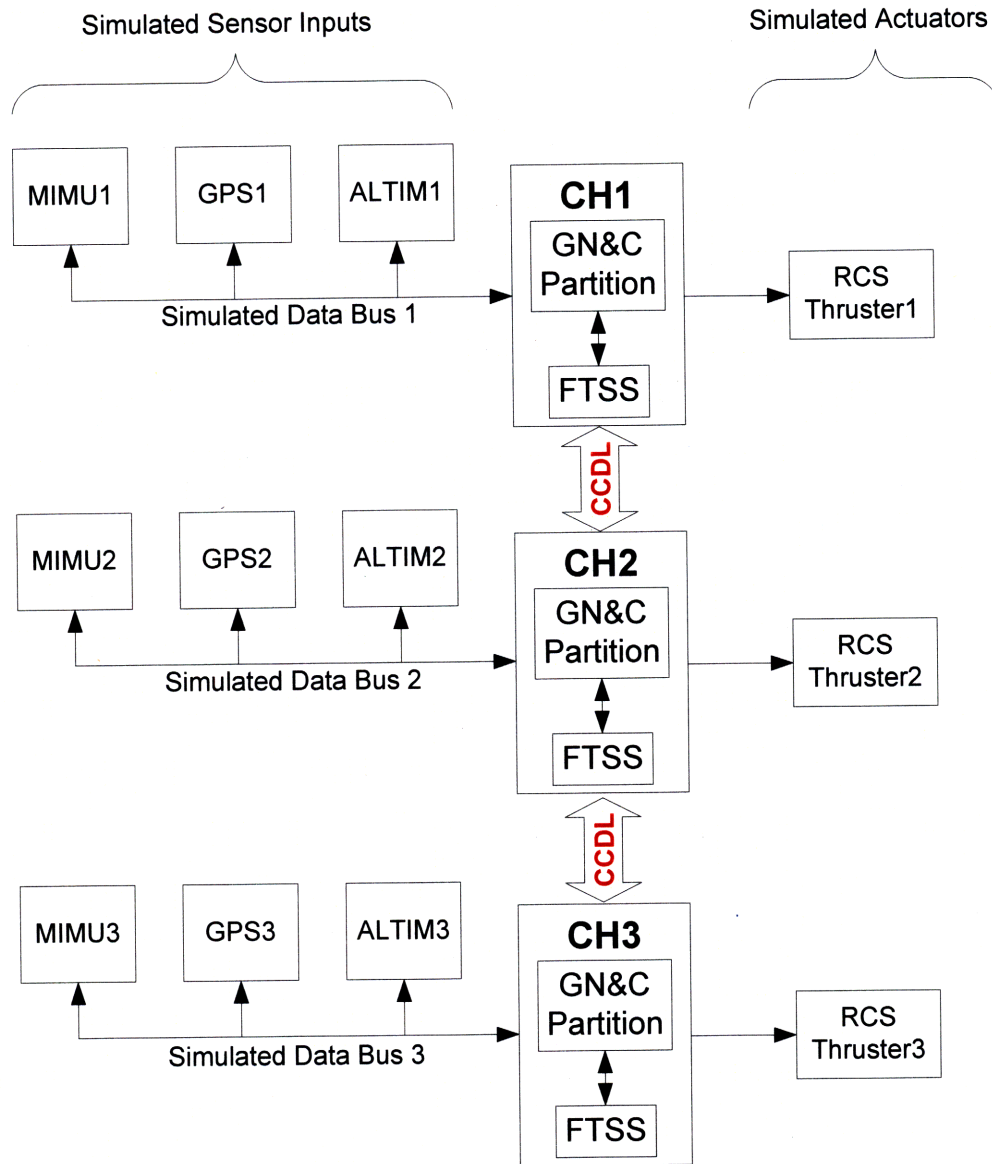


Figure 3.3-1 P-NMR Simulated Channelized Data Bus Architecture

An important point is that the GN&C partition (or any other partition) does not need to know how the sensors or actuators are physically connected to the redundant channels. For example, if GN&C on CH1 requires inputs from all three MIMU sensors then FTSS will make sure that the MIMU sensor data from CH2 and CH3 is reliably distributed to

CH1. User partitions are completely isolated from having to know the physical connections of the system. To the GN&C partition on CH1, it will appear as if all three MIMU sensors are connected directly to CH1. Also, the distribution of data to user partitions on different channels can be specified at design time using system configuration files. No software changes are required when changing the routing of data to application partitions. The reliable distribution methods used by FTSS are explained next.

3.3.1 Sensor Input Processing by FTSS

This section describes at a high level how FTSS distributes data from the simulated, redundant sensors to the three P-NMR channels for application processing. This exchange of sensor data is often termed *input congruency*. The method described is not Byzantine resilient because in order to tolerate one Byzantine fault, four channels are required with two rounds of data exchange. The P-NMR prototype only has three channels. Thus, the input distribution method described in this section uses a modified two round exchange that is meant to replicate the amount of data that would be exchanged in a four channel system. The processing is only performed for sensor inputs that require fault tolerant distribution. The selection of inputs that will use the FTSS service is done at design time using configuration files as will be described later. We now describe why sensor input distribution is necessary.

In a channelized data bus architecture, each redundant sensor is only connected to a single channel using a single data bus. However, it is still desirable to have each channel evaluate all the redundant sensors to determine if any of the sensors have a fault. The customary method for channels to exchange data with each other is by using the CCDLs which would include features such as electrical isolation so that one channel cannot corrupt or damage another. Thus, under nominal conditions, all three P-NMR channels would have access to the data from all three redundant sensors via the CCDLs but if two channels failed, the remaining operational channel would still have access to the sensors connected directly to it via its own local data bus.

Figure 3.3-2 shows a simplified sequence of events for distributing the redundant MIMU sensor inputs to all three channels in the P-NMR prototype using the CCDLs. The

two round exchange is managed by the FTSS input task executing on each channel. In reality, more than just one sensor would be distributed among the channels. The sequence starts out with the input simulation code producing sensor inputs on each channel. For this discussion, the sensor simulation results in the following: MIMU1 sent the value 7 to CH1, MIMU2 sent the value 8 to CH2 and MIMU3 sent the value 9 to CH3. These values represent three different readings of the SAME vehicle state such as angular roll rate. In other words, the three redundant sensors are reading slightly different values for the same parameter.

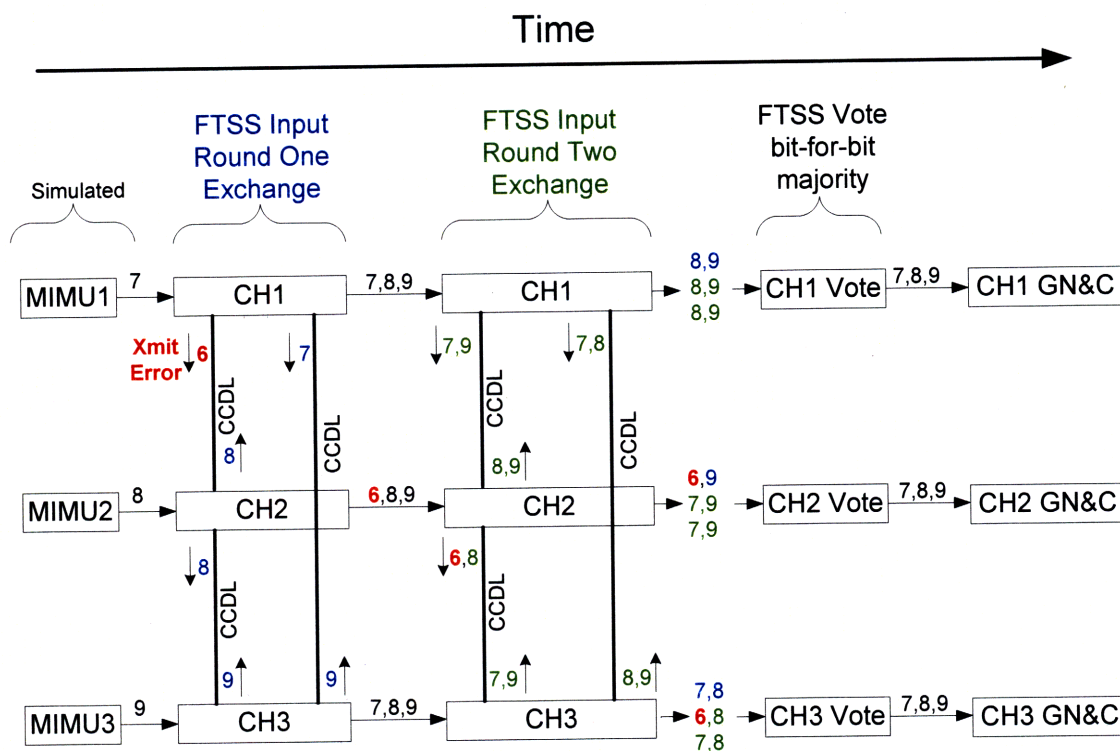


Figure 3.3-2 Sensor Input Processing by FTSS

The FTSS input task on each channel sends the received sensor data to all the other channels using the CCDLs. This is called “round one” of the two round input exchange (shown in blue). At the end of round one, each channel has the sensor data that it received from its own sensor (via the local data bus) and it also has the sensor data that it received from the other two channels via the CCDLs. For the purposes of this example, CH1 experienced a transmission fault during round one. It attempts to send the value 7 to CH2

but during the transmission the value is changed from 7 to 6 (shown in red) and the CCDL error detection mechanism (e.g. CRC) did not detect the fault. The send from CH1 to CH3 was successful and CH3 receives the value 7.

After round one, the FTSS input task on each channel resends its own local data and it forwards data it received from each channel during round one to other channels. For example, data that CH1 received from CH2 during round one is forwarded to CH3 but it is not echoed back to CH2. This is called “round two” of the two round exchange (shown in green). This round two exchange diverges from a traditional exchange for Byzantine fault tolerance since a traditional exchange would not resend the local channel data during round two. For example, a traditional exchange would have CH1 send its MIMU1 data to other channels only during round one. However, since the P-NMR prototype only has three channels, this divergence was necessary in order to obtain three sets of data that can be voted.

At the end of round two, each channel has three sets of sensor data from the other two channels. One set was received during round one. Two other sets were received during round two. The FTSS input task on each channel then sends the three sets of sensor data to the FTSS voter on each channel where bit-for-bit majority voting is performed on the three sets of input data in an attempt to determine if there were any faults during the exchange. It is during the voting of the three sets of data that the transmission error during round one from CH1 to CH2 is removed.

At the end of the two round exchange, the result of the FTSS voter is passed to the applications (e.g. GN&C partition). It is important to realize that all three sensor readings (7, 8, 9) are passed from FTSS to the GN&C application partition executing on each channel. It is up to the user application code to perform sensor specific fault detection (e.g. Kalman filters, statistical correlations) and to determine what value should be used for the single input (e.g. angular roll rate). The FTSS input processing is simply to make sure that local sensor inputs are correctly distributed to all channels via the CCDLs before the application partition starts to execute.

FTSS input processing makes no judgment about the “correctness” of a sensor value with respect to the other values. In this way, the FTSS voter can use a simple and fast bit-for-bit majority algorithm. The FTSS voter doesn’t have to be concerned about

median values, minimums, maximums or filters. It also doesn't need to know what the data represents – it doesn't care if it is an unsigned integer or a floating point real number. All it needs to know is the size of the data item in number of bits. This leads to a portable FTSS design. The disadvantage is that the application code must notify FTSS if a failed sensor is detected.

There are several places where the user sensor FDIR software could be located. For example, the sensor FDIR code may be a process in the GN&C partition that always executes before any other GN&C process (e.g. Control or Nav) during the partition window. A better option may be to have a separate GN&C FDIR partition. This way, the sensor FDIR and the actual GN&C algorithms are kept separate. However, this design comes at the cost of additional inter-partition (i.e. port) communication overhead (process to process communication within a partition is more efficient than ports). Figure 3.3-3 shows an example of the flow of data from the FTSS two round exchange to the GN&C FDIR partition and finally to the GN&C partition itself.

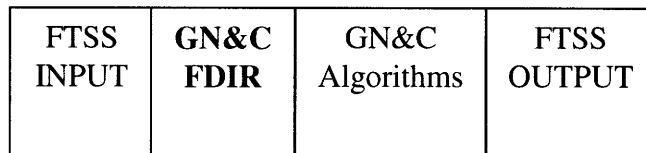


Figure 3.3-3 GN&C FDIR partition added to schedule

An important point is that the FTSS design does not require a two round exchange. It can easily perform one round exchanges. Also, the number of redundant sensors does not have to be perfectly symmetrical with respect to the number of channels. For example, there can be just two redundant sensor units and three channels or four channels. As will be explained, these changes in the design can be specified in the XML system configuration files.

In summary, the described two round exchange comes at the price of high CCDL bandwidth consumption and increased input latencies. Applications have to wait while the sensor inputs are being exchanged and voted. Also, the messages exchanged during round two are twice as large as the round one messages. This becomes worse as the

number of channels grows. The example shows only one type of sensor being used (MIMU) when in reality there would be many more involved. Also, the entire two round exchange must take place before every execution of the flight control process within the GN&C partition. Since the control process may be executing at 25 Hz or 50 Hz, the described FTSS input processing (the two round exchange) may also be executing at this rate. This is why CCDL performance is so critical in an NMR system that uses data exchange and voting for fault detection.

3.3.2 Application Output Processing by FTSS

This section describes at a high level how the outputs from application partitions which are replicated on all three P-NMR channels are voted before being sent to the simulated actuators. The distributed voting is how FTSS can identify a channel with a fault. Not all application outputs go through FTSS and the method for selecting the outputs at design time using XML system configuration files is discussed later.

Since the GN&C application replicated on each channel should compute the same output (the actuator command), only a one round exchange over the CCDLs is utilized instead of a two round exchange as is the case with input processing. The one round exchange is managed by the FTSS output task executing on each channel. Figure 3.3-4 shows this fault tolerant application output processing. This is a simplified example showing only one application output. In reality, many outputs would be voted, possibly from different application partitions.

The scenario starts with the GN&C application on each channel giving its value to the FTSS output task. For the purposes of illustration, the CH1 application has experienced a fault and so its value is not the same as CH2 and CH3. The exchange then takes place so that at the end of round one, the FTSS output task on each channel has the value computed by all three applications. The FTSS output task then passes the three values to the FTSS voter which performs the bit-for-bit majority vote. The voter masks out the erroneous (minority) value from CH1. The voted value on each channel would then be passed to an I/O Processing partition which would manage the transmission over

the data bus to the actuator (e.g. the RCS thruster). The voter also records that CH1 has experienced a fault.¹

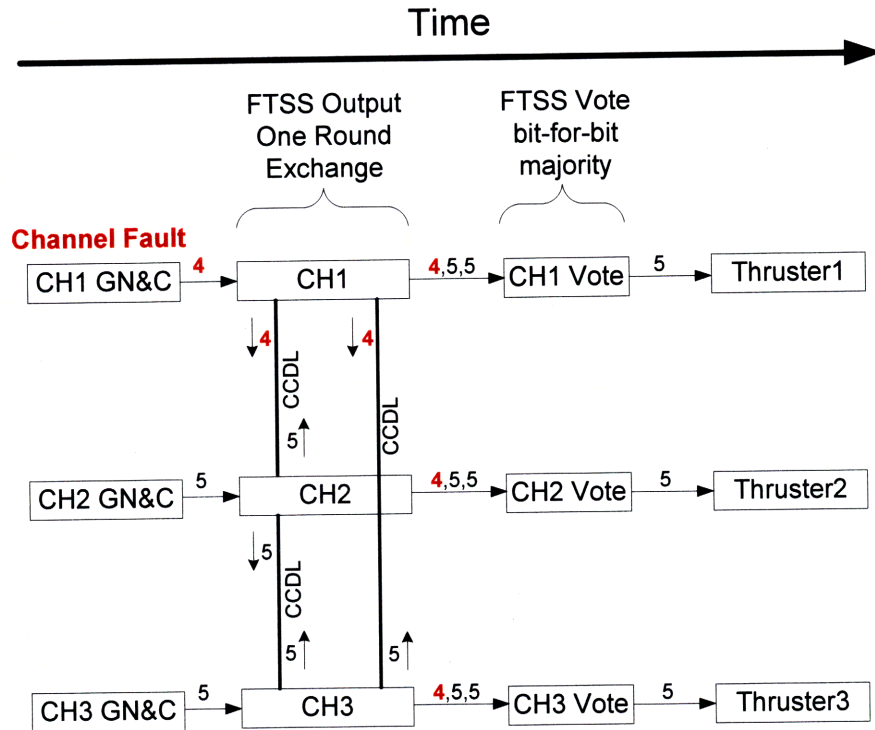


Figure 3.3-4 Application Output Processing by FTSS

As with the FTSS input processing, the FTSS output design is flexible in that it can perform a two round exchange instead of the one round exchange described here. Also, the number of redundant actuators does not have to match the number of computing channels. These settings can be changed by modifying the XML system configuration files.

3.4 Channel Hardware

The P-NMR system that was used for obtaining the results in this thesis is a low cost avionics prototype used for risk reduction in a lab environment. As such, the individual COTS hardware components used in the prototype are commercial grade and

¹ The group membership functionality that would normally use this information is not currently implemented in FTSS.

are not ruggedized for the space environment. In addition, there is no electrical isolation used for certain key elements such as the CCDL. Thus, if using the NASA Technology Readiness Level (TRL) scale [48], the P-NMR prototype would be between TRL 3 and 4. The intent is that once the basic design has been fully validated and performance metrics collected using the prototype then more expensive COTS hardware components can be acquired for testing in a space environment (prototype testing was still in progress at the time of this writing). These COTS components do not have to be radiation hardened since redundancy will be used to mitigate the effects of SEEs. The overall system performance metrics collected using the low cost commercial grade hardware components are reasonably accurate since the ruggedized COTS components with electrical isolation will have similar performance.

The P-NMR system prototype consists of three channels or FCRs and each channel consists of a single Wind River SBC750GX single board computer (SBC). This is a 6U CompactPCI board with a single PowerPC 750GX microprocessor. Normally, a single SBC would not be considered a channel. In IMA terms, the SBC would represent a single CPM that would reside in one slot in an IMA cabinet, as described in section 2.5.2. In this case, the cabinet would be a chassis with a CompactPCI backplane with several slots for processor boards similar to the SBC750GX and for various I/O boards. The channel or fault containment region would be the entire CompactPCI chassis with all the individual boards. However, the SBC750GX can be configured as a self contained SBC with its own power supply and so the P-NMR prototype did not use a CompactPCI chassis with multiple processing and I/O boards. Once the P-NMR architecture is validated using the current prototype then four CompactPCI chassis with several slots each will be acquired.

The choice of using the SBC750GX over other processor boards for the prototype was made for several reasons. First, the PowerPC processor family is a popular choice for radiation hardened space applications due to its excellent performance to power consumption ratio. Second, the CompactPCI backplane bus is being used for space applications. Finally, the board comes with a board support package (BSP) for the selected COTS ARINC 653 RTOS and a well documented software development environment.

3.4.1 SBC750GX Single Board Computer

The SBC750GX contains one PowerPC 750GX RISC microprocessor that runs at 933 MHz. The core OS, FTSS software and the user partitions all execute on this single processor. The other main features of the SBC750GX board include:

- Marvell Discovery II MV64360 system controller
- system bus running at 133Mhz
- 512MB of DDR266 SDRAM
- 64MB of flash memory
- three Gigabit Ethernet ports with RJ45 connectors
- three Gigabit Ethernet PHY devices (Broadcom 5461SA)
- CompactPCI interface via a Tundra Tsi310 PCIX P2P bridge
- one set of PMC connectors

A block diagram of the board's major components is shown in Figure 3.4-1.

3.4.1.1 PowerPC 750GX

For the P-NMR project, the PowerPC 750GX microprocessor operates at 933 MHz. There are two facilities of the PowerPC architecture that are particularly relevant to the P-NMR design. First, the processor has a MMU that an ARINC 653 compliant RTOS uses to ensure space partitioning: one user partition cannot corrupt the memory of another user partition and user partitions cannot corrupt the memory of the RTOS kernel. Second, the PowerPC has a decrementer register (DEC) which allows an RTOS to cause an exception after a programmable delay. An ARINC 653 compliant RTOS can use this hardware facility to ensure time partitioning: one partition cannot monopolize the processor causing a partition schedule violation. It should be noted that both of these PowerPC facilities are only available in supervisor mode. The significance of this will be explained in section 4.1.

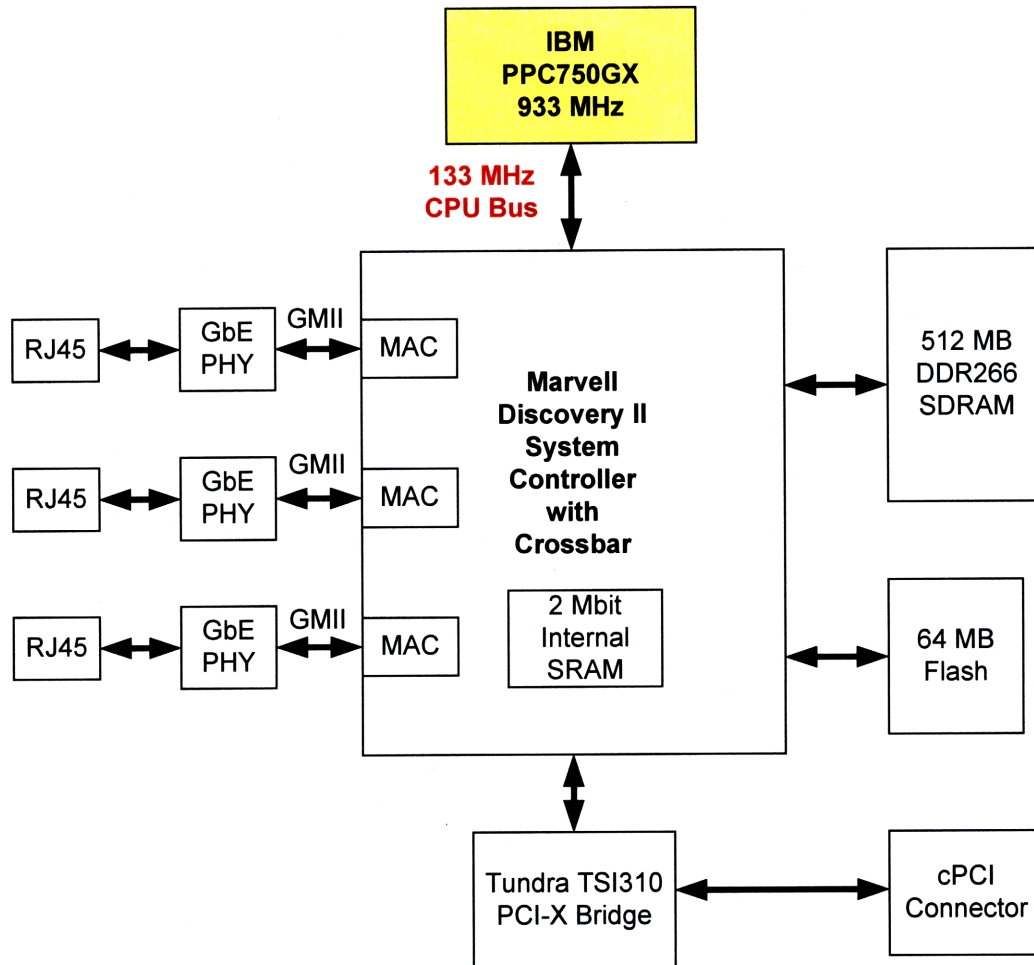


Figure 3.4-1 SBC750GX Board Components

3.4.1.2 Marvell MV64360 System Controller

The Marvell MV64360 system controller provides a single-chip, high performance solution for PowerPC based applications. It interfaces to the CPU bus, DDR SDRAM, flash memory and the Ethernet transceivers. It contains 2Mbit of internal, general purpose SRAM. Each of the interfaces uses dedicated read and write buffers. All of the interfaces are connected through a crossbar fabric. The crossbar enables concurrent transactions between devices. For example, the crossbar can simultaneously control:

- an Ethernet hardware fetching a direct memory access (DMA) descriptor from the internal/integrated SRAM,
- the CPU reading data from DDR SDRAM,
- a DMA moving data from flash to the PCI bus.

The MV64360 operates at the maximum 133 MHZ which allows the crossbar fabric to transfer up to 100 Gbps in aggregate throughput. The chip also fully supports PowerPC cache coherency by using snoop transactions.

The MV64360 chip has three Ethernet ports and each port has its own Media Access Control (MAC) logic. Each port supports 10/100 Mbps with MII or 1 Gbps with GMII. The 1 Gbps interface is termed Gigabit Ethernet (GbE). Receive and transmit buffer management is based on a buffer-descriptor linked list. Descriptors and data transfers are performed by port dedicated serial DMA. Each port has eight receive queues and eight transmit queues.

3.4.2 GbE CCDL

This section provides a brief overview of the Gigabit Ethernet hardware used for the CCDL. More details on the hardware and device driver can be found in Appendix C.

For the CCDL, each channel is connected to the other two channels in the TMR setup using two dedicated 1000BASE-T Category 6 Gigabit Ethernet copper cables. Each cable is two feet long with RJ-45 connectors on each end. Inside each 1000Base-T cable are four 100BASE-TX Category 5 shielded twisted pairs. No electrical isolation is present in the current prototype. Each connection is a point-to-point, full duplex, Gigabit Ethernet link. Gigabit Ethernet is defined in IEEE Standard 802.3 [49]. The Broadcom 5461SA device is used for the Ethernet PHY at each end of the cable (each SBC750GX has three such PHY devices). Each device has its own 25Mhz oscillator input. The PHYs connect to the Marvell MV64360 chip which contains a MAC function for each port. The MAC hardware implements the standard IEEE 802.3 CRC checksum which is used for error detection.

An Ethernet switch is not used for connecting the three channels since each CCDL is a dedicated point-to-point link. A switch is only used to connect the three channels to the PC used for development and testing. Since each CCDL is a point-to-point, full duplex connection, there can be no collisions or transmission delays due to two PHYs attempting to send data at the same time. Thus, the non-deterministic delays of a normal CSMA/CD (carrier sense, multiple access, collision detection) Ethernet network are avoided.

It should be noted that the MAC functions in the Marvell chip and the Broadcom PHYs are standard COTS devices that support the IEEE 802.3 Ethernet standard. As such, they do not implement any redundancy management functions such as clock synchronization, transmission schedule timing, group membership or protection from a “babbling node”. These are implemented by the FTSS software running on the PowerPC.

Using COTS Gigabit Ethernet for the CCDL has many advantages besides pure performance. The main advantage is the dominance of the technology for almost all commercial and industrial networking applications. Ethernet is a cheap, well tested and mature solution with many vendors and is being embraced by many aircraft avionics manufacturers and some space system designers [11, 12].

Chapter 4

P-NMR Software Design

This chapter provides a detailed description of the software that is executing on each P-NMR channel. Section 4.1 provides an overview of all the software including the COTS ARINC 653 RTOS, the FTSS component and the user application partitions. Section 4.2 provides an overview of the FTSS component executing in the core OS. The rest of the sections in the chapter describe detailed aspects of FTSS and how they interact with the ARINC 653 RTOS. Section 4.3 describes the Gigabit Ethernet CCDL device driver, section 4.4 discusses frame synchronization, section 4.5 presents the design for scheduling FTSS I/O processing, 4.6 explains how ARINC 653 communication ports are used and section 4.7 briefly describes the voting algorithm used by FTSS.

4.1 Channel Software Overview

Each channel in the P-NMR system is comprised of one core processing module, namely the COTS SBC750GX board which has one PowerPC 750GX processor. This core processor runs a COTS RTOS, the Wind River Systems VxWorks 653 product, which provides for the robust partitioning of user applications. VxWorks 653 is compliant with the ARINC 653 standard and it provides the APEX API which is used by user application partitions to call OS services. In addition, Wind River can provide a DO-178B Level A certification package. The FTSS component is software produced in-house by Draper Laboratory for the P-NMR project. Figure 4.1-1 shows the software layers that are present on each core processing module (i.e. on each SBC750GX single board computer).

