DAHLGREN DIVISION
NAVAL SURFACE WARFARE CENTER
Dahlgren, Virginia 22448-5100

SOFTWARE SAFETY TECHNOLOGY OVERVIEW

WORKING PAPERS

SYSTEMS SAFETY ENGINEERING DIVISION (CODE G70)

AUGUST 2012

This page intentionally left blank

CONTENTS

This page intentionally left blank

GLOSSARY

COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DCCP	Datagram Congestion Control Protocol
DNS	Domain Name System
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
ISO	International Standards Organization
FOSS	Free and Open Source Software
FPGA	Field Programmable Gate Array
LOR	Level of Rigor
Model of Maths	Representing infinitely many real numbers by a finite number of bits requires a computer system to perform an approximation for many real numbers. The method used by a computer system to perform this approximation is considered the system's "model of maths"
OS	Operating System
RIP	Routing Information Protocol
RTSJ	Real Time Specification for Java
SSH	Secure Shell
Static Code Analysis	The analysis of computer software that is performed without actually executing programs built from that software. In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called code review.
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

USB	Universal Serial Bus
WCET	Worst Case Execution Time

1 INTRODUCTION

1.1 Background

The use of software has become an ever-increasing reality in today's safety significant systems. As a result, several standards and guidelines have been written to address the use of software in safety-significant systems, including the Joint Software System Safety Engineering Handbook (JSSSEH). The JSSSEH, in particular, prescribes a process that identifies the Software Criticality Index (SwCI) and associated Level of Rigor (LOR) of a system's software components.

The determination of a software component's SwCI and associated LOR is an important starting point for the safety engineer, but, as they say, the devil is in the details. Software systems involve many layers of technology that are designed to work together to meet the overall requirements of the system. In order for the safety engineer to adequately perform LOR analyses, the safety engineer must utilize a well-defined model of these various layers of technology.

In addition to facilitating the creation of an accurate software system model, this paper presents an overview of the safety concerns associated with several common technology implementations. Each specific technology implementation, such as superscalar computer processors, Linux operating system distributions, and the C++ computer language, provides specific strengths and weaknesses related to system performance, reliability, and safety. This report provides a survey of available literature related to several technologies in order to present a handful of these strengths and weaknesses and guide the safety engineer through the software safety program planning and architectural and design analysis phases.

1.2 Scope

This paper presents an overview of the technologies involved in a software system in order to help the safety engineer develop an accurate model of the system. The intention is for this model to be utilized throughout the software safety process to facilitate proper requirements allocation, accurate design analysis, detailed code analysis, and pertinent safety-specific testing. It is important to understand that the content of this manual is a collection of data from many sources including conference papers, journal articles and other publications. There has been no formal assessment of the referenced data or endorsement by G70 on the stated conclusions or recommended approach from any given publication. This document simply provides a collection of that data as a resource for consideration during a DoD applicable software safety effort.

This report is not, and is never intended to be, a final release. The hope is that as the G70 workforce faces new technologies in their software systems, or performs research on existing technology, that this document will be updated to include any new knowledge gleaned as a result of that work.

Note: The technology-specific assessments are subject to the limitations at the time that the applicable article was written. It remains essential that the safety engineer research the available literature for new assessments that are performed as technologies mature.

1.3 Format

This document is structured in to four high-level technology sections: (1) Operating Systems, (2) Computer Languages, (3) Computer Processors, and (4) Networks. The first subsection of each technology section provides general criteria for the safety engineer to consider during the safety program planning phase. After presenting the general criteria for assessment, this document summarizes reports related to the use of several specific technologies in safety applications.

2 OPERATING SYSTEMS

Enclyclopædia Britannica [9] defines an operating system (OS) as a "program that manages a computer's resources, especially the allocation of those resources among other programs. Typical resources include the central processing unit (CPU), computer memory, file storage, input/output devices, and network connections. Management tasks include scheduling resource use to avoid conflicts and interference between programs. Unlike most programs, which complete a task and terminate, an OS runs indefinitely and terminates only when the computer is turned off."

