Technical Information Report

AAMI TIR32:2004

Medical device software risk management

AAMI
ASSOCIATION FOR THE ADVANCEMENT OF MEDICAL INSTRUMENTATION

Medical device software risk management

Approved 23 December 2004 by
Association for the Advancement of Medical Instrumentation

Abstract: This AAMI technical information report provides information useful to performing effective software risk management, a significant part of the overall risk management process for medical devices containing software. It does this in the context of ANSI/AAMI/ISO 14971:2000, *Medical devices—Application of risk management to medical devices*, and in the context of ANSI/AAMI SW68:2001, *Medical device software—Software life cycle processes*.

Keywords: hazard, risk, software

AAMI Technical Information Report

A technical information report (TIR) is a publication of the Association for the Advancement of Medical Instrumentation (AAMI) Standards Board that addresses a particular aspect of medical technology.

Although the material presented in a TIR may need further evaluation by experts, releasing the information is valuable because the industry and the professions have an immediate need for it.

A TIR differs markedly from a standard or recommended practice, and readers should understand the differences between these documents.

Standards and recommended practices are subject to a formal process of committee approval, public review, and resolution of all comments. This process of consensus is supervised by the AAMI Standards Board and, in the case of American National Standards, by the American National Standards Institute.

A TIR is not subject to the same formal approval process as a standard. However, a TIR is approved for distribution by a technical committee and the AAMI Standards Board.

Another difference is that, although both standards and TIRs are periodically reviewed, a standard must be acted on—reaffirmed, revised, or withdrawn—and the action formally approved usually every five years but at least every 10 years. For a TIR, AAMI consults with a technical committee about five years after the publication date (and periodically thereafter) for guidance on whether the document is still useful—that is, to check that the information is relevant or of historical value. If the information is not useful, the TIR is removed from circulation.

A TIR may be developed because it is more responsive to underlying safety or performance issues than a standard or recommended practice, or because achieving consensus is extremely difficult or unlikely. Unlike a standard, a TIR permits the inclusion of differing viewpoints on technical issues.

CAUTION NOTICE: This AAMI TIR may be revised or withdrawn at any time. Because it addresses a rapidly evolving field or technology, readers are cautioned to ensure that they have also considered information that may be more recent than this document.

All standards, recommended practices, technical information reports, and other types of technical documents developed by AAMI are *voluntary*, and their application is solely within the discretion and professional judgment of the user of the document. Occasionally, voluntary technical documents are adopted by government regulatory agencies or procurement authorities, in which case the adopting agency is responsible for enforcement of its rules and regulations.

Comments on this technical information report are invited and should be sent to AAMI, Attn: Standards Department, 1110 N. Glebe Road, Suite 220, Arlington, VA 22201-4795.

Published by

Association for the Advancement of Medical Instrumentation
1110 N. Glebe Road, Suite 220
Arlington, VA 22201-4795

Printed in the United States of America

ISBN 1–57020–233–8

Contents

Page

Glossary of equivalent standards v

Committee representation vii

Foreword ix

Introduction x

1 Scope 1

1.1 Purpose 1
1.2 Field of application 1
1.3 Usage 2
1.4 Organization 2
1.5 Limitations 2

2 References 2

3 Definitions 3

4 Perspective 1: Basic concepts of medical device software risk management 6

4.1 Medical device risk management 6
4.1.1 Software risk cannot be managed effectively in isolation 6
4.1.2 Software input is an important part of device risk management 7
4.2 Software risk management 7
4.3 Risk control 11
4.3.1 Risk control through development assurance levels 14
4.3.2 Achieving development assurance level requirements 14
4.4 Integration of risk management in the software life cycle 15
4.4.1 Risk management is also essential for software maintenance 16
4.5 Common confusion regarding software risk management 16