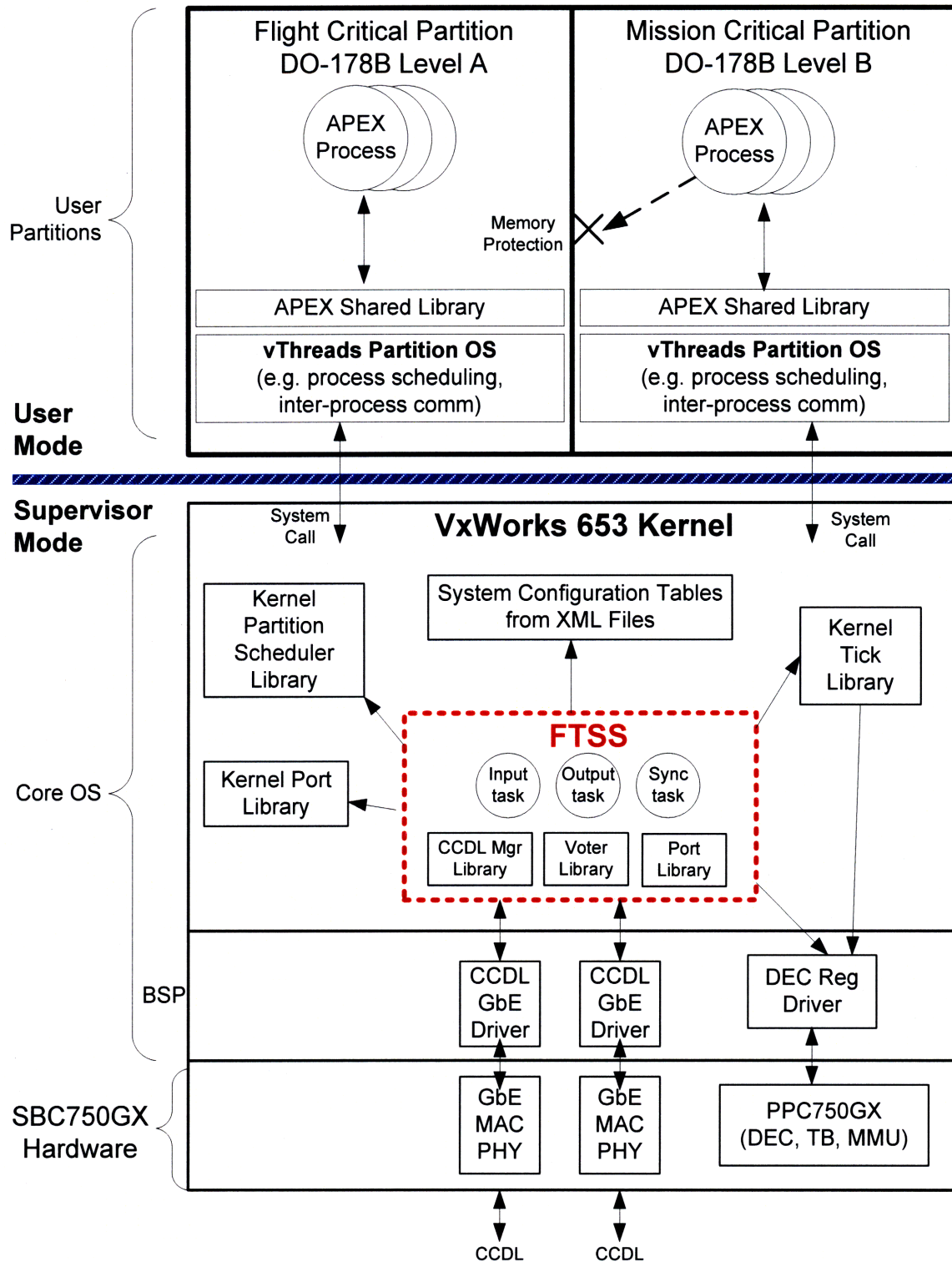


Figure 4.1-1 Channel Software Overview

As indicated in the figure, the PowerPC processor architecture has two levels of privilege: *user mode* and *supervisor mode*. These two levels of privilege govern the access to various registers on the processor. User mode is used by partition applications and it can also be used by the core OS. In this mode, access is granted to the 32 general purpose registers (GPRs), the 32 floating point registers (FPRs) and other standard PowerPC registers (branch etc). Supervisor mode, on the other hand, is used exclusively by the core OS. It is impossible for user partition code to execute in supervisor mode. In this mode, access is granted to the PowerPC special purpose registers (SPRs) that control functions such as memory management, exception handling and time keeping (i.e. the decremter register). This division of privilege allows the core OS to control what a partition application can access which in turn prevents a single application from corrupting the entire module. As a general rule of thumb, access to hardware resources on the SBC750GX board is accomplished by the core OS in supervisor mode. The rest of this section provides a brief overview of each software layer, starting at the upper most layer and then proceeding down to the SBC750GX hardware.

4.1.1 Application Partitions and the APEX API

The upper most software layer is occupied by the *user partitions*. As discussed in section 2.5.3, a partition can be thought of as a protective container for user application code to execute in. Normally, each partition contains one application such as GN&C. To the application developers, it appears as if their application is the only one executing on the processor. The VxWorks 653 RTOS allows applications executing on the same module to be written in different programming languages (C++, Ada etc). A partition application can be comprised of several APEX processes that implement the required functionality. These application processes can have different periods and priorities and they only run during the time slices given to their container partition by the core OS. The management of these processes and other OS services are accessed by calling routines in the VxWorks 653 APEX shared library.

The *APEX shared library* (SL) sits underneath the user applications. It is part of the COTS VxWorks 653 product. Whenever application code needs to make an OS call, it uses the APEX API which follows the ARINC 653 standard. As section 2.5.3 mentioned,

this promotes portability, reusability and modularity. The services provided by APEX include memory management, partition management, process management, intra-partition communication and inter-partition communication (which may go outside the module using some I/O device). The APEX SL sends all user requests down to the VxWorks 653 partition level OS called vThreads. The APEX SL provides a standardized wrapper around native Wind River vThreads services.

4.1.2 vThreads: The Partition OS

vThreads is the partition OS and is part of the COTS VxWorks 653 product. It has a kernel and it provides for the internal management of a single partition. The vThreads kernel performs such services as scheduling the user processes executing within a partition as well as inter-process communication within a partition. It should be noted that vThreads is not the module level core OS. The core OS does not have visibility into a partition and thus cannot see individual application processes. A vThreads kernel does not manage all the different partitions. It only manages things inside one single partition. It does not have any special privileges like the core OS kernel – vThreads tasks run in user mode just like user application processes. vThreads can be considered as a set of tasks, a system shared library (SSL) and a section of partition memory for objects created by vThreads tasks. Only the vThreads read-only code segment in the SSL can be shared amongst different partitions. The vThreads kernel tasks and the objects created by these tasks are unique to each and every partition. In other words, each partition has its own instance of a partition level OS.

When vThreads receives a service request from a user application process, it decides if the request can be handled internally by itself or if the request needs to be passed down to the core OS via a *system call*. Since vThreads executes in user mode just like the user application, sending a request down to the core OS is expensive since the core OS has its own separate memory domain and it executes in supervisor mode. Thus, Wind River has attempted to maximize the number of services (e.g. getting the current time) that are implemented internally by vThreads, minimizing the number of times that the thread of control needs to go from user mode in a partition to supervisor mode in the core OS. Finally, most system calls going to the core OS kernel are handled

synchronously – the requested core OS service executes immediately and then control returns to vThreads and then the user application process.

Even though vThreads tasks represent OS tasks, they are still at the partition level and thus only run during a partition's scheduled window, just like the application processes. vThreads does not receive hardware interrupts and so does not directly receive clock tick interrupts marking the passage of time. Instead, vThreads is made aware of the passage of time through *pseudo-interrupts* which are delivered to vThreads by the core OS only during the associated partition's schedule windows. Pseudo-interrupts are similar to software signals. As a performance optimization, multiple clock ticks are announced to vThreads in one pseudo-interrupt.

4.1.3 The core OS, FTSS and the BSP

Under all the partitions is the VxWorks 653 *core OS* which is the *module level OS* responsible for managing the module hardware resources as well as all the user partitions (including the partition OS). The core OS is also a partition in the sense that it has its own protected memory region which is separate from all the user partitions. It consists of

- *a kernel,*
- *a BSP,*
- *and any user defined code.*

The core OS kernel is completely separate and distinct from the vThreads kernel (i.e. the partition OS) in each partition. Any service request made by a partition application that cannot be handled internally by vThreads is passed down to the core OS. These requests include partition management (e.g. start, stop), inter-partition communication using APEX ports and access to module hardware such as timers and I/O devices. The core OS also monitors the health of each partition and the health of the entire module.

The core OS is responsible for enforcing robust partitioning of the different applications. This functionality is compliant with the ARINC 653 standard. Time partitioning is accomplished using time preemptive scheduling (TPS). The core OS makes use of the PowerPC DEC register in order to guarantee that one partition cannot monopolize the processor. Space partitioning is accomplished by setting up memory

regions for the core OS and for each partition. The core OS uses the PowerPC MMU to protect these regions; one partition cannot perform an illegal write in another partitions region or in the core OS region. It is possible to setup shared memory regions between partitions or between the core OS and one or more partitions. For example, these shared regions are used for the code in shared libraries.

The FTSS software is the “user defined code” that is included in the core OS memory region and runs in supervisor mode just like the core OS kernel code. It is not part of the COTS VxWorks 653 product and was created by Draper Laboratory to support the P-NMR research project. FTSS interacts with core OS kernel services and the BSP in order to provide the fault tolerance and redundancy management functionality. It also manages the CCDLs between channels.

Finally, the *board support package* software layer is located directly above the hardware layer. It is also considered part of the module core OS and provides the libraries required to support the core OS kernel on a specific hardware platform. For example, there is a specific BSP for the SBC750GX board supplied by Wind River. BSP libraries provide an identical software interface to different hardware boards. A BSP may include code for controlling things such as the processor MMU and cache, hardware clocks and timers, I/O devices and interrupts. FTSS uses some BSP services relating to the Gigabit Ethernet I/O driver. Since the BSP is specific to a particular hardware platform, it may be supplied by the hardware vendor and not the RTOS vendor. It also may or may not include a DO-178B certification package. In any event, the BSP needs to be considered along with the core OS kernel when certifying the avionics system.

4.1.4 XML System Configuration Files

It should be noted that the ARINC 653 standard suggests the use of XML files as a platform independent way to configure the core OS and user application partitions on a module. These files would normally be edited by the system integrator. The Wind River VxWorks 653 product has adopted this approach and has DO-178B certifiable XML tools that are used at build time to analyze and verify the system configuration tables specified in the XML files. Once verified, the XML files are converted into a binary file format containing the configuration tables and this data is then used by the VxWorks 653 core

OS and each partition OS (vThreads). The XML system configuration files include information about the hardware resources on the module, the size of memory segments for the core OS and the partitions, the partition schedules, the ports used for inter-partition communication and module/partition health monitoring rules. As will be explained, the FTSS component makes use of the existing data in these XML files but new data has also been added that is specific to fault tolerance.

4.2 FTSS Overview

The FTSS software component was written in the C programming language and was developed by Draper Laboratory for the P-NMR project. It currently performs the following fault tolerance and redundancy management functions:

- CCDL communication using point-to-point Gigabit Ethernet,
- frame synchronization,
- source input congruency (i.e. input exchange and voting)
- application output voting (i.e. computer fault masking).

It was written in-house because although the ARINC 653 standard mentions fault tolerance and voting in passing, it does not specify any details on how an RTOS should support these services and thus they are not present in any COTS RTOS products.

The FTSS software component will replace the functionality performed by the proprietary Network Element hardware in the Draper X-38 FTTP design. However, the version of FTSS used for this thesis was missing some functionality present in the NE hardware. For example, it does not include group membership services. FTSS can identify a faulty channel using voting but it does not remove the channel from future voting or other redundancy management activities. FTSS also does not reset a faulty partition or a faulty channel and it does not re-integrate a reset channel back into the operational group. Finally, as will be discussed, initial system cold start and synchronization is rather simplistic and needs to be made more robust. These functions will be added in future FTSS builds.

4.2.1 Design Goals

A major design goal for FTSS was that its operation be as transparent as possible to application developers. They should not have to know any details regarding the FTSS design or the hardware. The developers should still see the system as a simplex computer executing only one instance of their application. They should not need to know how the avionics hardware elements are physically connected. Thus, FTSS should not require that application developers use a custom FTSS API in order to take advantage of FTSS services. They should continue to be able to use the industry accepted APEX API. This leads to software that is portable, reusable and modular which facilitates enhancements and maintenance. This is crucial for a large project such as Constellation which could be active for several decades.

Also, it should be possible to change, add or remove FTSS from the system without having to make major changes to the existing COTS core OS kernel code. The goal was to design FTSS as a core OS kernel component that could be added to or removed from the kernel using existing build script tools. Few core OS kernel modifications are required when adding or removing FTSS. This is important if the COTS RTOS has a DO-178B Level A certification package. Conversely, an effort was also made to make the FTSS C code as portable as possible. This meant minimizing the use of VxWorks 653 kernel calls and maximizing the use of ANSI C.

Figure 4.2-1 shows a simplified example of how applications are isolated from the FTSS design. The figure depicts a GN&C partition sending its output data (the value 5) to an I/O Processing partition on the same module using ARINC 653 ports. The I/O partition will send the GN&C output over a data bus to an actuator. The output data from the GN&C partition is flight critical and requires voting before it is provided to the I/O Processing partition. Regardless of whether or not FTSS services are required, applications always use the same APEX Port API for inter-partition communication. The APEX SL and vThreads partition OS (not shown) then deliver the application data from the GN&C partition down to the core OS. Once in the core OS, the port data is managed by the VxWorks 653 kernel port library. This kernel library contains the core OS memory for storing the application data and it also has a standard interface for use by other code

in the core OS. The port library interface provides such functions as Port_Get and Port_Put.

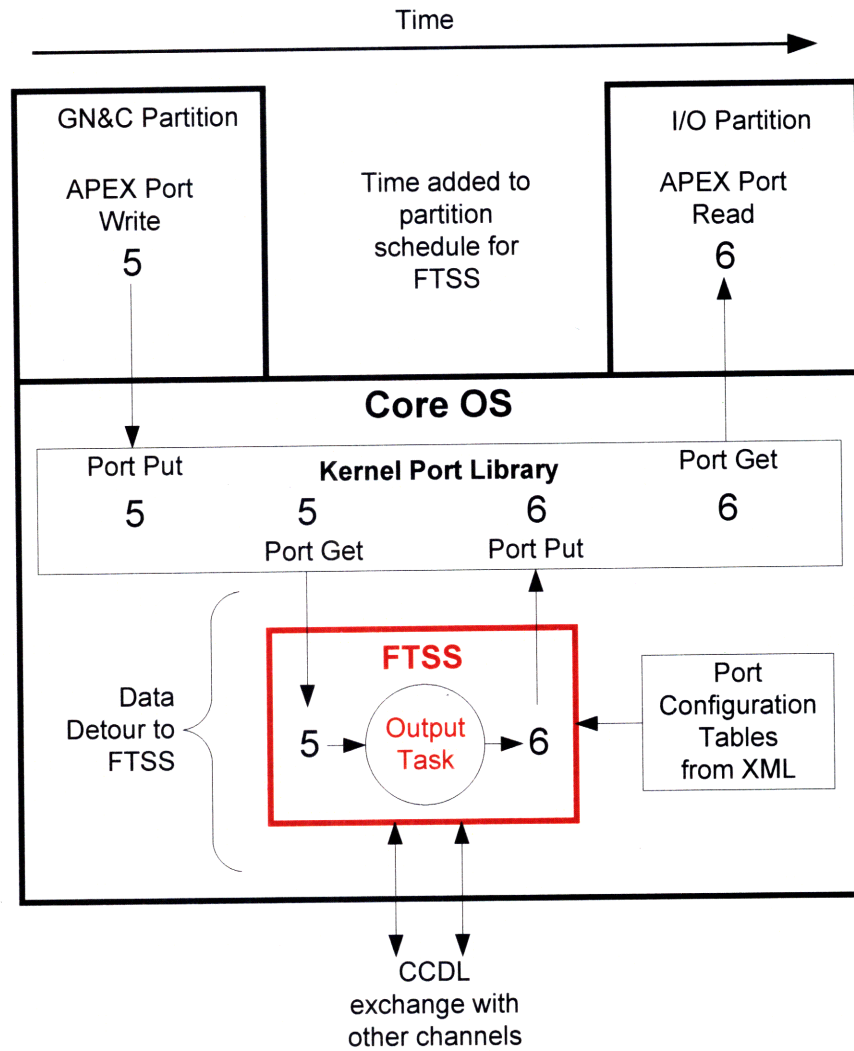


Figure 4.2-1 Application Output Detour to FTSS

When it is time for FTSS to execute in the schedule, it will use the kernel library service Port_Get to obtain the original data from the library and then it will perform any fault tolerance functions using the CCDLs. The figure shows the value changing from 5 to 6 due to the FTSS output exchange and vote. FTSS then uses the kernel library service Port_Put to replace the old application data (value 5) in the kernel memory with the new voted data (value 6). Then, when it is time for the I/O Processing partition to execute in

the schedule, it will call the standard APEX service to read the port. This will result in a system call down to the core OS where the kernel port library will return the voted data.

The detour (or redirection) to the FTSS component in the core OS is controlled by the XML port configuration files. By editing the XML files, the system designer can control when data is rerouted to FTSS before arriving at its normal destination port in a user partition. The application code does not need to change since it uses the standard APEX port API. Some of the XML modifications are specific to the P-NMR project and are thus not presently in the ARINC 653 standard. However, these modifications are used by FTSS code and not the kernel code and thus the modifications do not require that core OS kernel code change. In fact, no changes are required to the existing COTS core OS kernel code that handles ports (e.g. Port_Get, Port_Put). The detour to FTSS can be removed by simply removing the FTSS execution window from the partition schedule. No kernel code changes are required when adding or removing FTSS.

4.2.2 Run-Time Implementation

The FTSS functionality is implemented as three high priority core OS tasks that execute in supervisor mode on the PowerPC in each channel. The three tasks are part of the core OS. The tasks interact with the COTS VxWorks 653 core OS kernel and the SBC750GX BSP. Identical FTSS code executes on each of the three channels but each instance uses different sections of a static system configuration table which is loaded at startup. The following is a brief description of each FTSS task.

ftss_sync task

Executes once every major frame on all three channels. Responsible for executing the distributed, fault tolerant frame synchronization algorithm. All three channels attempt to “agree” on a global time. Uses the CCDL to send synchronization information to the other two channels. Does not interact with user partitions.

ftss_input task

Executes whenever input congruency is required during the major frame. Performs input exchange and voting so that all inputs are available on all channels. Uses the CCDL to distribute individual inputs to all channels. Uses ARINC 653 ports and the port configuration tables from the XML files to prepare inputs for user applications.

ftss_output task

Executes whenever application output voting (i.e. fault masking) is required during the major frame. Uses the CCDL to distribute application outputs to other channels for voting. Uses ARINC 653 ports and the port configuration tables from the XML files to obtain outputs from user applications.

4.2.3 Scheduling core OS FTSS Tasks

Normally, core OS tasks are not included in an ARINC 653 partition schedule. Only user partitions are scheduled. However, the ARINC 653 standard does allow “system partitions” to be placed in a core processing module schedule. System partitions are defined as:

“...partitions that require interfaces outside the scope of APEX services, yet constrained by robust spatial and temporal partitioning. These partitions may perform functions such as managing communication from hardware devices or fault management schemes. System partitions are optional and are specific to the core module implementation.”

The VxWorks 653 RTOS product provides system partitions and the P-NMR design makes use of them in order to schedule core OS FTSS tasks. However, the P-NMR use of system partitions is slightly different than the ARINC 653 definition. Three following system partitions are created:

FTSS_SYNC partition – controls when ftss_sync task executes

FTSS_INPUT partition – controls when ftss_input task executes

FTSS_OUTPUT partition – controls when ftss_output task executes

However, these FTSS system partitions have no application code that executes in user mode. They are essentially empty partitions but they can still be placed in the core module schedule. A separate facility that the VxWorks 653 product provides is the ability to *assign* a core OS task to a specific partition in the *time domain*. In other words, a core OS task assigned to a partition will only run during that partitions time window and it will not run during a core OS time window. Using this assignment method, the three different FTSS tasks in the core OS are inserted into the module schedule using the three FTSS system partitions. This scheduling is discussed further in subsequent sections.

4.2.4 Rationale for placing FTSS in the core OS

The decision to place FTSS in the core OS domain instead of a user-level partition is a significant one that merits some initial discussion. The main reason for the decision was performance. At the start of the P-NMR project, there were concerns that a software implementation of the X-38 FTTP Network Element functionality running on a COTS general purpose processor would be too slow. The FTSS overhead during execution of flight control software had to be kept to an absolute minimum to reduce the overall system response time. In addition to this, FTSS would need frequent access to the module hardware (e.g. CCDL, PowerPC supervisor registers) and to core OS kernel services which only execute in supervisor mode. For these reasons, it was decided at an early stage to place FTSS in the core OS instead of in a user-level partition.

This decision has an impact on robust partitioning since FTSS shares the same memory region as the kernel and it has the same privileges as the kernel. Thus, it is possible for an FTSS fault to corrupt the entire module. Placing FTSS in the core OS also has serious implications for system certification costs. This is because any modification to the core OS can invalidate portions of the DO-178B certification package provided by the RTOS vendor. Subsequent sections of this chapter and the Conclusions chapter elaborate further on these issues.

The rest of this chapter describes the detailed design of each FTSS function.

4.3 GbE CCDL Device Driver

This section provides an overview of the GbE CCDL device driver that is part of FTSS. More details on the driver can be found in Appendix C.

Many aspects of the GbE driver have been optimized for performance. This is because the CCDL is used heavily by all FTSS functions and is a major contributor to overall system response times (e.g. the time between sensor reading and actuator activation). Large or variable communication latencies also have a negative impact on distributed clock synchronization algorithms.

The three channels communicate with each other using IEEE Standard 802.3 Gigabit Ethernet CCDLs. There is a dedicated, 1000 Mbps, point-to-point, full-duplex link between every pair of channels. Each channel has two Ethernet ports that go to the other two channels. A hardware Ethernet switch is not used. There is a separate device driver for each hardware port on the SBC750GX board. For performance reasons, the device driver uses raw Ethernet MAC frames; the IP, UDP or TCP protocols are not used. The standard IEEE-Std-802.3 MAC CRC checksum is used for error detection. The driver executes in the core OS and operates in polled mode, not interrupt mode.

A significant portion of the CCDL I/O driver is comprised of SBC750GX BSP code that has been modified to improve performance. This BSP code is highly specific to the Marvell system controller chip which implements a dedicated MAC and serial DMA for each Ethernet port. The code performs the memory mapping of the Marvell hardware registers to the core OS memory space. It also controls when data is moved between SDRAM and the PowerPC cache. Above the modified BSP code is FTSS CCDL code that performs higher level connection management services. This FTSS CCDL code interacts with other FTSS functions which interact with core OS kernel services.

4.3.1 Core OS Driver Model

An important decision for the overall FTSS software design is whether to place the CCDL Ethernet device driver in the core OS or to place it in a partition. It is important because the rest of the higher level FTSS functions will tend to be collocated with the CCDL driver in order to increase performance. For example, if the CCDL driver is in the

core OS but the other FTSS functions (frame synchronization, voting etc) are in a partition instead of the core OS, there will be additional PowerPC mode switches and byte copies as data moves between the FTSS partition and the CCDL driver in the core OS. It is more efficient if the entire FTSS component (including the driver) is in the same memory domain; either all in the core OS or all in a partition. Thus, the location of the CCDL device driver heavily influences the location of the entire FTSS component.

The P-NMR prototype uses the core OS device driver architecture. Thus, the entire FTSS component is also in the core OS. Figure 4.1-1 depicts a simplified view of this design. FTSS uses the SBC750GX BSP code to memory map the Marvell Ethernet DMA registers to the core OS memory region where FTSS code has access to them. The driver code executes in supervisor mode like other kernel code.

The core OS approach was chosen because of the performance benefits. User applications like GN&C use APEX ports to move data down into the core OS memory region. This is one mode switch (user to supervisor) and one byte copy. Once in the core OS memory, FTSS uses kernel services to gain access to the data and move the data to the Ethernet driver DMA buffers. This is another byte copy. Thus, only one mode change and two byte copies are required to move data from a user partition to the CCDL Ethernet driver. This is a more efficient design than placing the CCDL device driver and the rest of FTSS in a partition (please see Appendix C for more details).

The main disadvantage of the core OS device driver approach is weak partitioning. The driver executes in supervisor mode and shares the same memory region as the kernel. It can thus corrupt the kernel causing a module reset. More details on the core OS device driver architecture as well as an alternative partition driver design can be found in Appendix C.

4.4 Frame Synchronization

The P-NMR system uses a frame synchronous design and thus it requires that individual channels agree on what time it is. This section describes the implementation of the Byzantine fault tolerant, distributed frame synchronization algorithm that allows the major and minor schedule frames to be aligned on each channel. This section also assesses how this frame synchronization affects ARINC 653 partition and process time

services. The frame synchronization algorithm has two parts, initial system cold start and steady state, both of which are described. However, the introduction of a new channel into an operational set of channels that are already synchronized is not currently implemented.

4.4.1 The ARINC 653 Partition Schedule

A major component of the ARINC 653 robust time partitioning concept is the partition schedule. Individual partitions are scheduled on a fixed, cyclic basis. The *major frame* is a fixed duration of time and it contains a fixed sequence of all the individual partition time windows. A partition window within the major frame is defined by its offset from the start of the major frame and its duration. The offset is calculated by simply summing the durations of previous windows in the major frame. All partitions that need to execute must have at least one window within the major time frame. Each window can be any duration; window durations do not need to be multiples of each other. However, windows must be multiples of the kernel tick time (described in 4.4.2) and the smallest duration for a partition window is one kernel tick.

The total duration of the major frame must be a multiple of the longest partition period. A particular partition can occur more than once during the major frame if it has a smaller period. For example, if the major frame is one second long then a 50 Hz partition will have 50 windows in the major frame. The smallest partition period is sometimes referred to as the minor frame. It is also permissible to have idle time in the major frame (partition ID = 0). Idle time can be used by core OS or background tasks.

Figure 4.4-1 shows a simple partition schedule. Partition A operates at 3 Hz and so its period is 333 ms. Partition B operates at 2 Hz and so its period is 500 ms. Partition C operates at 1 Hz and so its period is 1 second. Since partition C has the longest period, its duration is chosen for the major frame. Since partition A has the shortest period, its duration is chosen for the minor frame. There is also some spare time in the first and second minor frames that can be used by background tasks. In this example, partition A is always the first partition to execute in each minor frame.

The partition sequence within the major frame is executed repeatedly by the core OS kernel since it contains time windows for all partitions that need to run. Finally, the

partition windows that make up the major frame are defined in the XML system configuration files which are automatically verified by a set of tools at build time. Appendix B contains an example of an XML schedule file.

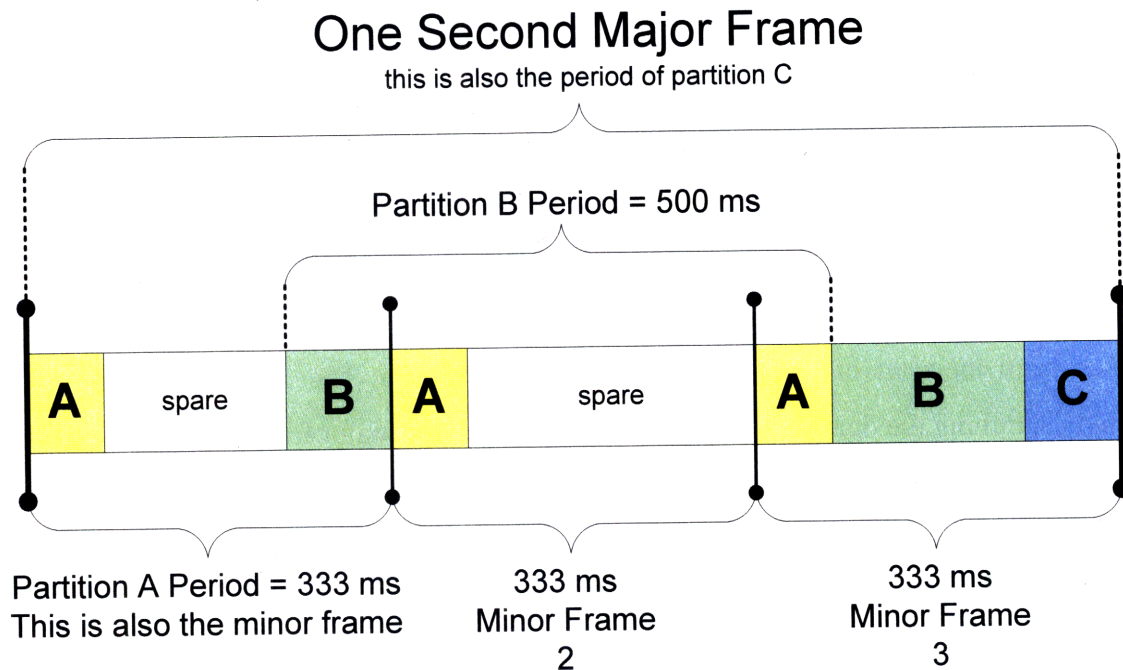


Figure 4.4-1 ARINC 653 Partition Schedule

4.4.1.1 Multiple Module Schedules

One more aspect of ARINC 653 partition scheduling that should be discussed is the notion of having more than one partition schedule defined for a module. Multiple partition schedules are defined in Part 2 of the ARINC 653 standard and the VxWorks 653 product supports this concept. The core OS will use the first schedule (schedule ID = 0) from the XML configuration data after the module powers up. It is up to the system designer to decide what schedule is going to be the first schedule. A core OS service is also provided to switch to a different schedule during normal operation. Three options are provided for when to change to the new schedule after calling the switch service:

- at the end of the current kernel tick,
- at the end of the current partition's window in the current major frame,
- at the end of the current major frame.

As will be discussed, multiple schedules are useful for accomplishing system cold start.

4.4.2 PowerPC DEC Register and Core OS Kernel Ticks

Kernel ticks allow the core OS to keep track of the passage of time on the module. They are at the heart of all scheduling facilities and robust time partitioning provided by the VxWorks 653 core OS kernel. Nearly all core OS and partition OS services that deal with time, such as watchdog timers, periodic waits and task delays, use kernel ticks. Thus, they are also a key element used for implementing the FTSS frame synchronization function.

A kernel tick is a hardware interrupt provided by the PowerPC DEC (decrementer) register. This is a supervisor level, 32 bit register and thus it is only available to core OS code, not user partition code. The operation of the DEC register is quite simple. Supervisor code executing on the PowerPC can load a value into the DEC at any time. The register then starts to decrement this value at a frequency derived from the system clock (e.g. 100 MHz or 133 MHz). The value in the register will go down to zero and then become negative. When it goes from zero to negative, a PowerPC decrementer interrupt is generated.

Note that the value placed in the DEC register is not equal to the desired number of microseconds or milliseconds per tick. For example, if each tick should be 250 μ s apart then a value of 250 is not placed in the DEC register. Instead, the value stored is calculated using a multiple of the system clock frequency.

The core OS kernel installs an interrupt handler for the DEC that is the start of all kernel time keeping and partition scheduling functions. The first operation that this handler does is reload the DEC register with the same value so that the next interrupt will occur one tick duration in the future. The core OS kernel then evaluates the fixed, cyclic partition schedule to determine if the partition that was executing should retain the use of the processor or if it has reached the end of its time window in the schedule. The core OS kernel will also deliver the tick to the current partition OS (vThreads) using a pseudo interrupt so that the partition OS can perform the APEX process scheduling within the partition.

In general, the decremter interrupt occurs regardless of what code (kernel, user partition) is currently executing on the PowerPC. The interrupt is maskable and so there are a few instances of kernel code that lock out this interrupt but these sections of code are extremely small and fast and thus the probability of them delaying the handling of the interrupt is small.

The value chosen to be loaded into the DEC register after each interrupt determines how many kernel tick interrupts there will be. The VxWorks 653 product specifies that there can be anywhere from 3 ticks per second (each tick is 333.33 ms) up to 5000 ticks per second (each tick is 200 μ s). The number that is chosen depends on several factors such as the duration of the smallest partition window in the fixed module schedule, the duration of the smallest application process capacity, the duration of the smallest watchdog timer or task delay, and the overhead associated with processing the DEC interrupt.

