DOT/FAA/AR-11/2

Air Traffic Organization
NextGen & Operations Planning
Office of Research and
Technology Development
Washington, DC 20591

Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems

February 2011

Final Report

This document is available to the U.S. public through the National Technical Information Services (NTIS), Springfield, Virginia 22161.

This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The United States Government assumes no liability for the contents or use thereof. The United States Government does not endorse products or manufacturers. Trade or manufacturer's names appear herein solely because they are considered essential to the objective of this report. The findings and conclusions in this report are those of the author(s) and do not necessarily represent the views of the funding agency. This document does not constitute FAA certification policy. Consult your local FAA aircraft certification office as to its use.

This report is available at the Federal Aviation Administration William J. Hughes Technical Center’s Full-Text Technical Reports page: actlibrary.tc.faa.gov in Adobe Acrobat portable document format (PDF).

Technical Report Documentation Page

1. Report No.
DOT/FAA/AR-11/2

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
HANDBOOK FOR THE SELECTION AND EVALUATION OF MICROPROCESSORS FOR AIRBORNE SYSTEMS

5. Report Date
February 2011

6. Performing Organization Code

7. Author(s)
Bob Green[1,] Joseph Marotta[2], Brian Petre[3], Kirk Lillestolen[4], Richard Spencer[5], Nikhil Gupta[6], Daniel O'Leary[7], Jason Dan Lee[6], John Strasburger[5], Arnold Nordsieck[8], Bob Manners[9], and Dr. Rabi Mahapatra[6]

8. Performing Organization Report No.

9. Performing Organization Name and Address

[1]BAE Systems
600 Main Street
Johnson City, NY 13790

[2]Honeywell Aerospace
9201 San Mateo, Blvd., NE, MS C01
Albuquerque, NM 87113

[3]GE Aviation
3290 Patterson Ave., SE
Grand Rapids, MI 49512-1991

[4]Hamilton Sundstrand Corporation
One Hamilton Road, MS 3-2-K2
Windsor Locks, CT 06096

[5]Federal Aviation Administration
Technical Programs and Continued
Airworthiness Branch, AIR-120
Washington, D.C. 20024

[6]Texas A&M University
College Station, TX 77845

[7]Lockheed Martin
PO Box 745, MZ8667
Ft. Worth, TX 76101

[8]Boeing
PO Box 3707
Seattle, WA 98124

[9]Lumark Technologies, Inc.
4904 Tydfil Court, Ste 100
Fairfax, Virginia 22030

10. Work Unit No. (TRAIS)

11. Contract or Grant No.

12. Sponsoring Agency Name and Address
U.S. Department of Transportation
Federal Aviation Administration
Air Traffic Organization NextGen & Operations Planning
Office of Research and Technology Development
Washington, DC 20591

13. Type of Report and Period Covered
Final Report

14. Sponsoring Agency Code
AIR-120

15. Supplementary Notes
The Federal Aviation Administration Airport and Aircraft Safety R&D Division COTR was Charles Kilgore.

16. Abstract

This Handbook provides research information intended to help aerospace system developers and integrators and regulatory agency personnel in the selection and evaluation of commercial off-the-shelf microprocessors for use in aircraft systems.

This Handbook is based on the cooperative research accomplished by contributing members of the aerospace industry and the Federal Aviation Administration (FAA) as part of the Aerospace Vehicle Systems Institute Microprocessor Evaluations Projects 1 through 5. The project objectives were to (1) identify common risks of using systems-on-a-chip (SoC) and mitigation techniques to provide evidence that they satisfy regulatory requirements and (2) evaluate existing regulatory policy and guidelines against the emerging characteristics of complex, nondeterministic microprocessors and SoCs to support the certification of aircraft and qualification of systems using these devices.

Complex aircraft system development requires more robust consideration of system failure and anomaly detection, correction, and recovery. The safety net approach identified in this Handbook also may provide a means to reduce the growing difficulties and costs of design assurance for highly integrated, complex, nondeterministic airborne electronic hardware and software within aircraft systems and reduce the labor burden for FAA regulation compliance and design assurance. The safety net approach documented in this Handbook is consistent with current FAA policy and guidelines.

17. Key Words
Microprocessor, System-on-a-chip, Safety net, Aircraft certification, System qualification, Airborne electronic hardware, Aircraft safety, System architecture, Simulation, Operational level error detection and recovery, Integrated circuits, Avionics safety, Architecture patterns

18. Distribution Statement
This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	60	

ACKNOWLEDGEMENTS

The authors would like to thank the following people for their direct contributions to this Handbook and their persistent efforts throughout the course of this project:

- Floyd Fazi, Lockheed Martin
- Bob Green, BAE Systems
- Nikhil Gupta, Texas A&M University
- Charles Kilgore, FAA
- Jason Lee, Texas A&M University
- Kirk Lillestolen, Hamilton Sundstrand
- Rabi Mahapatra, Texas A&M University
- Bob Manners, Lumark Technologies, Inc.
- Joseph Marotta, Honeywell
- Arnold Nordsieck, Boeing
- Daniel O'Leary, Lockheed Martin
- Brian Petre, GE Aviation
- Richard Spencer, FAA
- John Strasburger, FAA

The authors acknowledge the following individuals and organizations for providing support to the creation of this Handbook:

- Praveen Bhojwani, Texas A&M University
- Amita Biswas, Texas A&M University
- Luis Bonet, Glenn Beck, John McLellan, and John Abrams, Freescale Semiconductor
- David Beal, Chris Hall, Chad Margolin, and Jakob Engblom, Virtutech, Inc.
- Bob Chobot, BAE Systems
- Rob Clements, BAE Systems
- Robert Cox, Hamilton Sundstrand
- Wolfgang Denk, DENX Software Engineering
- Kyle Deutsch, BAE Systems
- Jon Diekama, GE Aviation
- Chris Eckert, GE Aviation
- Fred Fisher, AVSI
- John Gerard, Smiths Aerospace
- Dorina Hester, Boeing
- Gary Horan, FAA
- Yoonjin Kim, Texas A&M University
- Tod Lanham, GE Aviation
- John Lapointe, FAA
- Aaron Larson, Honeywell
- John Lewis, FAA
- Barbara Lingberg, FAA

- David Lund, AVSI
- Emmanuel Papadopoulos, FAA
- Eric Peterson, Honeywell
- James Peterson, Honeywell
- David Redman, AVSI
- Leanna Rierson, Digital Safety Consulting
- Tom Sherman, GE Aviation
- Tim Theriault, GE Aviation
- Craig Treece, Lockheed Martin
- Kurt Woodham, NASA

The authors would also like to thank all other technical experts and document reviewers who supported the AVSI AFE43 project.

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

LIST OF ACRONYMS

AEH	Airborne electronic hardware
AVSI	Aerospace Vehicle Systems Institute
BIST	Built-in self test
BIT	Built-in test
CCB	Core Complex Bus
CCSRBAR	Configuration control and status base address register
COTS	Commercial off-the shelf
CRI	Certification review items
DEC	Decrementer
DoS	Denial of service
EMI	Electromagnetic interference
FAA	Federal Aviation Administration
FPGA	Field-programmable gate arrays
I/O	Input/output
IDE	Integrated development environments
IP	Intellectual Property
L2	Level 2
MMU	Memory management unit
OS	Operating system
PCI	Peripheral component interconnect-express
RAM	Random access memory
SEU	Single event upset
SoC	System-on-a-chip
UART	Universal asynchronous receiver/transmitter
WCET	Worst-case execution time
WDM	Watchdog monitor

EXECUTIVE SUMMARY

This Handbook provides research information intended to help aerospace system developers and integrators and regulatory agency personnel in the selection and evaluation of commercial off-the-shelf (COTS) microprocessors for use in aircraft systems. Airborne electronic hardware (AEH) includes modern state-of-the-art and highly integrated COTS microprocessors that (1) may not provide adequate visibility and debug features to reveal internal functionality, (2) are less predictable due to the interaction of advanced features, (3) have programmable configuration capabilities available to application software, and (4) share resources across multiple cores and devices. These highly complex COTS microprocessors are becoming more challenging to test and to determine that they satisfy applicable functional and safety-related requirements.

Resolutions to certification process challenges should offer the possibility of establishing and maintaining standards that support the continual change and growth of technologies and operations. Such resolutions can include

- establishing qualitative as well as quantitative methods to certify aircraft with embedded nondeterminate complex or critical applications.

- establishing accepted standards of architectural patterns for critical, complex systems and methods for validation and design assurance.

- establishing industrywide accepted methods for design assurance of COTS microprocessor and microprocessor-based systems.

- streamlining the certification process.

This Handbook is based on the cooperative research accomplished by contributing members of the aerospace industry and the Federal Aviation Administration (FAA) as part of the Aerospace Vehicle Systems Institute Microprocessor Evaluations Projects 1 through 5. The research underlying this Handbook addressed the use of COTS microprocessors and systems-on-a-chip (SoC) in safety-critical avionics. The project objectives were to (1) identify common risks of using SoCs and mitigation techniques to provide evidence that they satisfy regulatory requirements and (2) evaluate existing regulatory policy and guidelines against the emerging characteristics of complex, nondeterministic microprocessors, and SoCs to support the certification of aircraft and qualification of systems using these devices.

Considering the growing complexity of microprocessors, the research revealed the increasing impracticality of providing safety assurance at the device level alone. The combination of growing complexity of both software and hardware will drive the need to evaluate large complex systems at the system level. The rate of change and growing complexity is accelerating, and the time between new generations of hardware and software is shrinking. The complexity of most COTS components has grown and continues to grow beyond the capability to exhaustively test them. The overlapping phases of development, certification, deployment, and life cycle maintenance together with COTS obsolescence accentuate the need for rapid and evolving

methodologies. FAA policy, guidelines, and practices may need to be updated to accept these methodologies.

Complex aircraft system development requires more robust consideration of system failure and anomaly detection, correction, and recovery. The safety net approach identified in this Handbook also may provide a means to reduce the growing difficulties and costs of design assurance for highly integrated, complex, nondeterministic AEH and software within aircraft systems, and to reduce the labor burden for FAA regulation compliance and design assurance.

A safety net is defined herein as the employment of mitigations and protections at the appropriate level of aircraft and system design to help ensure continuous safe flight and landing. The safety net methodology focuses on the assumption that a microprocessor will misbehave. The ability to protect against unexpected behavior, damage, injury, and instability over the service life outside, or at a level above the device itself, is necessary as appropriate for the design assurance level.

The safety net approach is an alternative way to mitigate the risks associated with COTS microprocessors via both passive and active methods designed into aircraft systems. If it is not feasible to show that complex aircraft systems are sufficiently free of anomalous behavior by evaluating system components, the safety net alternative can mitigate unforeseen or undesirable COTS microprocessor operation by detecting and recovering from anomalous behavior at the operational system level. This approach requires the safety net to be designed as a function within the aircraft system. The safety net can include passive monitoring functions, active fault avoidance functions, and control functions for recovery of system operations. System architecture and control and recovery functions should be designed to facilitate effective system recovery from anomalous events. Safety nets should show that systems are sufficiently impervious to anomalous behavior by ensuring continuous functional availability and reliability, satisfying applicable regulations, and meeting airworthiness requirements. This includes verifying any disabled functionality from the COTS will remain inactive in the specific application.

A multilevel safety net approach is required for complex and critical applications in systems that cannot be fully assured at the component level and is significantly linked to the assigned design assurance level required by regulation, contractual obligation, and the integrated complexity at the device level. The design of safety nets, in general, is becoming a complex, application-specific art form that will be required to detect, resolve, and validate recovery in a run-time environment to the required levels of availability and safety. The safety net approach documented in this Handbook is consistent with current FAA policy and guidelines.

The use of COTS microprocessors may be predominant in future aerospace systems. The use of COTS microprocessors avoids the drastically increasing costs of custom microprocessor design. This Handbook does not address the use of custom device design.

1. INTRODUCTION.

This Handbook is based on Aerospace Vehicle Systems Institute (AVSI) Microprocessor Evaluations Projects 1 through 5 [1-5]. Microprocessors and systems-on-a-chip (SoC) have become extremely complex and densely packaged. Recent changes in commercial off-the-shelf (COTS) microprocessors can be characterized as both physical and functional changes. Physically, transistor density has continued its exponential increase, allowing for hundreds of millions to billions of transistors to be placed on a single device. As of 2010, 65- and 45-nm devices are common in the COTS marketplace, and 32 nm and smaller devices are beginning to enter the marketplace or are on the near horizon. In addition to decreased device size, the functional capability of COTS devices has expanded. It is no longer necessary for different system components to be implemented as discrete devices. Instead, a single COTS SoC may contain multiple microprocessor cores, input/output (I/O) devices, memory controllers, and other functionality. As a result, deterministic performance is difficult or impossible to predict in some cases. These devices require additional evaluation methods beyond that identified in current regulatory requirements to achieve the resilience required to meet safety and reliability requirements. Aircraft systems that contain these COTS devices may require multilevel safety nets to be designed into them.

This Handbook has been written for experienced system designers and regulatory personnel. It is intentionally not prescriptive in nature. It is intended to support the development of new approaches to the design assurance and safety evaluation leading to the approval of airborne systems. The design of an airborne system, the selection of airborne electronic hardware (AEH) (e.g., microprocessor) devices within the system, and the architecture of the system will be unique for each application. The potential sources of nondeterminism and the challenges of design assurance of AEH devices must be determined for each system. This Handbook does not attempt to identify sources of microprocessor nondeterminism because they will be unique for each system and will proliferate in the future. System designers have to evaluate the risks associated with the candidate microprocessors and design the system architecture and the safety nets to mitigate these risks.

This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance; rather, it is the result of FAA- and industry-funded research and may contribute to future policy or guidance.

The purpose of this Handbook is to

- document common areas of concerns regarding the use of COTS microprocessors in complex and/or safety-critical systems.

- provide approaches, information, and examples for mitigating the concerns through a safety net.

- provide access to the research on which the content of this Handbook is based.

- provide example approaches to resilient systems through methods defined in this Handbook under the overarching term safety nets.

- reveal how existing regulatory policy and guidance may be augmented to support the creation of resilient systems through safety net approaches safeguarding the use of microprocessor technologies in complex and/or safety-critical systems.

1.1 SCOPE.

This Handbook can be used for the development and approval of aerospace systems containing embedded COTS microprocessors.