2.1 General Criteria for Assessment

Pierce et al. [11] explain that "the following OS features should be used as a minimum to assess the sufficiency or completeness of the safety requirements set on the OS:

- Executive and scheduling – the process switching time and the employed scheduling policy of the OS must meet all time-related application requirements;
- Resource management (both internal to the OS and provided to the application software) – the OS's own internal use of resources must be predictable and bounded;
- Internal communication – the OS inter-process communication mechanisms must be robust and the risk of a corrupt message affecting safety adequately low;
- External communication – the OS communication mechanisms used for communication with either other computers in the network or some external system must be robust and the risk of a corrupt message affecting safety adequately low;
- Internal liveliness failures – the OS must allow the application to meet its availability requirements;
- Partitioning – if the OS is used to partition functions of differing [safety criticality], functions of lower [safety criticality] should not interfere with the correct operation of higher [safety criticality] functions;
- Real-time – timing facilities and interrupt handling features must be sufficiently accurate to meet all application response time requirements;
- Security – only if the OS is used in a secure application;
- User interface – when the OS is used to provide a user interface, the risk of the interface corrupting the user input to the application or the output data of the application must be sufficiently low;
- Robustness – the OS must be able to detect and respond appropriately to the failure of the application processes and external interfaces;
- Installation – installation procedures must include measures to protect against producing a faulty installation due to user error."

It is important to note the emphasis of the ability of the OS to meet system requirements. While the following sections provide an overview of the capabilities provided by various OSs, it remains the job of

the developer and the safety engineer to identify the requirements of the system (e.g. timing, availability) and ensure that the features provided by the OS are used to meet those requirements.

2.1.1 Commercial OSs

Pierce et al. [11] state that, in general, the use of Commercial Off-The-Shelf (COTS) OSs cannot be justified in safety-critical systems. "While [an] application can be developed to [a] required degree of rigour *(sic)*, a pre-existing OS must be taken as given and safety assurance obtained by means other than control over the development process" (Pierce [11]). These other means of determining safety assurance include performing the appropriate Level of Rigor (LOR) analysis. Stottlar [6] explains that, with enough resources, "OS developers will sell limited access rights to their OS code and design. So even though one hasn't developed the OS, one could still perform a LOR 1 analysis."

During OS selection, the safety engineer must consider the documentation that would be included with the OS to support the various LORs. Depending on the level of criticality of the system application, and whether or not the level of documentation provided with the OS would support the associated LOR, the safety engineer can provide a justified recommendation on OS selection.

2.1.2 Free and Open Source Software

As a result of the open source code of free and open source software (FOSS) OSs, Kammerer [10] states that one pro for its use in safety systems is that it may be modified by the system developer in order to meet requirements not addressed by the basic capabilities of the OS. Additionally, any functionality provided by the OS that is not required by the system (e.g. the networking stack) can be excluded from the final OS kernel binary code.

The modifiable aspect of FOSS OSs allows the software safety engineer to provide recommendations to address identified deficiencies and thereby increase the safety of the system. However, this requires a detailed knowledge of the architecture, design, and code of the FOSS OS for the software safety engineer to properly assign a SwCI to the safety-significant functions performed by the OS and perform the appropriate LOR analysis.

3 COMPUTER LANGUAGES

3.1 General Criteria for Assessment

Cullyer and Wickman [13] state that the questions below should be the minimum set considered by a project manager and design team when agreeing on the choice of computer language:

- Wild Jumps: Can the control flow be totally determined? (i.e. can it be shown that the program cannot jump to an arbitrary store (memory) location?)
- Overwrites: Are there language features which prevent a store location from being arbitrarily overwritten?
- Semantics: Are the semantics of the language defined sufficiently for the translation process needed for static code analysis to be feasible?
- Model of Maths: Is there a rigorous model of both integer and floating point arithmetic within the language standard?
- Operational Arithmetic: Are there procedures for checking that the operational program obeys the model of the arithmetic when running on the target processor?
- Data Typing: Are the means of data typing strong enough to prevent misuse of variables?
- Exception Handling: If the software detects a malfunction at runtime, do mechanisms exist to facilitate recovery? (e.g. global exception handlers, which may in themselves introduce hazards if used unwisely.)
- Safe Subsets: Does a subset of the language exist which is defined to have properties that satisfy these requirements more adequately than the full language?
- Exhaustion of Memory: Are there facilities in the language to guard against running out of memory at runtime? (e.g. to prevent stack or heap overflow.)
- Separate compilation: Does the language provide facilities for separate compilation of modules, with type checking across the module boundaries?
- Well Understood: Will the designers and programmers understand the programming language sufficiently to write safety-critical software?