5 Perspective 2: Software considerations in medical device risk management 17

5.1 Risk analysis 17
5.1.1 Risk analysis procedure 18
5.1.2 Intended-use or intended-purpose identification 19
5.1.3 Identification of known or foreseeable hazards 20
5.1.4 Estimation of the risks for each hazard 23
5.2 Risk evaluation 25
5.3 Risk control 27
5.3.1 Options analysis 27
5.3.2 Implementation of risk control measures 29
5.3.3 Residual risk evaluation 29
5.3.4 Other generated hazards 30
5.4 Risk management report 30
5.5 Postproduction information 31

6 Perspective 3: Software risk management within a software life cycle 32

6.1 Risk management–life cycle integration 32
6.2 ANSI/AAMI SW68:2001 development process 35
6.2.1 Process implementation 35
6.2.2 Software requirements analysis 36
6.2.3 Software architectural design 38
6.2.4 Software detailed design 41
6.2.5 Code and unit test 42
6.2.6 Integration, system, and validation testing 46
6.2.7 Software release 48

6.3 ANSI/AAMI SW68 maintenance process 49
6.3.1 Process implementation....50
6.3.2 Problem and modification analysis and implementation....50

7 Perspective 4: Soft factors in software risk management....52

7.1 Intended-use and domain knowledge....52
7.2 Team dynamics....53
7.3 Management 53
7.4 Programming experience and attitude....53
7.5 Technical knowledge....54

Annexes

A Direct causes sample table 55

B Indirect causes and risk control measures table (failures due to unpredictable behaviors)....62

Tables

1 Examples of risk control measures 27

2 Life cycle–risk management grid....33

3 Types of analyses 37

4 Various methods of software redundancy 40

5 Methods to facilitate assurance that risk control methods are likely to perform as intended 47

Figures

1 Hazard–cause continuum 5

2 Software risk management context diagram 8

3 Types of functionality 9

4 Causal chains 10

5 First and last points of control context diagram....12

6 First and last points of software control in causal chains 13

7 Direct and indirect causes....22

Glossary of equivalent standards

International Standards adopted in the United States may include normative references to other International Standards. For each International Standard that has been adopted by AAMI (and ANSI), the table below gives the corresponding U.S. designation and level of equivalency to the International Standard.

NOTE—Documents are sorted by international designation.

Other normatively referenced International Standards may be under consideration for U.S. adoption by AAMI; therefore, this list should not be considered exhaustive.

International designation	U.S. designation	Equivalency
IEC 60601-1-2:2001 and Amendment 1:2004	ANSI/AAMI/IEC 60601-1-2:2001 and Amendment 1:2004	Identical
IEC 60601-2-04:2002	ANSI/AAMI DF80:2003	Major technical variations
IEC 60601-2-19:1990 and Amendment 1:1996	ANSI/AAMI II36:2004	Major technical variations
IEC 60601-2-20:1990 and Amendment 1:1996	ANSI/AAMI II51:2004	Major technical variations
IEC 60601-2-21:1994 and Amendment 1:1996	ANSI/AAMI/IEC 60601-2-21 and Amendment 1:2000 (consolidated texts)	Identical
IEC 60601-2-24:1998	ANSI/AAMI ID26:2004	Major technical variations
IEC TR 60878:2003	ANSI/AAMI/IEC TIR60878:2003	Identical
IEC TR 62296:2003	ANSI/AAMI/IEC TIR62296:2003	Identical
ISO 5840:200x[1]	ANSI/AAMI/ISO 5840:2005	Identical
ISO 7198:1998	ANSI/AAMI/ISO 7198:1998/2001/(R)2004	Identical
ISO 7199:1996	ANSI/AAMI/ISO 7199:1996/(R)2002	Identical
ISO 10993-1:2003	ANSI/AAMI/ISO 10993-1:2003	Identical
ISO 10993-2:1992	ANSI/AAMI/ISO 10993-2:1993/(R)2001	Identical
ISO 10993-3:2003	ANSI/AAMI/ISO 10993-3:2003	Identical
ISO 10993-4:2002	ANSI/AAMI/ISO 10993-4:2002	Identical
ISO 10993-5:1999	ANSI/AAMI/ISO 10993-5:1999	Identical
ISO 10993-6:1994	ANSI/AAMI/ISO 10993-6:1995/(R)2001	Identical
ISO 10993-7:1995	ANSI/AAMI/ISO 10993-7:1995/(R)2001	Identical
ISO 10993-9:1999	ANSI/AAMI/ISO 10993-9:1999	Identical
ISO 10993-10:2002	ANSI/AAMI BE78:2002	Minor technical variations
ISO 10993-11:1993	ANSI/AAMI 10993-11:1993	Minor technical variations
ISO 10993-12:2002	ANSI/AAMI/ISO 10993-12:2002	Identical
ISO 10993-13:1998	ANSI/AAMI/ISO 10993-13:1999/(R)2004	Identical
ISO 10993-14:2001	ANSI/AAMI/ISO 10993-14:2001	Identical
ISO 10993-15:2000	ANSI/AAMI/ISO 10993-15:2000	Identical