For example, take the case where the smallest partition window is 100 ms and the smallest APEX process duration is also 100 ms. Assume the selected kernel tick rate is 5000 ticks per second (i.e. each tick is 200 μ s). While the 100 ms APEX process is executing in its partition, the PowerPC will be interrupted 500 times by the DEC register. During each kernel tick interrupt, the core OS kernel will determine that the partition that was executing should continue to execute. The core OS will then deliver the tick to the partition OS and the partition OS will also determine that the process that was executing is still the highest priority process that is ready to execute and so it should keep executing. Basically, the kernel tick interrupt does nothing except verify that what was running should keep running. Thus, 5000 kernel ticks per second is probably overkill. For reasons that will be made clear in subsequent subsections, the P-NMR design uses 4000 ticks per second which equates to 250 μ s between tick interrupts.

4.4.3 PowerPC Time Base Register

One complication to the kernel tick interrupt is that when an integer value is used to load the DEC, the value will not equal the exact desired duration since the fractional portion is lost. The integer value will result in a duration close to the desired time but if there are many interrupts per second, the error will accumulate. For example, if there are

supposed to be 4000 ticks per second then each interrupt should take place exactly 250 μ s after the previous interrupt and the 4000 ticks should take exactly 1.0E6 μ s (one second). However, with the loss of the fractional part, the 4000 ticks could actually take 1.0E6 μ s \pm 120 μ s or \pm 0.012%. If the integer value is incremented or decremented by one to fix the problem, the error will simply switch from being positive to negative.

Another issue is the fact that the time it takes for the PowerPC to load and start executing the installed interrupt handler is not constant. There will be small variations in the number of nanoseconds between the time that the DEC register goes negative and the time that the first instruction of the handler is executed. Recall that the first thing the handler does is reload the DEC register. This variation in DEC reload times can become more significant if there are several thousand interrupts per second.

In order to solve these problems, the VxWorks kernel interrupt handler also uses the PowerPC Time Base (TB) register. The TB is a 64 bit register that increments at the same frequency as the DEC register. However, unlike the DEC register, it does not generate an interrupt. It simply increments for a long period (more than 6 hours) and then rolls over. It can be used for accurate (nanosecond) time stamping among other things. When the DEC interrupt handler executes, it uses the TB register value to save an accurate time of when the handler *should* start executing after the next DEC interrupt. This is called the *expected TB value*. When the next interrupt occurs, the handler code examines the current value of the TB register and compares it with the expected TB value. It uses the difference to adjust the value that is reloaded into the DEC register. The adjustment is actually quite small and the accuracy of the adjustment is discussed in section 5.5.2. It should be noted that since the `ftss_sync` task will update the DEC register, it must also update the expected TB value used by the DEC interrupt handler.

4.4.4 Steady State Frame Synchronization Algorithm

The chosen frame synchronization algorithm is based on the work done by Srikanth and Toueg in [22]. The authors present two versions of a clock synchronization algorithm, one for use with message authentication and one for use without message authentication. Since the current P-NMR prototype does not use message authentication, the latter algorithm is used. The algorithm has several other properties that are directly

applicable to the P-NMR system. It can provide synchronized logical clocks that have the same accuracy as the underlying physical clocks. The algorithm is optimal in the sense that no other algorithm can exceed the provided accuracy. The algorithm handles several failure models including asymmetrical or Byzantine faults. Finally, time stamping of received messages at the physical layer of the CCDL is not required. Due to implementation issues on the channel hardware, the P-NMR system uses a slightly modified version of the algorithm and this section provides a brief discussion of the selected approach.

Figure 4.4-2 shows the steady state algorithm used by the P-NMR system. It is a two round exchange. The “f” is the number of simultaneous Byzantine (asymmetric) faults to be tolerated. For this discussion, $f = 1$ and so 3 ECHO messages ($2 \times 1 + 1$) are required to accept the synchronization. Also note that if $f = 1$ then an NMR system requires $N = 4$ channels ($3f + 1 = 3 \times 1 + 1$) to tolerate one Byzantine fault. Since the P-NMR prototype currently has only three channels, each channel would receive only two ECHO messages which presents a problem. Thus, the current implementation of the algorithm increments the number of messages received as soon as it sends its own message. For example, when a channel sends an ECHO message, it immediately sets the number of ECHOs received to one. It will then receive an ECHO from the two other channels bringing the total number of ECHOs received to three. This workaround will be removed when the fourth channel is added to the P-NMR prototype.

The sync algorithm is executed periodically. Let the duration of the period be P . The value of P can be any value that keeps the physical clock drift on each channel within some pre-defined limits (P can be every .001 seconds, every 0.1 seconds, every 2.0 seconds etc). Each period also has an associated “round” number that increments with each execution of the algorithm (e.g. 0, 1, 2, 3, ..., k , ...). At some point the round will wrap back to 0. All messages sent have the round number in the message. If all channels remained perfectly synchronized then they would each send an INIT message to ALL other channels every P seconds (i.e. $0P$, $1P$, $2P$, $3P$, ..., kP , ...). Stated another way, each channel would send an INIT message exactly P seconds after the previous send of an INIT message.

```

wait for  $cur\_time = kP - (1/2 \text{ window})$  to start listening for msgs
{
  if  $cur\_time = kP$  then
    send (INIT, round  $k$ ) to ALL
    if ECHO not yet sent then
      {
        if received (INIT, round  $k$ ) from at least  $(f + 1)$  channels then
          send (ECHO, round  $k$ ) to ALL
        if received (ECHO, round  $k$ ) from at least  $(f + 1)$  channels then
          send (ECHO, round  $k$ ) to ALL
      }
    if received (ECHO, round  $k$ ) from at least  $(2f + 1)$  channels then
      {
        accept (round  $k$ ):
        adjust local clock: compare  $cur\_time$  with  $kP + (2 * t\_del)$ 
      }
  }
wait for  $cur\_time = kP + (1/2 \text{ window})$  to stop listening for msgs

```

Figure 4.4-2 Steady State Frame Sync Algorithm

As soon as a channel receives $(f + 1)$ INIT messages for the current round k , and the channel has not yet sent an ECHO, it will send an ECHO message to ALL other channels. Also, as soon as a channel receives $(f + 1)$ ECHO messages for the current round k , and the channel has not yet sent an ECHO, it will send an ECHO message to ALL other channels. As soon as a channel receives $(2f + 1)$ ECHO messages, it will “accept” the round k synchronization and it will adjust its local clock based on the current time. This is the end of the synchronization algorithm for the current round k .

The CCDLs are point-to-point connections. Assume that a channel can send the same message on different CCDLs at the exact same time (i.e. the send over multiple CCDLs occurs simultaneously and not serially). Also assume that a channel can receive messages simultaneously on multiple CCDLs. Let the time required to send a message over all CCDLs and have it be received by all the other channels be t_del (i.e. the communication time delay). If all three channels remain perfectly synchronized then each channel should expect to receive $(f + 1)$ INIT messages at exactly time $kP + t_del$ (i.e.

t_{del} seconds after it sent its own INIT msg to all other channels). Also, in a perfect system, each channel should receive $(2f + 1)$ ECHO messages and accept the synchronization round at time $(kP + 2t_{del})$. This is because it takes t_{del} seconds for an INIT messages to travel across the CCDLs to the other channels and then it takes another t_{del} seconds for the ECHO messages to travel across the CCDLs to all the other channels.

In a perfect system, the accept should take place at exactly time $(kP + 2t_{del})$. However, in a real system each channel's physical clock will drift at a certain rate. The amount of drift on each channel will be different. One clock may be a little slow and another may be a little fast. All channels will wait until their local clock indicates exactly kP before sending the INIT. Now assume there is a "true" clock on the wall that keeps perfect time. Each time we see an INIT leave a channel, we examine the true clock and see that each channel is sending a little early, a little late or exactly on time. It is because of this drift rate with respect to true time that a channel cannot expect to receive INIT messages at exactly $(kP + t_{del})$ or ECHO messages at exactly $(kP + 2t_{del})$. A channel has to start listening for messages BEFORE time $(kP + t_{del})$ and it has to keep listening for messages after time $(kP + 2t_{del})$ so that it does not miss any messages. The size of this listening window depends on how much drift there can be between any two physical clocks.

When a channel receives $(2f + 1)$ ECHO messages, it will *accept* the synchronization round and adjust its clock. The clock is really the DEC register described above. It will compare its local clock (cur_time) with the expected accept time of $(kP + 2t_{del})$. If $cur_time \leq (kP + 2t_{del})$ then the local clock is a bit slow and time will be added to it. If $cur_time > (kP + 2t_{del})$ then the local clock is a bit fast and time will be subtracted from it. The adjustment cannot be too large and so is set to a maximum of t_{del} (half the total communication latency required to accept i.e. half of $2t_{del}$). Also, any adjustment of the local clock cannot cause it to move past the accept time (i.e. $kP + 2t_{del}$). Figure 4.4-3 summarizes the local clock adjustment after accepting the current round synchronization.

```

adjustment = t_del;
if cur_time ≤ (kP + 2t_del) then
{
    if (cur_time + adjustment) ≤ (kP + 2t_del) then
        cur_time = cur_time + adjustment
    else
        cur_time = kP + 2t_del
}
else if cur_time > (kP + 2t_del) then
{
    if (cur_time - adjustment) ≥ (kP + 2t_del) then
        cur_time = cur_time - adjustment
    else
        cur_time = kP + 2t_del
}

```

Figure 4.4-3 Clock Adjustment Algorithm

4.4.5 Scheduling the FTSS SYNC Partition

The frame sync algorithm is implemented in the core OS *ftss_sync* task that executes in supervisor mode. This core OS task needs to execute periodically. In order to add it to the partition schedule major frame, an FTSS_SYNC system partition is used. As described in section 4.2.3, there is no user application code associated with this system partition. It is simply an empty partition that is used to reserve time in the partition schedule major frame for the core OS *ftss_sync* task. The *ftss_sync* task is *assigned* in the time domain to the FTSS_SYNC partition so that it only runs during the schedule windows allocated to the partition.

A determination has to be made as to how often the clock sync algorithm should execute. In other words, what should the period or frequency of the FTSS_SYNC partition be? The selected period cannot be so long that a local clock drifts too far away from the clocks on the other channels. The Wind River SBC750GX documentation indicates a worst case clock drift of 100 parts per million (ppm) or 100 μ s per second when the system clock driving the PowerPC is operating at 133 MHz. This is the drift with respect to some “true” clock. Testing in the lab indicated that the drift between any two channels was quite small over a one second period (see section 5.5.1 for details).

Thus, it was decided to schedule the FTSS_SYNC partition (and hence the clock sync algorithm) once per second. Since this period (one second) was the longest of all the partitions, it is also the duration of the schedule major frame i.e. the major frame is one second long and all partitions (user and system) are scheduled as needed during the one second. The major frame is then executed repeatedly. The FTSS_SYNC partition executes once at the end of each major frame. This is shown in Figure 4.4-4. This is called the NORMAL schedule and is used after system initialization has completed.

FTSS keeps a major frame counter that is incremented once at the start of each major frame. It keeps incrementing until it rolls over. All messages that go across the CCDL have a basic FTSS header that includes the major frame number, the minor frame number and the partition window number so that FTSS code can verify that all received messages are fresh and not stale.

The duration of the FTSS_SYNC partition window within the major frame is another variable that needs to be set. Since there are 4000 kernel ticks per second, the smallest duration that can be selected for any partition in the major frame is one tick or 250 μ s. If the send of the INIT message (i.e. time kP) is placed in the middle of the 250 μ s single tick window then the ftss_sync task can start listening for messages (i.e. INIT and ECHO) 125 μ s before this time (at the start of the tick window) and it can continue listening for another 125 μ s after this time (at the end of the tick window). Thus, even if two channels managed to drift 100 μ s apart during a one second period, the 250 μ s sync window would still be large enough to receive early or late INIT and ECHO clock sync messages. In reality, the chosen synch window length is slightly smaller than 250 μ s, as will be discussed in the next section.

4.4.6 The FTSS sync task

The ftss_sync task implements the steady state clock sync algorithm described above. It is a high priority core OS task that executes in supervisor mode. It only executes once per second during the FTSS_SYNC partition window at the end of the major frame. No user level processes execute during the FTSS_SYNC partition window – only the single core OS task executes. The duration of the FTSS_SYNC partition window is one kernel tick or 250 μ s.

For the following explanation of the clock sync algorithm implementation, we will use terms such as “ μs remaining in the tick”. This is because the `ftss_sync` task makes extensive use of the PowerPC DEC register which starts with a value equal to one tick (250 μs in this case) and then the value decrements until it reaches zero, causing an interrupt which is the next tick. The `ftss_sync` task reads the register to determine how many μs remain before the next tick interrupt.

The scenario starts with a PowerPC DEC interrupt indicating a kernel tick has expired. The kernel tick interrupt handler executes the module partition scheduler which determines that it is time for the `FTSS_SYNC` partition to execute i.e. `FTSS_SYNC` is the next partition in the sequence of partitions that make up the schedule major frame. In our case, it is also the last partition in the major frame. The partition OS (vThreads) will not start any user level processes since none exist for this system partition. However, the core OS kernel will schedule the `ftss_sync` task to execute.

Even though the `FTSS_SYNC` partition has been allocated one full kernel tick (250 μs) in the major frame, the total size of the clock sync window used by the `ftss_sync` task is set to 200 μs . One reason for this is that it takes a few μs for the kernel to load and start executing the sync task. Another reason is that there are a few operations that must be completed after the clock sync algorithm completes and so a little bit of time must be reserved near the end of the 250 μs tick window.

Figure 4.4-4 depicts the main events that occur while the core OS `ftss_sync` task is executing during the `NORMAL` schedule. The current implementation has the 200 μs clock sync window opening 25 μs into the tick (i.e. with 225 μs left until the end of the tick). The broadcast of the `INIT` message to `ALL` takes place exactly in the middle of the 200 μs window so that there is 100 μs to either side of the broadcast so that early or late messages can still be received. Thus, the `INIT` broadcast takes place 125 μs into the tick (125 μs left until the end of the tick). This is time `kP`. The reception of the `INITs` from other channels will start to take place at time $(kP + t_del)$. In this example, the time to send and receive a message (`t_del`) is 5 μs . After the first reception of an `INIT`, the `ECHO` message is broadcast to `ALL` (since we only have three channels, we only wait for one `INIT` instead of two). The reception of the `ECHOs` take place at time $(kP + 2t_del)$. Once both `ECHOs` have been received from the other two channels, the local channel *accepts*

the synchronization round and *adjusts the DEC register* – this is the adjustment or synchronization of the “local clock” using the algorithm in Figure 4.4-3.

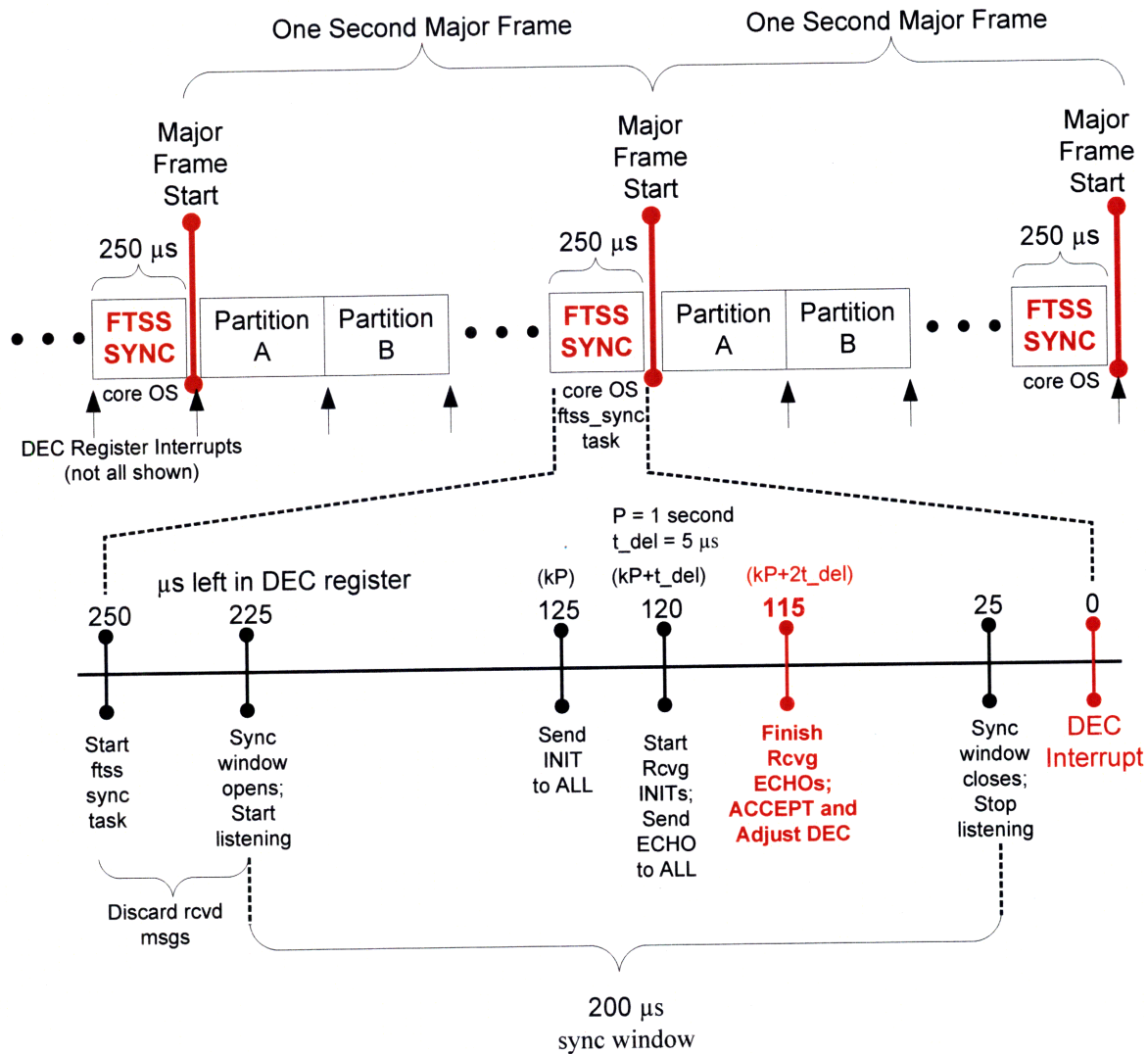


Figure 4.4-4 Major Frame Synchronization

The adjustment of the DEC register *changes* when the next interrupt will occur. Since the next interrupt is the start of the next major frame, the DEC adjustment changes when the next major frame begins. Since all three channels are performing the DEC adjustment to obtain convergence, the start time of the major frames on all three channels will also converge. If major frames are aligned on each channel then so are the minor frames within the major frame.

There are some issues with the described design. It should be apparent that the adjustment of the DEC by `ftss_sync` can contribute to some partition start time jitter as observed across multiple major frames. However, since the drift rate of physical clocks tend to stabilize over time, the jitter should be reduced after a certain amount of steady state operation. In any event, the maximum jitter is equal to the maximum allowed DEC adjustment `t_del`.

Another issue is border line timing. While the `ftss_sync` task is waiting for the 200 μ s sync window to open 25 μ s into the tick, it continually discards any received messages. Thus, if another channel sends a clock sync message that arrives too early (e.g. 24.999 μ s into the tick) it will be rejected. Thus, it is still possible for asymmetrical results to occur due to messages arriving right on the edge of a timing event (e.g. window open). For example, CH1 could send a message early and CH2 will indicate that it received it inside its window whereas CH3 will indicate that it never received it. This is not necessarily a problem as CH2 and CH3 will send ECHOs at a later time which will cause CH1 to synchronize its fast clock.

Also, the broadcasting of the INIT message exactly in the middle of the 200 μ s sync window is non-trivial. This is because all clock sync processing takes place during a single 250 μ s kernel tick. This means that all timing services provided by the kernel that use ticks as a parameter (i.e. watchdog timers, etc.) cannot be used to determine when it is time to broadcast the INIT. Also, the reception of all CCDL messages is done using driver polling. Thus, there are no CCDL I/O interrupts during the 200 μ s sync window indicating fresh data. Thus, the `ftss_sync` tasks loops continually, polling for new messages while at the same time watching the DEC register count down to see if it is getting close to the middle of the 200 μ s window (time `kP`) when it is time to broadcast the INIT.

The issue is that the receive function that polls the CCDL drivers for new messages can take 3 to 5 μ s if fresh data is actually present from both of the other channels. If no fresh data is present then the receive routine takes less than 1 μ s. Thus, if the receive function is called right before the time to broadcast the INIT then there is a chance the receive function will not return until 5 μ s after it was time to do the broadcast. Thus, the `ftss_sync` task stops polling for new messages 3 μ s before time `kP` and simply waits for

the DEC register to indicate time kP (i.e. 100 μ s into the 200 μ s sync window) before calling the broadcast service.

Finally, if an accept does not occur as planned because of a missing ECHO, the ftss_sync task will continue to loop, checking for new messages each time through the loop. During every loop it checks the DEC register to see if it is smaller than the value corresponding to the closing of the 200 μ s sync window. If the DEC is smaller then the main loop will be exited and the synchronization round will fail. When synchronization fails, the module will switch to an entirely different partition schedule, called the INIT_SYNC schedule. This schedule is completely distinct from the NORMAL schedule and is explained more in the next section. Figure 4.4-5 shows the algorithm from section 4.4.4 with additional details for monitoring the DEC register.

4.4.7 Frame Synchronization After System Cold Start

The previous sections explained how frame synchronization works when the P-NMR system is in steady state (i.e. NORMAL) operation. The FTSS_SYNC partition was scheduled for one tick at the end of the major frame after all other user partitions such as GN&C had executed. This section describes how synchronization is accomplished for system cold start when all three channels are starting after power up.

The startup partition schedule, called INIT_SYNC, is completely distinct from the steady state NORMAL schedule described in section 4.4.5. The only partition in the INIT_SYNC major frame is the FTSS_SYNC partition. No user partitions are given any time. The duration of the FTSS_SYNC partition is one second. This means the duration of the INIT_SYNC major frame is one second. The choice of one second is arbitrary. However, the number of kernel ticks per second is still the same: 4000.

```

Start round  $k$  due to DEC interrupt /* DEC is at maximum */

/* wait for window to open – discard rcvd msgs */
while DEC > window_open
    empty_all_CCDL_buffers()

/* window is now open */
/* wait for accept or for window to close */
while (not yet accepted) AND (DEC > window_close)
{
    check_for_new_msgs()

    if received (ECHO, round  $k$ ) from at least  $2f + 1$  channels then
    {
        accept (round  $k$ )
        adjust DEC: compare cur_dec with expected_dec
        EXIT
    }

    if received (INIT, round  $k$ ) from at least  $f + 1$  channels then
        send (ECHO, round  $k$ ) to ALL

    if received (ECHO, round  $k$ ) from at least  $f + 1$  channels then
        send (ECHO, round  $k$ ) to ALL

    /* if we are close to middle of window i.e. time  $kP$  */
    if DEC < (window_middle + buffer)
    {
        /* wait for middle of window */
        while (DEC > window_middle)
            keep waiting
        send (INIT, round  $k$ ) to ALL
    }

} /* wait for accept or window close */

if (no accept) then
    switch to INIT_SYNC partition schedule

```

Figure 4.4-5 Clock Synch Algorithm using DEC Register

Since the only partition executing is FTSS_SYNC, the only task executing is the core OS ftss_sync task. The operation of the sync task is basically the same. During each 250 μ s kernel tick, the sync task will attempt to synchronize with other channels. It will still require all three channels to be operational for the sync algorithm to be successful. The only difference is that the sync window within the tick window is made as large as possible - close to 250 μ s instead of 200 μ s. This is so that all channels are listening for other channels nearly all the time. Thus, the sync algorithm will execute during every single tick or once every 250 μ s (i.e. 4000 times per second) until all three channels obtain the first accept. The ftss_sync task will then verify that the next ten sync efforts during the next ten ticks also result in successful synchronizations. This is just to make sure that the first successful synchronization was not an aberration.

Once there has been ten successful sync attempts, ftss_sync will transition the module to another schedule called APP_STARTUP. This schedule is similar to the NORMAL steady state schedule except that each user partition gets a little bit of extra time. This is required because APEX partitions and processes require extra time to initialize for the first time and this initialization can offset normal steady state timing. This is explained further in section 4.5.2. The APP_STARTUP schedule has all the user partitions in it and the FTSS_SYNC partition gets the usual 250 μ s at the end of the major frame. The APP_STARTUP schedule is one second long just like the NORMAL schedule. Once the APP_STARTUP schedule executes successfully once, ftss_sync transitions the module to the NORMAL schedule.

4.4.8 Impact of Frame Synchronization on Robust Time Partitioning

This section assesses the impact of the frame synchronization activities on robust time partitioning. It is essential that the DEC register adjustment not steal time from the next partition/process in the schedule or violate the robust time partitioning in any manner. Also, the frame synchronization activities at the core OS level should be transparent to APEX processes running in user partitions.

There are two types of APEX time services that a partition process can use: a request for the system time and a request to wait. Both types are handled by the partition OS (i.e. vThreads), not the core OS. Both types of services use the

SYSTEM_TIME_TYPE . The ARINC 653 standard defines this type as a 64 bit signed integer with the LSB equal to one nanosecond.

4.4.8.1 Requesting the System Time

The system time is the time since module power up and is the same for all processors on the module. When an APEX process in a user partition needs to obtain the system time, it calls the following APEX service defined in the ARINC 653 standard:

```
void GET_TIME (
    /*out*/ SYSTEM_TIME_TYPE *SYSTEM_TIME,
    /*out*/ RETURN_CODE_TYPE *RETURN_CODE );
```

Calling this service results in the partition level OS (vThreads) reading the PowerPC 64 bit TB register and then converting this value into nanoseconds. As discussed in section 4.4.3, the TB register is incremented at a frequency based on the system clock (e.g. 133 MHz). This is the same frequency used to decrement the DEC register. The TB can be read but not written to in user mode. The TB register is not altered when the DEC register is adjusted after the frame synchronization algorithm completes successfully in the ftss_sync task. Adjusting the DEC register only causes a slight change in the release point of partitions and processes with respect to the TB register. Note that the adjustment of the DEC register does not change the release points with respect to the start of a major frame.

An APEX process that uses GET_TIME several times within one partition window or within one major frame will not be affected by the adjustment of the DEC register since this occurs once at the end of each major frame. However, the adjustment of the DEC register will manifest itself if an APEX process saves one system timestamp by calling GET_TIME and then saves another timestamp but in a different major frame. If the process calculates the duration between the two timestamps, the result will not be the same as a system that is not running the ftss_sync task. Compared to the TB register, the adjustment of the DEC register alters the release point of the user process in the next major frame and so the TB time when the process calls the second GET_TIME will not be the same as when the DEC is not adjusted. The maximum difference between these

two results is equal to the maximum adjustment allowed for the DEC register. This is currently `t_del` which is the maximum CCDL communication delay.

4.4.8.2 Requesting to Wait

When an APEX process in a user partition needs to wait for a certain duration, it can choose between two APEX services defined in the ARINC 653 standard:

```
void PERIODIC_WAIT (
    /*out*/ RETURN_CODE_TYPE *RETURN_CODE );

void TIMED_WAIT (
    /*in */ SYSTEM_TIME_TYPE DELAY_TIME,
    /*out*/ RETURN_CODE_TYPE *RETURN_CODE );
```

Calling one of these services causes the process state to change from `RUNNING` to `WAITING`. This will cause the partition OS (vThreads) to swap the process out of the PowerPC and another `READY` process will run in its place. The `PERIODIC_WAIT` does not have a `DELAY_TIME` parameter because periodic processes call this service when they have completed their work for the current period and want to wait until their next release point. The period of the process is specified (in nanoseconds) during process creation and so the partition OS has access to when the next release point is. For the `TIMED_WAIT` service, the passed in `DELAY_TIME` represents the number of nanoseconds to wait. Both of these services are handled by the partition OS, not the core OS.

Even though both of these services start with a delay time specified in nanoseconds, the actual implementation converts the nanoseconds into core OS kernel ticks. The result is rounded up to the nearest tick. This is because the partition OS relies on the core OS to inform it of the passing of time. Recall from section 4.4.2 that the core OS kernel is made aware of the passing of time through kernel ticks using the DEC register interrupt. The core OS kernel then passes these hardware interrupts to a partition OS using a pseudo-interrupt mechanism. Only the currently active partition receives these pseudo-interrupts. The core OS may batch several hardware interrupts into one single pseudo-interrupt in order to reduce the amount of processor throughput consumed by vThreads handling the

pseudo-interrupts. Thus, the APEX wait services cannot actually provide accuracy at the nanosecond level. They must use the granularity of kernel ticks.

As with the GET_TIME service, the execution of the ftss_sync task and the DEC register adjustment only affects the wait services when the number of ticks to wait spans a major frame boundary. The last tick in the major frame is where the DEC adjustment occurs. Even though ftss_sync does not change the number of ticks that an APEX process is waiting for, the absolute time (as recorded by the TB register) will be slightly different compared to when ftss_sync is not executing. Thus, the DEC adjustment will introduce some jitter to the start time of the first process or task in the next major frame (in our case, this is the ftss_input task). Again, the maximum jitter will be t_{del} which is the maximum CCDL communication delay. It should be pointed out that the release points for processes with respect to the start of a major frame will not change. The release points only change with respect to the TB register.

4.4.9 Summary

The adjustment of the DEC register after accepting a synchronization round is the key to major frame synchronization. By increasing or decreasing the value that was in the DEC register at the time of the accept, the ftss_sync task can hasten or delay the start of the next major frame. This is because once the DEC register is loaded with an adjusted value, it simply continues to count down as usual until it goes negative causing the next kernel tick interrupt. Since FTSS_SYNC is the last partition to execute in the major frame, the next tick interrupt signifies the start of the next major frame. Since all three channels will adjust their DEC register to achieve convergence, the start of the next major frame will occur at approximately same time on all three channels. If the major frames are aligned on each channel then so are the minor frames within the major frame.