Reinhardt [1] provides an argument framework for better understanding the importance of these specific guidelines. In general, each of these specific criteria support the ability of the language to implement the design of the system, the ability of the programmer(s) to correctly implement the software design, and/or the ability of the language tools to correctly implement the system design. Each of these three aspects play an important role in correctly translating the software design to executable code.

3.2 Language-Specific Considerations

3.2.1 Structured Assembly Language

Cullyer and Wickman [13] provide an assessment of the Structured Assembly Language:

- Offers good protection against wild jumps and exhaustion of memory, and can be well understood by a subset of designers
- Care must be taken with respect to ensuring against overwrites, semantics, model of maths, operational arithmetic, and data typing
- Does not provide a recovery mechanism in the language; exception recovery must be addressed by external system
- Several commercially-available structured assembly-level languages have been developed to address the shortfalls of assembly language; long-term support and documentation must be addressed when considering commercial solutions

3.2.2 C

Cullyer and Wickman [13] provide an assessment of the C Language:

- The C language is well understood, with several courses and textbooks available
- Some risk of malfunction related to wild jumps, exception handling, and exhaustion of memory
- No facility for handling overwrites, semantics, model of maths, operational arithmetic, data typing, separate compilation

3.2.3 Pascal

Cullyer and Wickman [13] provide an assessment of the Pascal Language:

- Provides a firm model of maths, and is a well understood language
- Implementation requires attention to wild jumps, overwrites, semantics, operational arithmetic, data typing, exhaustion of memory, and separate compilation
- No facility for implementing exception handling
- Commercially-available subsets of Pascal (SPADE Pascal, in particular) provide well-understood methods for handling the concerns related to International Standards Organization (ISO) Pascal with the exception of exception handling; long-term support and documentation must be addressed when considering commercial solutions

3.2.4 Ada

Cullyer and Wickman [13] provide an assessment of the ADA Language:

- Provides sound protection against wild jumps, strong data typing rules, strong exception handling features, and type checking across module boundaries under separate compilation
- Provided that all low-level facilities are avoided, there is no way an Ada program can overwrite a random location

3.2.5 Java

As described by Whitford, Loutzenhiser, and Mears [12], several factors hinder or prevent the use of Java in real-time systems:

- "Automatic garbage collection preempts application processing for significant amounts of time at unpredictable processing points
- Byte-code interpretation renders execution time unpredictable
- Priority inversion among sets of Java threads cause high priority threads to be unnecessarily delayed by lower priority threads"

However, as detailed by Whitford, Loutzenhiser, and Mears [12], several new Java technologies can be utilized to enhance the reliability and adequacy of Java for use in safety applications. These include Just In Time (JIT) and Ahead Of Time (AOT) compilation and Real-Time Garbage Collection (RTGC), which are appropriate for many soft real-time, safety critical, command and control applications. For hard real-time applications, such as missile flight control, radar control, or weapon control, Real-Time Specification for Java (RTSJ), and the Ravenscar Profile for Java are appropriate. RTSJ is designed to support hard real-time applications with scheduling properties with provisions for periodic and aperiodic tasks, support for deadlines and CPU time budgets, and tools to let tasks avoid garbage collection delays. The Ravenscar Profile for Java is a high-integrity subset of RTSJ that provides predictability of memory utilization, timing, and control and data flow.

It remains essential for the safety engineer to perform thorough analysis of the timing of the system under development in order to fully characterize the functionality of the system.

3.2.6 C++

Reinhardt [1] concludes "that it is possible to use C++ to write software for use in high integrity and safety critical applications. This may be achieved through the use of the safer C++ subset ([High Integrity C++] and additional rules) and a number of software tools for enforcement, static analysis and dynamic testing. In conjunction, some degree of manual code reviews and design inspection are also required."