[1] Currently at FDIS stage

International designation	U.S. designation	Equivalency
ISO 10993-16:1997	ANSI/AAMI/ISO 10993-16:1997/(R)2003	Identical
ISO 10993-17:2002	ANSI/AAMI/ISO 10993-17:2002	Identical
ISO 11134:1994	ANSI/AAMI/ISO 11134:1993	Identical
ISO 11135:1994	ANSI/AAMI/ISO 11135:1994	Identical
ISO 11137:1995 and Amdt 1:2001	ANSI/AAMI/ISO 11137:1994 and A1:2002	Identical
ISO 11138-1:1994	ANSI/AAMI ST59:1999	Major technical variations
ISO 11138-2:1994	ANSI/AAMI ST21:1999	Major technical variations
ISO 11138-3:1995	ANSI/AAMI ST19:1999	Major technical variations
ISO TS 11139:2001	ANSI/AAMI/ISO 11139:2002	Identical
ISO 11140-1:1995 and Technical Corrigendum 1:1998	ANSI/AAMI ST60:1996	Major technical variations
ISO 11140-5:2000	ANSI/AAMI ST66:1999	Major technical variations
ISO 11607:2003	ANSI/AAMI/ISO 11607:2000	Identical
ISO 11737-1:1995	ANSI/AAMI/ISO 11737-1:1995	Identical
ISO 11737-2:1998	ANSI/AAMI/ISO 11737-2:1998	Identical
ISO 11737-3:2004	ANSI/AAMI/ISO 11737-3:2004	Identical
ISO TR 13409:1996	AAMI/ISO TIR13409:1996	Identical
ISO 13485:2003	ANSI/AAMI/ISO 13485:2003	Identical
ISO 13488:1996	ANSI/AAMI/ISO 13488:1996	Identical
ISO 14155-1:2003	ANSI/AAMI/ISO 14155-1:2003	Identical
ISO 14155-2:2003	ANSI/AAMI/ISO 14155-2:2003	Identical
ISO 14160:1998	ANSI/AAMI/ISO 14160:1998	Identical
ISO 14161:2000	ANSI/AAMI/ISO 14161:2000	Identical
ISO 14937:2000	ANSI/AAMI/ISO 14937:2000	Identical
ISO TR 14969:2004	ANSI/AAMI/ISO TIR14969:2004	Identical
ISO 14971:2000 and A1:2003	ANSI/AAMI/ISO 14971:2000 and A1:2003	Identical
ISO 15223:2000, A1:2002, and A2:2004	ANSI/AAMI/ISO 15223:2000, A1:2001, and A2:2004	Identical
ISO 15225:2000 and A1:2004	ANSI/AAMI/ISO 15225:2000 and A1:2004	Identical
ISO 15674:2001	ANSI/AAMI/ISO 15674:2001	Identical
ISO 15675:2001	ANSI/AAMI/ISO 15675:2001	Identical
ISO TS 15843:2000	ANSI/AAMI/ISO TIR15843:2000	Identical
ISO TR 15844:1998	AAMI/ISO TIR15844:1998	Identical
ISO 15882:2003	ANSI/AAMI/ISO 15882:2003	Identical
ISO TR 16142:1999	ANSI/AAMI/ISO TIR16142:2000	Identical
ISO 17664:2004	ANSI/AAMI ST81:2004	Major technical variations
ISO 25539-1:2003	ANSI/AAMI/ISO 25539-1:2003	Identical