The execution of the FTSS frame synchronization algorithm does not violate any ARINC 653 robust time partitioning rules. The adjustment of the DEC register does not steal time from any user APEX process in the next major frame. However, it does introduce a small amount of jitter to the start time of the first task or process in the next major frame.

4.5 Scheduling FTSS Input/Output Processing

This section discusses two options for scheduling the FTSS Input and FTSS Output processing in an ARINC 653 partitioned module. The FTSS exchanges over the CCDL have to take place while all the user partitions are executing on the module and while the data bus is sending and receiving data. The schedule design is extremely important in terms of overall system performance and reliability. The schedule must ensure that channels stay in sync, deadlines are met and response times are kept to a minimum. Thus, this design activity is can be quite complicated.

4.5.1 Scheduling Data Bus I/O

An important consideration when designing the schedule for the FTSS I/O processing is how access to the data bus I/O hardware is scheduled within the channel. Unfortunately, the method used for data bus access depends heavily on the I/O hardware specifics and the low level driver. For example, is the data bus I/O hardware integrated directly onto the core processing module (e.g. the SBC750GX board) or is it a separate I/O module inserted into the channel backplane? Does the data bus I/O board have its own processor? Does the PowerPC on the SBC750GX need to *pull* the data from the I/O hardware into the local SDRAM or will the I/O hardware *push* the data into the SDRAM autonomously without requiring the involvement of the PowerPC (e.g. DMA engines). Does the I/O hardware have a local copy of a timing table that dictates when to transmit during the major frame of the PowerPC on the SBC750GX board?

The answers to all of these questions could influence the scheduling of the partitions on the SBC750GX module. Whenever code on the PowerPC needs to move (pull) data from the data bus I/O hardware to SDRAM, it is likely that an I/O or Gateway partition would be required in the schedule to manage this activity. Figure 4.5-1 shows a possible sensor input sequence with five partitions executing on the SBC750GX module. The I/O Processing Partition moves data from the data bus I/O hardware to the SDRAM, the FTSS_INPUT performs the data exchange to obtain input congruency, the user GN&C FDIR partition performs sensor data reduction using the congruent data, the GN&C partition containing the algorithms executes and finally the FTSS_OUTPUT

partition performs output voting. Another option is to merge the data bus I/O processing and the FTSS I/O processing into one partition and one core OS task.

I/O Processing	FTSS INPUT	GN&C FDIR	GN&C	FTSS OUTPUT	I/O Processing
---------------------------	-----------------------	--------------------------	-----------------	------------------------	---------------------------

Figure 4.5-1 I/O Processing added to partition schedule

The current P-NMR prototype only simulates a data bus. The simulation places generated sensor input data into the main SDRAM on the SBC750GX board. The simulation executes at the beginning of the ftss_input task processing. Thus, the simulation assumes that some mechanism would be available for moving data from the actual bus I/O hardware into SDRAM. Since the data bus I/O processing is currently simulated, a more detailed discussion of how to schedule data bus accesses within the channel is not included in this thesis. The reader is referred to [41] for a detailed analysis of data bus scheduling.

4.5.2 ARINC 653 Partition and Process Scheduling Overview

Before the options for scheduling FTSS I/O processing can be discussed, a brief review of the ARINC 653 two level scheduling design is presented. The term “two level” scheduling is used because there are two distinct scheduling activities on an ARINC 653 module.

First, partitions are scheduled in a fixed, cyclic manner as discussed in section 4.4.1. Partitions have no priority. A partition schedule is simply a sequence of partition windows of varying durations. The sequence of partition windows make up the major frame. Thus, TPS is used to schedule the partitions. The static partition schedule (major frame) must be constructed at design time. It is usually specified in the system configuration files (e.g. the XML files). The module core OS is responsible for scheduling the partitions using data from the configuration files.

Each partition contains one or more processes. Each partition also has its own partition OS. The partition OS, not the core OS, is responsible for scheduling the processes. The core OS has no visibility into a partition and cannot see or control processes. A classical priority preemptive scheduling (PPS) method is used by the partition OS to schedule the processes within a partition. In other words, the highest priority process that is ready to run within a partition is given the CPU. The priority of a process can be changed at run time. Processes only run during their partition's window in the major frame.

Periodic processes have fixed *period*, *time capacity* and *deadline type* attributes. The period specifies how often the process should execute. Time capacity specifies the elapsed true time within which the process should complete its execution during the period. It is important to realize that the time capacity is real time ("wall clock time") and not processor execution time. Thus, it must always be equal to or greater than the worst case execution time (WCET) for the process. The time capacity is always less than the period but it may span several partition windows within the major frame. The deadline type can be hard or soft. A hard deadline means the partition OS will perform some action if a periodic process deadline is missed. The action is specified at design time in the Health Monitoring system configuration files. Actions include ignore but log, wait for n occurrences before taking action, restart the process, stop the process, restart the partition (cold or warm) or stop the partition.

Each process in a partition has several *release points* corresponding to the start time of each of its periods. A process will move to the READY state when a release point arrives. If it is the highest priority process in the partition its state will change to RUNNING and it will be given the CPU. Release points can occur outside the partition window in which case the process will not execute until the start of the next partition window (if it is the highest priority ready process at that time). Processes also have a variable attribute called *deadline time* which is initially set to the current release point plus the time capacity. A periodic process must finish its work and call the APEX PERIODIC_WAIT service before the deadline time. Note that a deadline time can occur outside all the parent partition windows and inside another partition's window but it will not be acted upon until the next owner partition window. A process can also change its

own deadline time within a period by calling the APEX REPLENISH service but it must not cause the new deadline time to go past the next periodic release point.

Finally, the first release point for a periodic process is usually the start of the first partition window within the major frame. However in the case where a partition occurs more than once within the major frame, the first release point for a process can be set to a partition window other than the first one.

As can be seen from the above discussion, some aspects of process scheduling are non-trivial and the overall scheduling design needs careful consideration. What is important to note is that the system integrator has clear visibility into when partitions execute because of the fixed, cyclic nature of the partition schedule which is clearly specified in the system configuration files at design time. On the other hand, identifying when processes execute within partition windows in the major frame is more challenging. Processes are not specified in any way in the system configuration files and they can run at any time based on their state, priority and release points.

This is an important distinction to consider when attempting to synchronize the FTSS I/O activities among the three synchronized channels. The FTSS tasks execute as core OS tasks but they must also interact with the user partitions and processes. There are basically two methods for scheduling the FTSS I/O processing with the user applications and they are described next.

4.5.3 Implicit FTSS I/O Scheduling

This method does not explicitly add FTSS I/O partition windows to the partition schedule specified in the XML system configuration files. The partition schedule is made up of user partitions only. When a user APEX process within a partition needs FTSS I/O services, it makes custom FTSS API system calls. These calls are not part of ARINC 653 APEX standard. There is a system call to obtain congruent inputs from FTSS and there is a system call to send application outputs to FTSS for voting. The API calls result in a system call down into the core OS (after going through vThreads) where an FTSS task begins its processing (i.e. the two round input exchange or the one round output exchange). The user process blocks until FTSS has completed its exchange over the CCDL. The system call then returns to the user process and it continues to execute.

Note that several different user processes can and usually do execute during a single partition window in the major frame. It is possible that two or more processes may require FTSS I/O services in a single partition window. In this case, FTSS input processing may be called twice and FTSS output processing may be called twice in the same partition window.

As a simple example, consider user partition “A” which contains two APEX processes, A1 and A2. Process A1 is a high priority, flight critical process that requires congruent sensor inputs from FTSS and it sends its actuator command outputs to FTSS for voting. Process A1 must always execute first at the start of every partition “A” window in the schedule. It is not interrupted by any other process until it has completed all its processing. It is assumed that the release point of process A1 is at the start of the partition window. Once process A1 has completed its work, the goal is to have the lower priority process A2 execute for at least 3.0 ms before the end of the partition window. Process A2 does not need FTSS services and its release point and deadline are not in the current partition window. If process A1 finishes early and calls the APEX PERIODIC_WAIT service before its deadline, process A2 will start early and get more than the minimum 3.0 ms during the partition window.

Certain constant durations are known. The time required for process A1 to startup after its release point and invoke the FTSS system call to obtain congruent inputs takes 0.01 ms. The ftss_input task in the core OS takes 0.24 ms to perform the two round input exchange. Once process A1 has the congruent inputs from FTSS, it executes its control algorithms. The WCET of these algorithms is 2.0 ms. Process A1 will then make an FTSS system call to have its actuator command outputs voted. The ftss_output task in the core OS also takes 0.24 ms to perform the output exchange and vote with other channels. When the FTSS output system call returns, process A1 performs some cleanup and then calls the APEX PERIODIC_WAIT service indicating that it has finished all its work for the current period. This takes 0.01 ms. Thus, the total capacity required for process A1 to complete all its work is 2.5 ms as shown in Table 4.5-1. Since the desire is to have the low priority A2 process execute for at least 3.0 ms after process A1 finishes, the total duration for the partition “A” window in the partition schedule is 5.5 ms (2.5 ms for A1 and 3.0 ms for A2).

Activity	Duration (ms)
Process A1 Startup	0.01
FTSS core OS input processing	0.24
Process A1 algorithms WCET	2.0
FTSS core OS output processing	0.24
Process A1 Cleanup	0.01
TOTAL CAPACITY FOR PROCESS A1	2.5

Table 4.5-1 Process A1 System Call Durations

Figure 4.5-2 shows the execution of the A1 and A2 APEX processes within the 5.5 ms partition “A” window. Process A1 has a deadline that is 2.5 ms after its release point at the start of the partition window. The process deadline is the release point plus the capacity. It takes process A1 only 1.8 ms (out of the allotted 2.0 ms) to execute its algorithms and calculate actuator commands. Thus, A1 calls PERIODIC_WAIT 0.2 ms before its deadline. This causes the state of process A1 to change to WAITING and the partition OS then schedules process A2 to run since it was already in the READY state. Thus, A2 executes for 3.2 ms within the 5.5 ms partition window (it runs for an extra 0.1 ms).

The advantage of this implicit method of scheduling FTSS processing is that the FTSS services will be used whenever they are required. The system integrator does not need to know what order the processes execute in within a user partition window. As long as the process execution order and start times are the same on all three channels, the FTSS exchanges will be synchronized and successful. Another advantage of this method is that FTSS activities occur immediately, without any delay. When a user process has completed its computations and requests FTSS output processing, the FTSS output exchange over the CCDLs starts almost immediately. The FTSS processing does not have to wait for another partition window within the major frame to start its processing. This reduces overall system response times.

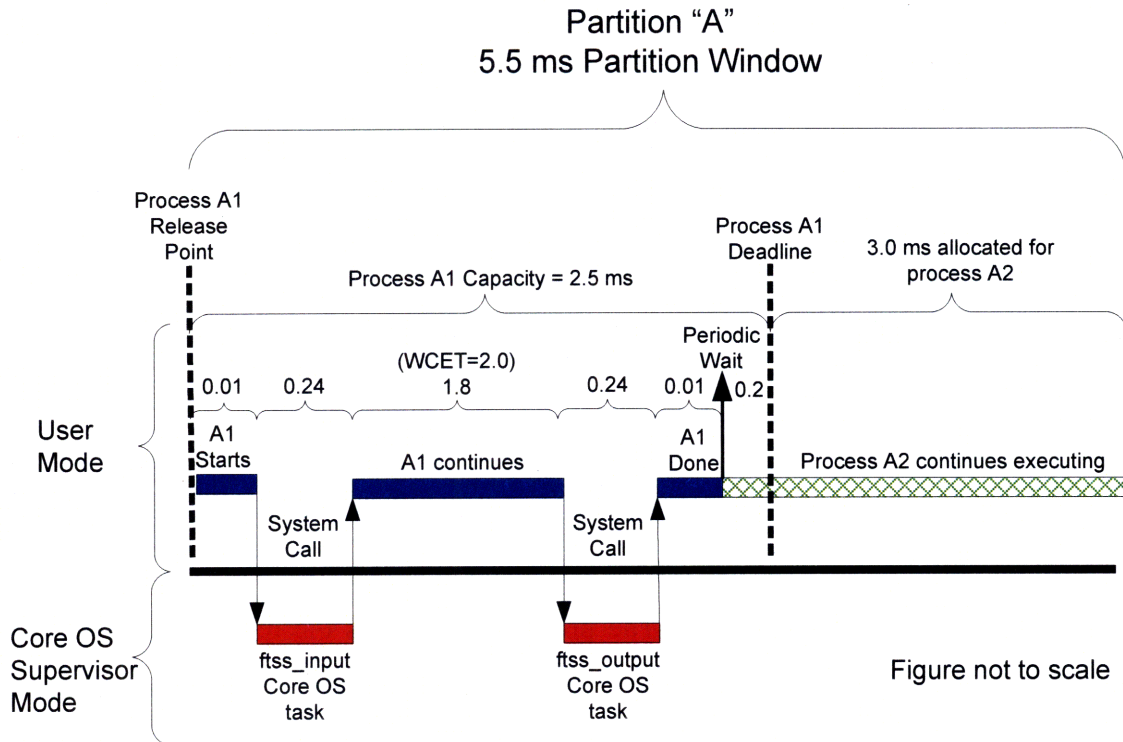


Figure 4.5-2 Implicit Scheduling of FTSS I/O in Partition Schedule

A significant disadvantage of this method is that the FTSS activities are not clearly specified in the module schedule in the XML system configuration files. The FTSS activities are scheduled by API calls in the user application code. This implicit scheduling of FTSS activities could make it more difficult for the system integrator to clearly understand the timing of the system. This is because the time required by the FTSS core OS tasks is included in the time required by the user process that made the FTSS API calls down into the core OS. The application developer must now consider the time required by the FTSS tasks in the core OS when deciding on process capacities and deadlines. The process capacities and deadlines will then influence the duration of the particular partition window which includes the execution of those processes. Thus, the use of FTSS will not require a change to the sequence of user partitions in the major frame in the XML system configuration file but it will require a change to the durations of the user partitions. Furthermore, a change to FTSS core OS code that modifies the time required for the data exchange over the CCDLs may require a change to the user process and/or partition durations.

Another disadvantage of this method is that user applications will have custom FTSS API calls down into the core OS. These calls are not part of the ARINC 653 APEX API standard. These non-standard API calls reduce the portability of the applications. Because of these disadvantages, the implicit method for scheduling FTSS I/O processing described in this section was not selected. Instead, an explicit scheduling method was chosen and it is described next.

4.5.4 Explicit FTSS I/O Scheduling

The explicit method for scheduling FTSS I/O activities involves adding FTSS partition windows to the static partition schedule (i.e. the major frame) in the XML system configuration files. Two system partitions are created, FTSS_INPUT and FTSS_OUTPUT, that have the core OS FTSS tasks assigned to them in the time domain. The core OS ftss_input task only runs during FTSS_INPUT partition windows and the core OS ftss_output task only runs during FTSS_OUTPUT partition windows in the major frame. FTSS I/O processing does not take place during user partition windows in the major frame. Thus, the time allocated to user processes and user partitions is used only for user application processing.

In order to illustrate this method, we use the channelized architecture shown in Figure 3.3-1 with a GN&C user partition replicated on all three channels. A major frame duration of one second is also assumed. The major frames are synchronized among all three channels as discussed in section 4.4.6. This example also assumes that an I/O Processing Partition has already moved the sensor data from the data bus I/O hardware into SDRAM on the processing module (i.e. the SBC750GX) where FTSS has access to it. The execution of the I/O Processing Partition is not included in this scenario.

The GN&C user partition contains two APEX processes, shown in Table 4.5-2. The high rate Control and Navigation (CN) process executes at 50 Hz (period = 20 ms) with a time capacity of 2 ms. In other words, the CN process must finish its work 2 ms into the 20 ms period. Since the intent is that the CN process will run uninterrupted, the 2 ms capacity is equal to the CN process WCET plus some amount of spare. The spare is for the seven DEC interrupts that occur during the 2 ms partition window. The 20 ms period of the CN process will be the “minor frame” in the module schedule. The low rate

Guidance and Navigation (GN) process executes at 1 Hz (period = 1000 ms, the same as the major frame) with a time capacity of 900 ms. Thus, the GN process has much more time within the one second major frame to finish its work. The high rate CN process has a much higher priority than the low rate GN process and so it will always execute before the GN process if both processes are in the READY state.

Process Name	Priority	Period (ms)	Capacity (ms)	Deadline Type
CN	63	20	2.0	HARD
GN	10	1000	900	HARD

Table 4.5-2 GN&C Apex Processes

The two processes communicate with each other using APEX inter-process communication, namely blackboards. Blackboards do not allow queuing of messages; there is only one message buffer. The 1 Hz GN process will write to the blackboards and the 50 Hz CN process will read the blackboards. The GN process only writes to the blackboards once per second after all of the GN algorithms have completed. Since the 50 Hz CN process executes fifty times between each GN write to the blackboards, the CN process simply keeps reading the same blackboard values until they change.

The 50 Hz CN process requires inputs from the three redundant MIMU sensors that are channelized and it will produce RCS thruster commands that need to be voted before being sent to the thrusters. The response time between the sensor reading and actuator activation must be kept to an absolute minimum. The 1 Hz GN process requires the inputs from the three redundant, channelized GPS receivers and the three redundant, channelized altimeters. The GN process does not produce any actuator outputs that require FTSS output voting. Instead, it writes to the APEX blackboards which are read by the high rate CN process. The GN process only reads the GPS inputs once per period (i.e. once per second).

There are several ways to schedule the GN&C partition so that the high rate CN process executes fifty times per second and the low rate GN process executes once per second. At a minimum, the GN&C partition must be scheduled fifty times for 2 ms each time so that the CN process completes its computations and sends actuator commands on

time. The GN process has more work to do but it has 900 ms out of the 1000 ms period to finish. The CN process will have 50 release points and deadlines per major frame whereas the GN process will only have one release point and one deadline per major frame. The first CN release point and the only GN release point will occur at the start of the first GN&C partition window within the major frame.

Some examples of how the GN process can be scheduled include making every 50 Hz GN&C partition window a little longer than the minimum 2 ms required for the CN process or ten out of the fifty windows can be made much longer than 2 ms. Note that the smallest partition window that can be used in the schedule is one kernel tick which in our case is 250 μ s. All partition windows must be a multiple of this duration. Also, many other user partitions are executing during the one second major frame - the GN&C partition does not run all the time.

Figure 4.5-3 shows a segment of the module schedule used by the P-NMR prototype. It contains one 20 ms minor frame within the major frame. It is not the first minor frame in the major frame and so the GN process has already obtained its GPS and altimeter inputs from FTSS. The GN release point and deadline are not in the depicted minor frame. The FTSS_INPUT system partition is placed into the partition schedule immediately preceding a 2 ms GN&C partition window. The FTSS_INPUT window is only one kernel tick in duration (i.e. 0.25 ms). It is during this window that the ftss_input core OS task performs the two round exchange of CN inputs. This window will occur at almost the exact same time on all three channels. Only the sensors required for CN (i.e. the MIMUs) are exchanged. The GN sensors (GPS and altimeters) are not included in this exchange. Once the two round exchange is complete, the ftss_input core OS task writes the congruent inputs to core OS memory that will be read by the GN&C application using APEX sampling ports. It is important to note that if the ftss_input core OS task completes early within the 0.25 ms FTSS_INPUT partition window, the GN&C partition will not start early. The GN&C partition must still wait for its partition window to open. While it is waiting the CPU is idle or it is performing background tasks.

Figure not to scale

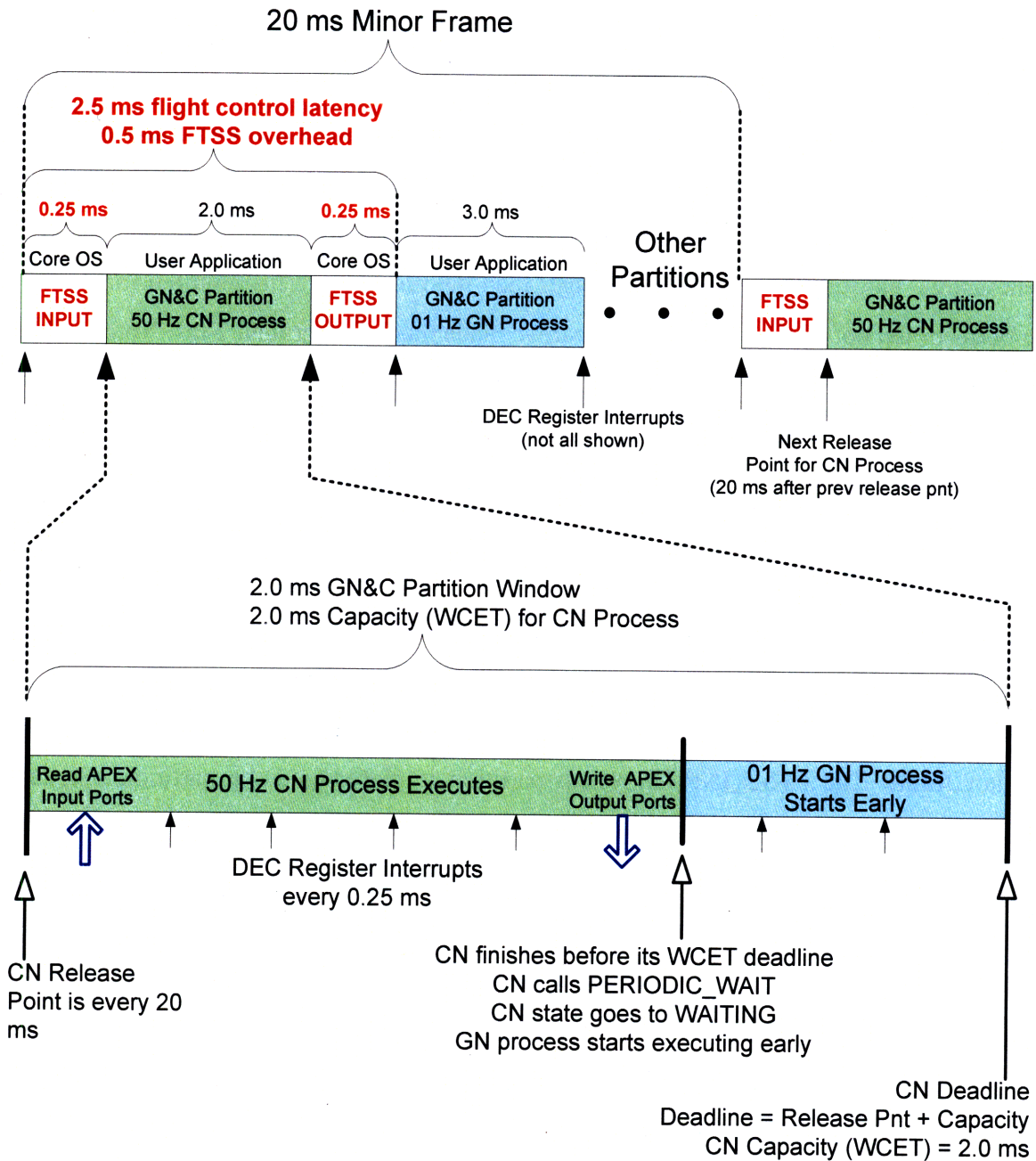


Figure 4.5-3 Explicit Scheduling of FTSS I/O in Partition Schedule

The GN&C partition is next in the 20 ms minor frame. It is always the first user partition in every 20 ms minor frame since the flight critical 50 Hz CN process must execute immediately. When it is time for the GN&C partition to execute, the partition OS

will schedule the CN process to run since it has a higher priority than the GN process. This first GN&C partition window in the minor frame is exactly 2 ms in duration since the CN process has a time capacity of 2.0 ms. The first thing the CN process does is read the APEX sampling ports that correspond to the three redundant MIMU sensors. This causes the input data to be moved from core OS memory up into the CN process (via the partition OS). The CN process then analyzes the three MIMU readings and performs data reduction to obtain one MIMU reading. It then executes its control algorithms and writes the resulting thruster command to the output APEX sampling port. This results in the immediate transfer of the data down into core OS memory that corresponds to the output sampling port. At this point the CN process has completed its work and so it will call the APEX PERIODIC_WAIT service which will cause the process to transition to the WAITING state. This must be done within the first 2 ms of the 20 ms minor frame since this is the time capacity of the CN process.

Since the CN time capacity is set to a duration that is greater than the WCET of the code, the CN process will always finish a little early, before the 2 ms GN&C partition window is closed. When this happens, the partition OS will schedule the low rate GN process since it will already be in the READY state. The GN process continues running until the end of this first 2 ms GN&C partition window.

The FTSS_OUTPUT system partition is placed in the partition schedule immediately after the first 2 ms GN&C window. Its duration is only one kernel tick or 0.25 ms. This window will occur at almost the exact same time on all three channels. The ftss_output core OS task reads the core OS memory corresponding to the output sampling port used by the CN process. It then performs the one round exchange and votes with the other channels, as described in section 3.3.2. Once the vote is complete, the result can be sent to the I/O Processing Partition (not shown) for transmission over the data bus to the actuator (i.e. RCS thruster). The I/O Processing Partition can be on another module (i.e. circuit board) within the channel. Alternatively, the ftss_output core OS task may be able to write the voted result to the appropriate memory which the data bus I/O driver uses. Note that if the ftss_output processing finishes early within the 0.25 ms FTSS_OUTPUT partition window, the next GN&C partition window will not start early.

After the FTSS_OUTPUT window, there is another GN&C partition window that is 3 ms in duration so that the 1 Hz GN process can continue executing. This second GN&C window does not have to come immediately after the FTSS_OUTPUT window. It does not even have to be in the current 20 ms minor frame. It could actually be placed anywhere in the major frame as long as the GN process completes its work 900 ms (the time capacity) after the first GN&C partition window (the GN process release point). However, in our example, it comes immediately after the FTSS_OUTPUT window. The high priority CN process will still be in the WAITING state since its next release point has not yet arrived. Thus, it will not execute. The lower priority GN process which is still in the READY state will execute for the entire 5 ms. After this, other user partitions will execute during this 20 ms minor frame. This sequence will repeat every 20 ms. Thus, the total FTSS input and output latency added to the control algorithm execution time is 0.5 ms. This is comprised of one 0.25 ms window for the FTSS sensor input processing and one 0.25 ms window for the FTSS application output processing. A sample of the above schedule taken from the XML system configuration files is contained in Appendix B.1.

Figure 4.5-4 shows the same scenario as above except that it shows the data movement between the FTSS core OS tasks executing in supervisor mode and the GN&C partition process executing in user mode. It also shows how communication channels are created using a source port and a destination port. Communication channels are explained more in section 4.6.

For a sensor input, a source port is associated with the FTSS_INPUT partition and a destination port is associated with the GN&C partition. A communication channel with a unique ID links the source and destination ports. During the FTSS_INPUT partition window, the ftss_input task will execute in the core OS in supervisor mode. Once the sensor data exchange and vote is complete, the ftss_input task updates the port memory associated with the sensor data (e.g. MIMU1). At a later time, the 50 Hz CN process will start to execute in user mode. In order to obtain the sensor data, it will make a destination port read request to the partition OS which results in a system call down to the core OS. This will cause a change from user mode to supervisor mode. The core OS code will then copy the port data (e.g. MIMU1) to the user memory buffer. The system call will then return to the CN process which will continue to execute.

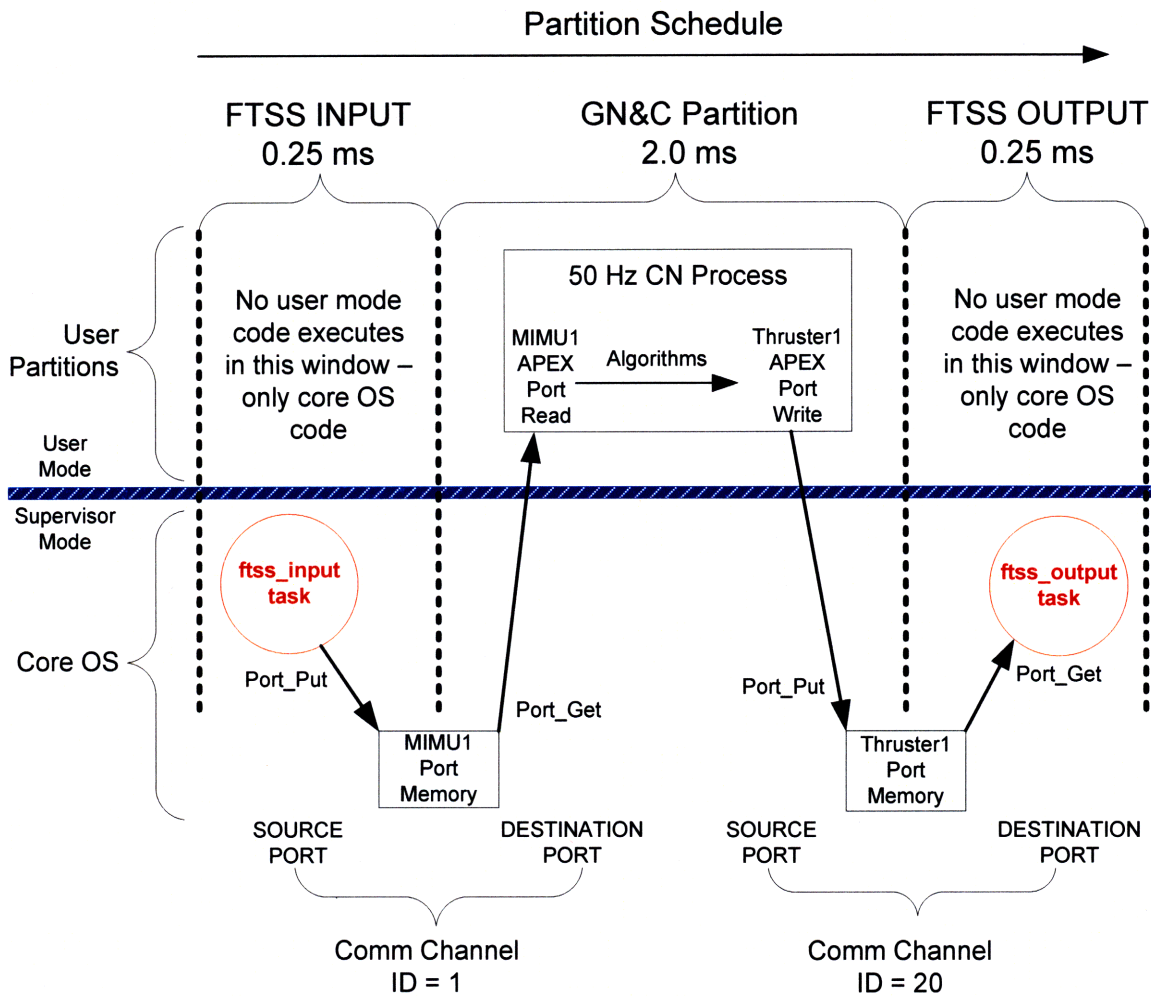


Figure 4.5-4 Explicit Scheduling of FTSS with APEX Ports

For an actuator command output, a source port is associated with the GN&C partition and a destination port is associated with the FTSS_OUTPUT partition. Again, a communication channel with a unique ID links the source and destination ports. Once the user CN process has finished executing the control algorithms, it will make a request to write an actuator command (e.g. thruster1) to the output port (these are source ports). Again, this results in a system call to the core OS which transfers the data from user partition memory to the core OS memory corresponding to the output port. At a later time, the ftss_output task will execute in supervisor mode during the FTSS_OUTPUT

partition window. It will obtain the actuator command from the port memory in the core OS and then it will perform the data exchange and vote with the other channels.