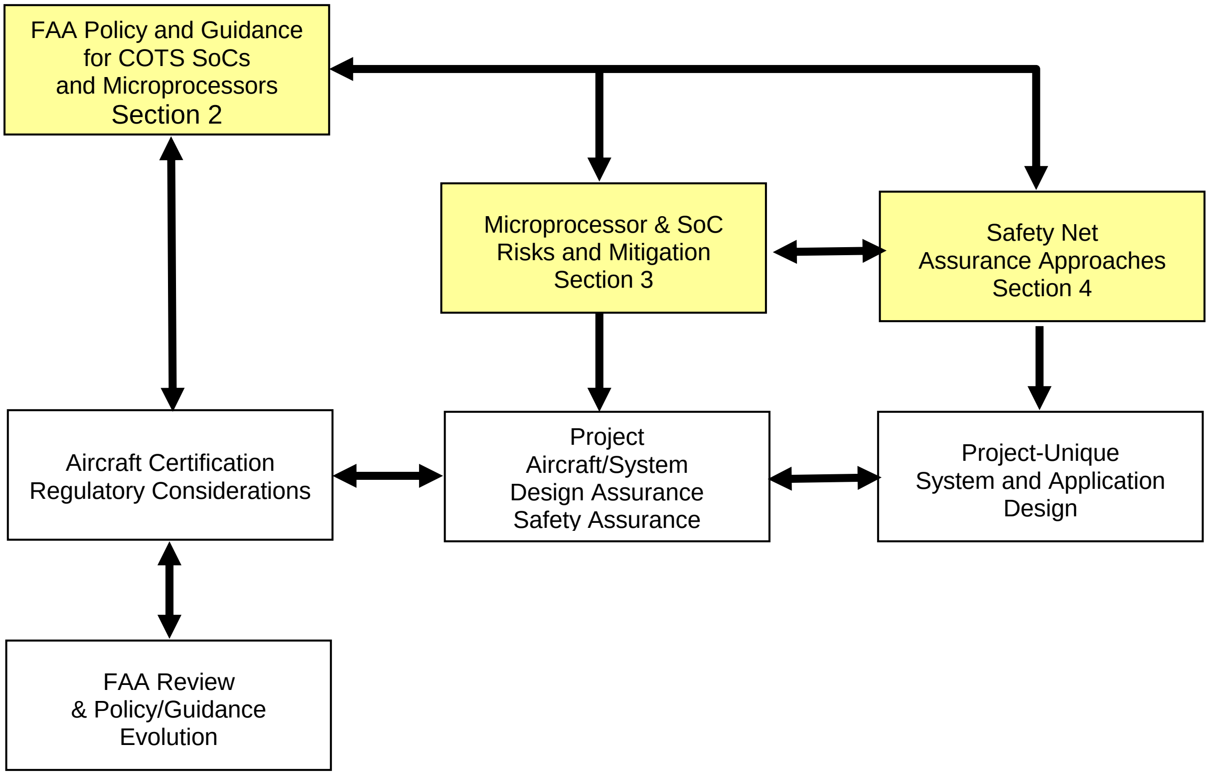
Note: Within this Handbook the term "microprocessors" shall be understood to include COTS microprocessors and SoCs unless otherwise specified. The term SoCs will be used when the text refers to only SoCs. The overloaded term microprocessor may tend to camouflage the differences between the two; therefore, it is highly recommended that users planning to use SoCs in aircraft systems fully understand the technical differences between microprocessors and SoCs.

The intended users include system developers, integrators, and all personnel in both industry and the FAA responsible for aircraft certification and system qualification.

As shown in figure 1, the Handbook relates current FAA policy and guidance (section 2) to both microprocessor and SoC risks and risk mitigation (section 3) and safety net assurance approaches (section 4). As described in section 3, the device risks and risk mitigation considerations may be used by project personnel to consider in the use of COTS microprocessors in aircraft systems. The safety net assurance approaches may be considered by project personnel as an approach to mitigating these risks. The information in this Handbook is intended to provide system design and safety assurance approaches that facilitate the use of complex COTS microprocessors without incurring the difficulties associated with legacy methods of design assurance by test and analysis of components. This Handbook is also intended to aid FAA personnel during aircraft certification and policy review. The authors hope the use of COTS microprocessors and their associated safety net approaches in aircraft systems will lead to enhanced FAA policy and guidance over time.

Both the FAA and aerospace industry members participated in the research on which this Handbook is based. Use of the Handbook content should be coordinated between project and regulatory personnel.

Acceptance of the safety net and the related assurance approaches documented in this Handbook are predicated upon the mutual development and acceptance of the project-unique design and assurance methodologies and their effectiveness in meeting safety and airworthiness requirements.



Note: The yellow blocks identify the scope of the Handbook.

Figure 1. Scope of the Handbook

1.2 DOCUMENT ORGANIZATION.

- Section 1 identifies the purpose and scope of this Handbook and identifies the research on which it is based.

- Section 2 provides an overview of the current regulatory considerations for using COTS microprocessors in airborne systems.

- Section 3 provides an overview for using COTS microprocessors, and some of the risks of using microprocessor technologies are described. These risks are described in terms of hardware and software considerations; tool use and limitations; development environments, information requirements and constraints, performance monitoring, fault insertion and analysis; simulation uses and constraints; and programmable characteristics (e.g., configuration control, excessive capabilities and disabling/enabling features, emerging technologies, and the resource-sharing effect on timing).

- Section 4 introduces safety net approaches; related changes to design, development, and safety; integrated device characteristics; operational considerations of safety nets, error/anomaly detection, recovery, and validation mechanisms; and architectural requirements for safety nets.

- Section 5 summarizes the Handbook and the future research that can be based on its content.

- Section 6 provides references.

- Section 7 contains a glossary defining the terminology used in this Handbook.

2. REGULATORY CONSIDERATIONS FOR MICROPROCESSOR-BASED AIRBORNE APPLICATIONS.

Current FAA policy and guidance does not directly address the use of COTS microprocessors and SoCs in aircraft systems. However, the existing policy and guidance can be used as a basis from which this Handbook builds upon to help provide an applicant with an acceptable means of demonstrating that their system meets the applicable airworthiness requirements.

FAA Advisory Circular AC 20-152 [6] notes that hardware life cycle data may not be available to satisfy the objectives of RTCA/DO-254, and therefore, alternative methods or processes are required to ensure that COTS microprocessors perform their intended functions and meet airworthiness requirements. No other guidance regarding COTS microprocessors is provided.

FAA Order 8110.105 [7] limits the discussion of COTS components to COTS Intellectual Property (IP), which it defines as commercially available, functional logic blocks that are implemented within custom micro-coded components, such as programmable logic devices, field-programmable gate arrays (FPGA), or similar programmable components. Although no specific policy or guidance is provided for COTS microprocessors, it does mention COTS processor cores. Many of the concerns presented in FAA Order 8110.105, section 4-9.b also apply to COTS microprocessors:

> "(1) COTS components, including IP, are developed by a company other than the applicant and hardware developer. Intended to provide specific functions or abilities in many different applications, COTS components may or may not have been developed using a rigorous design assurance method (such as RTCA/DO-254). Given this, we must ensure that the applicant and hardware developer show that using COTS IP complies with the applicable airworthiness requirements, regulations, policy and guidance for that project." [7]

> "(2) Availability of COTS IP doesn't automatically guarantee that it can be used in a manner that complies with airworthiness requirements, regulations, policy and guidance. Depending on the complexity of the COTS IP and the availability of IP documentation, applicants and/or hardware developers may have significant work to show compliance for the system or equipment." [7]

> "(3) Using a COTS IP in a simple electronic hardware/complex electronic hardware (SEH/CEH) device that is installed in airborne systems or equipment should satisfy applicable functional and safety-related requirements. RTCA/DO-254 Section 11.2 may not be sufficient for design assurance of a

> COTS IP implemented in a SEH/CEH that supports level A and B aircraft, and other safety critical, functions. As a result, applicants may need to develop or augment system architectural mitigation, component verification, testing, analysis and other life cycle data of a COTS IP. All this is needed to demonstrate its intended function, show it is free from anomalous behavior, satisfies applicable regulations, and meets airworthiness requirements." [7]

Although FAA Order 8110.105 does not specifically address COTS microprocessors, many of the issues applicable to COTS IP also apply to COTS microprocessors and SoCs, and therefore, some of the guidance of section 4-9.b.(4) of the Order can be used as a basis for the techniques presented in sections 3 and 4 of this Handbook:

- Paragraph (b) states that the applicant and system developer can use extensive testing and analysis to gain detailed information about the functionality, and how it operates during boundary and failure conditions. This should include testing and analysis of any functionality that will not be used or activated in the specific application. Section 3 of this Handbook presents some examples of activities that the system developer can perform to gain a better understanding of the system and, therefore, design mitigation techniques to avoid or contain an undesirable or catastrophic action by the microprocessor.

- Paragraph (c) states that architectural mitigations at the device, board, line replaceable unit, or system level can be employed to detect and/or mitigate unforeseen or undesirable operation. Sections 3 and 4 of this Handbook present some examples of mitigation techniques that the system developer can employ to eliminate or minimize the effect of an error that occurs as a result of an event either internal or external to the microprocessor.

Note that it is not intended for the techniques in this Handbook to be applied to COTS IP soft processor cores, since it is expected that the developer will have the appropriate life cycle data available and DO-254 processes apply [8].

The techniques presented in this Handbook are intended to help developers provide additional layers of protection through the development of mitigations that are incorporated at the appropriate level of the design. As microprocessor technologies continue to evolve, these techniques will become more critical in assuring the safe operation of the airborne systems in which they are employed. However, it is important to note that these techniques, such as safety nets, are in addition to, and do not replace, current FAA-accepted methods for selecting a microprocessor and designing and testing the system in which it is used. These methods include:

- Testing operating system and hosted applications on the actual hardware to be approved.

- Monitoring current errata sheets and technical notes so that developers are aware of known or newly discovered problems, undocumented features, and microprocessor limitations that may adversely affect system operation. There should be a process in place to determine if any actions are required as a result of changes to the errata sheets or technical notes.

- Establishing and maintaining a configuration control plan for the microprocessor throughout the life cycle of the system. There should be a process established for the microprocessor manufacturer to provide notification of any changes to the microprocessor, and for the design approval holder to review the changes and determine the effect on the operation of their system.

- Microprocessors used in applications where failure could result in catastrophic or hazardous failure conditions should have appropriate service experience so that the applicant has confidence that the device is mature and design deficiencies are known.

- The microprocessor should be operated within the environmental limits established by the microprocessor manufacturer. If these limits are exceeded in the operational environment, the applicant should verify through testing that all production microprocessors will meet the system environmental requirements.

As previously noted, the airborne market for these devices is insignificant when compared to other markets such as consumer electronics, automobiles, and telecommunications. Applications in these other markets are less susceptible to anomalous behavior resulting from internal and external events. Therefore, the aviation industry should learn how to cope with the increasing complexity and resulting vulnerabilities of these devices. The authors of this Handbook hope that the techniques and guidance provided herein will be used by system developers to better understand and mitigate the vulnerabilities of their devices, and that certification authorities will embrace these techniques and continue to pursue relevant policy and guidance to the aviation community.

3. THE COTS MICROPROCESSOR COMMON RISKS AND RISK MITIGATION.

This section provides an overview of using COTS microprocessors and the risks presented by their deployment in aerospace systems.

It is important to distinguish between discrete COTS microprocessors and SoCs when discussing certain elements of risk and safety analysis. The task of ascertaining safety considerations is more complicated for SoCs due to the broad variety in SoC designs. Unlike the case of discrete COTS microprocessors, where the majority of features of interest are similar across most microprocessors, SoC components tend to vary significantly based on the product selected, making safety analysis more complicated. Not only are the safety concerns due to the features of individual IP cores an issue, but interaction amongst them presents a verification challenge to system designers. Additionally, certain systems using SoCs may not require particular on-chip cores and would require the disabling of those cores for safety reasons.

Resources used by safety nets are expected to change as the underlying technology of the system it is protecting (or implemented within the safety net itself) evolves. An example of this occurs when configuration register changes (the addition of new, removal of old, exposure of hidden/reserved registers) are needed because separate, discrete microprocessors; system controllers; and other components are replaced with a single SoC. The safety net design must be examined and perhaps adjusted when such technology changes are implemented or when the device must be approved within a new application. The discussion above needs to be considered while formulating a safety net.

After reviewing the designs of a variety of modern COTS microprocessors across a group of manufacturers, product families, and technology generations, three areas of common risk were identified for all these devices [4]:

- Visibility and Debug—the inability to observe the internal operation of the device during system use and development (see section 3.1.).

- Configuration-Related Issues—software accessibility to device configuration during system operation (see section 3.2).

- Resource-Sharing Considerations—performance unpredictability due to multiple on-chip shared resources (see section 3.3).

3.1 VISIBILITY AND DEBUG.

During background research, a physical target computer environment and a simulated target computer environment were setup to perform experiments. Setting up these environments exposed visibility and debug challenges that could arise during system development and analysis; some of these challenges are described in the remainder of this section. The system developer should document the configuration of the test environment and identify any differences, limitations, and constraints of the simulated environment, if used, in relation to the physical environment. Based on the visibility and debug challenges identified, system developers should consider the following items when setting up their evaluation environment.

3.1.1 Target Computer Environment.

A target, or physical, computer environment was established to perform experiments and provide evidence and insight, which led to the conclusions of the research. The target computer environment included a development board for the SoC selected for evaluation, board support software, a COTS operating system, and COTS tools for development and debug. (Details can be found in reference 5.) The following software and hardware considerations were noted during the target computer environment evaluation.

Software considerations:

- Software integration—Operating systems, license agreements, and board support software are typically included with target development boards. System developers should work closely with the integrated circuit device manufacturer when setting up the development environment to ensure compatibility of tools selected for the development environment.

- Debug capabilities—Tools provide visibility and debug capabilities to assist with system and application software development. These tools should be evaluated to verify that they provide the capabilities needed for the system being developed.

- Multicore support—Additional tool capabilities are required to effectively use, debug, and develop multicore SoC applications. Beyond the typical compile and debug capabilities used in single-core development, individual and group control of the processing cores are important in testing multicore applications. Also, some projects may require programming language support for multicore development, including multithreading capabilities and shared access to on-chip devices.

- Integrated development environments (IDE) for safety-critical embedded systems—Safety-critical embedded system designers may find that they are limited in the available choices for IDE compared to standard embedded system development. Applicants should be aware of the IDE's suitability with respect to their specific project requirements in addition to tool qualification issues.

Hardware considerations:

- Limited visibility into device internal operation—manufacturer documentation is intended to support the use of the device based on the capabilities described in the User's Manual and device characteristics documented in the data or errata sheet. The internal operation is frequently proprietary information. For example, the User's Manual may document how to configure the use of cache memory, but the cache algorithm being used internally by the device may be proprietary.

- Performance monitoring—Hardware performance monitors may be provided by the manufacturer to provide insight into the internal operation of a microprocessor. These monitors allow system designers to track various system activities and performance statistics during application development and execution. Typical uses of hardware performance monitors include gathering level 1 and level 2 cache statistics and measuring cycle counts to estimate application execution times. Performance monitors may also allow a system designer to observe the activity of functions resident on the microprocessor, including memory controllers, PCI-Express (PCIe) and Ethernet controllers, and direct memory access engines. These performance monitors are uniquely designed for each COTS microprocessor, and access to the performance monitors typically requires custom software and is not fully supported by COTS operating systems.