Committee representation

Association for the Advancement of Medical Instrumentation

AAMI Medical Device Software Committee

This technical information report was developed by the Software Risk Management Task Group of the AAMI Medical Device Software Committee. Approval of this TIR does not necessarily imply that all committee members voted for its approval.

At the time this document was published, the **AAMI Medical Device Software Committee** had the following members:

Cochairs: Sherman Eagles
John F. Murray, Jr.

Secretary: Nancy George

Members: Robert G. Britain, National Electrical Manufacturers Association (NEMA)
Warren P. Dickinson, Ion Beam Applications
Sherman Eagles, Medtronic Inc.
Christine M. Flahive, Christine M. Flahive Associates
John J. Flynn, Hill-Rom Company
Richard C. Fries, Instrumentarium USA Inc.
Larry A. Fry, Draeger Medical
Nancy George, Software Quality Management Inc.
Larry Gillum, Cardinal Health Medical Products and Services Group
Steven Gitelis, Guidant Corp.
Lori Haller, STERIS Corp.
James P. Hempel, Tyco Healthcare/US Surgical
Neil Holland, Abbott Laboratories
David R. Jones, Philips Medical Systems
Christopher Keegan, Welch Allyn Inc.
Charlie D. King, Siemens Medical Systems
Alan Kusinitz, SoftwareCPR
Bernie Liebler, Advanced Medical Technology Association (AdvaMed)
Mary Beth McDonald, St. Jude Medical
E. Paul Morozoff, Spacelabs Medical Inc.
Paul Mueller, Bausch & Lomb Inc.
John F. Murray, Jr., U.S. Food and Drug Administration
Harvey Rudolph, PhD, Underwriters Laboratories Inc.
Carla Sivak, Mitek/Johnson & Johnson
Christine Strysik, Baxter Healthcare Corp.
Steven D. Walter, Becton Dickinson
Gregory Whitney, CR Bard

Alternates: Christopher P. Clark, Bausch & Lomb Inc.
Brian J. Fitzgerald, U.S. Food and Drug Administration
Christopher D. Ganser, CR Bard
Florentino Kimpo, Medtronic Physio-Control
Gretel Lumley, Philips Medical Systems
David McCall, STERIS Corp.
Dennis Mertz, Becton Dickinson
Carl Pantiskas, Spacelabs Medical Inc.
Raj G. Raghavendran, Ethicon Endo-Surgery/Johnson & Johnson
Robert Smith, St. Jude Medical
Fayez Sweiss, Abbott Laboratories
Donna Bea Tillman, PhD, U.S. Food and Drug Administration
Stephen Vastagh, National Electrical Manufacturers Association (NEMA)

The AAMI Medical Device Software Committee gratefully acknowledges the work of its **Software Risk Management Task Group**. At the time this document was published, the task group had the following members:

Cochairs: Paul L. Jones
Alan Kusinitz
Secretary: Annette M. Hillring
Members: Sherman Eagles, Medtronic Inc.
Lucille Ferus, SoftwareCPR
Stan Hamilton, SoftwareCPR
Annette M. Hillring, Johnson & Johnson
Paul L. Jones, U.S. Food and Drug Administration
Alan Kusinitz, SoftwareCPR
Eric Linner, Baxter Healthcare Corp.
John F. Murray, Jr., U.S. Food and Drug Administration
Brian Pate, GE Healthcare
Raj G. Raghavendran, Ethicon Endo-Surgery/Johnson & Johnson
Yves Theriault, STERIS Corp.