However, Reinhardt [1] continues by suggesting that industry acceptance of the use of C++ in safety critical applications is not yet mature. This highlights the importance of static analysis, dynamic testing, manual code reviews and design inspection of safety critical systems that utilize the C++ language.

This page intentionally left blank

4 COMPUTER PROCESSORS

The central processing unit within a computer system carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system. The fundamental operation of the CPU is to execute a sequence of stored instructions, called a program, typically generated by a compiler from developer-derived source code.

4.1 General Criteria for Assessment

Bate et.al [4] identify the following framework for assessing processors in safety systems:

- Execution of Instructions – Adequate software timing analyses depend on knowing the maximum time a piece of software takes to execute. As such, both an understanding of processor instruction execution as well as testing of instruction execution is important to characterizing the safety of the system.
- Memory Access – The levels of cache memory implemented in a processor introduce greater complexity and therefore greater uncertainty in the execution times of the instructions executed by the processor. Verification and validation of the processor memory access model (through testing and/or analysis) is important in characterizing the safety of the system.

When selecting a processor for system development, Bate et.al [4] suggest that there are two options: (1) use COTS processors or (2) design and manufacture a purpose-built processor. Each of these design solutions presents their own pros and cons.

Bate et al. [4] state that the "principal advantages of COTS processors are that the widespread use stems from the extensive operational experience and thoroughly exercised support tools such as compilers. However, the admissibility of this operational experience when not directly related to safety-critical applications is a point of debate. Disadvantages of COTS processors lie in the black-box nature of the design and development process, and the fact the processors are not necessarily designed for predictability and their design goals may not be appropriate."

As described by Bate et al. [4], "principal advantages of the bespoke processors lie in the ability to define a processor only using features that can easily be analysed *[sic]* allowing the designer to trade-off processor worst-case performance versus predictability, and that the process can be treated as 'white-box.' The principal disadvantages are that its use is not widespread resulting in a lack of operational experience, lack of freely available tools and that other users may not necessarily accept the design."

The decision to procure a COTS processor or design an in-house solution must be made for each of the processor types described in the following sections.

4.2 Specific Technology Considerations

4.2.1 Scalar Processors

As described by Bate et al. [4], scalar processors (e.g. the Motorola 680x0 family of processors) sequentially execute instructions in their original order. This is an asset for the safety engineer, as "any particular basic block [will] always execute in the same number of [processor] clock cycles" (Whitham

[5]). This defined execution time allows the safety engineer to perform Worst-Case Execution Time (WCET) analysis of safety-critical computer tasks during design and code analysis phases.

Additionally, Whitham [5] states that scalar processors do not typically utilize memory caches between the CPU and system memory. This aspect of scalar processors also helps the safety engineer accurately estimate the WCET of safety-critical tasks performed by the system.

However, Bate et al. [4] describes that scalar processors may be incapable of meeting modern system performance requirements and are generally no longer in production. This presents concerns with maintaining a system through deployment due to potential exhaustion of supply and/or storage degradation (if the developer makes a lifetime buy of remaining processors).

4.2.2 Superscalar Processors

Whitham [5] posits that superscalar processors, such as the Intel x86 family of processors, introduce two technologies that impact the ability of the safety engineer to perform WCET analyses: pipelines and caches. Bate et al. [4] explains that the use of pipelines, allowing for the execution of multiple instructions per clock cycle, and caches, which provide faster access times for certain data depending on execution history, require the safety engineer to create much more complex models in order to accurately determine the timing of the system.

However, research does exist to aid the safety engineer in the creation of a superscalar processor model in support of performing a timing analysis (see Bate et al. [4] and Whitham [5]) by the experienced safety engineer.

4.2.3 Field-Programmable Gate Arrays

As described by Bobrek et al. [7], "Field Programmable Gate Array (FPGA) devices are fundamentally complex software designs implemented by hardware engineers." As such, the safety engineer must be concerned with "FPGA design processes, including software design tools and development methodologies (similar to that used for current software reviews)."