- Fault insertion—The ability to insert faults internal to the microprocessor may be limited or not achievable. An applicant may not be able to demonstrate that faults are correctly detected if the faults cannot be injected. The safety net methodology should mitigate these types of faults.

- Industry benchmarks—The use of industry benchmarks may help demonstrate timing characteristics and the potential effect of shared resources on timing. The benchmarks that were investigated used memory to simulate the hardware I/O and did not exercise the microprocessor-shared hardware resources on the target computer. The limitations of industry benchmarks to fully exercise microprocessor behavior should be understood and augmented with other tests.

3.1.2 Simulated Computer Environment.

A simulated computer environment was established to perform experiments and provide evidence and insight, which led to the conclusions of the research. The environment included a host environment, a simulation of the microprocessor device selected for evaluation, COTS simulation support software, a COTS operating system, and COTS tools for development and debug. The following software and hardware considerations were noted during the simulated computer environment evaluation.

The software considerations were:

- Applicants should be aware that the primary focus of the microprocessor simulation is on application software development. Typically, hardware evaluation is a secondary concern, if addressed at all.

- Modeling configuration and internal registers—The simulated computer environment typically models the minimum set of configuration and internal registers to support software execution. The differences between the simulated computer environment and the target computer should be documented by the system developer as part of the test environment. Evaluating the system response in a simulated environment requires accurate modeling of all configuration registers.

The hardware considerations were:

- Limited modeling of hardware—Hardware interfaces and modeling of microprocessor functions may be limited to those items required for application software development.

- The timing and cycle accuracy of the simulated target computer should be assessed. If the target computer model is not cycle accurate, functions which are timing sensitive should be verified in the target computer environment.

- Modeling of device performance—Internal microprocessor performance monitors may not be modeled in the simulator.

- Focus of microprocessor models is on the simulation of core central processing units, modeling of other microprocessor-resident functionality may be limited.

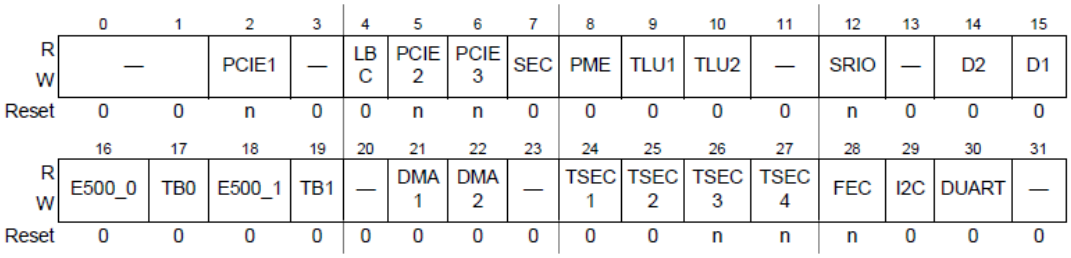
3.2 CONFIGURATION-RELATED ISSUES.

Configuration register changes are a growing concern for avionic microprocessor applications since continued device integration has allowed an increasing proportion of the entire system to be configured through software. If a microprocessor is not configured properly, erroneous behavior, including improper data processing, stack overflows, erroneous interrupts, machine checks, data loss, data corruption, or inadequate throughput, may occur.

Over time, the number of configuration registers per microprocessor has grown significantly. For example, the Freescale™ MPC8572 SoC has more than 500 software-accessible configuration registers that control basic functionality of the processing cores and on-chip devices [9]. In addition to these configuration registers, various device functions may be set externally via pullup/pulldown pins, which are sampled shortly after a hardware reset or internally via software. Incorrect settings or inadvertent changes are areas of concern that must be addressed.

In general, the capabilities of most microprocessors exceed what is required by typical applications. Care should be taken to provide assurance that unused capabilities are properly disabled. In legacy avionics with many discrete system devices, this concern was addressed through physical disconnection of the unused devices to power sources and the rest of the system. However, COTS SoCs have removed the physical separation between devices and processors and have given control of device configuration to software. Therefore, the deactivation of unused features has become an additional consideration within the set of configuration-related issues. In addition to proper deactivation of unused features and devices, system designers should assure that those features and devices cannot be reactivated through erroneous software or environmental effects. Inadvertent activation of an unused device may cause unintended and undesirable operation, such as erroneous interrupts, data loss, data corruption, machine check cycles, and stack overflow, among others.

As an example of how the deactivation of unused features has changed for modern COTS microprocessors, figure 2 shows the Freescale MPC8572 device disable register.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	—		PCIE1	—	LB C	PCIE 2	PCIE 3	SEC	PME	TLU1	TLU2	—	SRIO	—	D2	D1
Reset	0	0	n	0	0	n	n	0	0	0	0	0	n	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	E500_0	TB0	E500_1	TB1	—	DMA 1	DMA 2	—	TSEC 1	TSEC 2	TSEC 3	TSEC 4	FEC	I2C	DUART	—
Reset	0	0	0	0	0	0	0	0	0	0	n	n	n	0	0	0

Figure 2. Freescale MPC8572 Device Disable Register [9]

This register is used by the system to determine whether each of the major on-chip devices of the system should be enabled or disabled by setting particular fields of the register to 0 (enabled) or 1 (disabled). This configuration register contains 24 single-bit readable and writeable fields that are associated with the 24 on-chip devices of the SoC. For example, bits 16 and 18 control the two E500 processing cores [10], bits 2, 5, and 6 control the three PCIe controllers, and bits 24 through 27 control the four triple-speed Ethernet controllers. A single bit flip in this register can disable a critical on-chip device or one of the processors, and research has demonstrated that erroneous writes to this register can render the system inoperable [5]. Therefore, system designers should be aware of the risks of software erroneously modifying the system configuration space.

Several microprocessor devices also offer the user the flexibility of locating the configuration registers in external memory space, which makes them more vulnerable to corruption. This capability also makes the system susceptible to loss of all configuration data if the pointers to the external memory locations are corrupted.

Configuration registers may change by software errors (inadvertent writes), by single event upsets (SEU), hardware defects, hardware faults (such as a noisy power supply core voltage, signal integrity issues, ground bounce, etc.), or electromagnetic interference (EMI). Refer to appendix B for a description of power supply-related issues.

In light of the risks identified with the current technology and trends and the criticality of proper microprocessor configuration register settings, the risks associated with incorrect configuration register settings should be understood. Each register should be assessed for:

- Intended setting for each register bit within the system being implemented and the reason for selected setting

- Identification of each operational phase at which each register is being set (i.e., initial power up, power on reset, built-in test (BIT), and exception processing)

- Identification of disabled functions

- Impact to system if the state of the register bit is unintentionally changed

    - It is recommended that the impact of an inadvertent change to critical registers or registers of questionable impact, be verified through simulation, if possible.

    - The simulation environment should implement an algorithm that has the ability to randomly change configuration settings.

- Timing and rate of impact to system of an unintentionally changed register bit.

- Errata sheet information

Once this analysis has been completed, safety net methodologies can be employed to mitigate the identified risks. Refer to section 4 for information on safety net methodologies.

3.3 RESOURCE-SHARING CONSIDERATIONS.

Modern microprocessors differ from previous technology in that many processing, memory, and I/O components reside within a single device, and many of these components are designed to be shared to optimize system performance. The multiple processing components compete to initiate requests to their memory and I/O targets. Additionally, the I/O components can initiate requests to memory targets through direct memory access. Whether initiated by a processor or I/O controller, requests travel over shared on-chip interconnects, typically a single on-chip bus. This Handbook defines shared resources as any on-chip or off-chip components that are accessible to multiple initiators. Examples of shared resources include on-chip memory controllers, hardware accelerators, level 2 (L2) and/or level 3 caches, on-chip busses and, on-chip Ethernet controllers.

Modern microprocessors feature a number of shared resources. As shown in figure 3, the two e500 cores of the Freescale MPC8572 share a single L2 cache [9]. Additionally, the two processing cores share a single on-chip bus to access the other major components of the system, including the various I/O controllers and memory controllers.

It is highly recommended that the system designer analyze the access protocol for these shared resources and the run-time behavior of all programs that share a given resource. Based on this analysis, the designer should ensure that even with the sharing of resources, the system will continue to run in a predictable manner (continuous operation). In some conditions, hard shutdowns can be an acceptable safety protection mechanism.

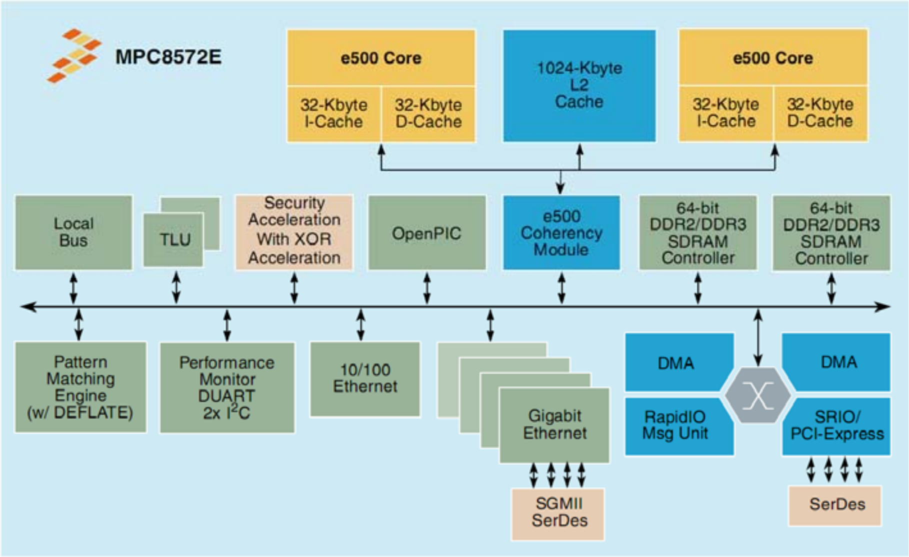


Figure 3. Freescale MPC8572 Block Diagram [9]

Sharing resources is a major contributor towards nondeterminism and worst-case execution time (WCET) analysis challenges in modern COTS microprocessors. Nondeterminism arises because the availability of a shared resource becomes largely dependent on the run-time behavior of other processes sharing the same resource. In many cases, the run-time behavior of programs is data-dependent and cannot be predicted offline. WCET analysis depends on understanding all conditions that lead to timing delays and then bounding for worst-case conditions. Multiple shared resources on a single device complicate this analysis due to the great increase in the number of delay conditions.

Pellizoni, et al. [11], describe the challenges in predicting the WCET in a multitasking system. They show that, due to the interference between cache-fetching activities and I/O peripheral transactions, tasks can suffer computation time variance of up to 46% in a typical embedded system.

To further assess the timing delays and nondeterminism caused by resource sharing, additional experiments were performed using a simple matrix multiplication program on the Freescale MPC8572 platform considering the L2 cache as the shared resource [5]. Each processing core executed its own copy of a matrix multiplication program that required using the shared L2 cache due to the program size. The experiments showed that the execution time of the matrix multiplication program can increase by as much as 17% as L2 cache interference increases.

Moscibroda, et al. [12], describe that in a multicore system, multiple programs running on different cores can interfere with each other's memory access requests, thereby adversely affecting performance. They show that a competing program running on one processing core can result in a denial of service (DoS) on the other processing core, due to the inherent unfairness in memory controller access policy. The performance of a blocked application can be reduced by as much as 2.9 times in a typical dual-core system. Moscibroda, et al., identify the memory access scheduling algorithm as the main source of inequality in memory access, allowing the DoS to occur. Shared resource access policy and scheduling is addressed in section 4.3.1.

Industry trends indicate that the ratio of processing cores to various shared resources in COTS SoCs will increase over time. Instead of two processing cores sharing a common cache, memory controllers, and I/O devices, there will be four, eight, or more processing cores sharing these resources. This will increase the competition for shared resources among processing cores, worsening potential unpredictability issues.

4. SAFETY NETS.

A safety net in the context of this Handbook is defined as the employment of mitigations and protections at the appropriate level of aircraft and system design to help ensure continuous safe flight and landing.

Assuring that the microprocessor is absent of uncertainty at the device level alone is no longer feasible; therefore, a multilevel safety net becomes essential. A multilevel safety net concept considers the discrete devices, the circuits, individual components directly supporting the use of microprocessors, and higher-level system architectures. The multilevel safety net can be viewed as an architectural approach.

Multilevel safety net protection is significantly linked to the assigned design assurance level required by regulation and/or contractual obligation and the integrated complexity at the device level. Modern microprocessor technology necessitates the need for a multilevel safety net. For example, failure of a flight control system may result in a catastrophic condition, whereas the failure effects of a lavatory system may be limited to passenger discomfort. Therefore, a safety-critical system using a particular microprocessor requires a higher level of safety than a nonessential system using the same microprocessor. This demonstrates the need for a multilevel safety net approach where the safety net may respond differently in different levels of criticality. A failure in the flight control system will most likely require a high-priority response with limited recovery time, while a failure in the lavatory may only require a low-priority response with a longer recovery time. Error detection and recovery from anomalous conditions may occur at different hierarchical levels within the system (e.g., chip level, board level, subsystem, or system).

Since the criticality of the lavatory system is lower than the flight control system, fundamentals for a multilevel safety net should be established as an acceptable approach.

Industry research has shown that the susceptibility of random access memory (RAM)-based devices to external stimuli, such as cosmic rays (resulting in single event effects), lightning, and EMI, is generally greater than the devices that employ flash-based technology [13]. However, disregarding the potential effects of these types of threats on flash-based memory could lead to the same risks as the threats on RAM-based memory. For example, a multilevel safety net implementation in RAM-based designs that incorporate periodic memory content verification (scrubbing) to help ensure error-free operation in a flight-critical system may also incorporate the use of a non-RAM-based monitoring circuit and/or the use of an external discrete monitoring circuit. Since the safety net implementation would require the ability of both the primary and monitoring circuits to work together to provide the overall integrity, the monitoring circuit should be less susceptible to external threats as the primary circuit. A safety net implementation using the same device in a low-criticality system, such as the lavatory example, may use only a high-level monitor component, since system response time is of little concern.

Another example that continues to challenge the use of modern, highly integrated, complex devices are the core voltage (rail voltage) of COTS microprocessors. As the core voltages continue to decrease, it becomes more difficult to ensure that the devices are not susceptible to supply voltage fluctuations, noise, and environmental effects. Current industry data show that, as the size of the device decreases, the margin between the applied voltage and the parasitic energy of the device leads to unpredictable and unstable conditions.

The safety net methodology assumes that a microprocessor will misbehave. The absence of a properly implemented safety net reduces the ability to protect a system against the unexpected behavior, damage, injury, or instability of a device during its service life. Architectural designs, in general, are becoming a more complex, application-specific art form requiring the ability to detect, resolve, and validate component failure in a run-time environment to the required levels of availability and safety. The safety net methodology does not ensure determinism solely at the device level. Rather, safety nets are intended to provide the ability to protect against unintended or misleading behavior using a multilevel approach.