The AAMI Medical Device Software Committee and its Software Risk Management Task Group would like to thank the following individuals for their invaluable contribution to the development of this TIR:

Sean M. Beatty, High Impact Services Inc.
Doug Lichorwic, Northrup Grumman
David Vogel, Intertech Engineering Associates, Inc.

NOTE—Participation by federal agency representatives in the development of this technical information report does not constitute endorsement by the federal government or any of its agencies.

Foreword

Effective software risk management is a significant part of the overall risk management process for medical devices containing software. This technical information report (TIR) provides information useful to performing effective software risk management. It does this in the context of ANSI/AAMI/ISO 14971:2000, *Medical devices—Application of risk management to medical devices*, and in the context of ANSI/AAMI SW68:2001, *Medical device software—Software life cycle processes*.

Introduction

Software is often an integral part of medical device technology. Establishing the safety and effectiveness of a medical device containing software requires knowledge of what the software is intended to do and demonstration that the implementation of the software fulfills those intentions without causing any unacceptable risks.

Accidents are often preceded by a belief that they cannot happen. People often believe that software is designed to work properly and that testing ensures that it will work properly, despite a general recognition that neither quality nor safety can be "tested into" software. The fact is that most software testing does little more than exercise a small sampling of the software logic in all but the simplest of programs.

Ignoring the possibility of defects in software can lead to the release of medical device software that can fail in ways that affect device safety and effectiveness. This report is predicated on the notion that, as part of the software development process and general device risk management process, specific focus is required on

— identifying software's relationship to potential device hazards, both in terms of intended software functionality and the effects of potential software defects;

— identifying adequate risk control measures in terms of software and nonsoftware design; and

— verifying the implementation and effectiveness of risk control measures.

The cost of diligence in this regard is inconsequential compared to the cost of an accident in any way one would like to define cost (e.g., human, financial, legal, or regulatory exposure).

Many standards have taken the approach of having separate "safety" and "performance" standards for medical electrical equipment. This approach was a natural extension of the historical approach taken at the national and international level with other electrical equipment standards (e.g., those for consumer electronics), where basic physical safety is regulated through mandatory standards but other "performance" specifications are regulated by market pressure. In this context, one could say, "The ability of an electric kettle to boil water is not critical to its safe use!"

This is not the situation with medical devices. Responsible organizations must depend on standards to ensure effectiveness as well as basic safety. The accuracy with which the equipment controls the delivery of energy or therapeutic substances to the patient is of concern because a lack of effectiveness can become a safety issue. Likewise, the manner in which medical device software processes and displays physiological or diagnostic data is of concern because it can affect patient management. Medical authorities are equally concerned about the ability of the equipment to prevent hazards and to perform clinical functions effectively. An increasing amount of the clinical functionality of many medical devices is controlled by a software subsystem of the medical device.

Thus, it is sometimes difficult to make clear distinctions between safety and effectiveness. As such, where this report uses the term "safety," it is intended to include effectiveness in cases where the lack of effectiveness can be a safety issue.

The benefit derived by the patient must always be considered along with the risk in using a medical device. This implies that the need for protection from risk caused by the medical device differs depending on the risk to the patient of not receiving treatment, and these varying needs must be considered as part of the overall risk evaluation of the device.

Many standards and publications—national, international, and sector specific—address risk management, and others address software life cycle development processes. However, none of these documents address software system safety design issues in the context of medical device systems. Additionally, software-related aspects of medical device risk management need to be explained in software terms with software examples. The AAMI Medical Device Software Committee recognizes a need for this information and is addressing the issue by way of this technical information report.