5 NETWORKS

Bonaventure [8] describes a five layer model of computer networks. These layers are: (1) Physical, (2) Data Link, (3) Network, (4) Transport, and (5) Application. The following sections provide some general criteria for assessment of a computer network as well as an overview and some safety considerations of each layer. The safety engineer should have a complete understanding of the network(s) utilized in a system (e.g. the protocols of each layer) to effectively assess the interface against the criteria identified in Section 5.1.

5.1 General Criteria for Assessment

Curtis and France [3] make an important distinction between the time-triggered domain and the event-triggered domain of computer networks.

Interface protocols that utilize the time-triggered domain must be designed around the system under development. As described by Curtis and France [3], "the determination of when a node can place a message on [a time-triggered] network is made at design time." This solution allows for a bounded understanding of the system response time, allowing the safety engineer to identify potential risks and provide recommendations to the system developers.

On the other hand, many non-critical interface protocols utilize the event-driven domain to reduce design complexity (i.e. each node on the network issues signals at a rate that is not determined at design time). This presents potential issues related to network overload and, if the network supports a priority scheme, the "babbling idiot" failure mode in which a node begins to send a high-priority message repeatedly, blocking all lower priority messages from transmission. (Curtis and France [3])

McKinlay [1] provides recommendations for simple safety guidelines when assessing computer networks:

- Analyze (complete) time and timing.
- Analyze what is acceptable time.
- Identify safety critical messages and any associated timing requirements.
- Identify safety critical data to be sent and received and assess parity failures, memory failures, possible out-of-sequence (age) failures.
- Consider wrap-around messaging for the must-know-it-got-there messages. This may or may not include "break engagement" of a friendly type of broadcast message versus the hazard of shooting down that friendly. Be aware that mandating message wrap-around may not be globally selectable, or renounceable, leading to a global traffic jam in reply to broadcast messages. Always consider the timing of the entire network and inability to turn-off the safety message.
- The entire system and the communication subsystems must be sized for the worst-case, multiple messages sent and lost, and the maximum overhead scenario. Anything less will probably result in lost or deadlocked systems, or unknown safety states at remote weapons stations.

5.2 Protocol Considerations

5.2.1 Physical Layer

As described by Bonaventure [8], two communicating devices are linked through a physical medium that transfers an electrical or optical signal between them. This physical layer allows the communicating devices to exchange bits between one another.

Bonaventure [8] also describes a failure mode associated with the physical layer, stating that the physical layer may deliver more or fewer bits to the receiver than originally sent by the sender. "This is mainly due to the fact that the communicating devices use their own clock to transmit and receive bits at a given bit rate. Consider a sender having a clock that ticks one million times per second and sends one bit every tick. Every microsecond, the sender sends an electrical or optical signal that encodes one bit. The sender's bit rate is thus 1 megabit per second (Mbps). If the receiver clock ticks exactly every microsecond, it will also deliver 1 Mbps to its user. However, if the receiver's clock is slightly faster, than *[sic]* it will deliver slightly more than one million bits per second. This explains why the physical layer may lose or create bits." Mitigations against bit loss may be implemented in the higher network layers. (Bonaventure [x, p. 22])

Physical layer protocols include Universal Serial Bus (USB) physical layer, 1000BASE-T (and other varieties), Bluetooth physical layer, and RS-232.

5.2.2 Data Link Layer

"The data link layer builds on the service provided by the underlying physical layer. The data link layer allows two hosts that are directly connected through the physical layer to exchange information. The unit of information exchanged between two entities in the data link layer is a frame. A frame is a finite sequence of bits. Some data link layers use variable-length frames while others only use fixed-length frames. (Bonaventure [8])"

Data link layer protocols include Ethernet, 802.11 wireless Local Area Network, Point-to-Point Protocol, and most forms of serial communication.

5.2.2.1 Ethernet

As described by Lee et al. [14], Ethernet supports either a bus based or switch based physical network. "In a bus-based system, a single bus connects all the nodes in the network, and data transmission can be carried out in only one direction at a time by any one node" (Lee et al. [14]). In a bus-based configuration, if two nodes transmit messages simultaneously, the messages collide and are destroyed (i.e., data is lost). Ethernet uses a Carrier Sense Multiple Access with Collision Detection (CSMA/CD) mechanism to address these conditions in a bus-based configuration and resend the lost message(s). Bus-based implementations remain attractive due to lower implementation and maintenance costs.