4.1 MICROPROCESSOR SELECTION AND SAFETY NET CONSIDERATIONS.

DO-254 does provide some guidance (see section 2) for the use of COTS products [8]. Further guidance and clarifications to DO-254 are provided in FAA Order 8110.105 Change 1 [7]. The safety net and system architecture should be designed concurrently, employing one or more of the techniques previously discussed as well as taking into consideration first-time use, availability, suitability, stability, and testability traits of the microprocessor.

Device selection continues to be a regulatory concern as microprocessor technology advances. This section was developed in part from research, from DO-178B [14] and DO-254 [8], and largely from existing issue papers and certification review items (CRI) that have been levied on a project-by-project basis by the regulatory agencies. The specific CRI and issue papers are considered to be confidential to the project for which they are levied against; therefore, the information contained within this Handbook is presented from a generalized viewpoint.

It has been expressed by the regulatory authorities through CRI and issue papers and DO-254 that in using a microprocessor, the integrator is responsible for managing the device service life through an electronic components management plan, taking into consideration the information contained within the device selection safety net considerations. These device selection safety net considerations are detailed in sections 4.1.1 through 4.15.

4.1.1 First-Time Use.

The first time a COTS microprocessor is selected for use in an aircraft system it should be evaluated with great attention to detail. Consider the following relevant questions: How long has the microprocessor been fielded? What has it been used for? Is there substantiated service history that can be evaluated? Additional care should be considered in the evaluation of its availability estimates (see section 4.1.2), stability evaluation (see section 4.1.3) and suitability (see section 4.1.4). The length of time the microprocessor has been fielded, the number of microprocessors currently in use, and the number of existing applications for that microprocessor provide insight during the selection process.

4.1.2 Availability.

Availability focuses on the maturity of the device and includes libraries, tape-out iterations, launch customer, and number of units currently fielded. The review attempts to discover unknowns as related to the manufacturers' intended use. The maturity of the manufacturers' supporting libraries can provide insight to determine when a device could become suitable for use and when variation is less likely to occur.

Tape-out maturity can provide an indication of where the manufacturer is within their development cycles. Manufacturers typically plan on several tape-outs prior to production release for the purpose of incrementally debugging the device operation and fabrication. Using a device with fewer tape-outs, as measured against the manufacturer's normal number of tape-outs, can indicate the risk of the manufacturer making changes without notifying the customer.

Interviewing initial customers of a COTS microprocessor and assessing their applications associated with the device can provide insight to the device maturity level as driven by the in-service use. Initial in-service use has become a debugging tool employed by manufacturers. In today's rapidly changing environment coupled with demands by customers that are not in the aerospace industry, manufacturers are generally given less time to extensively debug a product compared to past years. Industry trends indicate that the first generation use of a newly released device is less stable, thus requiring changes to be made by the manufacturer in most cases without the requirement to inform the users.

4.1.3 Stability.

Assessment of the device life cycle history can provide insight into the microprocessor's stability based on publicly available data, such as application notes and errata sheets. Microprocessors that are variations of an existing family may exhibit early operational stability versus a device that is not based on an existing family of devices.

4.1.4 Suitability.

Suitability involves the review of the intended application against the manufacturer's published performance data. When using microprocessors, available useful performance (actual rates) should be expected to be less than that implied by raw clock rates (published rates). Reasons for the decreased available performance includes the following: cache miss rate, interference between multiple execution units running simultaneously in parallel, and instructions that require multiple clock cycles to complete. In addition, the microprocessor manufacturer's fabrication process uses proprietary libraries that are intentionally not disclosed to customers and are accounted for in the overall performance and fabrication process.

Microprocessor startup times are an important aspect of suitability due to the stringent availability requirements for commercial avionics. If the microprocessor must be restarted due to error conditions during operation, the microprocessor startup time should be well understood when selecting devices and constructing safety nets. The following restart conditions should be considered for the safety net design:

- Time constraints for startup should cover all possible startup conditions based on the system design, design assurance level, and required response.

- Time constraints for startup should include the microprocessor configuration initialization time and any required initialization to satisfy system safety requirements.

Another important aspect of suitability involves the manufacturer changing device package sizes without making any additional changes, other than possibly core voltage. For example, experience has shown that a device that was manufactured in a 90-nanometer process and migrated to a 60-nanometer process will likely demonstrate a change in characteristics. There is no guarantee that simply changing package size will not cause a change in the application's performance and operation; and significant evidence shows that it can.

Another key area of suitability is the demonstration of an acceptable understanding of the embedded features within the device and the intended use. For those features that are not intended to be used, it has been expressed by the regulatory agencies that the integrator must demonstrate an acceptable understanding of potential misbehavior caused by the activation of an unused function. Additionally, the integrator must include the mitigation for disabling or deactivating a given feature.

4.1.5 Testability.

Tests to characterize device behavior and performance are typically performed at many levels, beginning with the manufacturer data. Designers may use configurable cores within their system that may not provide full visibility into their behavior due to encryption or other intellectual property limitations. The manufacturer's intended testability approach may not provide enough insight, may not be consistent with the integrator's testing approach, or may not be suitable for the intended application. Industry experience has shown that an unexpected behavior can occur with a difference in testing approaches between the integrator's expectation and the manufacturer's intentions. Some proven techniques for characterizing device behavior are:

- Manufacturer's intended testability approach
- Error-correcting code
- Built-in self test (BIST)
- Automatic test pattern generation

The multilevel safety net approach defined in section 4.2 encompasses safety compliance at various levels as appropriate to the associated safety risk. Research conducted reveals the escalating future impracticality of ensuring safety at the device level alone when COTS microprocessors and equivalent airborne electronic hardware technology are used [4].

4.2 SAFETY NET MONITOR CONSIDERATIONS.

The term safety net monitor is intended to convey technology approaches that could be employed through the use of hardware and/or software solutions. The safety net methodology presumes the device will misbehave during its service life, and therefore, the safety net monitors would likely need to be architecturally integrated at a level above the device itself.

It is the intent of this Handbook to help identify some safety net monitors that were discussed and/or physically tested through example design. The responsibility for defining and using safety net monitors belongs to the integrator developing the application-specific architecture.

4.2.1 Hardware Monitor Considerations.

Hardware monitors focus on hardware-based technology solutions to indicate when unexpected changes in behavior or physical characteristics occur. The list below identifies the types of hardware monitors discussed during the research. It is recognized that some of these considerations could also be implemented using software applications.

- Memory (parity/error correction code/SEU monitors)
- Internal buses (address and data parity)
- External buses (protocol checks, parity, checksums, activity patterns, cyclic redundancy code)
- Discrete digital signals (dualize I/O, BIT stimulation capability, wraparounds)
- General signals (wraparounds analog/digital to digital/analog, range checks, rate checks, etc.)
- Dissimilar hardware (address generic hardware fault coverage)
- Configuration register monitors

4.2.2 Software Monitor Considerations.

Software monitors focus on software application solutions to indicate when unexpected changes in behavior or physical characteristics occur. The list below identifies the types of software monitors discussed during the research. To better support the safety net methodology, the application-specific architecture would need to consider the risk of the microprocessor misbehaving and having an adverse effect on the monitor's ability to accurately report abnormal behavior.

- Additional system-level and box-level built-in-test (SBIT: Start-up BIT; IBIT: Interface, initiated, interruptive, or intermittent BIT; PBIT: Periodic BIT; MBIT: Maintenance BIT)
- Additional voting planes
- Data integrity checks
- Configuration register monitors

4.2.3 External Monitor Considerations.

External monitors are identified below where each microprocessor has its own custom dedicated watchdog monitor (WDM) for reporting abnormal behavior.

- WDM power up failed and must be properly serviced to be declared valid.

- WDM should be serviced once per frame.

- WDM has an independent clock source and voltage rail.

- WDMs are designed with custom requirements to the processor, design configuration, criticality, and the utilization.

- WDM failures are reported through cross-channel data links to the other channels and voting planes.

The following are types of techniques discussed as potential solutions during the research, but this list is not exhaustive.

- External monitoring of safety-related behavior
- External redundancy
- External watchdogs
- External architecture that allows a level of autonomous run-time correction

4.2.4 Internal Monitor Considerations.

The research acknowledges the use of internal monitors from the aspect of those that can be embedded, such as BIST, at the transistor level as associated with a multicore device. The level of research conducted with internal monitors was limited to the association of software monitors discussed in section 4.2.2.

4.3 ARCHITECTURAL SAFETY NET EXAMPLES.

The architectural safety net examples in sections 4.3.1 through 4.3.5 were identified by research as possible approaches and are discussed only briefly. The details behind using these examples are understood to be the responsibility of the integrator, as applicable to the application-specific solutions. These examples are not meant to be a complete list of approaches, as application-specific solutions may differ greatly from these examples. It is not the intent of the research or this Handbook to provide implementation details.

4.3.1 Shared Resource Approach.

A safety net design to mitigate shared resource effects depends on how access is granted to those shared resources. This section details the different forms of shared resource access policy and provides an example of how that policy may affect safety.

Access to any shared resource will be controlled by an arbiter. An arbiter can be implemented in hardware or software, and it is possible that the arbiter can be configured through configuration registers. There are three arbiter properties that affect the behavior of the system when using shared resources:

- Access fairness
- Tenure fairness
- Disconnection policy

Access fairness provides a mechanism that allows agents to request permission to use a shared resource. Access fairness does not enforce how long the “winning” requesting agent may use the resource (tenure) nor does it assure productive utilization versus nonproductive occupation. Basic types of mechanisms to implement access fairness include the following:

- First-come, first-served—Initiators are granted access in order of the arrival of their requests at the arbiter. This can lead to potential access starvation if an initiator constantly requests access.

- Round robin—Initiators are granted access in a prearranged, fixed sequence, bypassing initiators that do not request access.

- Priority—Initiators are granted access according to a weighted priority system, with “higher”-priority requests being granted access before “lower”-priority requests.

- Weighted round robin—Initiators are allocated multiple access opportunities, either sequentially or nonsequentially, from a fixed number of possible opportunities.

- Multilevel (round robin with priorities)—This is a derivative of round robin where initiators on an higher-priority level are granted access in sequence, along with a reservation for a single initiator at a lower-priority level. After all the initiators at the higher-priority level have been granted an access opportunity, the reservation is offered to the first of the initiators at the lower-priority level. The process repeats with all the higher-priority level initiators again given access opportunity, followed by the reservation offered to the next initiator in the lower-priority level sequence.

Shared resource tenure fairness (utilization allocation) enforces how long the requesting initiator may use the shared resource once it has acquired access to it. It is possible that an arbiter does not implement tenure fairness, and an initiator is allowed indefinite access to a shared resource. The two most common mechanisms for enforcing maximum access duration are the following:

- Timers—Enforce tenure by number of clock cycles
- Counters—Enforce tenure by number of events

Disconnection policy can also affect the timing behavior of a COTS microprocessor. The arbiter can either forcibly disconnect an initiator or signal the initiator to disconnect. Signaling the initiator to disconnect is less temporally rigid than forcing disconnection, but this allows the initiator to gracefully terminate its transaction.

For example, standard memory controllers implement a priority-based access fairness policy based on the request address to maximize bandwidth. There is no notion of tenure fairness or disconnection policy in memory controllers because a typical access has a very short duration, and memory controllers only consider the request address, not the initiator, in scheduling memory requests. Therefore, a memory controller can starve other processing cores of access if it can ensure that its requests always receive the highest priority. A safety net should account for this shared resource access policy.

Another safety net example for shared resources is register access control. The memory management unit (MMU) can be used to manage register access by only allowing a limited subset of the software to access these registers, regardless of assigned criticalities. This has the potential of minimizing the possibility of corruptions. This technique has also been found to be effective when used for other memory-addressable, microprocessor-shared resources. Through MMU enforcement, only privileged software that is intended to use a given resource is allowed to do so.

4.3.2 Lock-Step Approach.

The research recognizes that the lock-step approach has proven to be a successful approach for ensuring safety. However, the research also acknowledges that the ability to ensure the same level of safety as previously shown is becoming more difficult as the technology advances toward the SoC and multicore microprocessor. This is believed to be inherently due to the embedded management between the cores. Thus, the integrator has less insight into the SoC, which challenges its ability to tightly synchronize multicores. No detailed research was performed on this approach.

4.3.3 Frame-Lock Approach.

Like the lock-step approach, the frame-lock approach has proven to be successful for assuring safety. Unlike the lock-step approach, the comparison of the independent execution results are performed at the I/O boundaries rather than at the instruction level, and synchronization occurs at the completion of a predefined frame. The data from each processor is compared at the end of this frame. Because frame-lock compares the results at a higher level of granularity than lock-step, challenges involving WCET variation during comparison can be avoided since the results must be available by the end of the frame. However, erroneous microprocessor behavior may go undetected for longer periods of time compared to lock-step. This concern is accentuated in light of research that shows it is increasingly difficult to provide safety assurance at the device level. For example, unintended configuration register changes may result in significant variations in processor operation, and this erroneous behavior may occur for an unacceptable period of time. No detailed research was performed on this approach, but safety net methodologies should account for this type of behavior.

4.3.4 Dissimilar Architectures.

It is recognized that using dissimilar architectures has proven to be a successful approach for providing safety assurance. Unlike lock-step where technology challenges determine the feasibility of that approach, the feasibility in using dissimilar architectures is driven by the significant increase in the level of effort. This increase in effort is indirectly caused by the increase in microprocessor technology challenges, essentially requiring the integrator to develop a dual system to perform an identical operation across dissimilar architectures. Each dissimilar architecture used may include its own unique risks, and therefore, different risk mitigations and safety nets should be designed for each architecture. No detailed research was performed on this approach.

4.3.5 Aircraft-Level Safety Assurance.

It has been proven that aircraft-level architectural approaches are successful for providing safety assurance and are successfully handled through project-specific special conditions. No detailed research was performed on this approach.

4.4 SAFETY NET CONCLUSION.

The multilevel, safety net approach encompasses acceptable safety compliance at various levels as the safety risk increases. This research supports the future impracticality of ensuring safety at the device level alone when microprocessors and equivalent AEH technology is used [4].

5. RESULTS AND FUTURE WORK.

5.1 RESULTS.

This Handbook has identified common risk areas for COTS microprocessors (section 3) and recommends approaches to reduce/mitigate these risks (sections 3 and 4).

The research has justified the need for a multilevel safety net approach (section 4) to handle anomalous behavior from complex COTS microprocessors used in aircraft systems. Additionally, this Handbook relates the existing FAA guidance and policy to the provided safety net approach and identifies the path to include safety nets in future policy and guidance (sections 2 and 4).

The safety net approach described in section 4.1 is a potential solution for increasingly challenging components that will be tested and validated by detecting and recovering from anomalous microprocessor behavior during system operation.

The safety net approach should be implemented on a project-by-project basis. It is highly recommended to design the safety net concurrently with the system architecture. This can lead to standards for preferred architectural patterns for resilient systems with future COTS microprocessors and other complex AEH and their emerging technologies, features, and functionality.