Medical device software risk management

1 Scope

This technical information report (TIR) should be regarded as a reference for developing safe software systems to be used in medical devices. The information that it contains provides a framework within which experience, insight, and judgment are applied systematically to reduce medical device risks. The TIR does this in the context of ANSI/AAMI/ISO 14971:2000, *Medical devices—Application of risk management to medical devices*, and in the context of ANSI/AAMI SW68:2001, *Medical device software—Software life cycle processes*.

For readers to understand the scope of this document, it is important to understand the distinction between software safety and software reliability. The National Institute of Science and Technology information report [NISTIR 5589] on software hazard analysis states this distinction quite clearly:

> Software safety should not be confused with software reliability. Reliability is the ability of a system to perform its required functions under stated conditions for a specified period of time [IEEE610]. Safety is the probability that conditions (hazards) that can lead to a mishap do not occur, whether or not the intended function is performed [LEVESON86]. Reliability is interested in all possible software errors, while safety is concerned only with those errors that cause actual system hazards [LEVESON86]. . . . Software safety and software reliability are part of software quality. Quality is the degree to which a system meets specified requirements, and customer or user needs or expectations [IEEE610].

Many of the same techniques used to ensure software reliability and quality are relevant for ensuring software safety. This report does not discuss general aspects of software quality assurance.

1.1 Purpose

The goal of this TIR is to be a technical reference on risk management for medical devices. It is intended primarily for software engineers, software quality assurance personnel, and those responsible for medical device risk management. Others involved in medical device product development, quality assurance, regulatory affairs, and auditing may also find this document useful.

The report attempts to clarify process relationships outlined in ANSI/AAMI SW68:2001, *Medical device software—Software life cycle processes*, and ANSI/AAMI/ISO 14971:2000, *Medical devices—Application of risk management to medical devices*, in the context of software system safety, keeping in mind the varied interests of the audience.

Understanding the terminology and its proper context is key to understanding the associated processes. This report attempts to clarify some of those subtleties by looking at the components of risk management, and by using precise language to identify how those components relate to each other. For example, ISO definitions such as *hazard*, *harm*, and *safety* are clarified through use of additional terms and examples from a software perspective.

The report provides guidance for those new to the concepts of software system safety in the medical device industry and as an aide-memoire for medical device and software designers more familiar with the topic.

The first objective of this report is to provide those working “down in the trenches” with some insight into safety considerations when using software in a medical device. A second objective is to help risk managers understand the implications for risk management posed by the presence of software in the system. All too often, those charged with the responsibility for developing software and those charged with the responsibility of managing risk operate independently of each other. It is a goal of this report to help bridge this divide by fostering communication and a shared understanding of the relationship between software engineering and risk management.

1.2 Field of application

This TIR contains information applicable to risk management for the entire array of medical device software, including:

— Embedded software systems (e.g., glucose meter firmware)

— Stand-alone software systems (e.g., dosage calculations programs)

— Information systems (e.g., clinical information systems)

— Accessory systems (e.g., radiation planning systems)

— Telemedicine systems (e.g., remote imaging systems)

1.3 Usage

This report discusses medical device risk management in the context of software. The report is technical in nature and does not dwell on processes specified in other relevant consensus standards, ANSI/AAMI/ISO 14971:2000 and ANSI/AAMI SW68:2001 in particular. The expectation is that software engineers will use the principles of software risk management described in this report together with other general risk management and software engineering standards and reference and educational material.

1.4 Organization

This report is organized into four major sections that contribute to the subject matter from a different perspective:

— Perspective 1: Basic concepts of medical device software risk management (section 4)

  This section presents key concepts and terms useful for ensuring software safety and effectiveness.

— Perspective 2: Software considerations in medical device risk management (section 5)

  This section goes through major elements of the medical device risk management process defined in ANSI/AAMI/ISO 14971:2000 and provides a software perspective for each risk management activity.