Figure 4.5-5 shows the special case of the first minor frame in the major frame where the 1 Hz GN process requires its inputs at the same time that the CN process requires its inputs. In this case, FTSS_INPUT is scheduled twice. The first occurrence is for the MIMU exchange for CN and the second occurrence is for the GPS/altimeter exchange for GN. It would be possible to *pipeline* the output exchange for CN with the input exchange for GN so that only one 0.25 ms window is used for both but this facility is not currently required given the performance of the CCDL.

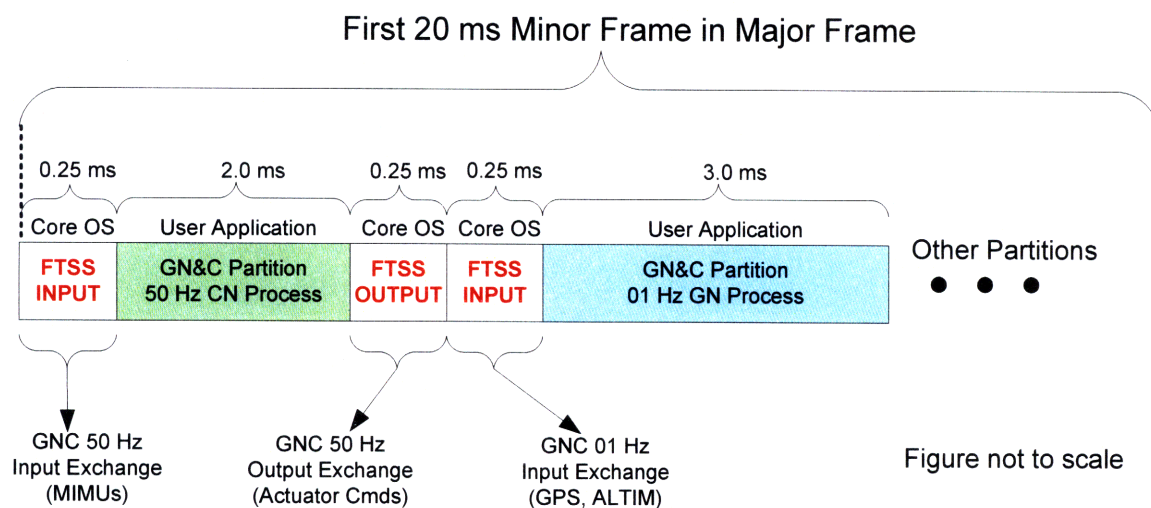


Figure 4.5-5 First 20 ms minor frame

Another option for the first minor frame would be to always include the GN sensors (GPS, altimeter) in every input exchange for CN sensors but this is suboptimal since the FTSS input exchange for CN inputs takes place fifty times a second and the GN sensors change only once per second. Another option is to perform a two round exchange of all inputs only on the first minor frame and then only exchange CN inputs on the other 49 frames. This is discussed more in the APEX Ports section.

4.5.4.1 Timing Within FTSS I/O Tasks

The method for timing the one and two round exchanges in the FTSS core OS I/O tasks is similar to the `ftss_sync` task described earlier. The events for the `ftss_input` task are shown in Figure 4.5-6. The input processing begins with a kernel tick. The core OS determines that it is time for the FTSS_INPUT partition to execute which causes the core OS `ftss_input` task to start executing. At this point the DEC register contains a value corresponding to almost the full 250 μ s (i.e. one kernel tick). It will continue to count down until the next PowerPC interrupt. The `ftss_input` task builds the round one message using the sensor data in SDRAM brought in by the I/O Processing Partition (or by the data bus simulation in our case). The round one message only contains the local sensor data. The input task sends this round one message to all other channels using the dedicated CCDLs. It then waits a predetermined time to receive the round one messages from the other channels.

Since all input processing takes place within one kernel tick, the input task must use the DEC register to keep track of the passage of time. It cannot use normal kernel time services like watchdog timers since these are based on kernel ticks. If the DEC counts down past a certain threshold value without `ftss_input` receiving a round one message from another channel, that channel is marked as being suspect. The channel that did not receive the round one data from another channel will continue with the round two exchange.

Once all the round one messages are received from the other channels, the `ftss_input` task builds the round two message which contains the sensor data from all the channels including the local channel. The input task then sends this round two message to all the channels using the CCDLs and then waits for the round two messages from the other channels. Again, the input task uses the DEC register to watch time and if the DEC gets low enough without receiving a round two messages from another channel, that channel is marked as being suspect. Once the `ftss_input` task has all the round two messages from the other channels, it sends all three sets of data (one set from round one and two sets from round two) to the `ftss_voter` for voting. The `ftss_voter` is a function call that returns the voted values. The `ftss_input` task then places the values in the appropriate

APEX sampling ports which will be read by the user application later. The ftss_output task operates in a similar fashion.

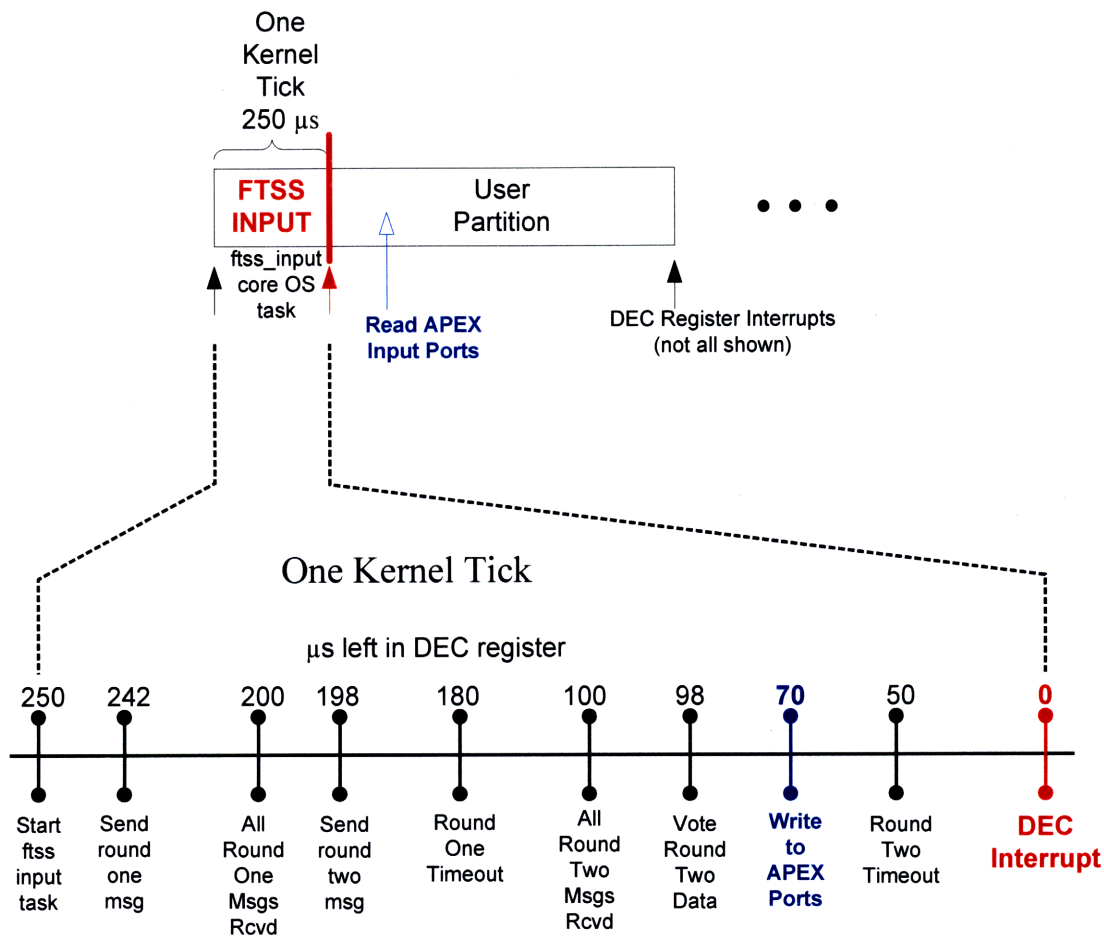


Figure 4.5-6 Timing within FTSS Input Task

It should be noted that the DEC register values picked for the round one and round two timeouts are currently selected at design time and are hard coded. However, the time required for building a message and transmitting it over a CCDL varies depending on the data size of the sensor inputs or application outputs. Thus, if the system integrator changed the amount of data that needed to be exchanged, either for input or output, the DEC register deadline times would also need to change in the code. Thus, the FTSS code needs to be updated so that it uses the system configuration tables (from the XML files) to calculate the DEC register timeouts.

4.5.5 Channel Specific Schedules

The P-NMR frame synchronous architecture with explicit scheduling of FTSS does not require that each channel execute identical partition schedules. It only requires that partitions that need to use FTSS services (i.e. congruent inputs, voting of outputs) be aligned in time on each channel. Stated another way, not all partitions need to be replicated on all channels. For example, if the flight critical GN&C partition is the only partition that uses FTSS then it is the only partition that needs to be replicated and aligned within the major frame on all three channels. As long as this requirement is satisfied, the non-flight critical partitions, if any, can be different on each channel and they can start at different times within the major frame.

Figure 4.5-7 shows an example of this. The flight critical FTSS_INPUT, GN&C and FTSS_OUTPUT partition windows are all in the same place within the minor frame of the partition schedule on each channel. Thus, with frame synchronization, they will all execute at the same time on each channel which allows for data exchange and voting to take place. However, after these flight critical partitions have completed their work, each channel starts its own non-flight critical partition within the minor frame. CH1 starts partition 1A, CH2 starts partition 2A and CH3 starts partition 3A. Note that these partitions do not have to have the same durations. Also note that partition 1B on CH1 starts even though other channels are not starting a new partition.

The advantage of this flexible approach is that it facilitates processor load balancing among the channels. Different non-replicated applications can be hosted on each channel. The disadvantage of this flexibility is that the static analysis tools that analyze the XML system configuration files need to be able to cross check the partition schedule defined for each channel in order to verify that flight critical partitions are properly aligned.

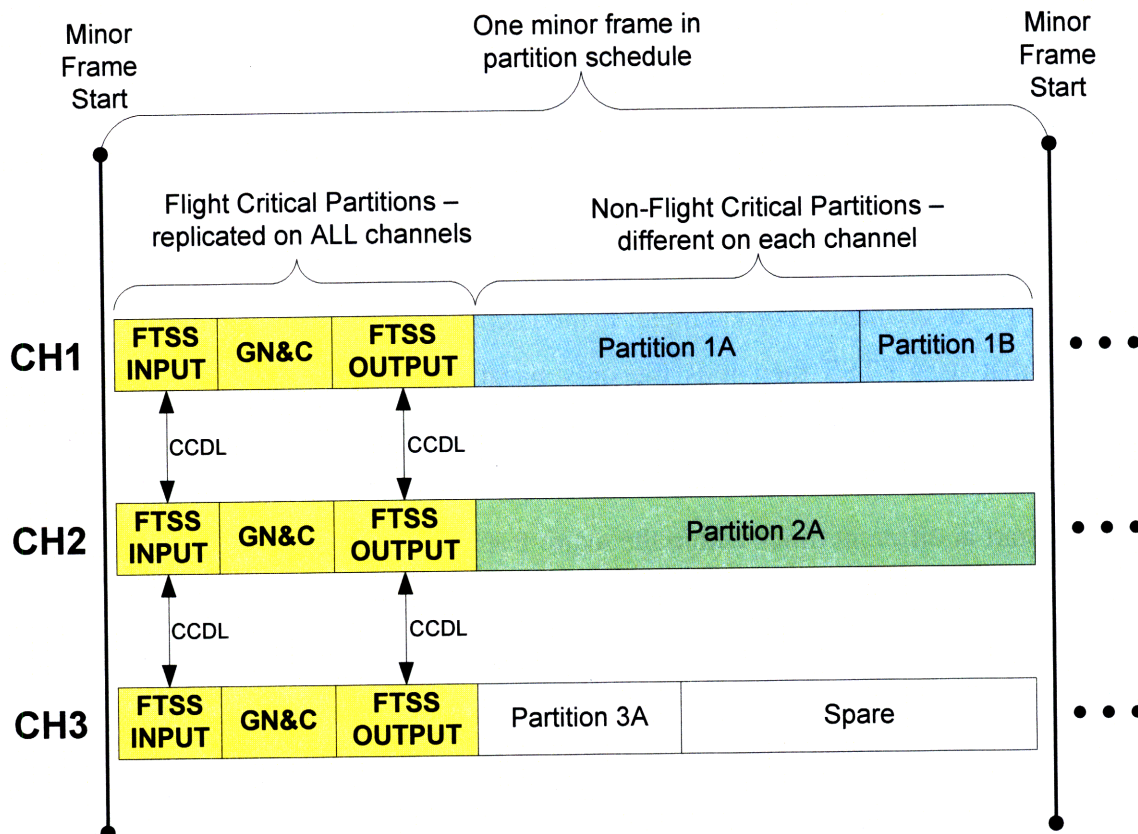


Figure 4.5-7 Channel Specific Schedules

4.5.6 Summary

The disadvantages of the explicit method of scheduling where FTSS system partitions are placed in the module schedule is that there can be an increase in latency compared to the implicit method. When the `ftss_input` task has finished the sensor exchange, the user CN process cannot start executing until the GN&C partition window opens. When the user CN process has calculated an actuator command, the `ftss_output` exchange cannot begin until the FTSS_OUPTUT partition window opens. However, the amount of idle CPU time in the FTSS_INPUT window will usually be less than 50 μ s, depending on the amount of sensor data that requires the two round exchange. Also, the total FTSS overhead added to the flight critical processing (the CN process) is just 0.5 ms; 0.25 ms (one kernel tick) for the FTSS input processing and 0.25 ms (one kernel tick) for the FTSS output processing.

Another disadvantage of this method is that it may require that a single user partition window in the schedule be broken up into separate windows with FTSS I/O partitions in between them. This will occur if several processes execute within the partition window and it is essential that the output from one process be sent to FTSS immediately before other processing takes place. This is the case in Figure 4.5-3 where the output from the CN process must be sent to FTSS for output voting before the GN process executes. The additional partition changes is not a significant impact on latency because it only takes a few microseconds for the core OS to change partitions. However, more careful analysis of the individual process release points and deadlines is required in order to ensure no process deadlines are missed.

The advantage of this method is that the scheduling of FTSS processing can be done in a clear and precise way at design time using the XML system configuration files. It is much easier for the system integrator to understand when the FTSS exchanges will take place on all three synchronized channels. The activation of FTSS processing is not buried inside user code. The time allocated to user partitions in the schedule is only used for user code and not for FTSS code and CCDL exchanges. Thus, the time required by FTSS is decoupled from the time required by user applications. In addition, user applications can use ARINC 653 APEX ports to communicate with the FTSS tasks in the core OS. They do not need to use any custom FTSS API calls. It is mainly for these reasons that this explicit method was chosen for the P-NMR system. In addition, this method allows for channel specific schedules.

4.6 Fault Tolerant ARINC 653 Ports

Previous sections have mentioned the use of ARINC 653 ports for inter-partition communication. This section describes in more detail what ports are and how the P-NMR system uses them.

4.6.1 Communication Channel and Port Overview

Inter-partition communication is defined by the ARINC 653 standard as communication between two or more partitions executing either on the same core module

or on different core modules. It may also mean communication between an APEX partition on a core module and non-ARINC 653 equipment that is external to the core module. All inter-partition communication is accomplished by using messages. A message is defined as a continuous block of data of finite length.

The basic mechanism for linking partitions is the *communication channel* (this channel is completely unrelated to the redundant computer channels used in a fault tolerant NMR architecture). A communication channel defines a logical link between *one source and one or more destinations*, where the source and the destinations may be one or more partitions. It also specifies the mode of transfer of messages from the source to the destinations together with the characteristics of the messages that are to be sent from that source to those destinations.

Partitions have access to communication channels via defined access points called ports. A partition owns its ports and is responsible for creating them. Any user process within the partition can access a port that is owned by the partition. A port can be either a *source* or a *destination* but it cannot be both. A communication channel consists of one source port and one or more destination ports. A port provides the required resources (e.g. memory) that allow a specific partition to either send or receive messages in a specific channel. A partition is allowed to exchange messages through multiple channels using their respective source and destination ports. A channel describes the route connecting one sending port to one or more receiving ports.

Communication using channels and ports can cross several boundaries. An ARINC 653 partition can send a message to another ARINC 653 partition that is on the same module, on a different module in the same cabinet or on a module in a different cabinet. It can even communicate with traditional hardware LRUs that are not running an ARINC 653 RTOS (such as sensors and actuators). The communication can take place using shared memory, a backplane bus (e.g. CompactPCI) or a data bus (e.g. Mil-Std-1553B, IEEE 1394b). In the case of a message going from an ARINC 653 partition to an entity outside the local module, the notion of a pseudo partition and port is used. These are explained later. By using pseudo ports, the core OS takes care of encapsulating the application data in a bus specific format and it routes the message to the appropriate location in the module (e.g. I/O driver memory) for transmission.

The core OS also manages incoming messages from outside the module in the same way. It strips away any data bus specific headers and only delivers the original application data to the destination ARINC 653 partition. For example, when a partition receives sensor data through a port, it does not need to know if it came from a sensor connected to a Mil-Std-1553B data bus connected to the local module or if it came over an Ethernet CCDL from another cabinet.

All the port and channel information is specified in the XML system configuration files at design time. Port definitions are included in the configuration data for a partition application. A separate configuration table defines the channels between partitions on the local module. If a message needs to go outside the local module via the backplane or data bus then a pseudo partition and pseudo port is defined. The system integrator is responsible for managing the connections between partitions, modules and cabinets.

The advantage of the above channel and port approach is that it isolates application software from the underlying hardware. User applications do not contain code that is specific to a particular processing module in a particular cabinet. This makes it much easier for the system integrator to move partitions between processing modules in one cabinet or even between cabinets. This activity is often done during system integration in order to reduce latencies, balance processor loads or to reduce data bus bandwidth consumption. It is also easy to add or delete communication paths within the system by editing the system configuration files. However, this would usually result in partition code changes as well.

Another advantage of using the static system configuration files to define the communication paths within the avionics system is that the system integrator can use sophisticated analysis tools to predict overall system timing and performance. The use of such tools is almost a necessity since the interaction between the partition schedule, backplane access and data bus I/O access can become complex and difficult to verify using testing in the lab [41].

4.6.2 Sampling and Queuing Ports

Each communication channel can be configured to operate in one of two transfer modes: sampling and queuing. The single source port and the one or more destination

ports that comprise the channel must be homogenous in terms of the transfer mode; they must all be sampling ports or they must all be queuing ports.

In queuing mode, the ports involved are allowed to buffer multiple messages in queues. Existing messages in the queue are not overwritten. A message sent by the source partition can be buffered in the source port queue until the core OS has time to move the message to the destination ports. Similarly, a destination port can buffer received messages until the receiving partition requests them. The oldest message in the destination queue is always the first one given to the receiving partition. Once a message is removed from a queue, it cannot be re-read. The send order is always maintained in the destination queues. Queuing ports have a maximum queue size specified at design time in the configuration files. Queuing ports may be more suitable for aperiodic messages or events where it is important that the event not be lost such as pilot inputs. The P-NMR prototype does not currently use queuing ports.

Sampling ports have only one single buffer for a message. On the send side, an old message remains in the source buffer until it is transmitted or until it is overwritten by a new message from the source partition. On the receive side, a message remains in the destination buffer until it is overwritten by a new message from the sending partition. The message is not removed from the destination buffer when the destination partition performs a read. If a destination partition reads the destination sampling port several times, the same data in the buffer will be returned until the buffer is overwritten with a new message from the source partition.

Sampling ports have a *required refresh rate*. This is the rate at which new messages should be received by a partition. The source port does not have to have the same refresh rate as a destination port. However, if the source partition transmits at a rate that is slower than the destination port's refresh rate, the OS will indicate to the destination partition that the messages are stale. Thus, careful consideration is required when designing sampling ports and the module partition schedule. Both the source and destination partitions need to be scheduled properly in the major frame so that messages are not declared stale.

Sampling ports are more suited for periodic data that is always transmitted at the same rate such as sensor inputs or actuator outputs used in flight control. Although an

occasional overwrite of a value is not desirable, it usually can be tolerated for one minor frame in the partition schedule. It is for these reasons that the P-NMR system uses sampling ports.

In summary, ports have the following attributes:

Partition Identifier

The partition that owns the port.

Port Name

A character string. It must be unique within the partition. This is usually specific to the data in the message such as “Altitude”. The name should not contain any reference to the specific source or destination partition e.g. “Altitude_From_Partition_X”.

Direction

Can be either SOURCE or DESTINATION.

Message Size

Size of the message in bytes.

Refresh Rate

For Sampling ports only. The rate at which the message in the sampling channel should be updated. Specified in microseconds or milliseconds.

4.6.3 Pseudo Partitions and Ports

Although not formally discussed in the ARINC 653 standard, pseudo partitions and pseudo ports are described in an appendix to the standard which details the use of XML system configuration files. Pseudo partitions and ports are used when an ARINC 653 partition needs to communicate with an entity outside the local module. The external entity can be another ARINC 653 partition, software running on a non-ARINC 653 computer or a hardware device with no software. Pseudo partitions and ports are added to the system configuration files just like regular partitions and ports. However, pseudo ports have additional attributes not present in normal ports such as the address of an I/O driver routine or the address of a memory location used by the I/O driver. Also, pseudo partitions are not placed in the partition schedule and do not get processor time.

Configuration data associated with a pseudo partition and pseudo port can be used by the core OS to correctly locate the appropriate I/O driver routine or memory location that should be used for transmitting or receiving a message over the physical data bus. In general, a proper ARINC 653 I/O partition will interact with pseudo partitions and ports. User partitions like GN&C would send their output data to the I/O partition using a normal communication channel since both the GN&C and I/O partition reside on the same module. The I/O partition would then send the GN&C output data to a pseudo partition and port which is really a mapping to an I/O driver routine or an I/O driver memory location. The data would then leave the module via the bus I/O driver.

The P-NMR prototype currently simulates a data bus and so does not make use of an I/O partition or a pseudo partition. However, it does use pseudo ports to specify the location of the simulated bus data. FTSS also uses its own internal computer channel and CCDL configuration data, described in Appendix C.4. It uses this data to perform the fault tolerant exchanges, as will be described next.

4.6.4 The P-NMR Port Design

The GN&C partition is used to illustrate how ARINC 653 sampling ports can be used with FTSS to implement fault tolerant application inputs and outputs. As discussed in section 3.3, the P-NMR prototype currently simulates a channelized bus architecture. Redundant sensors and actuators are connected to redundant channels using a channel specific data bus. A single sensor or actuator is not connected to more than one channel using a single data bus.

Connected to each channel are three sensor types: MIMU, GPS and barometric altimeter (ALTIM). All three sensor types are triple redundant so that each channel has its own set of three sensors. In this scenario, the GN&C partition executing on each channel requires all three readings from each sensor type. The GN&C partition will perform sensor specific fault detection and will reduce the three redundant readings for each sensor type into one set of values for that sensor type. Thus, the GN&C partition needs nine inputs (three sensor types and each type is triple redundant).

The CN process within the GN&C partition requires that the MIMU inputs be refreshed at a rate of 50 Hz. The GN process within the GN&C partition requires that the

GPS and altimeter inputs be refreshed at a rate of 1 Hz. As explained in section 4.5.4, the CN and GN processes may execute at different times within a schedule minor frame and may even execute in completely different minor frames. It is assumed that each sensor type sends 60 bytes every frame. Recall that ports are defined on a partition basis and not on a process basis. Given the above scenario, the system configuration files will indicate that the GN&C partition has nine *destination* ports as shown in Table 4.6-1. The first three rows correspond to 50 Hz inputs and the next six rows correspond to 01 Hz inputs. This port table will be identical on all three channels.

Port Name	Direction	Message Size (bytes)	Refresh Rate (seconds)
MIMU1	Destination	60	0.02
MIMU2	Destination	60	0.02
MIMU3	Destination	60	0.02
GPS1	Destination	60	1.00
GPS2	Destination	60	1.00
GPS3	Destination	60	1.00
ALTIM1	Destination	60	1.00
ALTIM2	Destination	60	1.00
ALTIM3	Destination	60	1.00

Table 4.6-1 GN&C Partition Input Ports

Since all of the above sensors come from an FTSS two round exchange for channelized inputs as described in section 3.3.1, the source of the above sensor ports is the FTSS_INPUT partition. Recall that the FTSS_INPUT partition does not have any actual application code running at the partition level in user mode. It only has the ftss_input core OS task. The partition is only used for temporal scheduling of the ftss_input task. Thus, the source for the nine sensor ports is not in a user partition but in the core OS. The ftss_input task can use kernel services to access and modify the port buffers in kernel memory.

The source ports associated with ftss_input are pseudo ports since they do not originate from an actual ARINC 653 partition. Taking CH1 as an example, MIMU1 comes from the simulated data bus, MIMU2 comes from the CH2 CCDL and MIMU3 comes from the CH3 CCDL. For sensors that come from the simulated data bus,

ftss_input uses an internal mapping to a memory location containing the data. In the future, this internal mapping will be migrated to using the “Driver Function” pseudo port parameter. This is an address to a data bus I/O driver routine (e.g. Read, Write). On CH1, the Driver Function parameter for the MIMU1 port will be set to the appropriate data bus driver read function whereas the Driver parameter for MIMU2 and MIMU3 will be set to null because ftss_input manages the CCDL drivers and does not need a driver mapping for these. Note that this setup means that the nine source ports will have different configuration data on each channel. For example, on CH2, MIMU1 comes from the CCDL and not the data bus as is the case on CH1. Table 4.6-2 shows the source port configuration data for each channel. The first four columns of the table are identical on all three channels but the last three columns represent channel specific data.

Port Name	Direction	Msg Size (bytes)	Refresh Rate (seconds)	CH1 Driver Function	CH2 Driver Function	CH3 Driver Function
MIMU1	Source	60	0.02	Read Fcn	null	null
MIMU2	Source	60	0.02	null	Read Fcn	null
MIMU3	Source	60	0.02	null	Null	Read Fcn
GPS1	Source	60	1.00	Read Fcn	Null	null
GPS2	Source	60	1.00	null	Read Fcn	null
GPS3	Source	60	1.00	null	null	Read Fcn
ALTIM1	Source	60	1.00	Read Fcn	null	null
ALTIM2	Source	60	1.00	null	Read Fcn	null
ALTIM3	Source	60	1.00	null	null	Read Fcn

Table 4.6-2 FTSS_INPUT Pseudo Source Ports going to GN&C

Given the source ports in Table 4.6-2 and the destination ports in Table 4.6-1, the corresponding connections in the communication channel table in the configuration files will look like Table 4.6-3.

Channel Id	Source		Destination	
	Partition	Port	Partition	Port
1	FTSS_INPUT	MIMU1	GN&C	MIMU1
2	FTSS_INPUT	MIMU2	GN&C	MIMU2
3	FTSS_INPUT	MIMU3	GN&C	MIMU3
4	FTSS_INPUT	GPS1	GN&C	GPS1
5	FTSS_INPUT	GPS2	GN&C	GPS2
6	FTSS_INPUT	GPS3	GN&C	GPS3
7	FTSS_INPUT	ALTIM1	GN&C	ALTIM1
8	FTSS_INPUT	ALTIM2	GN&C	ALTIM2
9	FTSS_INPUT	ALTIM3	GN&C	ALTIM3

Table 4.6-3 Sensor Input Channels Between FTSS_INPUT and GN&C

In terms of user partition outputs to the FTSS_OUTPUT partition for voting, the sampling port setup is similar to inputs except that now the GN&C partition has source ports and the FTSS_OUTPUT partition has destination ports. For this example, there are just three outputs from the GN&C partition that require cross channel voting (the one round exchange). The GN&C CN process writes to the source ports at 50 Hz and each message is 20 bytes. Just as the case was with FTSS_INPUT, the FTSS_OUTPUT destination ports do not actually go up into a user partition. The ftss_output core OS task interacts with the port buffers in core OS memory. Table 4.6-4 shows the three GN&C partition source ports, Table 4.6-5 shows the three FTSS_OUTPUT destination ports and Table 4.6-6 shows the communication channels.

Port Name	Direction	Message Size (bytes)	Refresh Rate (seconds)
Thruster1	Source	20	0.02
Thruster2	Source	20	0.02
Thruster3	Source	20	0.02

Table 4.6-4 GN&C Partition Output Ports

Port Name	Direction	Message Size (bytes)	Refresh Rate (seconds)
Thruster1	Destination	20	0.02
Thruster2	Destination	20	0.02
Thruster3	Destination	20	0.02

Table 4.6-5 FTSS_OUTPUT Destination Ports

Channel Id	Source		Destination	
	Partition	Port	Partition	Port
20	GN&C	Thruster1	FTSS_OUTPUT	Thruster1
21	GN&C	Thruster2	FTSS_OUTPUT	Thruster2
22	GN&C	Thruster3	FTSS_OUTPUT	Thruster3

Table 4.6-6 Command Output Channels Between GN&C and FTSS_OUTPUT

By using sampling source ports for the CN process outputs, the ftss_output core OS task can determine when the CN process has stopped placing fresh data in the communication channels since the destination ports will indicate stale data (i.e. the data was not refreshed at a 50 Hz rate). Stale ports could also indicate that the CN process did not meet its 2 ms deadline and was still executing when the GN&C partition was swapped out of the processor.