5.2 FUTURE WORK.

The FAA has suggested a follow-on AVSI project to evaluate other AEH beyond COTS microprocessors and an update to this Handbook to reflect the results of the follow-on project. AEH considered in the follow-on project is expected to include FPGA-based SoCs.

Additionally, the follow-on project proposes to guide one or more pilot projects in the use of safety nets for microprocessor-based aircraft systems. Pilot projects in the continuing development, application, and refinement of the safety net approach could provide additional content and definition to this Handbook in the form of examples and real-world analyses of the effectiveness of safety net approaches.

It is the intention of the FAA to assess the use of safety nets in future microprocessor-based aircraft systems. Using the safety net approach, resulting in certification of aircraft containing systems with safety nets, can lead to enhancements in FAA policy and guidance to formally accept, implement, and provide guidance for the continued use of safety nets.

Additional research should

- investigate additional functionality that can be accomplished within safety nets.

- investigate new trends in microprocessor design that may aid or hinder the implementation of safety nets.

- investigate architectural and functional requirements for the safety net monitoring itself.

- investigate the implementation of the architectural safety net examples identified in section 4.3.

6. REFERENCES.

1. Mahapatra, R.N. and Ahmed, S., “Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 1 Report,” FAA report DOT/FAA/AR-06/34, December 2006.

2. Mahapatra, R.N., Bhojwani, P.S., and Lee, J.D., “Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 2 Report,” FAA report DOT/FAA/AR-08/14, June 2008.

3. Mahapatra, R.N., Bhojwani, P.S., and Lee, J.D., “Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 3 Report,” FAA report DOT/FAA/AR-08/55, February 2009.

4. Mahapatra, R.N., Lee, J.D., Gupta, N., and Manners, B., “Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 4 Report,” FAA report DOT/FAA/AR-10/21, September 2010.

5. Mahapatra, R.N., Lee, J.D., Gupta, N., and Manners, B., "Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 5 Report," FAA report DOT/FAA/AR-11/5, to be published.

6. Advisory Circular 20-152, "RTCA, Inc., Document RTCA/DO-254, Design Assurance Guidance for Airborne Electronic Hardware," Federal Aviation Administration, AIR-100, 2005.

7. FAA Order 8110.105, "Simple and Complex Electronic Hardware Approval Guidance," Federal Aviation Administration, AIR-100, 2008.

8. DO-254, "Design Assurance Guidance for Airborne Electronic Hardware," RTCA, Inc., 2000.

9. "MPC8572E PowerQUICC III Integrated Host Processor Family Reference Manual," Freescale Semiconductor, Rev. 2, May 2008.

10. "PowerPC e500 Core Family Reference Manual," Freescale Semiconductor, Rev. 1, 2005.

11. Pellizoni, R. and Caccamo, M., "Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems," *IEEE Transactions on Computers*, Vol. 59, Issue 3, pp. 400-415.

12. Moscibroda, T. and Multu, O., "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," *Proceedings of the 16th USENIX Security Symposium*, 2007, pp. 257-274.

13. Agosteo, S., et al., "First Evaluation of Neutron Induced Single Event Effects on the CMS Barrel Muon Electronics," *Proceedings of the 6th Workshop on Electronics for LHC Experiments*, 2000, pp. 240-244.

14. DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, Inc., 1992.

7. GLOSSARY.

Arbiter—A device that controls access to a shared resource.

Architecture—The configuration of any equipment or interconnected system or subsystems of equipment that is used in the automatic acquisition, storage, manipulation, management, movement, control, display, switching, interchange, transmission, or reception of data or information, which includes computers, ancillary equipment, and services, including support services and related resources.

Availability—That proportion of time that a system is in a functioning condition. Timely, reliable access to functionality and data for authorized users.

Certification—"Legal recognition by the certification authority that a product, service, organization or person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization or person and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval or other documents as are required by national laws and procedures. In particular, certification of a product involves:

a. The process of assessing the design of a produce to ensure that it complies with a set of standards applicable to that type of product so as to demonstrate an acceptable level of safety.

b. The process of assessing an individual product to ensure that it conforms with the certified type design.

c. The issuance of a certificate required by national laws to declare that compliance or conformity has been found with standards in accordance with the above two items" RTCA/DO-254, April 19, 2000.

Initiator—A system component that requests a service from another system component.

Launch customer—The first delivery of a microprocessor to a customer.

Lockstep—A fault-tolerant mode of operation in which redundant systems perform the same activity at the same time in parallel. This allows the output of these activities to be compared so that any differences in output can be identified and resolved.

Softcore IP cores—A reusable processor or peripheral core purchased or licensed from a third party. This core is provided as a configurable piece of software code that can be used to generate a hardware design. These IP cores may provide full design visibility to the user, or it may be encrypted by the vendor due to intellectual property concerns. These IP cores are typically used in FPGA-based designs, but may be used in Application-Specific Integrated Circuit designs.

System-on-Chip (SoC)—A single integrated circuit that includes processing cores, cache memory, memory and interface controllers, timing and power management circuits, and other functionality. SoCs have replaced many discrete system components with a single device.

Tape Out—The final step in the design cycle for hardware. The design's photomask is sent to the device fabricator for production.

Walking one pattern—The sequential setting and resetting of each bit in a configuration register. Setting a bit assigns the value 1 to that bit, and resetting a bit returns its value to 0. Essentially, the value 1 "walks" the configuration register one bit at a time. The purpose of a walking one pattern test is to verify the functionality of a system during the individual modification of each bit in a set of configuration registers. The following example is a typical walking one pattern for a three bit input: 000, 100, 010, 001. Variations to the walking one pattern test exist. This includes the walking XOR pattern where each bit is flipped and then returned to its original value sequentially, as in the experiment described in appendix A, section A.3.

APPENDIX A—RESEARCH EXPERIMENTS

This appendix describes experiments on the Freescale™ MPC8572DS platform. These experiments were performed to evaluate the severity of the common risk areas identified in section 3 of the main report. Figure A-1 shows the complete system diagram of the experimental platform.

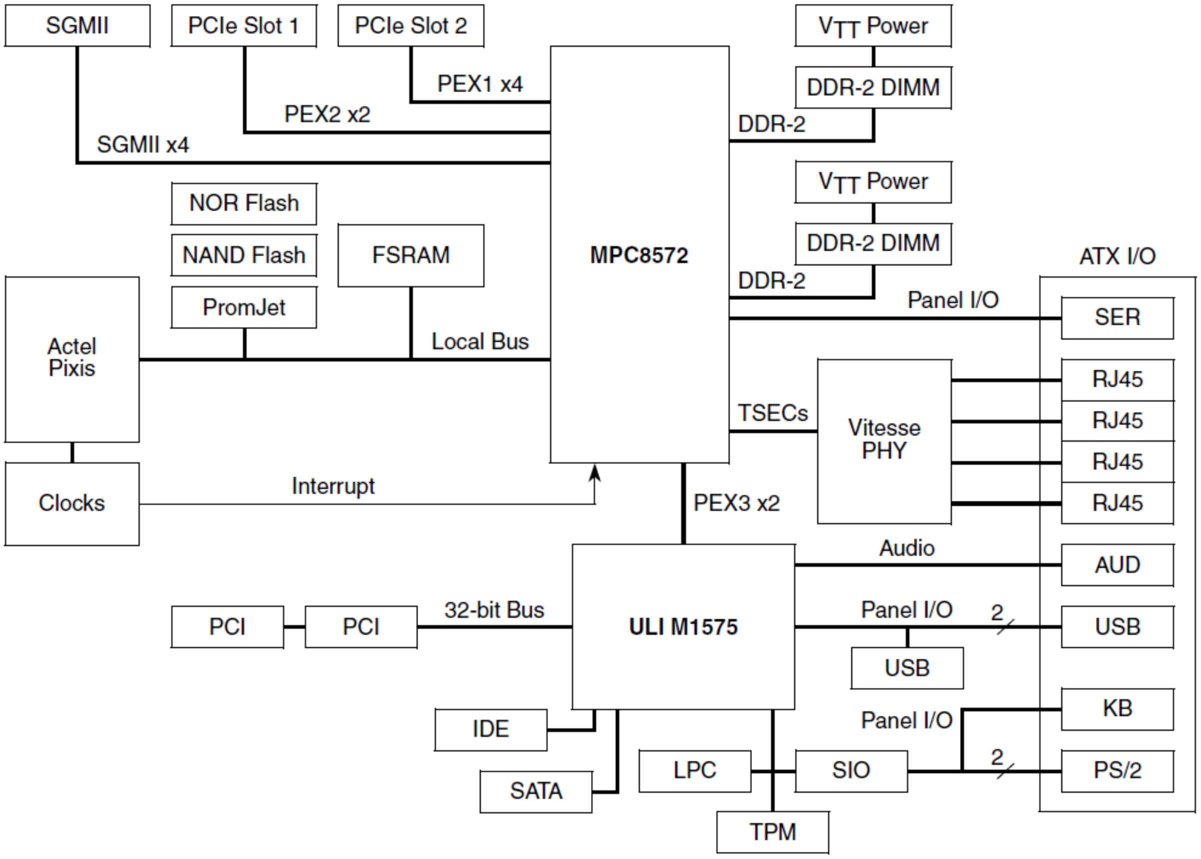
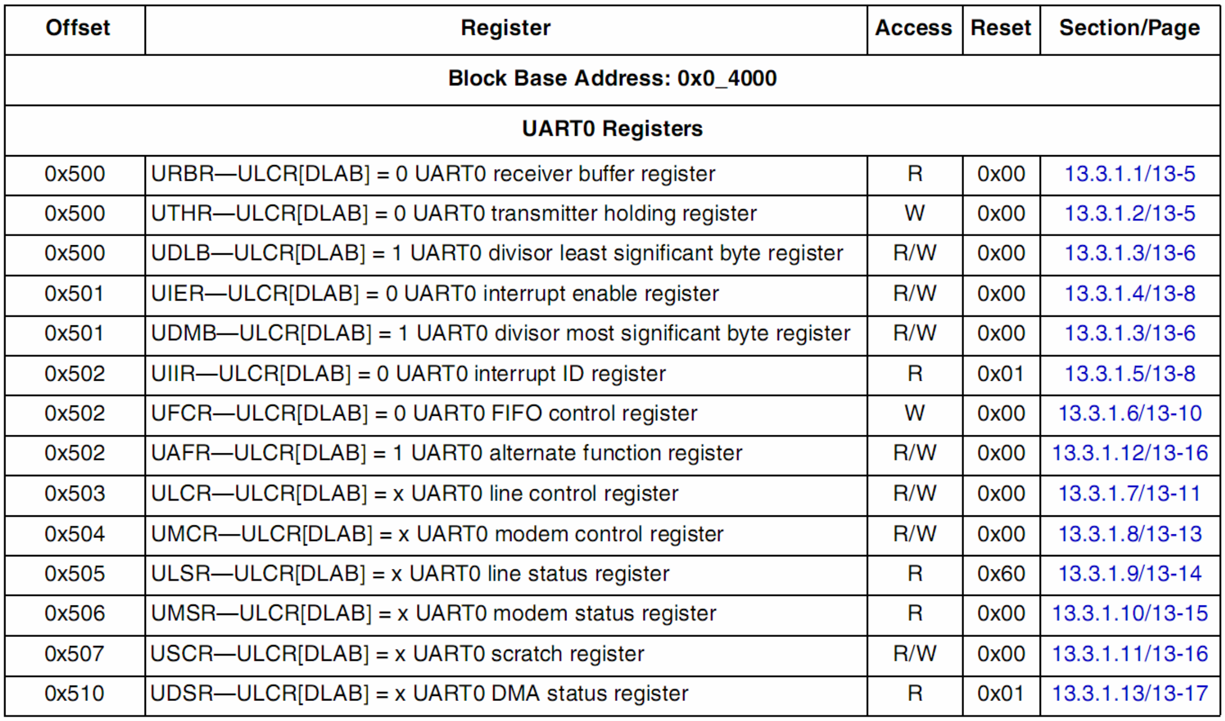


Figure A-1. Freescale MPC8572DS System Diagram (Section 3, of reference A-1)

A.1 TEST DESCRIPTION.

This experiment was designed to test the effects of bit changes in configuration registers. In this experiment, register bits in the universal asynchronous receiver/transmitter (UART) configuration space were changed. The Configuration Control and Status Base Address Register (CCSRBAR) holds the base address of all the memory-mapped configuration registers on the MPC8572E. The UART configuration registers are located at an offset of 0x4500 from CCSRBAR. Table A-1 shows all the configuration registers for UART0. The registers for UART1 are similarly located, starting at 0x4600. Figure A-2 shows the details of UART0 registers.



Offset	Register	Access	Reset	Section/Page
Block Base Address: 0x0_4000				
UART0 Registers				
0x500	URBR—ULCR[DLAB] = 0 UART0 receiver buffer register	R	0x00	13.3.1.1/13-5
0x500	UTHR—ULCR[DLAB] = 0 UART0 transmitter holding register	W	0x00	13.3.1.2/13-5
0x500	UDLB—ULCR[DLAB] = 1 UART0 divisor least significant byte register	R/W	0x00	13.3.1.3/13-6
0x501	UIER—ULCR[DLAB] = 0 UART0 interrupt enable register	R/W	0x00	13.3.1.4/13-8
0x501	UDMB—ULCR[DLAB] = 1 UART0 divisor most significant byte register	R/W	0x00	13.3.1.3/13-6
0x502	UIIR—ULCR[DLAB] = 0 UART0 interrupt ID register	R	0x01	13.3.1.5/13-8
0x502	UFCR—ULCR[DLAB] = 0 UART0 FIFO control register	W	0x00	13.3.1.6/13-10
0x502	UAFR—ULCR[DLAB] = 1 UART0 alternate function register	R/W	0x00	13.3.1.12/13-16
0x503	ULCR—ULCR[DLAB] = x UART0 line control register	R/W	0x00	13.3.1.7/13-11
0x504	UMCR—ULCR[DLAB] = x UART0 modem control register	R/W	0x00	13.3.1.8/13-13
0x505	ULSR—ULCR[DLAB] = x UART0 line status register	R	0x60	13.3.1.9/13-14
0x506	UMSR—ULCR[DLAB] = x UART0 modem status register	R	0x00	13.3.1.10/13-15
0x507	USCR—ULCR[DLAB] = x UART0 scratch register	R/W	0x00	13.3.1.11/13-16
0x510	UDSR—ULCR[DLAB] = x UART0 DMA status register	R	0x01	13.3.1.13/13-17

Figure A-2. MPC8572 UART Configuration Registers (Section 13.2.2 of reference A-2)

A.2 TEST SETUP.

The program was run through the Freescale Semiconductor, Inc. CodeWarrior™ integrated development environment (IDE) by loading it into memory. The program used to change the register bits was also used to test the UART functioning. This was done by printing the register and bit number each time a register bit was changed.

A.3 EXPERIMENTS.