"In a switch-based system, separate conductors are used for sending and receiving data. In a fully switched network, Ethernet nodes only communicate with the switch and never directly with each other. Thus, they can forego the collision detection process and transmit at will. Hence, data transmission between the switch and the nodes takes place in a collision-free environment […and] introduces more determinism into the system" (Lee et al. [14]).

5.2.3 Network Layer

The task of the network layer is to exchange information between hosts that are not attached to the same physical medium (e.g. a router exists between the two communicating devices), as described by Bonaventure [8]. "Network layer entities exchange *packets.* A *packet* is a finite sequence of bytes that is transported by the data link layer inside one or more frames. A package usually contains information about its origin and destination, and usually passes through several intermediate devices called routers on its way from its origin to its destination" (Bonaventure [8]).

Network layer protocols include Internet Protocol (IP), Internet Control Message Protocol (ICMP), and Routing Information Protocol (RIP).

5.2.4 Transport Layer

"Ensuring the reliable delivery of the data produced by applications is the task of the transport layer. Transport layer entities exchange segments. A segment is a finite sequence of bytes that are transported inside one or more packets. A transport layer entity issues segments to the underlying network layer entity. (Bonaventure [8])"

Transport layer protocols include Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Datagram Congestion Control Protocol (DCCP).

5.2.4.1 Transmission Control Protocol

As described by Lee et al. [14], TCP provides stream data transfer, reliability of transmission, efficient flow control, full-duplex operation (separate paths for sending and receiving data), and multiplexing (sending/receiving multiple signals or streams of information at the same time). "However, TCP's use of an acknowledgement and retransmission scheme for ensuring reliability makes it inherently nondeterministic" (i.e. message delivery times cannot be determined in advance) (Lee et al. [14]).

5.2.4.2 User Datagram Protocol

"UDP is a connectionless transport layer protocol. Unlike TCP, UDP adds no reliability, flow-control, or error-recovery functions to IP. Because of UDP's simplicity, UDP headers contain fewer bytes and consume less network overhead than TCP. UDP is useful in situations where the reliability mechanisms of TCP are not necessary, such as cases where a higher-layer protocol might provide error and flow control. UDP is equally nondeterministic." However, if the application layer protocol performs traffic regulation and resource reservation, determinism can be achieved by using UDP without the overhead of TCP. (Lee et al. [14])

5.2.5 Application Layer

Bonaventure [8] states that the application layer "includes all the mechanisms and data structures that are necessary for the applications of the sender and receiver. " The sender and receiver applications must both agree on a set of syntactical and semantic rules in order to ensure that each other are able to understand queries sent between them. These rules define the format and ordering of the messages exchanged, and is called an application-level protocol. (Bonaventure [8])

Application layer protocols include Domain Name System (DNS), File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), and Secure Shell (SSH).

This page intentionally left blank

6 GENERAL LESSONS/EXAMPLES

- General principles for software engineering can be gleaned from other engineering disciplines' mistakes. For example, the Tacoma Narrows bridge collapse and the Challenger Shuttle explosion. (Holloway, 1999)
    - Relying heavily on theory, without adequate confirming data, is unwise
    - Going well beyond existing experience is unwise
    - In studying existing experience, more than just the recent past should be included
    - When safety is concerned, misgivings on the part of competent engineers should be given strong consideration, even if the engineers cannot fully substantiate these misgivings
- Currently System Safety Engineers do not communicate well with Software Engineers, especially in dealing with what system safety actually is. Software Engineers need to learn the concepts and language of system safety. (Lavine, 1989)
    - Several analyses that were found to be particularly useful (with the last two found to be very labor intensive) were:
        - Software fault-tree analysis (soft tree) where the hazardous condition was then controlled through the software interface
        - Failure Modes, Effects and Criticality Analysis (FMECA)
        - Petri-net analysis on the entire software component of one system or on a single, highly critical module - Patriot Missile Failure due to software fault during the Gulf War (GAO/IMTEC-92-26)
    - Inaccurate system calculation of the time since boot caused the system to ignore an incoming scud missile, which killed 28 soldiers and injured 100.
    - The operational environment must be considered when designing a system. This particular event would not have occurred (theoretically) if the system had not been running for 100 hours continuously. If the system had been rebooted the timer would have reset, erasing the accumulated timing error.
- Faults in hardware can be poorly managed within the software as evidenced by the following:
    - In 1990 a single switch at one of AT&T's 114 switching centers suffered a minor mechanical problem and shut down the center. When the center came back up, it sent a message to other switching centers, which in turn caused them to shut down and brought down the entire AT&T network for 9 hours, resulting in 75 million missed phone calls. This was caused by a single line of buggy code in a complex software upgrade implemented to speed up calling that caused a ripple effect that shut down the network.