4.6.4.1 Scheduling Port Groups

Recall that several different partitions may be using FTSS I/O services during the schedule major frame. Figure 4.6-1 shows an example of two different partitions executing during the major frame, each of whom use FTSS I/O services. As the figure shows, the ports used during each FTSS input or output window can be completely different since they are for different partitions. Any user partition that required inputs from FTSS_INPUT would have to make sure that the appropriate source ports were added to Table 4.6-2. Similarly, any user partition wanting to send application outputs to

FTSS_OUTPUT would have to make sure that the appropriate destination ports were added to Table 4.6-5. Thus, these tables would in reality be much larger since FTSS tasks in the core OS could be serving multiple partitions, not just the GN&C partition.

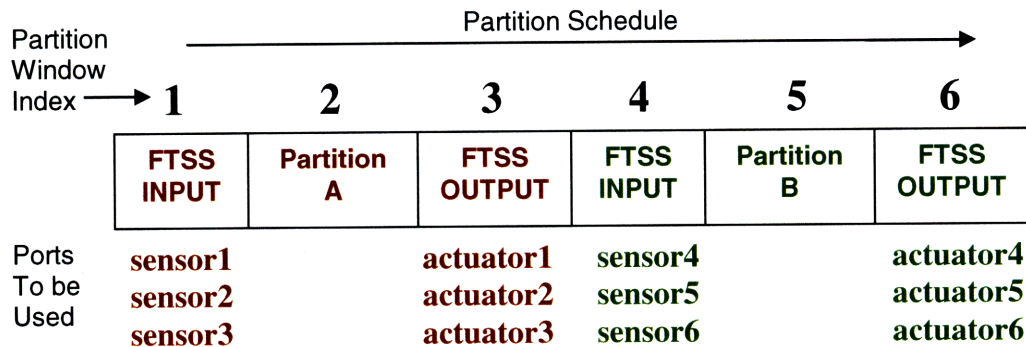


Figure 4.6-1 Different Partitions using FTSS services

Given that the FTSS input and output port tables could be quite large, a mechanism is required that allows FTSS to determine which ports should be used for exchanges during any given FTSS I/O window in the partition schedule major frame. Using sensor inputs as an example, an argument could be made that if FTSS knew what user partition came after the current FTSS_INPUT window in the schedule then it could use only those source ports whose destination ports are associated with that user partition. However, this would be a suboptimal design because a single user partition could have multiple windows in the schedule corresponding to when different APEX processes execute *within* the partition. Each process may require different congruent inputs. Thus, different source ports would be used by FTSS_INPUT at different times in the schedule major frame for the *same* partition. This is illustrated in Figure 4.5-5 where the CN and GN processes within the GN&C partition both require FTSS input services.

A more straightforward method would be to associate *port groups* with specific schedule windows. Port groups (or signal groups) are not a new concept and are used in several system modeling languages such as Simulink, SysML and AADL. The goal is to clearly indicate which source ports in the FTSS_INPUT port table should be used during

a particular FTSS_INPUT schedule window in the major frame. A similar destination port assignment is needed for the FTSS_OUTPUT schedule windows.

Basically, ports in the FTSS port tables are *grouped* together and the port group is given a unique name. A port group can be made up of other port groups. Port groups also have two other attributes: *Direction* and *Exchange_Type*. The direction attribute can be INPUT or OUTPUT. It is used to verify that input port groups are not accidentally assigned to FTSS_OUTPUT windows and vice versa. The Exchange_Type attribute can be ONE_ROUND or TWO_ROUND. It is used to tell FTSS what kind of exchange to perform using the port group.

Since ports are specific to a partition (e.g. FTSS_INPUT partition), so are the port groups. Once the port groups are defined in the partition configuration data, the port group names can be assigned to individual FTSS I/O windows in the partition schedule. The FTSS core OS tasks know which schedule window they are currently in and can examine the partition schedule in the configuration tables to determine which port group to use for a particular input or output exchange.

In addition to the port name, a *Source Channel* parameter is also associated with each port in a port group. The source indicates which computer channel is responsible for sending the data. For example, in the channelized bus architecture of Figure 3.3-1, MIMU1 comes from CH1, MIMU2 comes from CH2 and MIMU3 comes from CH3.

As an example, we consider the GN&C sensor input ports defined in Table 4.6-2 for the FTSS_INPUT partition. The first three rows are for the MIMU sensors required by the 50 Hz CN process. These are the sensors that need to be exchanged in the first FTSS_INPUT window in Figure 4.5-5. The next six rows in Table 4.6-2 are the GPS and altimeter sensors required by the 1 Hz GN process. They need to be exchanged during the second FTSS_INPUT window in Figure 4.5-5. Thus, two port groups are created for the FTSS_INPUT partition as shown in Table 4.6-7.

Port Group Name : GNC_50Hz_Inputs Direction : INPUT Exchange_Type : TWO_ROUND		
	Port Name	Round One Source Channel
CH1 Round One Msg {	MIMU1	CH1
CH2 Round One Msg {	MIMU2	CH2
CH3 Round One Msg {	MIMU3	CH3

Port Group Name : GNC_01Hz_Inputs Direction : INPUT Exchange_Type : TWO_ROUND		
	Port Name	Round One Source Channel
CH1 Round One Msg {	GPS1	CH1
	ALTIM1	CH1
CH2 Round One Msg {	GPS2	CH2
	ALTIM2	CH2
CH3 Round One Msg {	GPS3	CH3
	ALTIM3	CH3

Table 4.6-7 FTSS_INPUT Sensor Port Groups For GN&C Partition

This FTSS_INPUT port group configuration data will be the same on all three channels. Note that Table 4.6-7 is part of the FTSS_INPUT partition configuration data and not the GN&C partition configuration data. User partitions do not need to care about port groups. The ports groups are only used by the FTSS I/O core OS tasks.

Figure 4.6-2 shows how the two port groups in Table 4.6-7 are assigned to the partition schedule. The GNC_50Hz_Inputs port group is assigned to the first FTSS_INPUT partition window in the partition schedule (window index = 1). The GNC_01Hz_Inputs port group is assigned to the second FTSS_INPUT partition window in the partition schedule (window index = 4). When the ftss_input task is activated, it has access to the current schedule window index. It uses this index to examine the partition schedule table to determine which port group to use for the two round input exchange. It

obtains the port group details from the FTSS_INPUT partition port group configuration table.

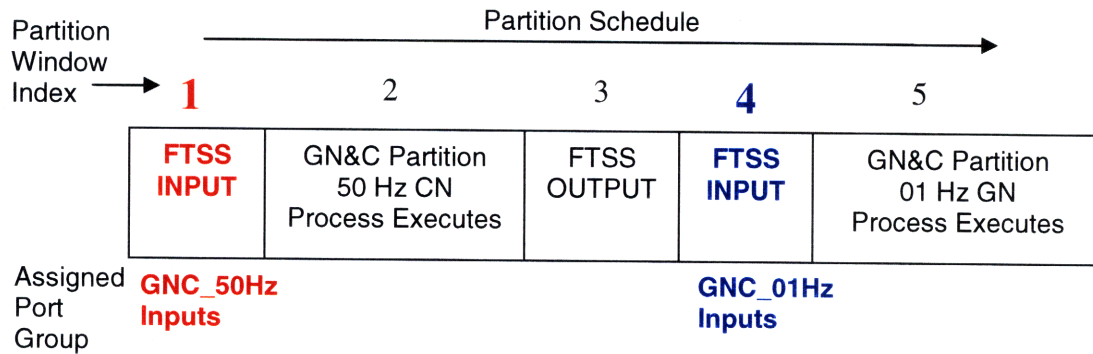


Figure 4.6-2 Port Groups Assigned to Schedule Windows

It is important to note that the *order* of the ports in a port group is important. Ports must be ordered by the source channel parameter. All CH1 ports must come first, followed by all CH2 ports followed by all CH3 ports. This is so that FTSS knows how to build round one and round two messages using data received over the CCDLs.

For example, consider a two round input exchange using the GNC_01Hz_Inputs port group in Table 4.6-7. On CH1, the ftss_input task will loop through the port group until it finds the first port whose source indicates CH1. In our case, this is the first row in the GNC_01Hz_Inputs port group corresponding to GPS1. CH1 then uses the port details in Table 4.6-2 to obtain the GPS data using the data bus I/O driver Read Function. It does the same for ALTIM1, the next port in the port group with source channel equal to CH1. It uses this local GPS1 and ALTIM1 data to build the round one message sent to CH2 and CH3.

On CH2, the ftss_input task will also loop through the port group until it finds the first entry whose source is CH2. In this example, that is GPS2. The next row, ALTIM2, also has a source of CH2 and so ftss_input on CH2 will build a round one message using local GPS2 and ALTIM2 data. This round one message is then sent to CH1 and CH3. The ftss_input task on CH3 uses the port group data in a similar fashion to send a round one message with GPS3 and ALTIM3 data to CH1 and CH2.

After sending the round one messages, `ftss_input` on all three channels will wait to receive round one messages from the other channels via the CCDLs. When `ftss_input` receives the round one messages from other channels, it knows what data is in each message by looking at the port group. For example, it knows that the CH2 round one message contains GPS data followed by ALTIM data, not the other way around. This becomes important when the number of sensors is not symmetric with respect to the number of computer channels, as will be explained in the next section.

Once all round one messages have been received, the `ftss_input` task on each channel builds the round two messages using the port group in Table 4.6-7. For example, consider the round two message from CH1 to CH2. The `ftss_input` task on CH1 will place its own local GPS1 and ALTIM1 data in the first segment of the round two message, followed by the round one data from CH3 (i.e. GPS3, ALTIM3). For the message from CH1 to CH3, the `ftss_input` task on CH1 will place its own local GPS1 and ALTIM1 data in the first segment of the round two message, followed by the round one data from CH2 (i.e. GPS2, ALTIM2). The `ftss_input` task on the other channels will build round two messages in a similar fashion.

Each channel then sends its round two message to the other two channels and then waits to receive round two messages. After round two, each channel will have three sets of data: one set from round one and two sets from round two. The order of the data is the same as the `GNC_01Hz_Inputs` port group except that the local channel data is not present since it is not voted. Only the data from other channels will be present in each data set. Once the round two messages are received, the `ftss_voter` uses the port group to vote individual fields in the three copies of data. The `ftss_voter` uses the port group to determine the offset of each data item (e.g. GPS2, ALTIM3) in the messages.

Figure 4.6-3 shows the three data sets that should be present on CH1 after the two round exchange just prior to voting. After the vote there will only be one set of data which will be written to the 01 Hz destination ports of the GN&C partition, shown in Table 4.6-1.

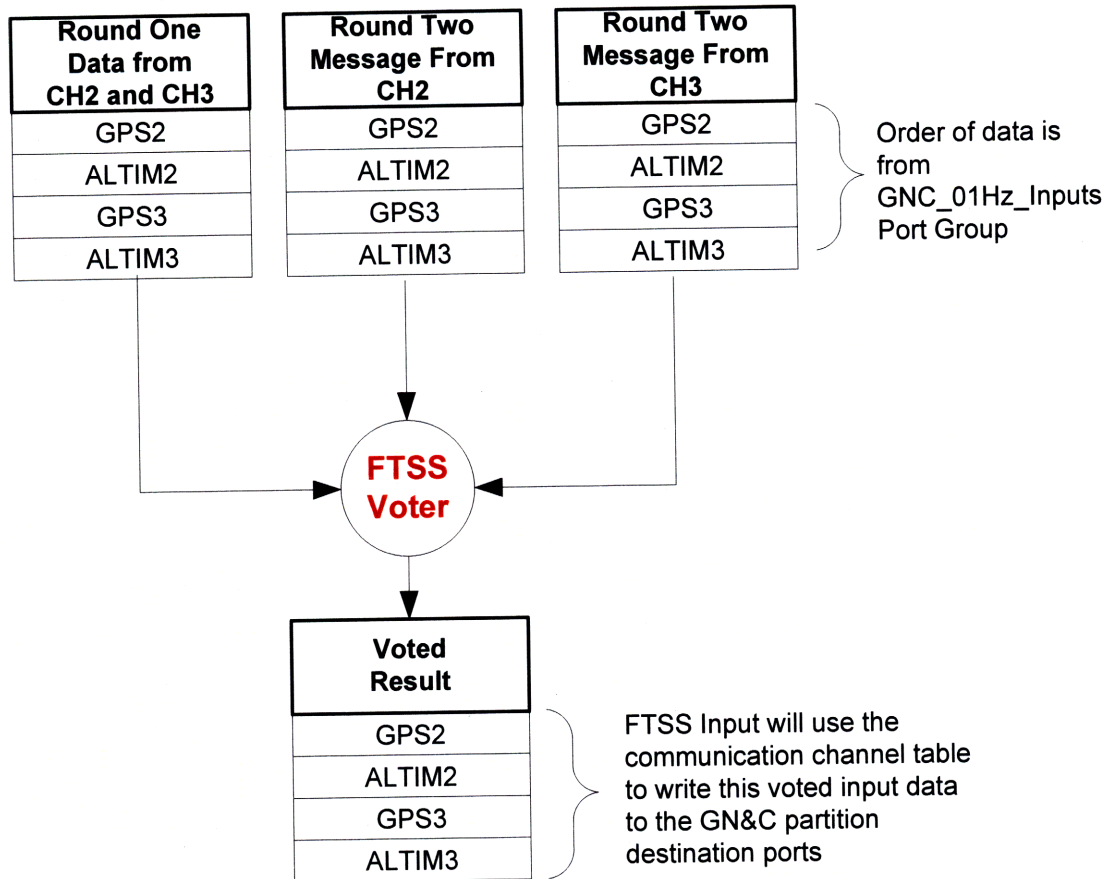


Figure 4.6-3 CH1 GNC_01Hz_Input Voting

4.6.4.2 Handling Asymmetric I/O

Until this point, the channelized system examples have assumed symmetric inputs and outputs. That is to say, the number of redundant sensors and actuators matches the number of computing channels. For example, Figure 3.3-1 shows three computing channels, three redundant MIMUs, three redundant GPS units and three redundant altimeters. This symmetry makes setting up the ports, communication channels and port groups easier. However, it is entirely possible that the number of redundant sensors or actuators may not match the number of computing channels.

We consider an example where there are only two MIMU units; one connected to CH1 and one connected to CH2. CH3 does not have a MIMU connected to its data bus. The GN&C partition is still replicated on all three channels and the 50 Hz CN process on

each channel still requires the MIMU1 and MIMU2 inputs. Thus, the GNC_50Hz_Inputs port group from Table 4.6-7 will now only have the entries in Table 4.6-8.

Port Group Name : GNC_50Hz_Inputs Direction : INPUT Exchange_Type : TWO_ROUND		
	Port Name	Round One Source Channel
CH1 Round One Msg {	MIMU1	CH1
CH2 Round One Msg {	MIMU2	CH2
NO Data from CH3 { During Round One		

Table 4.6-8 Example of Asymmetric Inputs

When the ftss_input task executes on each channel, it will determine that it should use the GNC_50Hz_Inputs port group for a two round exchange. CH1 will loop through the port group and will send MIMU1 as its round one message. CH1 will send this round one message to CH2 and CH3. CH2 will loop through the port group and send MIMU2 as the round one message to CH1 and CH3. However, CH3 will loop through the port group and will not find any ports whose source channel parameter is set to CH3. Thus, CH3 will determine that it does not need to send a round one message. CH1 and CH2 will also notice that CH3 does not have a round one message and will not wait to receive one from CH3. CH3 will receive the round one messages from CH1 and CH2 and will build the appropriate round two messages. The round two exchange will then take place as expected. Thus, port groups can be used to implement asymmetric inputs and outputs.

4.6.5 Summary

Assigning port groups to schedule windows allows a clear and explicit way of selecting APEX ports which need to be exchanged and voted. By placing this data in the XML system configuration files at design time, *offline analysis tools* can be created to determine if there will be sufficient processor throughput and CCDL bandwidth to perform the exchanges within the allotted FTSS time windows in the partition schedule.

Note that the original port configuration tables are not modified in any way. Instead, the port group table is a new table added to the FTSS partition configuration data. Thus, port groups do not violate any aspect of the existing ARINC 653 standard. Instead, they are an extension to the XML system configuration files and should be considered for inclusion in future releases of the ARINC 653 standard.

4.7 FTSS Voter

The FTSS voter is a C library function call. The FTSS input and output tasks use the voter. The function expects three data buffers since the P-NMR prototype currently has three channels. Each data buffer corresponds to a port group. An individual signal (e.g. altitude, roll rate) in a buffer corresponds to a single port in the port group and can be comprised of one or more 32 bit words. However, when voting an individual port in the group, the data is voted on a 32 bit word basis. The voter knows where each port begins and ends in the port group buffer by using the port group definition in the XML files. Each set of three 32 bit words are voted using the “Fast Majority Vote” algorithm described in [50].

Chapter 5

P-NMR System Performance Results

5.1 Overview

All results are for the TMR instance of the P-NMR architecture. The SBC750GX system clock (SYSCLK) was operating at 133 MHz and the PowerPC 750GX was operating at 933 MHz. The C code had not been optimized using the Wind River supplied C compiler.

5.2 Data Collection Methods

Each channel (i.e. SBC750GX board) has two connections to the development and test Windows PC: an Ethernet connection and an RS-232 connection. The Ethernet connection is used by the Wind River Workbench Debug and System Viewer tools on the PC. The RS-232 connection is used for logging text output from a task or process running on a channel in a terminal window on the PC.

A certain amount of care is required when designing a method for collecting timing and performance data. This is because certain time related data points are small (less than one microsecond) and an overly obtrusive data collection mechanism can greatly influence the observed timing of the system. Thus, three methods were used at various stages of development for collecting performance data.

First, test data structures and code were added to the software running on each module. This was done at the user and core OS levels. The Workbench Debug tool was then used to stop the execution of a task or process when the data structure had been filled with measurement data. This data structure was then exported to Excel. This test code could easily be deactivated using the C compiler preprocessor directives.

Second, the Wind River System Viewer tool was used. This tool executes on the PC and communicates with each channel using test Ethernet connections. These test connections are completely separate from the CCDLs. The System Viewer tool can collect detailed event data such as when a kernel tick occurs, when a core OS task

executes and when a user partition executes. The collected data is then sent to the PC where it was exported to Excel for analysis.

Finally, text output statements were added to the software under test which printed various parameters to the PC terminal window on a delayed basis so as not to affect timing. The terminal window was then used to log data to a file over several seconds.

5.3 CCDL Gigabit Ethernet Performance

Table 5.3-1 provides an example of the CCDL Gigabit Ethernet performance. The CCDL code and the GbE driver in the SBC750GX BSP operate in polled mode, not interrupt mode. The smallest packet that can be sent and received by the Ethernet hardware is 40 bytes and so this is the size that is used for the clock synchronization messages (i.e. the INIT and ECHO) and it is the size used for this example test. The test involves sending a 40 byte packet from one GbE port to another GbE port on the same SBC750GX board. Using the same board facilitated the measuring of latency since the send and receive code resided in the same core OS task which executed on one PowerPC. The core OS task performed a send on one port and then simply started polling the other port until the DMA receive engine indicated fresh data. The TB register was used to timestamp the send and receive.

To confirm the accuracy of these results, another test was performed involving two boards. Board one would send a packet to board two. As soon as the packet is received on board two, it sends a packet back to board one. Board one records the total time between the send and the receive. This test confirmed the single board results described in this section.

The results in Table 5.3-1 include the time required to move the data from a core OS FTSS buffer in SDRAM to the DMA buffers in the Marvell internal memory and vice versa. The size of FTSS data moved is 4 bytes. Note that these results do not include any time required to move data from/to a user partition. All byte copies are done in the core OS in supervisor mode. Time required to move data between the core OS and a user partition is described in section 5.7.

All the numbers in the table are averages for 100 test cases. The deviation between each test case was insignificant. To summarize, it took approximately 4.685 μ s to send 40

bytes (using 4 bytes of data) from one channel to another. Since this message size is used for the frame synchronization INIT and ECHO messages, this is also the **t_del** time.

Activity (function call)	Duration (μ s)	Comment
Poll Send	0.555	Time to move data from FTSS buffer to DMA buffer
Poll Rcv (empty)	0.197	Time to verify that no new data is present
Total Empty Rcv	2.165	11 poll rcvs occur before fresh data received
Poll Rcv (new data)	1.750	Time to move new data to FTSS buffer
TOTAL	4.685	t_del for 40 bytes send and receive (4 byte user payload)

Table 5.3-1 CCDL Gigabit Ethernet Latency (t_del for 40 bytes)

5.4 Time to Process One Kernel Tick

The FTSS_SYNC partition window is the last partition window in the major frame. Preceding this window was some spare time. Recall that the FTSS_SYNC partition does not have APEX processes that run in user mode. It only has the ftss_sync task that executes in supervisor mode in the core OS.

Test code was added at the beginning of the core OS ftss_sync task to record the value of the DEC register. This value should be reasonably constant and indicates the time between reloading the DEC register by the interrupt handler and the execution of the first statement in the ftss_sync core OS task. This time includes the interrupt handler code which evaluates the partition schedule to determine if a new partition should execute and it includes the code which determines that the core OS ftss_sync task is eligible to run during the FTSS_SYNC partition window. However, it does not include the execution of the partition OS which would normally run to evaluate which APEX process should run next within a partition.

Different durations were observed on different channels. CH1 durations ranged between 3 and 4 μ s and CH2 and CH3 durations ranged between 4 and 5 μ s. Thus, the core OS task start time jitter was always less than 1.0 μ s. This jitter impacts the accuracy of other measurements discussed below.

5.5 Frame Synchronization Performance

For all of the sections below, the kernel was using 4000 ticks per second and so the duration between each tick (i.e. DEC interrupt) was 250 μ s.

5.5.1 Observed Relative Clock Drift

Wind River indicated that the maximum drift for the SBC750GX system clock is 100 parts per million (ppm) when operating at 133 MHz. This is the drift with respect to “true” time. When all three channels had been running for at least one hour at room temperature in the lab, the maximum observed clock drift between any two channels over a one second period was less than 0.5 μ s. This is not the drift with respect to true time but is the relative drift between any two channels. In other words, two channels could both be drifting at 100 ppm with respect to true time but they could both be slow or both be fast and thus the drift between them could be quite small.

Testing this was accomplished as follows. One channel sends a 40 byte message over the CCDL to another channel every two seconds (every 8000 ticks) according to its local clock. The send timing was accomplished by a high priority core OS task using a watchdog timer. The watchdog timer is a kernel service based on kernel ticks. The timer is evaluated every time the DEC interrupt occurs. When the timer expires, the kernel activates the send task. The first thing the send task does is record the TB register value and then it sends a message over the CCDL. It then sets the watchdog timer to go off in another 8000 ticks (2 seconds).

Unfortunately, as mentioned in the previous section, the time it takes for the core OS task to start executing after the DEC interrupt is variable (± 1 μ s) and different on each channel. Thus, the recording of the TB register value will also be variable due to this start time jitter. This makes measuring the drift more difficult since the drift is still small over a two second period (longer periods were also used). Some of this effect is negated by swapping the send direction (the sender becomes the receiver) and recording the times again.

On the receiving channel, a high priority core OS task would simply loop forever, polling the CCDL receive buffers. As soon as a message is received, the TB register

value would be saved. The time between two sends (2.0E6 μ s) on one channel should be the same as the time between two receives on another channel. Table 5.5-1 shows the observed differences between the channels for a two second interval. Hundreds of measurements were taken before taking the median.

Comparison for two second interval	μ s
CH1 clock is FASTER than CH2 clock by	0.78
CH1 clock is FASTER than CH3 clock by	0.30
CH3 clock is FASTER than CH2 clock by	0.35

Table 5.5-1 Relative Clock Drift Between Channels

5.5.2 Accuracy of 4000 DEC Interrupts Per Second

The 4000 ticks (i.e. DEC interrupts) should take exactly 1.0E6 μ s (one second). However, there may be a small amount of error. This error is reduced through the use of the TB register discussed in section 4.4.3. When the system clock (SYSCLK) is operating 133 MHz, 1 μ s equals exactly 33.25 register decrements and so 250 μ s (one tick) equals exactly 8312.5 decrements.

However, because an integer must be loaded and because of the variable amount of time required to start executing the interrupt code, a value between 8299 and 8308 is loaded in the DEC by the interrupt handler. The value changes from one instance to the next. Using the TB register, it was determined that the error for 4000 DEC reloads per second was less than $\pm 1.0 \mu$ s. This measurement was done by recording the TB register at the start of the core OS ftss_sync task which was set to execute every 4000 ticks in the partition schedule. Again, this measurement is difficult given core OS task start time jitter.

5.5.3 Frame Synchronization Performance

The FTSS_SYNC partition is scheduled once per major frame or once per second. It is the last partition window in the major frame and has a duration of one kernel tick or 250 μ s. The scheduling of this partition causes the core OS ftss_sync task to execute during the 250 μ s.

The ftss_sync task waits until the middle of the tick (DEC register indicates 125 μ s remaining) before broadcasting the INIT message to all other channels. Table 5.5-2 shows the sequence of frame synchronization events and their durations. The synchronization algorithm starts with the broadcasting of the INIT message and ends with the reception of the final ECHO message and the DEC register adjustment. As can be seen, a “broadcast” of the INIT is not done in parallel due to the polled nature of the CCDL driver. Instead, it is done sequentially. This is why the send order is different on each channel (see Appendix C.3.1). The total time required from broadcasting the INIT message to receiving the final ECHO and accepting the frame synchronization round was 13 μ s. Thus, the expected ACCEPT time in the DEC register was $125 - 13 = 112 \mu$ s.

All three channels accepted within 3 μ s of this time. Two of the channels always accepted within 1.0 μ s of this time. Thus, the DEC register adjustment never exceeded 3 μ s. It should be noted that no test cases involved the intentional use of bad timing on one or more channels. These test cases need to be added. Finally, synchronization after system power up occurred within 10 ms after the third channel became operational.

Event	Dest/Src	μ s
send INIT to	CH2	0.550
send INIT to	CH3	0.550
wait	NA	2.000
rcv INIT from	CH3	1.850
return from check_for_msgs		0.200
send ECHO to	CH2	0.550
send ECHO to	CH3	0.550
enter check_for_msgs		0.200
rcv INIT from	CH2	1.850
wait		0.500
rcv ECHO from	CH3	1.850
return/re-enter check_for_msgs		0.200
rcv ECHO from	CH2	1.850
Total Sync Time		12.700

Table 5.5-2 CH1 Frame Synchronization Events

5.6 FTSS Input Exchange Performance

Table 5.6-1 shows two test cases (TC1, TC2) used for benchmarking the performance of the two round input exchange performed by the core OS `ftss_input` task using the CCDLs. The test cases use the two round exchange of GN&C sensor data described in section 3.3.1. Recall that this is not a true Byzantine resilient two round exchange since there are only three channels in the prototype. Also, all three sensor types (MIMU, GPS, ALTIM) are exchanged at 50 Hz as opposed to previous discussions which broke up the input exchange into 50 Hz sensors (MIMU) and 01 Hz sensors (GPS and ALTIM).

Test case one (TC1) uses a port message size of 60 bytes. This is similar to the size of messages used for benchmarking the performance of the X-38 FTTP. Test case two (TC2) uses 112 bytes per port message. Given the three sensor types (MIMU, GPS, ALTIM), 112 bytes is the maximum bytes per sensor (i.e. port) that can be exchanged during the 250 μ s `FTSS_INPUT` partition window in the schedule. A duration of 250 μ s is one kernel tick (there are 4000 ticks per second). It should be noted that in a real system, different sensor ports will be of different sizes.

The table does not include the time to move the sensor data from the data bus I/O driver memory to the SBC750GX memory since the data bus is simulated. The first section of the table deals with the round one exchange. The second section deals with the round two exchange, the voting and the writing of voted data to kernel port memory where it will be read later by the GN&C partition. The entire table corresponds to activities that take place in the core OS. There is no interaction with the user GN&C partition.

In summary, the 60 byte per sensor port test case uses just over half (136 μ s) of the 250 μ s partition window and the 112 byte test case uses the entire window (243 μ s). In both cases, the time to vote the final round two data is significant, approximately 23%.

	Units	TC1	TC2	Comments
Sensor Input Rnd 1 Exchange				
Number of sensor inputs per channel	num	3	3	e.g. MIMU1, GPS1, ALTIM1
Bytes per input port	bytes	60	112	
Round 1 bytes per CCDL msg	bytes	180	336	= num ports x bytes per port
CCDL Broadcast to ALL	μs	8.8	14.7	
Single CCDL read	μs	9.1	15.3	
Total Time for Rnd 1 Exchange	μs	29.2	52.9	Does NOT includes I/O memory reads
Sensor Input Rnd 2 Exchange				
Round 2 bytes per CCDL msg	bytes	540	1008	3 channels
CCDL Broadcast to ALL	μs	22.3	39.9	
Single CCDL read	μs	23.5	42.2	
Total for CCDL exchange	μs	71.5	128.8	
Time to vote	μs	30.2	55.7	
Total number of ARINC 653 input ports	num	9	9	Each CH gets ALL redundant sensors
Single kernel port write	μs	0.5	0.6	FTSS is writing to kernel memory, not user memory
Total for all port writes	μs	4.5	5.4	
Total Time for Round 2 Exchange and Vote	μs	107.0	190.2	Data not yet in partition memory
Total for Two Round Exchange and Vote	μs	136.2	243.1	

Table 5.6-1 FTSS Input Performance

5.7 APEX Port System Call Performance

Table 5.7-1 shows the performance of the APEX Port system calls used by the GN&C user partition. This table is a continuation of the FTSS Input table above and uses the same two test cases. The first section of the table shows how long it takes the GN&C 50 Hz CN process to read all nine input ports (3 redundant values each for MIMU, GPS, ALTIM sensors). The next section of the table shows how long it takes the CN process to write the three actuator commands to the three APEX output ports (Thruster1, Thruster2, Thruster3). This section of the test did not use different sizes for the output ports for each test case.