The register bits were changed by using an XOR (exclusive OR device) function with a mask in a walking one pattern (see section 7 of the main report). After printing the register and bit numbers, the bit was reset to its original value before changing the next bit.

A.4 RESULTS.

Table A-1 shows the results of the experiment. The table only shows the bit and register numbers for which there was a deviation from the normal execution.

Table A-1. Results of Experiment

Serial No.	Register No.	Bit No.	Register Name	Bit Description	Effects
1	0	3	UTHR0	Data	Newline character(s) deleted
2	0	5	UTHR0	Data	Extraneous space character inserted
3	0	6	UTHR0	Data	Extraneous at character inserted
4	2	1	UFCR0	Receiver trigger level	Extra character repeated
5	2	2	UFCR0	Reserved	Extra character repeated
6	2	7	UFCR0	First-in, first-out enable	Extra non-ASCII characters on reset
7	3	1	ULCR0	Set break	Some characters get translated to non-ACSII characters
8	3	6	ULCR0	Word length	Test hangs
9	3	7	ULCR0	Word length	Test hangs

It should be noted that, in each case, the program returned to the normal mode of operation after resetting the changed bit. The results can be classified based on their degree of criticality:

- Change in output data: The data output on the console was different from normal output. (For example, serial numbers 1-7 in table A-1)

- Change in program execution (serial numbers 8 and 9)

- Crash: None in this experiment.

A.5 INTERPRETATION OF RESULTS.

For serial numbers 8 and 9 in table A-1, a special diagnosis is due since the program did not execute normally. Through Codewarrior debugger it was observed that the execution could not come out of the function MPCDUARTReadPool() for result 8. The program did not exactly stop executing, but it seemed to be stuck in the function. The set break forced logic 0 to be on the serial out line and did not affect the UART buffers. In such a situation, the UART buffers get filled up, and hence, the call to printf() in the test program does not return.

A.6 TEST DESCRIPTION.

In this experiment, a safety net design was tested for the configuration-related issues. In an earlier test, it was observed that some of the UART configuration registers can lead to unwanted effects when their values are changed. It was also observed that the correct values of these registers are known for a particular use case. In this experiment, the UART configuration registers were periodically overwritten with their correct values. The researchers observed whether the unwanted effects still existed when overwriting with correct values. The period at which the system was able to perform correctly even when the register values change was also observed.

A.7 TEST SETUP.

The test was performed inside the CodeWarrior debug environment. The UART0 Line Control Register was used to corrupt and repair. The register was corrupted by changing its lowest bit, which changes the data to be transmitted on the UART. Through OPTP1, it was observed that this data translation led to the output of non-ASCII characters on the serial port. To repair the corrupted register, the default correct value of 0x03 was written on to the register. To detect errors, the value of the register was checked for 0x03. The corruption, repair and usage were all run within a single thread on one core.

A.8 TIMER SETUP.

The decrementer (DEC) counter was used in the e500 cores to set up a 1-ms timer. The DEC counter is decremented every 8 Core Complex Bus (CCB) clocks. In the default setup, the CCB runs at a 600-MHz frequency, so the DEC was loaded with a value of 750,000. The interrupt handler for the timer calls the timerInt function, where the register corruption, repair, and usage were performed. The pseudo code for the timerInt function is shown in figure A-3.

```
time++
if (time % usage_period == 0)
        if(register value incorrect)
                repairs++
if (time % repair_period == 0)
        repair register()
if (time == next_error_time)
        corrupt register()
        update next_error_time
```

Figure A-3. The timerInt Function

A.9 EXPERIMENTS.

Three different periods were used for corrupting and repairing the register value. The usage period and corruption period are an estimate on how frequently the register is expected to be used and corrupted, respectively. Different values of repair period were used and a number of errors were observed. For each repair period, a 10-second test was run.

A.10 TEST RESULTS.

This section discusses the results from the experiments performed for this test plan. Figures A-4 through A-6 show the number of results for different values of repair period, usage period, and corruption period. In all three results, no errors were detected if the repair period was less than 5 ms. At the same time, for certain values of the repair period, there was a spike in the number of errors. This happened because the interleaving of usage, repair, and corruption periods allowed the register to be used just after it was corrupted for the given corruption period. The usage period was kept constant at 15 ms.

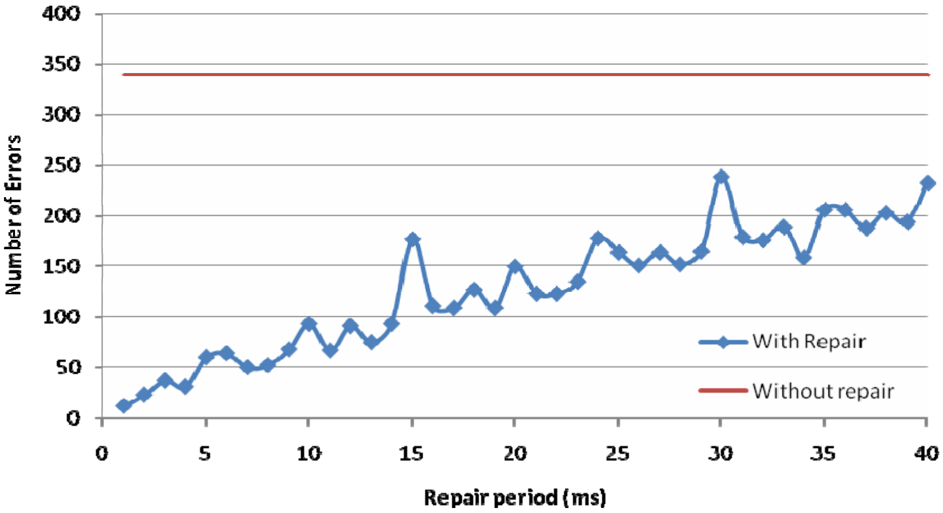


Figure A-4. Corruption Period = 100 ms

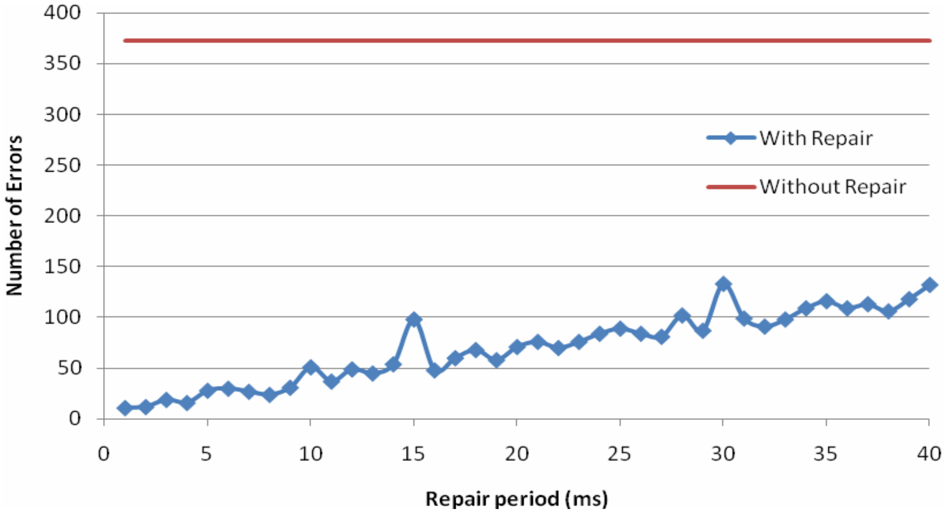


Figure A-5. Corruption Period = 200 ms

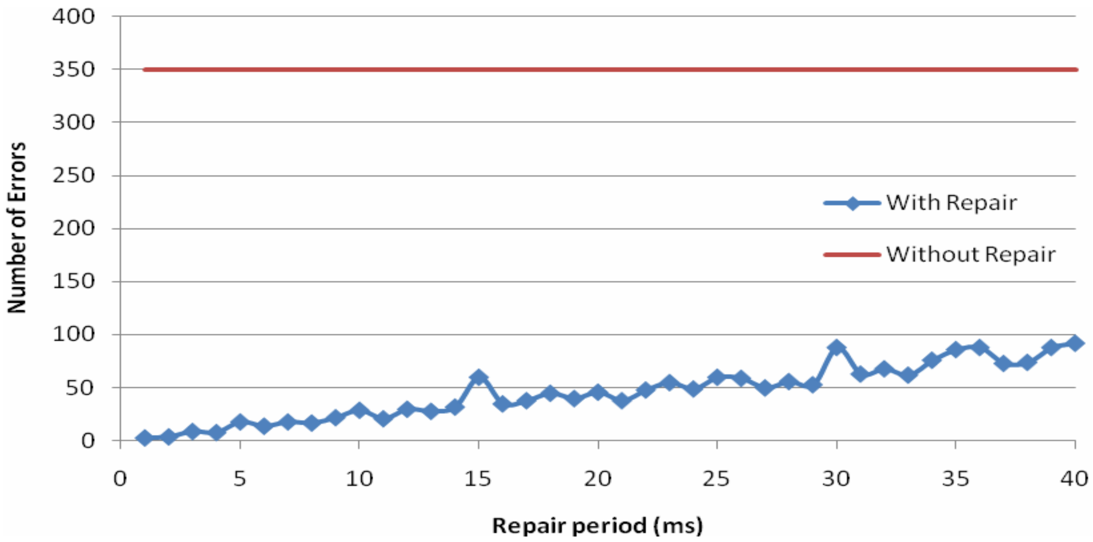


Figure A-6. Corruption Period = 300 ms

A.11 CONCLUSION.

Through these experiments, it was concluded that the number of detected errors depends greatly on the usage, corruption, and repair periods. However, if the configuration register is repaired at a sufficiently high rate, the system can run error-free.

A.12 RESOURCE SHARING EFFECT ON TIMING.

A.12.1 TEST DESCRIPTION.

In this experiment, the timing delays, which were due to the contention by cores on level 2 (L2) cache, were determined. First, a simple matrix multiplication program was used to determine its execution time by running it on only one core. This execution time was the baseline timing requirement. To test the effect of contention, the other core was activated and run on another matrix multiplication program, which stressed the shared L2 cache. The delays incurred by this contention on the execution time of the matrix multiplication program were observed and measured.

A.12.2 TEST SETUP.

The test was performed inside the Linux operating system environment from the board support package for the MPC8572 System. The taskset utility from the DENX ELDK software package was used to restrict the programs to run on a single core. The execution time of the programs was obtained by reading the real-time clock device set to a frequency of 8 MHz. The specific details about reading the real-time clock and running the programs can be obtained from the documentation in the code package for the one page test plan (Resource-Sharing-Effect-on-Timing One Page Test Plan). Two versions of the matrix multiplication program were implemented. The first one, matmult_timer, prints out the execution time whenever it is run.

The second one, matmult, is used for L2 cache-stressing purposes and does not print out the execution time. Figures A-7 through A-10 give a description of how the source code was used and the general content of the scripts.

```
Source Code
============
matmult.c - untimed matrix multiplication code (see figure A-8)
matmult - matrix multiplication executable(run on MPC8572Ds platform and see cannot be
viewed)

matmult_timer.c - timed matric multiplication code (see figure A-9)
matmult_timer - timed matrix multiplication executable (run on MPC8572DS platform and
cannot be viewed)

rtctest.c - reference code for Real Time Clock manipulation (see figure A-10)

Scripts
=======
For each of the figures A-7 to A-12 in the handbook, this package contains two
scripts. One to record the baseline execution times (named base-exec-time.sh)
and the other to record the execution time in the presence of contention
(named contention-exec-time.sh). These scripts utilize the matmult and
matmul_timer executables to produce the data displayed in the handbook
results.
```

Figure A-7. Readme.txt

```
\#include <stdio.h>
#include <stdlib.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

/* This program reads the dimensions of two matrices from command line.
 * The matrix contents are filled with random numbers and the result of
 * matrix is printed out.
 */


main(int argc, char **argv)
{
      int i, j, k, dim1, dim2, dim3, temp1, temp2, temp3, sum;
      int **mat1, **mat2, **matR;
```

Figure A-8. The matmult.c Source Code

```
/* Check command line arguments sanity
 */
if(argc != 4){
        printf("Usage: ./a.out dim1 dim2 dim3\n");
        exit(1);
}


/* Read the matrix dimensions from command line
 */

dim1 = atoi(argv[1]);
dim2 = atoi(argv[2]);
dim3 = atoi(argv[3]);


srand(dim1);
//printf("dim1 = %d, dim2 = %d, dim3 = %d\n", dim1, dim2, dim3);

/* Dynamically allocate the arrays and initialize the contents with
 *  random numbers
 */


mat1 = (int **) malloc(dim1 * sizeof (int *));
mat2 = (int **) malloc(dim2 * sizeof (int *));
matR = (int **) malloc(dim1 * sizeof (int *));

for (temp1 = 0; temp1 < dim1; temp1++){
        mat1[temp1] = (int *) malloc(dim2 * sizeof(int));
        for (temp2 = 0; temp2 < dim2; temp2++){
                mat1[temp1][temp2] = rand();
        }
}

for (temp1 = 0; temp1 < dim2; temp1++){
        mat2[temp1] = (int *) malloc(dim3 * sizeof(int));
        for (temp2 = 0; temp2 < dim3; temp2++){
                mat2[temp1][temp2] = rand();
        }
}

for (temp1 = 0; temp1 < dim1; temp1++){
        matR[temp1] = (int *) malloc(dim3 * sizeof(int));
}

for(temp1 = 0; temp1 < dim1; temp1++){
        for (temp2 = 0; temp2 < dim3; temp2++){
                sum = 0;
                for(temp3 = 0; temp3 < dim2; temp3 ++){
                        sum += mat1[temp1][temp3] * mat2[temp3][temp2];
                }
                matR[temp1][temp2] = sum;
        }
}
```

Figure A-8. The matmult.c Source Code (Continued)

```
        /*
        printf("Matrix1:\n");
        printMat(mat1, dim1, dim2);
        printf("Matrix2:\n");
        printMat(mat2, dim2, dim3);
        printf("Matrix Result:\n");
        printMat(matR, dim1, dim3);
        */

}

printMat(int **mat, int dim1, int dim2)
{
        int i, j;
        for(i = 0; i < dim1; i++){
                for(j = 0; j < dim2; j++){
                        printf("%d ",mat[i][j]);
                }
                printf("\n");
        }
}
```

Figure A-8. The matmult.c Source Code (Continued)