This page intentionally left blank

REFERENCES

[1] A. McKinlay, "Safety Considerations in Network Messaging," *25th International System Safety* Conference, Baltimore, MD, 2007.

[2] D. Reinhardt, "Use of the C++ Programming Language in Safety Critical Systems," M.S. thesis, Dept. of Comp. Sci., Univ. of York, England, 2004.

[3] H. Curtis, R. France. (1999). "Time Triggered Protocol (TTP/C): A Safety-Critical System Protocol." Available: http://users.ece.utexas.edu/~bevans/courses/ee382c/projects/fall99/curtis-france/litsurvey.pdf

[4] I. Bate, P. Conmy, T. Kelly, J. McDermid, "Use of Modern Processors in Safety-Critical Applications," *Computer Journal*, vol. 44, no. 6, pp. 531-543, 2001.

[5] J. Whitham, "Real-time Processor Architectures for Worst Case Execution Time Reduction," Ph.D. dissertation, Dept. of Comp. Sci., Univ. of York, UK, 2008.

[6] K. Stottlar, private communication, Jul 2012

[7] M. Bobrek, D. Bouldin, D.E. Holcomb, S.M. Killough, S.F. Smith, C. Ward, R.T. Wood, "Review Guidelines for Field-Programmable Gate Arrays in Nuclear Power Plant Safety Systems," United States Nuclear Regulatory Commission, Office of Nuclear Regulatory Research, Oak Ridge, TN, Rep. No. NUREG/CR-7006 ORNL/TM-2009/20, Feb. 2010.

[8] O. Bonaventure. (2011, October 31). *Computer Networking: Principles, Protocols and Practice* [Online]. Available: http://www.saylor.org/courses/cs402/

[9] "operating system (OS)." *Encyclopædia Britannica Online* [Online]. Available: http://www.britannica.com/EBchecked/topic/429897/operating-system

[10] R. Kammerer, "Linux in Safety-Critical Applications," Ph.D. dissertation, Dept. of Comp. Sci. and Eng. Studies, Vienna Univ. of Tech., Vienna, Austria, 2008.

[11] R. Pierce, S. Wilson, J. McDermid, L. Beus-Dukic, A. Eaton, "Requirements for the Use of COTS Operating Systems in Safety Related Air Traffic Services," *DAta Systems in Aerospace Conference (DASIA'99)*, Lisbon, Portugal, 1999, pp. 17-20.

[12] S. Whitford, D. Loutzenhiser, M. Mears, "Safety Concerns with the Use of Java in Real-Time Combat System Applications," G Department Technical Brief, Naval Surface Warfare Center, Dahlgren Division, Nov 2007.

[13] W.J. Cullyer, B.A. Wickman, "The Choice of Computer Languages for Use in Safety-Critical Systems," *Software Engineering Journal*, vol. 6, no. 2, pp. 51-58, 1991.

[14] Y. Lee, E. Rachlin, P. Scandura, Jr., "Safety and Certification Approaches for Ethernet-Based Aviation Databuses," U.S. Department of Transportation, Federal Aviation Administration, Office of Aviation Research and Development, Washington, D.C., Rep. No. DOT/FAA/AR-05/52, Dec. 2005.