	Units	TC1	TC2	Comments
GET SENSOR INPUTS				
Number of APEX input ports	num	9	9	Each CH reads ALL redundant sensors (3 x MIMU, GPS, ALTIM)
Bytes per input port	bytes	60	112	
Single APEX port read	μ s	3.5	3.5	System Call - moves data from kernel to partition memory
Total for all input port reads	μ s	31.5	31.5	
SEND CMD OUTPUTS				
Number of APEX output Ports	num	3	3	
Bytes per output port	bytes	20	20	
Single APEX port write	μ s	3.1	3.1	System Call - moves data from partition to kernel memory
Total for all output port writes	μ s	9.3	9.3	
Total time spent on APEX Port I/O	μ s	40.8	40.8	Includes Input Reads and Output Writes

Table 5.7-1 APEX Port System Call Performance

As can be seen from the “Single APEX port read” row, the time it takes for the APEX port read system call to move data from core OS memory up into the user partition memory does not vary when the size of the data is less than 112 bytes. It took 3.5 μ s to read 60 bytes and 112 bytes. The majority of the 3.5 μ s is taken up by the handling of the PowerPC exception when switching between user mode and supervisor mode and vice versa. Once the mode switch has completed, the actual byte copy is fast and is not a factor at these message sizes.

The total time spent by the 50 Hz CN process interacting with APEX ports is 40.8 μ s which is not insignificant given that the CN process only has a 2000 μ s time capacity to complete its work. These results indicate that it may be more efficient to have FTSS bundle different sensors into a few large APEX input ports rather than using many smaller ports. The 40.8 μ s may be longer than the final P-NMR implementation since these test cases have the 50Hz CN process reading MIMU, GPS and ALTIM sensors when in reality it would probably only read the 50 Hz sensors (i.e. only MIMU).

5.8 FTSS Output Exchange Performance

Table 5.8-1 shows the single test case (TC1) used for benchmarking the performance of the one round output exchange performed by the core OS ftss_output task using the CCDLs. This table is a continuation of the APEX Port System Calls table above. Only one output port size (20 bytes) was used by the GN&C user partition when writing to the three actuator command output ports. The table does not include the time required to move voted data to the data bus I/O driver memory. The results in this table will be proportional to the FTSS input round one exchange since they use the exact same exchange method. The time to perform the vote is 17% of the total time.

	Units	TC1	Comments
Kernel Output One Round Exchange			
Number of ARINC 653 output Ports	num	3	
Bytes per output port	bytes	20	
Single kernel ARINC 653 port read	μs	0.4	FTSS is reading kernel port memory
Total for all port reads	μs	1.2	
Round 1 bytes per CCDL msg	Bytes	60	= num ports x bytes per port
CCDL Broadcast to ALL	μs	4.6	
Single CCDL read	μs	4.4	
Total for CCDL exchange	μs	17.2	
Time to vote	μs	3.7	
Total Time for One Round Exchange	μs	21.2	

Table 5.8-1 FTSS Output Performance

5.9 Comparison with X-38 FTPP

The P-NMR FTSS input and output exchange numbers compare favorably with the X-38 FTPP system. Recall that the FTPP uses custom FPGA hardware (the Network Element) to perform frame synchronization, data exchange and voting. It also has an FTSS software component executing on a PowerPC that interacts with the Network Element. The FTPP 50 Hz flight critical sensor input exchange using message sizes similar to the P-NMR test case one (TC1) takes between 444 and 792 μs. The P-NMR TC1 input exchange takes 136 μs. The FTPP 50 Hz output exchange takes between 120

and 262 μ s whereas P-NMR takes 21 μ s. On FTTP, the time between the interrupt for starting the sensor input exchange to the time the first user GN&C process starts executing on the processor ranged from 433 to 531 μ s. On P-NMR, the worst case delay is 286 μ s. This duration comes from three sources: the 250 μ s (one kernel tick) FTSS_INPUT partition window; the 4 μ s at the start of the GN&C partition window used by the partition OS (vThreads) to start the 50 Hz CN process; and the 32 μ s used by the CN process while it reads nine APEX input ports (3 sets of MIMU, GPS, ALTIM). Thus, the P-NMR total is $250+4+32=286$ μ s.

Although the P-NMR numbers appear promising, certain factors make a direct comparison with FTTP difficult. The FTTP numbers are for a four channel system. The P-NMR prototype has only three channels. The P-NMR prototype uses a sensor input two round exchange that mimics the amount of data that would be exchanged if four channels were present. This exchange process is not the same as the FTTP system. The P-NMR prototype uses 1000 Mbps Ethernet whereas FTTP uses 125 Mbps optical fiber. The P-NMR PowerPC 750GX was operating at 933 MHz whereas the FTTP PowerPC 604e was operating at 300 MHz. It is likely that a final P-NMR system will reduce the clock speed in order to reduce slightly off specification timing problems. The FTTP Network Element hardware and the FTTP version of FTSS perform the complete set of redundancy management functions including group membership and channel resets. The P-NMR version of FTSS is not yet complete and does not perform all of the FTTP functions. Finally, FTTP limits messages to 64 bytes whereas P-NMR can send a single Ethernet packet that is several thousand bytes long.

This page intentionally left blank

Chapter 6

Conclusions

6.1 Use of ARINC 653 with an NMR Architecture

The P-NMR prototype demonstrates that the use of robust partitioning, as defined in the industry accepted ARINC 653 standard, is compatible with a classical, frame synchronous NMR architecture that implements distributed voting for flight critical applications. The idea of hardware fault containment regions in NMR avionics systems is not new and has been well documented in the literature. However, the addition of *software* fault containment regions, implemented as ARINC 653 partitions, to a traditional NMR hardware architecture can increase the safety and reliability of the overall system. Robust time, space and I/O partitioning greatly reduces the probability that one software application will be able to adversely affect another application executing on the same processor within the operational group.

The P-NMR fault tolerance and redundancy management functions are implemented in software in the FTSS component which resides in the module core OS. FTSS interacts with the COTS ARINC 653 kernel which in turn interacts with the user partitions. This approach allows user application partitions to utilize the industry accepted APEX API instead of a custom fault tolerant services API. Thus, user applications are completely isolated from the fault tolerance design. The application developers can view the system as a simplex computer executing only one instance of their application. They do not need to be concerned with how many redundant channels there are or how other avionics hardware elements are physically connected to the channels. The avionics architecture could change from a channelized data bus approach to a global data bus approach without any user application code changes. Use of the APEX API also means that applications are not tied to a specific RTOS vendor. This leads to software that is portable, reusable and modular which facilitates enhancements and maintenance. This is crucial for a large project such as Constellation which could be active for several decades.

The ARINC 653 fixed, cyclic partition schedule is synergistic with the idea of NMR frame synchronization. It allows the FTSS sync task in the core OS to be reliably scheduled within the major frame. The execution of the frame synchronization algorithm does not adversely impact ARINC 653 robust time partitioning. The adjustment of the PowerPC DEC register at the end of each major frame does not steal any time from any user APEX process or violate the partition schedule in any other manner. However, it does introduce a small amount of jitter ($< 3 \mu\text{s}$) to the start time of the first process in the next major frame.

The ARINC 653 standard defines a two level scheduling design. First, partitions are scheduled on a fixed, cyclic basis by the core OS. Then, APEX processes within a partition are scheduled on a priority basis by each partition OS. This design is compatible with the scheduling of fault tolerant I/O processing to support user applications. The system integrator can explicitly define FTSS I/O processing in the partition schedule. However, this explicit method of scheduling may require that some large user partition windows in the schedule be split up into smaller windows that correspond to when different APEX processes within the partition execute. This requires careful analysis of process release points and deadlines by the system integrator. Another disadvantage of this method is that there may be an increase in system response times due to processor idle time in FTSS and user partition windows. However, the P-NMR prototype demonstrated that a significant amount of data (336 bytes) could be exchanged and voted by FTSS within a 0.25 ms partition window. The total FTSS I/O overhead added to the execution time of the GN&C partition was 0.5 ms.

A significant portion of the ARINC 653 standard and APEX interface specification is devoted to inter-partition communication using ports. The P-NMR prototype has demonstrated that APEX ports can be applied to sensor input congruency and application output voting. By setting up the appropriate ports, port groups and communication channels between FTSS system partitions and user application partitions, the system integrator can easily accomplish fault tolerant I/O for flight critical applications that are replicated on all channels. By using APEX ports, user applications are isolated from the fault tolerant design and physical layout of the redundant avionics hardware. The port

groups can be assigned to specific FTSS windows in the partition schedule such that it is clear what data is being exchanged during the major frame.

The ARINC 653 standard also provides an example of how XML system configuration files can be used to specify many aspects of the system architecture such as the partition schedule, ports and communication channels between ports. This specification can be done early in the development lifecycle, before actual flight software is ready to be integrated on the hardware. Using a structured syntax for the system design data allows for the creation of useful offline analysis tools that can identify system performance problems early in the project schedule. System configuration files can help manage system complexity and they can facilitate a successful system integration and test phase.

Finally, the use of the ARINC 653 standard and the APEX API facilitates the integration of flight control computers and mission management computers into a single set of VMCs. Robust partitioning allows both flight critical and non-flight critical software to be hosted safely on the same processor within a single VMC channel. The partition schedule does not have to be identical on all VMCs. Non-flight critical partitions can be unique to a single VMC. Only the flight critical partitions need to be aligned within the major frame. By integrating the FCCs and MMCs into one set of VMCs, the avionics system can be reduced in terms of size, weight, power dissipation and wiring complexity. These reductions are extremely important for any vehicle being launched into space.

6.2 Use of COTS

6.2.1 COTS ARINC 653 RTOS and DO-178B Certification

It is possible that NASA may be willing to use certain elements of the DO-178B standard for the development of flight software on some Constellation projects. If this is the case then a significant cost advantage can be achieved through the use of a COTS ARINC 653 RTOS that comes with a DO-178B certification package. This package, provided by the RTOS vendor, contains all the necessary documentation that can be used when certifying the avionics system with the vehicle. The documentation includes the

evidence that the vendor followed the appropriate software development processes and that all objectives for each process were met (e.g. requirements were peer reviewed, code was unit tested, MC/DC results were obtained). The package may also include certification evidence for development tools such as XML parsers, compilers and linkers.

Although expensive, the purchase of a certification package would mean that a Constellation avionics project would not have to spend any time or resources on verifying and validating the RTOS software which could contain more than 30k SLOCs. This is extremely important for DO-178B Level A software which has the highest number of objectives that must be met. Thus, any modification of the COTS core OS by the customer invalidates some of the verification effort already performed by the vendor and is undesirable.

A disadvantage of the current P-NMR design is that the FTSS component is at the core OS level and not at the user partition level. Integrating FTSS with the VxWorks 653 kernel required changing one kernel source file dealing with the PowerPC DEC register and two BSP source files dealing with memory layout and Gigabit Ethernet driver parameters. Even if no changes were made to core OS kernel source files, the addition of user defined high priority tasks, like the FTSS tasks, in the core OS could invalidate some of the verification done by the vendor due to possible changes in kernel task scheduling.

Finally, the impact of BSP changes on the kernel certification package is not clear. Some COTS RTOS vendors claim that their kernel certification package is valid for several different BSPs. The RTOS vendor provides a standard set of test cases which the BSP must pass. Alternatively, some board manufacturers provide a BSP certification package that is claimed to be compatible with the kernel certification package. In any event, some kernel re-validation will be required if a standard BSP is customized for a space qualified processor board.

6.2.2 COTS ARINC 653 RTOS Performance

There was concern at the start of the P-NMR project that the performance of a COTS ARINC 653 RTOS would be inadequate to support strict timing goals. One option that was contemplated involved writing in-house code that implemented the APEX API but that was also optimized for performance. However, preliminary results with the

P-NMR project indicate that the performance of the COTS ARINC 653 product, including the APEX shared library, is sufficient when running on a PowerPC operating at 933 MHz. Handling of DEC interrupts takes less than 6 μ s. Moving data to and from the core OS using APEX ports takes less than 4 μ s for messages less than 200 bytes. Start time jitter for core OS tasks was less than 1 μ s. More testing is required to evaluate the time to switch to a new partition.

6.2.3 COTS Hardware for Fault Tolerance

The X-38 FTPP uses custom hardware to implement some fault tolerance functions such as clock synchronization and voting. On the other hand, the P-NMR system uses COTS hardware such as a PowerPC processor and Gigabit Ethernet (IEEE Std 802.3) hardware for the CCDL. The P-NMR fault tolerance functionality is implemented entirely in software running on the PowerPC.

The Gigabit Ethernet hardware (PHY, MAC and DMA engines) support small latencies and sufficient bandwidth for the CCDL. A 40 byte packet takes less than 5 μ s to go from the sender's core OS memory to the receiver's core OS memory. However, more research is required in the area of evaluating Gigabit PHY and MAC devices in a radiation environment.

The use of the PowerPC DEC and TB registers provide a solid basis for reasonably accurate frame synchronization. However, more P-NMR testing is required where one or more channels are intentionally setup to have off-nominal timing. Finally, the PowerPC processor has been used extensively in space applications although not at 933 MHz clock speeds.

Even with the differences between the X-38 FTPP system and the P-NMR prototype that make a direct comparison difficult, the initial performance results tend to indicate that the use of COTS hardware is a viable option for a fault tolerant avionics architecture.

6.3 Future Work

6.3.1 FTSS Improvements

The complete set of fault tolerance and redundancy management functions could be added to the P-NMR FTSS software component. For example, FTSS could implement a group membership algorithm. This would involve keeping track of the number of faults experienced by each channel. When the number of faults for a channel exceeds some threshold, the channel should be removed from the operational group and prevented from participating in future voting activities. Each instance of FTSS on each channel would exchange system status messages with the other FTSS instances so that the channels could agree on which channels are healthy and which channels have faults. The exchange of this system status data would allow the healthy channels to agree on which channel should be reset.

The frame synchronization algorithm could be upgraded to allow a channel that has been reset to become synchronized with the operational group. The operational group could then monitor the outputs being produced by the channel that was reset. Once the channel has provided correct outputs for a certain duration of time, it could be readmitted into the operational group.

Once the full set of FTSS functionality is implemented then benchmarking could continue in the lab with a four channel P-NMR system instead of the current three channel system. Four channels would mean that the current sensor input two round exchange design could be modified to use a traditional Byzantine fault tolerant design. These performance results should then be compared again with the X-38 FTTP performance results.

In addition to the above, all the VxWorks 653 specific kernel calls that are located in various parts of the FTSS code could be moved to a single low-level FTSS library file with a standard interface. All other FTSS code could then be written in ANSI C except for the use of the low-level library which contains RTOS specific function calls. This would allow an easier migration of FTSS to a different COTS ARINC 653 RTOS. Only the bodies in the low level FTSS RTOS library file would need to change.

6.3.2 Software Fault Containment Regions

The notion of software fault containment regions could be investigated further. In traditional NMR systems, when voting of application outputs indicates that a processor module has produced an incorrect value, the entire hardware module or channel is reset. However, with software partitions, it may be beneficial to only reset the offending partition while other partitions are allowed to continue to execute on the processor. This may make sense if the memory area of just one partition has been corrupted (e.g. by a radiation event). Only the memory of the that partition would need to be reset. An advantage of a software partition reset is that it is faster than a hardware reset of the entire module or channel.

However, if all other partitions that remain running need the outputs from the partition being reset then there is limited benefit to resetting just one partition on the module. Also, a memory corruption could have occurred in kernel memory space but it may only be affecting a single user partition. Resetting that single partition would not remove the fault in the kernel memory. For example, the FTSS port groups are in kernel memory and are for all user partitions. Parts of this memory could be corrupted.

Along these same lines, the use of ARINC 653 Health Monitoring services could be integrated with the FTSS software. The standard describes a simple and efficient way of managing errors. Errors can be handled on a process, partition or module level. Different error response mechanisms are available at each level. Errors that are not handled are passed up to the next level. All error responses can be defined in the XML system configuration files.

Partitions could also be used to prevent faults due to compiler or linker defects. Two or more partitions on the same processor could execute the exact same source code. However, each partition would use a different compiler and linker from different vendors. The outputs of the partitions could then be compared at run-time. This is a cheaper but less robust solution to using different processor families within the channel.

6.3.3 Scheduling

More research is required on the use of different partition schedules for each mission mode such as ascent, on-orbit, de-orbit, entry and descent. Part 2 of the ARINC 653 standard allows multiple partition schedules to be defined for the same module. The P-NMR system currently uses different schedules to accomplish module cold start. However, different schedules could also be defined for different mission modes. In this case, the time at which resources are created, such as partitions, processes and ports, needs to be carefully considered. Also, the method by which schedule transitions occur at run-time needs to be tested.

The notion of dynamic software reconfiguration could also be integrated into the P-NMR system. In order to reduce weight, it is likely that Constellation vehicles will use computer channels that integrate both the flight control and mission management functions. As discussed in section 4.5.5, some mission applications may not be critical enough to justify replication on all channels. They may be hosted on only one channel. If that channel experiences a permanent fault, it would be desirable if the mission management application could be migrated to another operational channel. This is especially useful on long missions where permanent hardware failures are possible. Dynamic reconfiguration would increase the chance of mission success.

A drawback of dynamic reconfiguration is that it involves some inherent complexities. Starting a new software partition on a channel that is already in the operational group is problematic. Either the existing schedule has pre-defined spare time or it needs to be altered to accommodate the new partition without affecting existing synchronization between the operational channels. The other option is to remove the healthy channel from the group while it is being reconfigured with a new partition schedule.

Permanent hardware faults are not the only reason to consider software reconfiguration. It should also be considered for the case where the vehicle itself is reconfigured. For example, a vehicle with three redundant channels may dock in orbit with another vehicle which also has three redundant channels. Each vehicle would have sensors and actuators. A robust avionics design would allow the computer channels on one vehicle to control the sensors and actuators on the other vehicle. In addition, it may

be beneficial to have the six channels form a single operational group for fault tolerance purposes. This system could still tolerate one Byzantine fault even after one of the channels permanently failed. Thus, dynamic reconfiguration is applicable to the Constellation program system-of-systems approach.

Additional work could also be done on integrating partition, backplane and data bus I/O scheduling. This is especially true if the data bus uses a table driven, time triggered approach.

6.3.4 Configuration Files and Static Analysis Tools

The ARINC 653 standard provides an example of using XML for the system configuration files for configuring the software on a processor module. This idea could be extended to certain fault tolerance and redundancy management design elements. For example, the files could specify which I/O signals (i.e. port groups) need FTSS services and when they need to be exchanged over the CCDLs during the major frame. They could specify different partition schedules for each mission mode (e.g. ascent, on-orbit) and they could specify how the avionics system should be reconfigured due to a hardware failure or due to a vehicle reconfiguration (e.g. docking).

In addition to specifying certain fault tolerance and redundancy management design parameters in the configuration files, analysis tools that use the files as an input could also be created. For example, a tool could verify that the time allocated in the schedule for FTSS exchanges is sufficient for the amount of data being exchanged and for the CCDL bandwidth. Another tool could verify that replicated flight critical partitions on each channel are properly aligned within each channel's major frame. Schedulability analysis could be performed on a processor, cabinet or system wide basis. It would be possible to calculate total system response times as well as a detailed breakdown of latencies for each subsystem or partition. Another useful tool might be one that analyzes the software partition schedule and the data bus I/O schedule in order to verify that data refresh rates will be satisfied.

This page intentionally left blank

References

- [1] "The Vision for Space Exploration," doc. no. NP-2004-01-334-HQ, NASA, Washington, D.C., Feb. 2004.
- [2] "NASA's Exploration Systems Architecture Study Final Report," doc. no. NASA-TIM-2005-21-214062, NASA, Washington, D.C., Nov. 2005.
- [3] "Human-Rating Requirements for Space Systems," NPR 8705.2A, NASA, Washington, D.C., Feb. 2005.
- [4] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382-401, July 1982.
- [5] R. Hammett, "Design by Extrapolation: An Evaluation of Fault Tolerant Avionics," *IEEE Aerospace and Electronic Systems Magazine*, vol. 17, no. 4, pp. 17-25, April 2002.
- [6] R. Charles and A. C. Brooks, "The Economic Impact of Avionics Standardization on the Airline Industry," Georgia State University, Nov. 2000.
- [7] C. Adams, "COTS Operating Systems: Boarding the Boeing 787," *Avionics Magazine*, April 2005.
- [8] Y. H. Lee, E. Rachlin, and P. A. Scandura, "Handbook For Ethernet-Based Aviation Databases: Certification and Design Considerations," DOT/FAA/AR-05/54, Federal Aviation Administration, Washington, D.C., Nov. 2005.
- [9] B. Filmer, "Open Systems Avionics Architectures Considerations," *IEEE Aerospace and Electronic Systems Magazine*, vol. 18, no. 9, pp. 3 - 10, Sept. 2003.
- [10] "Design Guidance for Integrated Modular Avionics," ARINC Report 651-1, Aeronautical Radio, Inc., Annapolis, Maryland, Nov. 1997.
- [11] J. Wlad, "A New Generation in Aircraft Avionics Design," *Embedded Control Europe*, pp. 14-17, May 2005.
- [12] C. Adams, "A380 Innovations: A Balancing Act," *Avionics Magazine*, March 2003.
- [13] "Avionics Application Software Standard Interface - Part 1 - Required Services," ARINC Specification 653-2, Aeronautical Radio, Inc., Annapolis, Maryland, Dec. 2005.

- [14] "Software Considerations In Airborne Systems and Equipment Certification," RTCA/DO-178B, RTCA, Inc., Washington D.C., Dec. 1992.
- [15] "Software Safety Standard," NASA-STD-8719.13B, NASA, Washington, D.C., July 2004.
- [16] Y. C. Yeh, "Design Considerations in Boeing 777 Fly-By-Wire Computers," *Proc. High-Assurance Systems Engineering Symposium*, IEEE, pp. 64 - 72, 1998.
- [17] D. C. Chau, "Authenticated Messages for a Real-Time Fault-Tolerant Computer System," Master of Engineering, Dept. of Electrical Eng. and Computer Science, Mass. Inst. of Tech., Cambridge, Mass., 2006.
- [18] M. Paulitsch, et al., "Coverage and the Use of Cyclic Redundancy Codes in Ultra-Dependable Systems," *Proc. Dependable Systems and Networks (DSN)*, IEEE, pp. 346 - 355, 2005.
- [19] K. Driscoll, et al., "The Real Byzantine Generals," *Proc. 23rd Digital Avionics Systems Conference (DASC)*, IEEE, pp. 6D4/1 - 6D4/11, 2004.
- [20] Y. C. Yeh, "Safety Critical Avionics for the 777 Primary Flight Controls System," *Proc. 20th Digital Avionics Systems Conference (DASC)*, IEEE, pp. 1C2/1 - 1C2/11, 2001.
- [21] F. Martin and D. Adams, "Cross Channel Dependency Requirements of the Multi-Path Redundant Avionics Suite," *Proc. Aerospace and Electronics Conference*, IEEE, pp. 200 - 206, 1992.
- [22] T. K. Srikanth and S. Toueg, "Optimal Clock Synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626-645, July 1987.
- [23] J. Rushby, "A Comparison of Bus Architectures for Safety-Critical Embedded Systems," NASA/CR-2003-212161, NASA, Hampton, Virginia, March 2003.
- [24] D. W. Caldwell, "Minimalist Fault-Tolerance Techniques for Mitigating Single-Event Effects in Non-Radiation-Hardened Microcontrollers," PhD dissertation, Computer Science, Univ. of California, Los Angeles, California, 1998.
- [25] L. P. Bolduc, "X-33 Redundancy Management System," *IEEE Aerospace and Electronic Systems Magazine*, vol. 16, no. 5, pp. 23 - 28, May 2001.
- [26] R. Racine, M. LeBlanc, and S. Beilin, "Design of a Fault-Tolerant Parallel Processor," *Proc. 21st Digital Avionics Systems Conference (DASC)*, IEEE, pp. 13D2/1 - 13D2/10, 2002.
- [27] D. J. Popp and R. L. Kahler, "C-17 Flight Control Systems Software Design," *Proc. 11th Digital Avionics Systems Conference (DASC)*, IEEE, pp. 580 - 585, 1992.

- [28] "Emergency Airworthiness Directive (Boeing 777)," AD 2005-18-51, Federal Aviation Administration, Washington, D.C., Aug. 2005.
- [29] J. R. Sklaroff, "Redundancy Management Technique for Space Shuttle Computers," *IBM J. Res. Develop.*, vol. 20, no. 1, pp. 20 - 28, Jan. 1976.
- [30] L. P. Bolduc, "Clock Synchronization in an N-Modular Redundant System," *Proc. 21st Digital Avionics Systems Conference (DASC)*, IEEE, pp. 9A5/1 - 9A5/8, 2002.
- [31] M. J. Morgan, "Integrated Modular Avionics for Next Generation Commercial Airplanes," *IEEE Aerospace and Electronic Systems Magazine*, vol. 6, no. 8, pp. 9 - 12, Aug. 1991.
- [32] T. J. Redling, "Integrated Flight Control Systems - A New Paradigm For An Old Art," *Proc. 19th Digital Avionics Systems Conferences (DASC)*, IEEE, pp. 1C4/1 - 1C4/8, 2000.
- [33] B. Witwer, "Developing the 777 Airplane Information Management System (AIMS): a view from program start to one year of service," *IEEE Aerospace and Electronic Systems Magazine*, vol. 33, no. 2, pp. 637 - 641, April 1997.
- [34] P. Binns, "A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach," *Proc. 20th Digital Avionics Systems Conference (DASC)*, IEEE, pp. 1B6/1 - 1B6/12, 2001.
- [35] D. Jensen, "FBW for the S-92," *Avionics Magazine*, March 2004.
- [36] "BAE Systems CsLEOS Chosen For Advanced Flight Control on U.S. Air Force C-17 Transport," Press Release 020/2003, BAE Systems, Jan. 2003.
- [37] "Electronic architectures for UAVs: Getting it together," Smiths Aerospace, 2004.
- [38] M. F. Younis and B. He, "Integrating Redundancy Management and Real-time Services for Ultra Reliable Control Systems," Honeywell International Inc., Columbia, Maryland
- [39] R. M. Keichafer, et al., "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 398 - 404, April 1988.
- [40] R. Black and M. Fletcher, "Next Generation Space Avionics: Layered System Implementation," *IEEE Aerospace and Electronic Systems Magazine*, vol. 20, no. 12, pp. 9 - 14, Dec. 2005.
- [41] Y.-H. Lee, et al., "Scheduling Tool and Algorithm for Integrated Modular Avionics Systems," *Proc. 19th Digital Avionics Systems Conference (DASC)*, IEEE, pp. 1C2/1 - 1C2/8, 2000.

- [42] J. C. N. Ventura and J. A. S. Neves, "Generic Avionics Scaleable Computing Architecture," *Proc. Data Systems in Aerospace (DASIA)*, 1999.
- [43] R. Obermaisser and P. Peti, "A Fault Hypothesis for Integrated Architectures," *Proc. International Workshop on Intelligent Solutions in Embedded Systems (WISES)*, IEEE, pp. 1 - 18, 2006.
- [44] R. C. Ferguson, B. L. Peterson, and H. C. Thompson, "System Software Framework for System of Systems Avionics," *Proc. 24th Digital Avionics Systems Conference (DASC)*, IEEE, pp. 8A1/1 - 8A1/10, 2005.
- [45] M. Agrawal, et al., "An Open Software Architecture for High-integrity and High-availability Avionics," *Proc. 23rd Digital Avionics Systems Conference (DASC)*, IEEE, pp. 8C2/1 - 8C2/11 2004.
- [46] N. C. Audsley and M. Burke, "Distributed fault-tolerant avionic systems-a real-time perspective," *Proc. Aerospace Conference*, IEEE, pp. 43-60, 1998.
- [47] J. Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance," NASA/CR-1999-209347, SRI International, Menlo Park, CA, June 1999.
- [48] J. C. Mankins, "Technology Readiness Levels," White Paper, NASA, Washington, D.C., April 1995.
- [49] "Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications," IEEE Standard 802.3 - 2002, IEEE, New York, NY, March 2002.
- [50] R. S. Boyer and J. S. Moore, "MJRTY - A Fast Majority Vote Algorithm," Technical Report ICSCA-CMP-32, Univ. of Texas at Austin, Austin, Texas, 1982.
- [51] P. Koopman and T. Chakravarty, "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks," *Proc. Int. Conf. on Dependable Systems and Networks (DSN)*, IEEE, pp. 145 - 154, 2004.
- [52] "System Design and Analysis Working Group Report - 25.1309," Draft R6X Phase 1, Federal Aviation Administration - Aviation Rulemaking Advisory Committee (ARAC), Washington, D.C., Aug. 2002.
- [53] "Approval of Flight Guidance Systems," AC 25.1329-1B, Federal Aviation Administration, Washington, D.C., July 2006.
- [54] "System Design and Analysis," AC 25.1309-1A, Federal Aviation Administration, Washington, D.C., June 1988.

- [55] "Criteria For Approval of Category III Weather Minima For Takeoff, Landing, and Rollout," AC 120-28D, Federal Aviation Administration, Washington, D.C., July 1999.
- [56] "Guidance For Integrated Modular Avionics (IMA) That Implement TSO-C153 Authorized Hardware Elements," AC 20.145, Federal Aviation Administration, Washington, D.C., Feb. 2003.
- [57] "Guidance for the Certification of Honeywell Primus Epic Systems," ANM-02-113-016, Federal Aviation Administration, Washington, D.C., May 2003.