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <time.h>

/* This program reads the dimensions of two matrices from command line.
 * The matrix contents are filled with random numbers and the result of
 * matrix is printed out.
 */


main(int argc, char **argv)
{
        int i, j, k, dim1, dim2, dim3, temp1, temp2, temp3, sum;
        int **mat1, **mat2, **matR;
        unsigned long long a, b;
        const char *rtc = "/dev/rtc0";
        unsigned long tmp, data;
        int fd, retval, timerFreq;
        double time;

        fd = open(rtc, O_RDONLY);

        if (fd ==  -1) {
```

Figure A-9. The matmult_timer.c Source Code

```
        perror(rtc);
        exit(errno);
}

/* Read periodic IRQ rate */
retval = ioctl(fd, RTC_IRQP_READ, &tmp);
if (retval == -1) {
        /* not all RTCs support periodic IRQs */
        if (errno == ENOTTY) {
                fprintf(stderr, "\nNo periodic IRQ support\n");
                exit(1);
        }
        perror("RTC_IRQP_READ ioctl");
        exit(errno);
}
//fprintf(stderr, "\nPeriodic IRQ rate is %ldHz.\n", tmp);

timerFreq = 8192;
retval = ioctl(fd, RTC_IRQP_SET, timerFreq);
if (retval == -1) {
        /* not all RTCs can change their periodic IRQ rate */
        if (errno == ENOTTY) {
                fprintf(stderr,
                        "\n...Periodic IRQ rate is fixed\n");
                exit(1);
        }
        perror("RTC_IRQP_SET ioctl");
        exit(errno);
}
//fprintf(stderr, "periodic frequency set to %ldHz:\n", tmp);
fflush(stderr);

/* Enable periodic interrupts */
retval = ioctl(fd, RTC_PIE_ON, 0);
if (retval == -1) {
        perror("RTC_PIE_ON ioctl");
        exit(errno);
}

/* This blocks */
retval = read(fd, &data, sizeof(unsigned long));
if (retval == -1) {
        perror("read");
        exit(errno);
}
a = data >> 8;

/* Check command line arguments sanity
 */
if(argc != 4){
        printf("Usage: ./a.out dim1 dim2 dim3\n");
        exit(1);
}


/* Read the matrix dimensions from command line
 */
```

Figure A-9. The matmult_timer.c Source Code (Continued)

```
dim1 = atoi(argv[1]);
dim2 = atoi(argv[2]);
dim3 = atoi(argv[3]);

srand(dim1);

//printf("dim1 = %d, dim2 = %d, dim3 = %d\n", dim1, dim2, dim3);

/* Dynamically allocate the arrays and00 initialize the contents with
 *  random numbers
 */

mat1 = (int **) malloc(dim1 * sizeof (int *));
mat2 = (int **) malloc(dim2 * sizeof (int *));
matR = (int **) malloc(dim1 * sizeof (int *));

for (temp1 = 0; temp1 < dim1; temp1++){
       mat1[temp1] = (int *) malloc(dim2 * sizeof(int));
       for (temp2 = 0; temp2 < dim2; temp2++){
              mat1[temp1][temp2] = rand();
       }
}

for (temp1 = 0; temp1 < dim2; temp1++){
       mat2[temp1] = (int *) malloc(dim3 * sizeof(int));
       for (temp2 = 0; temp2 < dim3; temp2++){
              mat2[temp1][temp2] = rand();
       }
}

for (temp1 = 0; temp1 < dim1; temp1++){
       matR[temp1] = (int *) malloc(dim3 * sizeof(int));
}

for(temp1 = 0; temp1 < dim1; temp1++){
       for (temp2 = 0; temp2 < dim3; temp2++){
              sum = 0;
              for(temp3 = 0; temp3 < dim2; temp3 ++){
                     sum += mat1[temp1][temp3] * mat2[temp3][temp2];
              }
              matR[temp1][temp2] = sum;
       }
}

/* This blocks */
retval = read(fd, &data, sizeof(unsigned long));
if (retval == -1) {
       perror("read");
       exit(errno);
}
b = data >> 8;

time = (b-(double)a)/timerFreq;
printf("execution time = %f\n", time);
/*
printf("Matrix1:\n");
printMat(mat1, dim1, dim2);
```

Figure A-9. The matmult_timer.c Source Code (Continued)

```
        printf("Matrix2:\n");
        printMat(mat2, dim2, dim3);
        printf("Matrix Result:\n");
        printMat(matR, dim1, dim3);
        */

        /* Disable periodic interrupts */
        retval = ioctl(fd, RTC_PIE_OFF, 0);
        if (retval == -1) {
                perror("RTC_PIE_OFF ioctl");
                exit(errno);
        }
}

printMat(int **mat, int dim1, int dim2)
{
        int i, j;
        for(i = 0; i < dim1; i++){
                for(j = 0; j < dim2; j++){
                        printf("%d ",mat[i][j]);
                }
                printf("\n");
        }
}
```

Figure A-9. The matmult_timer.c Source Code (Continued)

```
/*
        *      Real Time Clock Driver Test/Example Program
        *
        *      Compile with:
        *                gcc -s -Wall -Wstrict-prototypes rtctest.c -o rtctest
        *
        *      Copyright (C) 1996, Paul Gortmaker.
        *
        *      Released under the GNU General Public License, version 2,
        *      included herein by reference.
        *
        */

        #include <stdio.h>
        #include <linux/rtc.h>
        #include <sys/ioctl.h>
        #include <sys/time.h>
        #include <sys/types.h>
        #include <fcntl.h>
        #include <unistd.h>
        #include <stdlib.h>
        #include <errno.h>


        /*
```

Figure A-10. The rtctest.c Source Code

```
 * This expects the new RTC class driver framework, working with
 * clocks that will often not be clones of what the PC-AT had.
 * Use the command line to specify another RTC if you need one.
 */
static const char default_rtc[] = "/dev/rtc0";


int main(int argc, char **argv)
{
        int i, fd, retval, irqcount = 0;
        unsigned long tmp, data;
        struct rtc_time rtc_tm;
        const char *rtc = default_rtc;

        printf("%d\n",sizeof(unsigned long));
        switch(argc)
        {
        case 1:
                break;
        default:
                fprintf(stderr, "usage:  rtctest [rtcdev]\n");
                return 1;
        }

        fd = open(rtc, O_RDONLY);

        if (fd ==  -1) {
                perror(rtc);
                exit(errno);
        }

        fprintf(stderr, "\n\t\t\tRTC Driver Test Example.\n\n");

        /* Turn on update interrupts (one per second) */
        retval = ioctl(fd, RTC_UIE_ON, 0);
        if (retval == -1) {
                if (errno == ENOTTY) {
                        fprintf(stderr,
                                "\n...Update IRQs not supported.\n");
                        goto test_READ;
                }
                perror("RTC_UIE_ON ioctl");
                exit(errno);
        }

        fprintf(stderr, "Counting 5 update (1/sec) interrupts from reading %s:",
                        rtc);
        fflush(stderr);
        for (i=1; i<6; i++) {
                /* This read will block */
                retval = read(fd, &data, sizeof(unsigned long));
                if (retval == -1) {
                        perror("read");
                        exit(errno);
                }
                fprintf(stderr, " %d",i);
```

Figure A-10. The rtctest.c Source Code (Continued)

```
                    fflush(stderr);
                    irqcount++;
             }

             fprintf(stderr, "\nAgain, from using select(2) on /dev/rtc:");
             fflush(stderr);
             for (i=1; i<6; i++) {
                    struct timeval tv = {5, 0};      /* 5 second timeout on select */
                    fd_set readfds;

                    FD_ZERO(&readfds);
                    FD_SET(fd, &readfds);
                    /* The select will wait until an RTC interrupt happens. */
                    retval = select(fd+1, &readfds, NULL, NULL, &tv);
                    if (retval == -1) {
                            perror("select");
                            exit(errno);
                    }
                    /* This read won't block unlike the select-less case above. */
                    retval = read(fd, &data, sizeof(unsigned long));
                    if (retval == -1) {
                            perror("read");
                            exit(errno);
                    }
                    fprintf(stderr, " %d",i);
                    fflush(stderr);
                    irqcount++;
             }

             /* Turn off update interrupts */
             retval = ioctl(fd, RTC_UIE_OFF, 0);
             if (retval == -1) {
                    perror("RTC_UIE_OFF ioctl");
                    exit(errno);
             }

       test_READ:
             /* Read the RTC time/date */
             retval = ioctl(fd, RTC_RD_TIME, &rtc_tm);
             if (retval == -1) {
                    perror("RTC_RD_TIME ioctl");
                    exit(errno);
             }

             fprintf(stderr, "\n\nCurrent RTC date/time is %d-%d-%d,
%02d:%02d:%02d.\n",
                    rtc_tm.tm_mday, rtc_tm.tm_mon + 1, rtc_tm.tm_year + 1900,
                    rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

             /* Set the alarm to 5 sec in the future, and check for rollover */
             rtc_tm.tm_sec += 5;
             if (rtc_tm.tm_sec >= 60) {
                    rtc_tm.tm_sec %= 60;
                    rtc_tm.tm_min++;
             }
             if (rtc_tm.tm_min == 60) {
                    rtc_tm.tm_min = 0;
```

Figure A-10. The rtctest.c Source Code (Continued)

```
                rtc_tm.tm_hour++;
        }
        if (rtc_tm.tm_hour == 24)
                rtc_tm.tm_hour = 0;

        retval = ioctl(fd, RTC_ALM_SET, &rtc_tm);
        if (retval == -1) {
                if (errno == ENOTTY) {
                        fprintf(stderr,
                                "\n...Alarm IRQs not supported.\n");
                        goto test_PIE;
                }
                perror("RTC_ALM_SET ioctl");
                exit(errno);
        }

        /* Read the current alarm settings */
        retval = ioctl(fd, RTC_ALM_READ, &rtc_tm);
        if (retval == -1) {
                perror("RTC_ALM_READ ioctl");
                exit(errno);

        }

        fprintf(stderr, "Alarm time now set to %02d:%02d:%02d.\n",
                rtc_tm.tm_hour, rtc_tm.tm_min, rtc_tm.tm_sec);

        /* Enable alarm interrupts */
        retval = ioctl(fd, RTC_AIE_ON, 0);
        if (retval == -1) {
                perror("RTC_AIE_ON ioctl");
                exit(errno);
        }

        fprintf(stderr, "Waiting 5 seconds for alarm...");
        fflush(stderr);
        /* This blocks until the alarm ring causes an interrupt */
        retval = read(fd, &data, sizeof(unsigned long));
        if (retval == -1) {
                perror("read");
                exit(errno);
        }
        irqcount++;
        fprintf(stderr, " okay. Alarm rang.\n");

        /* Disable alarm interrupts */
        retval = ioctl(fd, RTC_AIE_OFF, 0);
        if (retval == -1) {
                perror("RTC_AIE_OFF ioctl");
                exit(errno);
        }

test_PIE:
        /* Read periodic IRQ rate */
        retval = ioctl(fd, RTC_IRQP_READ, &tmp);
        if (retval == -1) {
```

Figure A-10. The rtctest.c Source Code (Continued)

```
        /* not all RTCs support periodic IRQs */
        if (errno == ENOTTY) {

                fprintf(stderr, "\nNo periodic IRQ support\n");
                goto done;
        }
        perror("RTC_IRQP_READ ioctl");
        exit(errno);
}
fprintf(stderr, "\nPeriodic IRQ rate is %ldHz.\n", tmp);

fprintf(stderr, "Counting 20 interrupts at:");
fflush(stderr);

/* The frequencies 128Hz, 256Hz, ... 8192Hz are only allowed for root. */
for (tmp=2; tmp<=8192; tmp*=2) {

        retval = ioctl(fd, RTC_IRQP_SET, tmp);
        if (retval == -1) {
                /* not all RTCs can change their periodic IRQ rate */
                if (errno == ENOTTY) {
                        fprintf(stderr,
                                "\n...Periodic IRQ rate is fixed\n");
                        goto done;
                }
                perror("RTC_IRQP_SET ioctl");
                exit(errno);
        }

        fprintf(stderr, "\n%ldHz:\t", tmp);
        fflush(stderr);

        /* Enable periodic interrupts */
        retval = ioctl(fd, RTC_PIE_ON, 0);
        if (retval == -1) {
                perror("RTC_PIE_ON ioctl");
                exit(errno);
        }

        for (i=1; i<21; i++) {
                /* This blocks */
                retval = read(fd, &data, sizeof(unsigned long));
                if (retval == -1) {
                        perror("read");
                        exit(errno);
                }
                sleep(i);
                fprintf(stderr, " [%d,%lx]",i,data);
                fflush(stderr);
                irqcount++;
        }

        /* Disable periodic interrupts */
        retval = ioctl(fd, RTC_PIE_OFF, 0);
        if (retval == -1) {
                perror("RTC_PIE_OFF ioctl");
```

Figure A-10. The rtctest.c Source Code (Continued)

```
                        exit(errno);
                }
        }

done:
        fprintf(stderr, "\n\n\t\t\t *** Test complete ***\n");

        close(fd);

        return 0;
}
```

Figure A-10. The rtctest.c Source Code (Continued)

A.12.3 EXPERIMENTS.

In the following description, the timed and untimed versions of the matrix multiplication program are referred to as base matrix and contention matrix, respectively. Two different experiments were performed to test the results of contention. In the first experiment, the contention matrix size was kept constant at 10,000, while the base matrix size was varied from 20 to 700. In the second set of experiments, the base matrix size was kept constant, while the contention matrix sizes were varied.

A.12.4 TEST RESULTS.

Figure A-11 shows the execution times for the first experiment, where the base matrix size was varied. It is clearly shown that the execution times in presence of contention are greater than those without contention. However, the trend of the execution times was unexpected. As shown in figure A-11, spikes trend at matrix sizes 510, 590, and 670. The cause(s) of these spikes are probably the dynamic management of cache by the Linux operating system (OS) and needs additional investigation. See figures A-12 and A-13 for the applicable scripts.

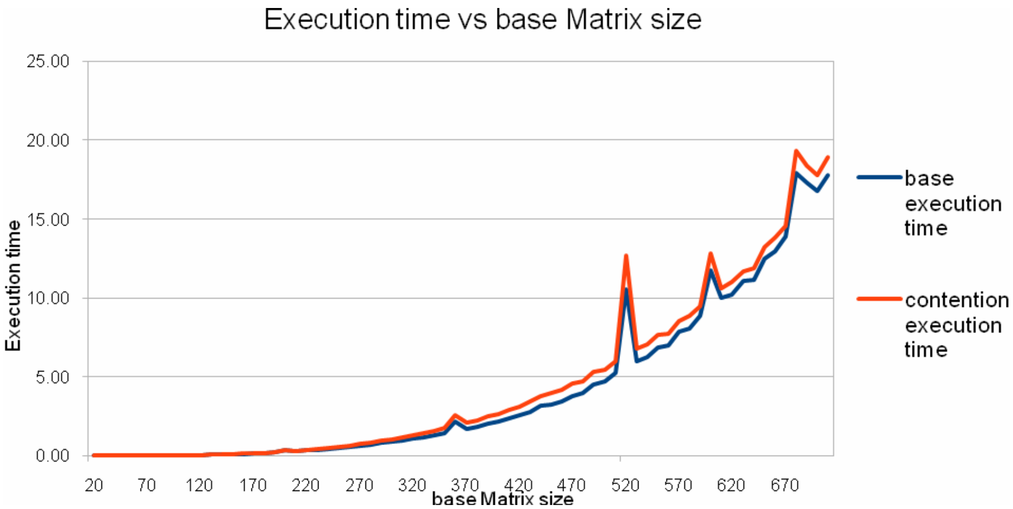


Figure A-11. Effect of Contention on Execution Time

```
#!/bin/bash

for (( i=20; i<=2000 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult $i $i $i &
../tools/taskset 0x02 ./matmult_timer 600 600 600 ; pkill -9 matmult
done
```

Figure A-12. The base-exec-time.sh Script

```
#!/bin/bash

for (( i=20; i<=700 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult 10000 10000 10000 &
../tools/taskset 0x02 ./matmult_timer $i $i $i ; pkill -9 matmult
done
```

Figure A-13. The contention.exec-time.sh Script

Figures A-14 through A-28 show the execution times, base-contention time scripts, and contention exec_time scripts for the second experiment, where the base matrix sizes were kept constant. Five different base matrix sizes, from 200 to 600, were used for this experiment. In each case, the contention matrix size was varied from a low number (around 20) to a size roughly equal to double the size of the base matrix. The reason for choosing this range was that a low contention matrix size would put very little stress on the L2 cache, and the execution time with contention would be roughly the same as without contention. This is clearly visible in the results. A contention matrix size of double the base matrix size ensures there is some contention on the L2 cache. As the contention matrix size is gradually increased beyond a certain value, the execution time with contention may suffer a significant increase, which is shown in figures A-8 through A-12. But as the contention matrix size is increased even more, the execution time does not continue to increase. This again might be a result of the Linux OS doing some intelligent reconfiguration of the cache that masks the effect of an increase in the contention matrix.

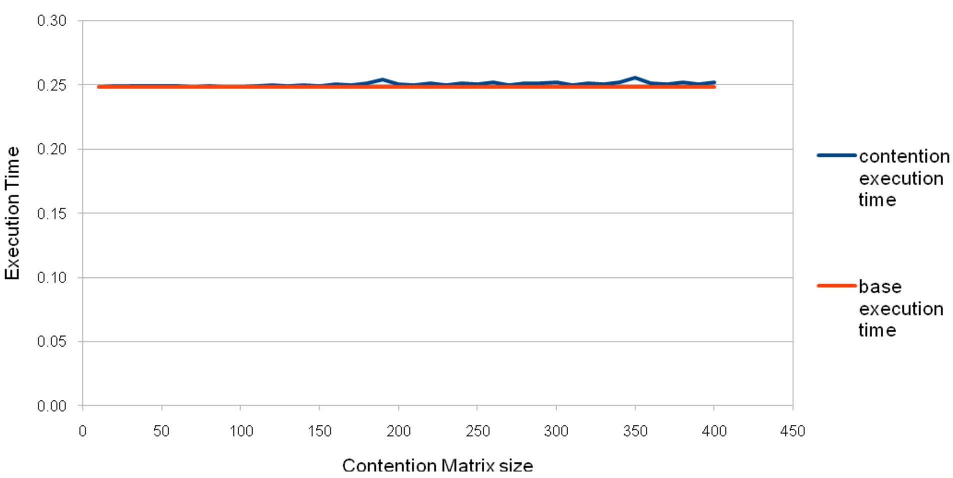


Figure A-14. Base Matrix Size = 200

```
#!/bin/bash

../tools/taskset 0x01 ./matmult_timer 200 200 200
```

Figure A-15. The base-exec-time.sh Script

```
#!/bin/bash

for (( i=20; i<=400 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult $i $i $i &
../tools/taskset 0x02 ./matmult_timer 200 200 200 ; pkill -9 matmult
done
```

Figure A-16. The contention-exec-time.sh Script

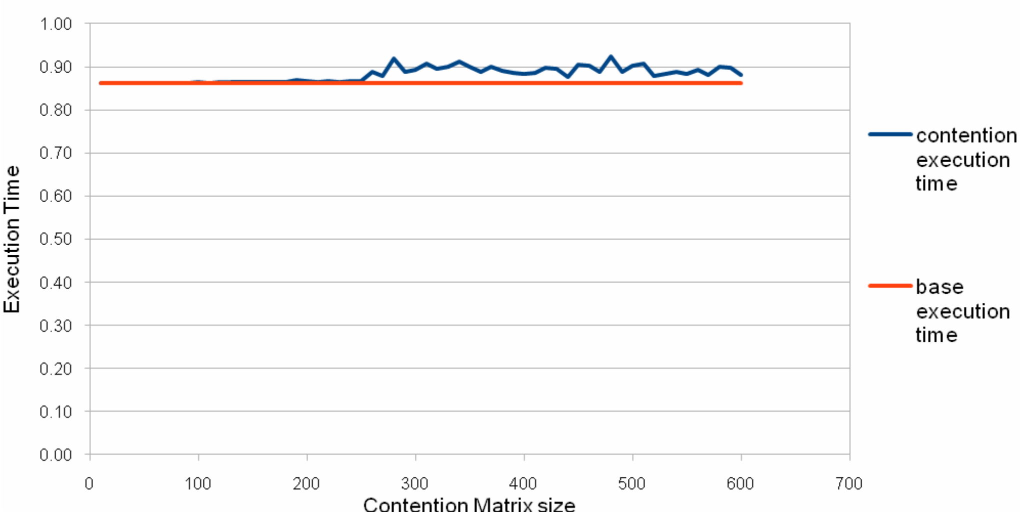


Figure A-17. Base Matrix Size = 300

```
#!/bin/bash

../tools/taskset 0x01 ./matmult_timer 300 300 300
```

Figure A-18. The base-exec-time.sh Script

```
#!/bin/bash

for (( i=20; i<=600 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult $i $i $i &
../tools/taskset 0x02 ./matmult_timer 300 300 300 ; pkill -9 matmult
done
```

Figure A-19. The contention-exec-time.sh Script

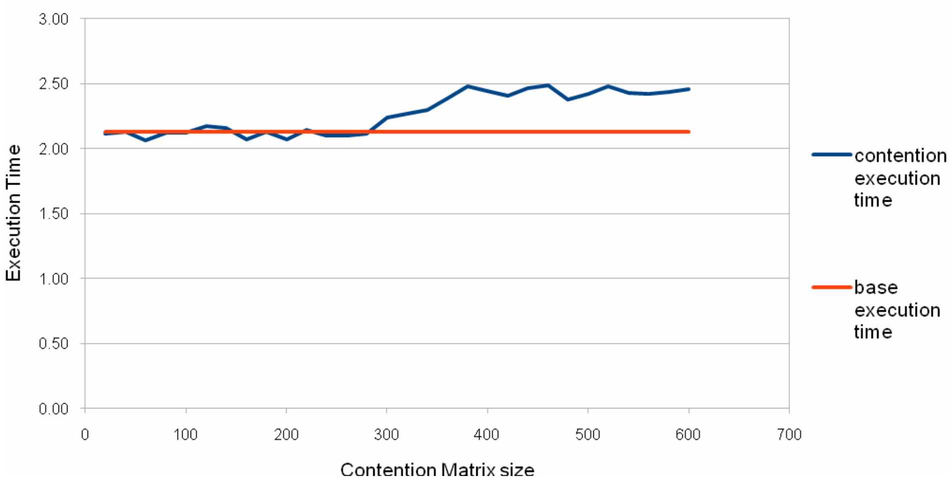


Figure A-20. Base Matrix Size = 400

```
#!/bin/bash

../tools/taskset 0x01 ./matmult_timer 400 400 400
```

Figure A-21. The base-exec-time.sh Script

```
#!/bin/bash

for (( i=20; i<=600 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult $i $i $i &
../tools/taskset 0x02 ./matmult_timer 400 400 400 ; pkill -9 matmult
done
```

Figure A-22. The contention-exec-time.sh Script

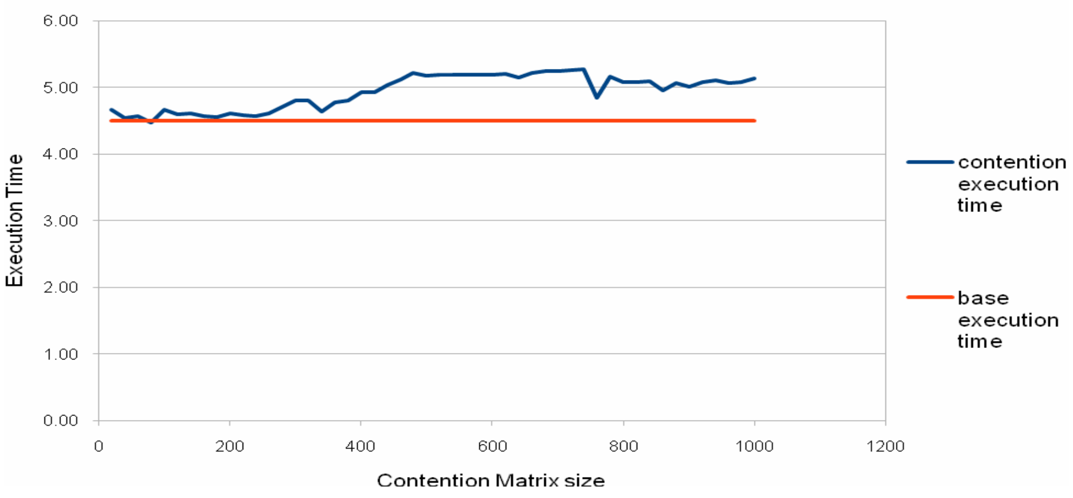


Figure A-23. Base Matrix Size = 500

```
#!/bin/bash

../tools/taskset 0x01 ./matmult_timer 500 500 500
```

Figure A-24. The base-exec-time.sh Script

```
#!/bin/bash

for (( i=20; i<=1000 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult $i $i $i &
../tools/taskset 0x02 ./matmult_timer 500 500 500 ; pkill -9 matmult
done
```

Figure A-25. The contention-exec-time.sh Script

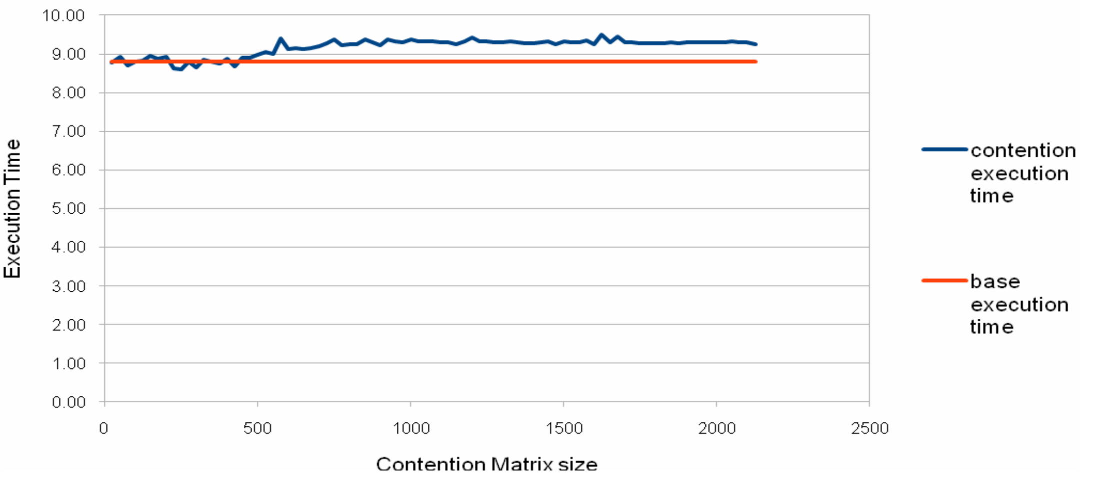


Figure A-26. Base Matrix Size = 600

```
#!/bin/bash

../tools/taskset 0x01 ./matmult_timer 600 600 600
```

Figure A-27. The base-exec-time.sh Script

```
#!/bin/bash

for (( i=20; i<=2000 ; i=$i+20 ))
do
../tools/taskset 0x01 ./matmult $i $i $i &
../tools/taskset 0x02 ./matmult_timer 600 600 600 ; pkill -9 matmult
done
```

Figure A-28. The contention-exec-time.sh Script

A.13 CONCLUSION.

Through these experiments, it was shown that the effects of contention might be masked by the Linux operating system (OS) management of the cache. Further, it was shown that even with the Linux OS management, the execution times can vary significantly, depending on the amount of contention present in the system. To assess the effects of contention in the absence of Linux OS management, the application should be run directly on the hardware, which would give a much clearer picture of the effects of contention on shared resources.

A.14 REFERENCES.

A-1. “MPC8572 Development System User’s Guide,” Freescale Semiconductor, Inc., Rev.1, Austin, Texas, January 2009.

A-2. “MPC8572E PowerQUICC III Integrated Host Processor Family Reference Manual,” Freescale Semiconductor, Inc., Rev. 2, Austin, Texas, May 2008.

APPENDIX B—CONFIGURATION-RELATED ISSUES BACKUP INFORMATION

The following information, concerning the risks associated with power supply designs, is provided for safety net designers. Safety nets may also be required to monitor power supplies and characteristics to accomplish error detection and recovery.

System-on-a-chip (SoC) power supply design is challenging because of high power supply output currents and low core voltages. Higher numbers of gate transitions per unit time and high clock frequencies reduce the noise margin and raise the noise floor of the processor. The SoC power supplies need to be properly sequenced often to provide assurance that the processor boots up correctly.

If noise margins are low, then the SoC will be more susceptible to electromagnetic interference (EMI). EMI may be radiated or conducted.

Generally, the SoC mechanical package has a lot of connections in a small area, which makes it challenging to place the decoupling caps close to the pins. The lead inductance of the pin and the printed circuit board trace may cause ringing with the decoupling capacitor. This ringing will reduce the noise margin. In general, the power demand of the core voltages require that switch mode power supplies be used for SoC core voltages.

Noise margin in electrical engineering is defined as the amount by which a signal exceeds the minimum amount for proper operation. It is commonly used in at least two contexts:

- In communications system engineering, noise margin is the ratio by which the signal exceeds the minimum acceptable amount. It is normally measured in decibels.

- In a digital circuit, the noise margin is the amount by which the signal exceeds the threshold for a proper 0 or 1. For example, a digital circuit might be designed to swing between 0.0 and 1.2 volts, with anything below 0.2 volt considered a 0, and anything above 1.0 volt considered a 1. Then the noise margin for a 0 would be the amount that a signal is below 0.2 volt, and the noise margin for a 1 would be the amount by which a signal exceeds 1.0 volt. In this case, noise margins are measured as an absolute voltage, not a ratio. Noise margins are generally defined so that positive values ensure proper operation, and negative margins result in compromised operation, or perhaps outright failure.

Noise floor, in signal theory, is the measure of the signal created from the sum of all the noise sources and unwanted signals within a measurement system.

In radio communication and electronics, this may include thermal noise, blackbody, and any other interfering signals. The noise floor limits the smallest measurement that can be taken with certainty, since any measured amplitude can, on average, be no less than the noise floor.