This page intentionally left blank

Appendix A

Acronyms

AEEC	Airlines Electronic Engineering Committee
AFDX	Avionics Full Duplex Switched Ethernet
AIMS	Airplane Information Management System
APEX	Application Executive
API	Application Programming Interface
ARINC	Aeronautical Radio Inc.
BIT	Built-In Test
BSP	Board Support Package
CCDL	Cross Channel Data Link
CN	Control and Navigation
COTS	Commercial Off The Shelf
CPM	Core Processing Module
CRC	Cyclic Redundancy Code
DEC	Decrementer register
DMA	Direct Memory Access
FAA	Federal Aviation Administration
FBW	Fly-By-Wire
FCC	Flight Control Computer
FCP	Flight Critical Processor
FCR	Fault Containment Region
FDIR	Fault Detection, Isolation and Recovery
FGS	Flight Guidance System
FHA	Functional Hazard Assessment
FMS	Flight Management System
FPGA	Field Programmable Gate Array
FTPP	Fault Tolerant Parallel Processor
FTSS	Fault Tolerant System Services
GALS	Globally Asynchronous, Locally Synchronous
GbE	Gigabit Ethernet
GN	Guidance and Navigation
GN&C	Guidance, Navigation and Control
GPS	Global Positioning System
ICP	Instrumentation Control Processor
IMA	Integrated Modular Avionics
I/O	Input / Output
IOM	I/O Module
IVHM	Integrated Vehicle Health Management
LRU	Line Replaceable Unit
MAC	Media Access Control
MIMU	Miniature Inertial Measurement Unit

MMC	Mission Management Computer
MMU	Memory Management Unit
MVS	Median Value Select
NE	Network Element
NMR	N-Modular Redundancy
OS	Operating System
P-NMR	Partitioned NMR
PPC	PowerPC
PPS	Priority Preemptive Scheduling
PSSA	Preliminary System Safety Assessment
RCS	Reaction Control System
RDC	Remote Data Concentrator
RMS	Redundancy Management System
RTOS	Real Time Operating System
SBC	Single Board Computer
SCP	Self Checking Pair
SEE	Single Event Effect
SEL	Single Event Latchup
SEU	Single Event Upset
SoS	System of Systems
SOS	Slightly Off Specification
TB	Time Base register
TDM	Time Division Multiplexing
TMR	Triple Modular Redundancy
TPS	Time Preemptive Scheduling
TRL	Technology Readiness Level
VMC	Vehicle Management Computer
WCET	Worst Case Execution Time
XML	eXtensible Markup Language

Appendix B

XML System Configuration Files

B.1 Partition Schedule

This following schedule is for a one second major frame with a 20 ms minor frame.

The following 20 ms minor frame is repeated 49 times:

```
<PartitionWindow PartitionNameRef="FTSS_INPUT_PARTITION"
  Duration="0.000250" ReleasePoint="1" />
<PartitionWindow PartitionNameRef="GNC_PARTITION" Duration="0.002000"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="FTSS_OUTPUT_PARTITION"
  Duration="0.000250" ReleasePoint="1" />
<PartitionWindow PartitionNameRef="GNC_PARTITION" Duration="0.005000"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="IVHM_PARTITION" Duration="0.010000"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="SPARE" Duration="0.002500"
  ReleasePoint="1" />
```

This is the last 20 ms minor frame with the FTSS_SYNC window:

```
<PartitionWindow PartitionNameRef="FTSS_INPUT_PARTITION"
  Duration="0.000250" ReleasePoint="1" />
<PartitionWindow PartitionNameRef="GNC_PARTITION" Duration="0.002000"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="FTSS_OUTPUT_PARTITION"
  Duration="0.000250" ReleasePoint="1" />
<PartitionWindow PartitionNameRef="GNC_PARTITION" Duration="0.005000"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="IVHM_PARTITION" Duration="0.010000"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="SPARE" Duration="0.002250"
  ReleasePoint="1" />
<PartitionWindow PartitionNameRef="FTSS_SYNC_PARTITION"
  Duration="0.000250" ReleasePoint="1" />
```

B.2 Ports

```
- <Application Name="ftss_input_app">
- <ApplicationDescription EntryPoint="usrRoot" InitializationTime="1" Period="1"
  ComputeTime="0.000250">
- <Ports>
  <SamplingPort Name="MIMU_1" Direction="SOURCE" MessageSize="60"
    RefreshRate="0.020000" />
  <SamplingPort Name="MIMU_2" Direction="SOURCE" MessageSize="60"
    RefreshRate="0.020000" />
  <SamplingPort Name="MIMU_3" Direction="SOURCE" MessageSize="60"
    RefreshRate="0.020000" />
  <SamplingPort Name="GPS_1" Direction="SOURCE" MessageSize="60"
    RefreshRate="0.100000" />
  <SamplingPort Name="GPS_2" Direction="SOURCE" MessageSize="60"
    RefreshRate="0.100000" />
  <SamplingPort Name="GPS_3" Direction="SOURCE" MessageSize="60"
    RefreshRate="0.100000" />
  <SamplingPort Name="BARO_ALT_1" Direction="SOURCE" MessageSize="60"
    RefreshRate="1.000000" />
  <SamplingPort Name="BARO_ALT_2" Direction="SOURCE" MessageSize="60"
    RefreshRate="1.000000" />
  <SamplingPort Name="BARO_ALT_3" Direction="SOURCE" MessageSize="60"
    RefreshRate="1.000000" />
</Ports>

- <Application Name="gnc_app">
- <ApplicationDescription EntryPoint="usrRoot" InitializationTime="1" Period="1"
  ComputeTime="0.002000">
- <Ports>
  <SamplingPort Name="MIMU_1" Direction="DESTINATION" MessageSize="60"
    RefreshRate="0.020000" />
  <SamplingPort Name="MIMU_2" Direction="DESTINATION" MessageSize="60"
    RefreshRate="0.020000" />
  <SamplingPort Name="MIMU_3" Direction="DESTINATION" MessageSize="60"
    RefreshRate="0.020000" />
  <SamplingPort Name="GPS_1" Direction="DESTINATION" MessageSize="60"
    RefreshRate="0.100000" />
  <SamplingPort Name="GPS_2" Direction="DESTINATION" MessageSize="60"
    RefreshRate="0.100000" />
  <SamplingPort Name="GPS_3" Direction="DESTINATION" MessageSize="60"
    RefreshRate="0.100000" />
  <SamplingPort Name="BARO_ALT_1" Direction="DESTINATION"
    MessageSize="60" RefreshRate="1.000000" />
  <SamplingPort Name="BARO_ALT_2" Direction="DESTINATION"
    MessageSize="60" RefreshRate="1.000000" />
```

```

<SamplingPort Name="BARO_ALT_3" Direction="DESTINATION"
  MessageSize="60" RefreshRate="1.000000" />
<SamplingPort Name="RCS_thruster_1" Direction="SOURCE"
  MessageSize="20" RefreshRate="0.020000" />
<SamplingPort Name="RCS_thruster_2" Direction="SOURCE"
  MessageSize="20" RefreshRate="0.020000" />
<SamplingPort Name="RCS_thruster_3" Direction="SOURCE"
  MessageSize="20" RefreshRate="0.020000" />
</Ports>

```

```

- <Application Name="ftss_output_app">
- <ApplicationDescription EntryPoint="usrRoot" InitializationTime="1" Period="1"
  ComputeTime="0.000250">
- <Ports>
  <SamplingPort Name="RCS_thruster_1" Direction="DESTINATION"
    MessageSize="20" RefreshRate="0.020000" />
  <SamplingPort Name="RCS_thruster_2" Direction="DESTINATION"
    MessageSize="20" RefreshRate="0.020000" />
  <SamplingPort Name="RCS_thruster_3" Direction="DESTINATION"
    MessageSize="20" RefreshRate="0.020000" />
</Ports>

```

B.3 Communication Channels

```
- <Connections xmlns="ARINC653"
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="ARINC653
    C:/Workspace/Module/module_component.xsd">
- <Channel Id="1">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION"
    PortNameRef="MIMU_1" />
  <Destination PartitionNameRef="GNC_PARTITION" PortNameRef="MIMU_1" />
</Channel>
- <Channel Id="2">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION"
    PortNameRef="MIMU_2" />
  <Destination PartitionNameRef="GNC_PARTITION" PortNameRef="MIMU_2" />
</Channel>
- <Channel Id="3">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION"
    PortNameRef="MIMU_3" />
  <Destination PartitionNameRef="GNC_PARTITION" PortNameRef="MIMU_3" />
</Channel>
- <Channel Id="4">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION" PortNameRef="GPS_1"
    />
  <Destination PartitionNameRef="GNC_PARTITION" PortNameRef="GPS_1" />
</Channel>
- <Channel Id="5">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION" PortNameRef="GPS_2"
    />
  <Destination PartitionNameRef="GNC_PARTITION" PortNameRef="GPS_2" />
</Channel>
- <Channel Id="6">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION" PortNameRef="GPS_3"
    />
  <Destination PartitionNameRef="GNC_PARTITION" PortNameRef="GPS_3" />
</Channel>
- <Channel Id="7">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION"
    PortNameRef="BARO_ALT_1" />
  <Destination PartitionNameRef="GNC_PARTITION"
    PortNameRef="BARO_ALT_1" />
</Channel>
- <Channel Id="8">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION"
    PortNameRef="BARO_ALT_2" />
```

```

    <Destination PartitionNameRef="GNC_PARTITION"
      PortNameRef="BARO_ALT_2" />
  </Channel>
- <Channel Id="9">
  <Source PartitionNameRef="FTSS_INPUT_PARTITION"
    PortNameRef="BARO_ALT_3" />
  <Destination PartitionNameRef="GNC_PARTITION"
    PortNameRef="BARO_ALT_3" />
  </Channel>
- <Channel Id="20">
  <Source PartitionNameRef="GNC_PARTITION"
    PortNameRef="RCS_thruster_1" />
  <Destination PartitionNameRef="FTSS_OUTPUT_PARTITION"
    PortNameRef="RCS_thruster_1" />
  </Channel>
- <Channel Id="21">
  <Source PartitionNameRef="GNC_PARTITION"
    PortNameRef="RCS_thruster_2" />
  <Destination PartitionNameRef="FTSS_OUTPUT_PARTITION"
    PortNameRef="RCS_thruster_2" />
  </Channel>
- <Channel Id="22">
  <Source PartitionNameRef="GNC_PARTITION"
    PortNameRef="RCS_thruster_3" />
  <Destination PartitionNameRef="FTSS_OUTPUT_PARTITION"
    PortNameRef="RCS_thruster_3" />
  </Channel>
</Connections>

```


B.4 Port Groups

Note that the XML parser for the port groups was not completed. Thus, some of the below data had to be hard coded in the FTSS software.

```
<PortGroups>
  <FtssPortGroup
    Name="gnc_50hz_inputs"
    ExchangeType="TwoRound"
    <Ports>
      <Port NameRef="gps_1" SrcCh="CH1" />
      <Port NameRef="mimu_1" SrcCh="CH1" />
      <Port NameRef="gps_2" SrcCh="CH2" />
      <Port NameRef="mimu_2" SrcCh="CH2" />
      <Port NameRef="gps_3" SrcCh="CH3" />
    </Ports>
  </FtssPortGroup>

  <FtssPortGroup
    Name="mp_50hz_inputs"
    ExchangeType="OneRound"
    <Ports>
      <Port NameRef="sen_1" SrcCh="ALL" />
      <Port NameRef="sen_2" SrcCh="ALL" />
      <Port NameRef="sen_3" SrcCh="ALL" />
    </Ports>
  </FtssPortGroup>

  <FtssPortGroup
    Name="gnc_50hz_outputs"
    ExchangeType="TwoRound"
    <Ports>
      <Port NameRef="act_1", SrcCh="ALL" />
      <Port NameRef="act_2", SrcCh="ALL" />
      <Port NameRef="act_3", SrcCh="ALL" />
    </Ports>
  </FtssPortGroup>
```


Appendix C

GbE CCDL Driver Details

C.1 IEEE 802.3 MAC Frame Structure

All messages that go across the CCDL use the IEEE Standard 802.3 Basic MAC Frame format [49]. The IP, UDP or TCP protocols are not used. The FTSS header starts immediately after the Length field. The smallest frame length supported by the hardware is 40 bytes (802.3 indicates 64 byte min) leaving 22 bytes for user data (14 bytes are used by the header and 4 bytes are used by the CRC). The longest frame length supported by the hardware is 9700 bytes (802.3 indicates 1518 byte max).

Each channel (i.e. SBC750GX board) has three MAC functions implemented in the Marvell chip. Each MAC has a dedicated destination address that is used to filter all messages. Messages that do not have the correct destination address are dropped. Since each CCDL is a dedicated point-to-point connection between two MACs (one on each channel), the source and destination address fields never change; they are only swapped depending on which direction a message is traveling.

C.1.1 Ethernet Frame Check Sequence

Every message frame that goes across a CCDL will contain a 32 bit CRC in the frame check sequence (FCS) field. The CRC is generated on the transmit side and checked on the receive side by the COTS Ethernet hardware. Any message received with a CRC error will be dropped by the hardware. The 32 bit FCS field in the MAC frame is an error detection value that is the result of performing polynomial division. The entire MAC frame (except the FCS field) is treated as a GF(2) polynomial (i.e. modulo 2: each coefficient is a 0 or 1). This polynomial is then divided by a generator (or CRC) polynomial $G(x)$. The remainder of this division is placed in the 32 bit FCS field. IEEE 802.3 specifies the following irreducible 32 bit CRC polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

The minimum number of bytes sent over a CCDL was 40 and so four errors would be detected and the maximum number of bytes sent was 1030 and so three errors would be detected using the results in [51].

It should be noted that this is not a cryptographic signature and thus it cannot be used to verify the identity of the sender as is required for message authentication. As section 2.3.3 discussed, message authentication would reduce the number of channels (fault containment regions) required to tolerate one Byzantine fault from four to three. Signatures are not currently used in the P-NMR system mainly because they would have to be calculated by the PowerPC 750GX. Digital signatures are computationally intensive and require a significant amount of processor time. Thus, they would increase FTSS overhead and overall system response times. The advantage of using the IEEE 802.3 CRC is that it is implemented in COTS hardware. The MACs in the Marvell chip perform all the CRC calculations without involving the 750GX. Work is ongoing at Draper Laboratory to design an efficient and reliable digital signature that could be calculated by the 750GX.

C.2 DMA, SDRAM and the PowerPC Cache

Data can be transmitted or received over the GbE CCDLs without involving the PowerPC processor. This is accomplished through hardware DMA engines implemented in the Marvell system controller chip. Each port on the chip has two engines, one for receive and one for transmit. Each DMA engine uses a linked list of buffer descriptors. Each descriptor has a pointer to buffer memory where data can be read/written and a pointer to the next descriptor in the list. The creation of the descriptors and the buffers which they point to is performed by the FTSS software executing on the PowerPC. This creation step is only done once at system startup.

Once the FTSS code has created the descriptor list and the buffers then the DMA engines can take over transmitting and receiving data. The FTSS code simply writes the SDRAM address of the first descriptor to a DMA hardware register and then it writes another value to the DMA control register that enables the DMA operation. For example, the receive DMA engine will obtain the first receive descriptor from the register. It will

then use the buffer pointer in the descriptor to locate the data buffer in SDRAM. The DMA operation will move received data to this buffer until it is full. The DMA engine will then obtain the address of the next descriptor in SDRAM and the process will repeat until all descriptors in the list have been used. While the DMA engine is filling up the buffers in SDRAM using one end of the descriptor list, the FTSS task executing on the PowerPC is reading the descriptors and buffers from the other end of the list. Each time FTSS reads a descriptor, it marks it as usable again.

The descriptors are located in 2 MB of SRAM that is internal to the Marvell chip. This reduces the PowerPC latency when accessing the descriptors since the Marvell does not need to go to the external SDRAM for the descriptors. Since the Marvell has an internal crossbar that allows concurrent access to devices, the PowerPC can be accessing a descriptor in the internal SRAM at the time that a DMA engine is accessing a buffer in the external SDRAM. This also reduces the external SDRAM bandwidth consumption by the PowerPC.

The descriptor memory in the Marvell internal SRAM has also been marked as “cache safe”. This eliminates cache coherency problems. Any time a transmit or receive descriptor in SRAM is modified by a DMA engine, it will also be updated in the PowerPC cache (or copies of the descriptors will not be allowed in the cache in the first place i.e. noncacheable memory). The data buffers that the descriptors point to are in the external SDRAM and thus the FTSS code uses cache flush and invalidate routines to maintain coherency with the PowerPC cache.

As mentioned, the MAC frame header includes 6 bytes for the source MAC address and 6 bytes for the destination MAC address. Since these 12 bytes are static, they are placed in the transmit data buffers in SDRAM at system startup in order to reduce the amount of data that is flushed from the PowerPC cache out to SDRAM.

The handling of small messages consisting of just a few bytes (such as clock sync messages) has also been optimized. The minimum Ethernet frame size supported by the Marvell MAC is 40 bytes. After the Ethernet header and CRC are accounted for, this leaves 22 bytes of data. If the user data being sent is smaller than 22 bytes then only it is flushed out of the cache to the SDRAM buffer. The rest of the buffer will contain stale or uninitialized values. The CRC will be calculated for the entire 40 byte Ethernet frame.

Channels receiving the data will use the FTSS message type in the FTSS header to determine how many of the 22 bytes are relevant.

C.3 Polled Mode vs. Interrupt Mode

Each Ethernet driver operates in polled mode and not interrupt mode. That is to say, FTSS tasks are not interrupted by the Ethernet device hardware or driver. It is up to the FTSS code to continually query the low-level Ethernet driver for a change of state (e.g. new data). Both transmit and receive services are implemented as function calls that return a status. Polled mode is more appropriate for a system with robust time partitioning. If an application needs to send or receive data over a device then these activities should take place during the partition window for that application and not during the time window of another partition. Interrupts could occur at any time and are thus less deterministic compared to polling. This is especially true in the case of a babbling node that is constantly transmitting random data on the CCDL.

C.3.1 Sending Data

In order to send data over a CCDL, an FTSS task calls the function:

status CCDL_poll_send (dest_ch, data_ptr)

The *dest_ch* parameter is the channel to send to and the *data_ptr* parameter is a pointer to the data. The *dest_ch* can be a single channel or it can be “ALL” which means broadcast the data to all channels in the operational set. For each Ethernet driver (there is a driver for each Ethernet port), this function will first wait for any in-progress transmit operation to complete. Thus, the FTSS task will be blocked while this occurs. If there is no in-progress transmission occurring on a particular port, the data will be copied to the buffer pointed to by the first transmit DMA descriptor. FTSS then flushes this data out of the cache and into SDRAM and then activates the DMA transmit operation. At this point, the transmit request is complete and the send function returns.

It should be noted that a broadcast to ALL does not mean that the data goes to all channels at the exact same time. The send is actually done serially since there are two individual drivers, one for each CCDL. For example, when FTSS on CH1 calls the

CCDL_poll_send function with ALL as a parameter, two individual sends are performed; one for the CCDL driver going to CH2 and one for the CCDL driver going to CH3.

Included in the CCDL system configuration data (explained below) is a channel specific send order that makes broadcasting more equitable among the three channels. For example, if FTSS on every channel always used [CH1, CH2, CH3] as the send order when broadcasting data then CH1 will always receive messages before CH3 does. In some applications such as clock synchronization, it would be more appropriate to make sure that each channel has an equal probability of receiving the message first.

C.3.2 Receiving Data

In order to receive data from a particular channel, an FTSS task calls the function:

status CCDL_poll_rcv(src_ch, data_ptr)

The dest_ch parameter is the channel to receive from and the data_ptr parameter is a pointer to the memory area that will store the data. Unlike the send function, the src_ch parameter cannot be “ALL”. The receive function will then obtain the DMA receive descriptor that is at the front of the list of descriptors. The flags in the descriptor will indicate if the DMA operation has placed fresh data in the buffer pointed to by the descriptor. If there is no fresh data, the returned status will indicate this. The FTSS code continually calls this function until the status indicates fresh data or until there is a timeout.

C.3.3 Protection Against a Babbling Node

The current driver does provide some limited protection against a babbling channel that is flooding a CCDL with invalid data. The receive DMA engine for the Ethernet port in question will continue to fill up the buffers pointed to by the receive descriptor list. If the babbling channel sends enough data, FTSS will not get a chance to read the descriptors and mark them as free again for DMA use. Thus, all descriptors will be marked as currently in use and the DMA engine will increment a “Frames Discarded Counter” register. When FTSS does run, it examines this register to determine if too many messages have been sent since the last time it ran. In addition to the above, FTSS

has the ability to immediately discard all data in all the receive buffers for a particular Ethernet port.

C.4 Static CCDL Configuration Table

Since the exact same FTSS software executes on each channel, a method is needed to determine which channel is hosting the FTSS and which Ethernet ports are connected to the other channels. In order to accomplish this, a CCDL system configuration table was created. At the present time this system configuration data is hard coded in the source but work has started on migrating this data to the ARINC 653 XML configuration files.

Each channel (i.e. SBC750GX board) has three ports and each port has a unique MAC address stored in non-volatile RAM. Only two of the ports (PORT_0 and PORT_1) on each channel are currently used for CCDLs since the system is currently in a TMR configuration. This port and MAC data is stored in the CCDL system configuration table. When FTSS starts up after system cold start, it queries NVRAM and finds the MAC address for PORT_0 on the local channel. FTSS then searches for this MAC address in the table. Once found, FTSS knows which channel it is executing on and which channels PORT_0 and PORT_1 are connected to.

C.5 Alternate Design: Partition Level CCDL Device Driver

An alternative to the core OS CCDL device driver design is one where the driver is located entirely in a user-level partition. This means the rest of the FTSS would also reside in the same partition as the driver. The BSP code that performs the memory mapping of Marvell Ethernet DMA registers can still be used except now the registers are mapped to a user-level FTSS partition memory region, not the core OS memory region. The FTSS code in the partition then has read/write access to the registers. All code executes in user mode. Supervisor privileges are not required to access the DMA registers.

The disadvantage of this design is a performance penalty. If data in another application partition like GN&C needs to go to the CCDL driver in the FTSS partition, it must use inter-partition communication via APEX ports. The GN&C partition would write data to the port which would transfer the data to the core OS memory. This is one mode switch (user to supervisor) and byte copy. At some later time, the FTSS partition would read the APEX port, moving the data up into the FTSS partition memory region. This is another mode switch (supervisor to user) and byte copy. FTSS code would then copy the data to one of the device driver buffers for transmission. Compared to the core OS driver model, this sequence would cause an additional mode changes and byte copy.

Another disadvantage of placing FTSS in a user-level partition is that new system calls would be required to access certain core OS kernel services not currently exported to user applications. This would mean more modifications to the core OS kernel code which adversely impacts the RTOS DO-178B certification package from the vendor.

The advantage of this design is more robust partitioning. FTSS is contained in its own memory region, separate from the kernel. It also executes in user mode instead of supervisor mode. A fault in the device driver or other FTSS code would not corrupt kernel memory. Thus, a malfunctioning FTSS would not immediately bring down the entire module.

This page intentionally left blank

Appendix D

FAA Regulations and Guidelines

This thesis proposes using certain technologies or standards that originated in the commercial aviation industry for use on space vehicles such as those being developed for the Constellation program. In particular, we propose the use of robust software partitioning, as defined by the open ARINC 653 standard, to host flight critical and non-flight critical applications on the same processor as is done in certain IMA applications in the aircraft avionics industry. We also propose using DO-178B as a means for qualifying certain software elements such as the operating system used on Constellation avionics projects.

Thus, it is appropriate to examine a few reliability and safety standards used by the aviation industry so that they can be compared with the NASA requirements discussed in section 2.1. This appendix focuses on Federal Aviation Administration (FAA) regulations and guidelines that pertain to avionics, especially computer based flight guidance systems (FGS) and IMA systems. It also presents some terminology and definitions used by the FAA regulations.

D.1 Airworthiness Regulations

The FAA is responsible for regulating all air travel in the United States. Thus, its regulations, rules and standards are quite complex and are distributed among several Code of Federal Regulations (CFRs) documents and various Advisory Circular (AC) documents which attempt to provide additional guidance for certain CFRs. The following discussion pertains to Part 25 of the regulations which deals with the airworthiness standards for transport category aircraft. These aircraft must be extremely safe due to the number of passengers that can be carried and due to the frequency of flights per day. For example, the Airbus A380 can carry over 600 passengers in an economy configuration.

A precise definition of *reliable* has been problematic for the FAA and aircraft manufacturers [52]. This is in part due to the different ways in which a *fault or failure*

can be defined. Failures can have different results in terms of severity and different methods can be used for calculating the probability that a given failure with a given severity will occur. Confusion has also arisen when attempting to calculate the probability of a severe system failure that is due to less severe failures in other systems (e.g. cascading failures).

More recent FAA advisory material in [53] define an *error* as an omission or incorrect action by a crewmember or maintenance personnel, or a mistake in requirements, design, or implementation. A *failure* is an occurrence that affects the operation of a *component, part, or element* such that it can no longer function as intended. This includes both *loss of function* and *malfunction*. Errors may cause failures but are not in themselves failures. A *failure condition* is a condition having an effect on the airplane which is caused or contributed to by one or more failures or errors. Aircraft manufacturers must ensure that designs for flight critical systems are capable of handling failure conditions.

The main philosophy used by the FAA when writing airworthiness regulations is based on a *fail-safe* design concept [54]. A fail-safe design assumes that during any one flight, in any system or subsystem, there will be a failure of a single element, component, or connection *regardless of its probability* and that these single failures must not prevent continued safe flight and landing. Subsequent failures during the same flight, whether detected *or latent*, and any combination of failures, should also be assumed by the designer.

Advisory material dealing with a computer based FGS also use the term *fail-passive* [53, 55]. When a fail-passive system fails, it will not cause a significant deviation of aircraft flight path or attitude. For example, a fail-passive autopilot will not cause a large control surface deflection when it fails. The control surfaces will simply remain where they are until direct inputs are received from the pilot controls (e.g. rudder pedals). The pilots would not be able to use the failed autopilot system but they would certainly be able to fly the aircraft manually without the autopilot. In other words, there is a loss of function but there is no malfunction. A system that can fail in an active way (e.g. causing the deflection of a control surface) is termed a fail-active system or a malfunctioning system and is highly undesirable (and never intentionally used) in flight critical systems.

Another term used in the context of regulations for autopilot systems is *fail-operational (fail-op)*. A fail-operational system is defined as a system capable of completing an operation, following the failure of any single element or component of that system, without pilot action. In other words, after a single component failure within the system, there is no loss of function. A fail-op system is always implemented using some form of *redundancy*. Redundancy is defined as the presence of more than one independent means for accomplishing a given function or flight operation. For an autopilot, this includes, but is not limited to, redundant sensors, data buses, computers, actuators and power supplies.

The FAA advisory material define four classes of failure conditions: *minor, major, hazardous and catastrophic* [53, 54, 55]. A catastrophic failure condition is one where the failure would result in multiple fatalities, usually with the loss of the airplane. The regulations specify that the occurrence of any catastrophic failure condition be “*Extremely Improbable*”. The definition of Extremely Improbable is one occurrence every one billion flight hours, often written as a 1×10^{-9} probability per flight hour or per event (e.g. per takeoff or landing). The intent of the term Extremely Improbable is to describe a failure condition that has a probability of occurrence so small that it is not anticipated to occur during the entire lifetime of all aircraft of the same type.

A hazardous failure condition is one where the failure would *severely* reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions. The occurrence of any hazardous failure condition must be “*Extremely Remote*” which is defined as a probability of occurrence between the orders of 1×10^{-9} and 1×10^{-7} per hour of flight, or per event. The intent of Extremely Remote is to describe a failure condition that is not anticipated to occur to each airplane during its total life, but which may occur a few times when considering the total operational life of all airplanes of the type.

In order to illustrate the use of the above terms for an actual application, we examine the FAA advisory material for a CAT IIIb fail-operational automatic landing system [55]. A CAT IIIb landing occurs when poor visibility conditions prevent the pilots from seeing the runway even when the aircraft is a few feet above the ground. In these conditions, an autopilot with an automatic landing function (autoland) is capable of flying

the aircraft all the way down onto the runway. This includes the approach, flare, touchdown and rollout. For CAT IIb operations, the “alert height” is defined as the height below which it is safer to continue the landing rather than have the pilot attempt a manual go around in the event of an autopilot failure. For large transport aircraft, this height is typically in the order of 50 to 200 feet.

The FAA stipulates that any single malfunction or any combination of malfunctions of the landing system that could prevent a safe landing or go around must be Extremely Improbable if the malfunctions cannot be detected and annunciated to the crew. If the malfunctions can be detected and annunciated then the probability of a system failure must be Extremely Remote. The exposure time for assessing failure probabilities is the average time to descend from 200 feet or higher to touchdown. Finally, following a single malfunction, the landing system must not lose the capability to perform lateral and vertical path tracking, alignment with runway heading, flare and touchdown.

D.2 IMA Guidelines

The FAA has published guidelines for the use of IMA [56, 57]. These guidelines indicate that IMA systems should be designed to provide capabilities to initiate recovery of functions whose loss is catastrophic. They should also be designed to avoid the need for crew-initiated recovery features.

The guidelines go on to state that the intended functions of the IMA system should be identified and evaluated for their impact on aircraft and engine safety. A functional hazard assessment (FHA) should be conducted at the aircraft level to determine and classify the hazards associated with both the loss and malfunction of each function provided by the IMA system. *Failure probabilities of the protection scheme(s)* must be commensurate with the failure condition classifications of the simultaneous malfunction of all IMA functions that it supports. The FAA recommends that design features that implement protection use *both hardware and software means*.

Based on the hazard classifications determined by the FHA, the proposed design and installation of the IMA system should be evaluated by a preliminary system safety assessment (PSSA) to establish the specific safety requirements of each component in the IMA system (for example, cabinet, rack, hardware modules, buses, connectors, displays,

sensors, control devices, and functional software). Unless appropriate measures are provided to protect an IMA cabinet or rack from *common-cause failures*, all of the functions provided by a single IMA cabinet or rack should be assumed to fail as the result of a *single failure*. Loss of all functions in each IMA cabinet or rack and/or hardware module should be addressed in the safety analysis, including common-cause issues.

The FAA also provides guidelines for certifying parts of a flight guidance system that may use an IMA architecture [53]. The FGS includes the autopilot, flight director and autothrottle functions. As such it includes the sensors, data buses, computers and actuators used for flight control. The guidelines recognize that modern digital avionics architectures are leading to an FGS implementation that is more closely integrated with other systems on the aircraft such as the FBW controls, flight management system and the crew displays.

The guidelines go on to state that the integration of other aircraft systems with the FGS has the potential of reducing the independence of failure effects and partitioning between functions (i.e. cascading failure modes). IMA architectures are specifically called out as a risk due to the sharing of hardware and software resources with other non-FGS functions. The guidelines state that when credit is taken for shared resources or partitioning schemes, these should be justified and documented within the system safety analysis.