— Perspective 3: Software risk management within a software life cycle (section 6)

  This section steps through the activities of the two primary software life cycle processes defined in ANSI/AAMI SW68:2001 (development and maintenance) and discusses risk management considerations for each activity of the software life cycle.

— Perspective 4: Soft factors in software risk management (section 7)

  This section identifies a number of soft factors that are important to generate safe and effective software.

Two important annexes (Annexes A and B) are referenced from several sections. These annexes contain tables that provide examples of typical medical device software functionality related to safety, of potential software failures that could contribute to hazardous situations, and of potential risk control measures corresponding to these failures.

Since risk management is addressed from different perspectives in each major section, there is some intentional redundancy in the information provided.

1.5 Limitations

This report is not a complete exposition of all of the software engineering tools and techniques used to manage risk. It discusses approaches to reduce risk but does not define specific acceptable risk levels. It was not developed to explicitly address software used in medical device manufacturing, quality systems, design, development, or electronic recordkeeping, although the concepts presented could be useful in some respects.

2 References

ANSI/AAMI SW68:2001, *Medical device software—Software life cycle processes*.

ANSI/AAMI/ISO 14971:2000, *Medical devices—Application of risk management to medical devices*.

DEF STAN 00-55, 00-56:1997, *Requirements for Safety-Related Equipment in Defence Equipment*. Ministry of Defence, Directorate of Standardization, Glasgow, Scotland.

IEC 60513:1994, *Fundamental aspects of safety standards for medical electrical equipment*.

IEC 61508-3:1998, *Functional safety of electrical/electronic/programmable electronic safety-related systems—Part 3: Software requirements*.

IEEE 610:1990, *IEEE computer dictionary—Compilation of IEEE standard computer glossaries*.

ISO 9000-3:1997, *Quality management and quality assurance standards—Part 3: Guidelines for the application of ISO 9001:1994 to the development, supply, installation, and maintenance of computer software*.

ISO/IEC Guide 51:1999, *Safety aspects—Guidelines for their inclusions in standards*.

ISO/IEC 12207:1995/Amendment 1:2002, *Information technology—Software life cycle processes*.

ISO/IEC 15026:1998, *Information technology—System and software integrity levels*.

ISO/IEC 60601-1-4:2000, *Medical electrical equipment—Part 4: Programmable electrical medical systems*.

NISTIR 5589:1995, *A study on hazard analysis in high-integrity software standards and guidelines*.

Pradhan, DK, *Fault-Tolerant Computer System Design*. Upper Saddle River (NJ): Prentice-Hall, 1996.

3 Definitions

3.1 corrective measures: Risk control measures meant to detect a potential cause of a hazardous situation and reduce the severity or probability of harm occurring.

3.2 direct cause: Defect whose results are predictable and that can contribute to one or more hazards.

3.3 diversity: Characteristic of risk control measures that uses different architectures, components, algorithms, development processes, or other techniques to reduce the risk of a single software defect resulting in a hazard.

3.4 failsafe: Management of failure modes to prevent harm; performance characteristics necessary to maintain risk within acceptable limits. (See IEC 60601-1-4:2000.)

3.5 harm: Physical injury or damage to the health of people, damage to property, or damage to the environment. (See ANSI/AAMI/ISO 14971:2000.)

3.6 hazard: Potential source of harm. (See ANSI/AAMI/ISO 14971:2000.)

3.7 hazardous event: Any occurrence of a hazard. (See ISO/IEC Guide 51:1999.) A hazardous situation that results in harm.

3.8 hazardous situation: Circumstance in which people, property, or the environment are exposed to one or more hazards. (See ANSI/AAMI/ISO 14971:2000.)

3.9 indirect cause: Defect that could have unpredictable side effects and result in one or more hazards (e.g., a memory overflow that could corrupt safety-related code or data).

3.10 intended use/intended purpose: Use of a product, process, or service in accordance with the specifications, instructions, and information provided by the manufacturer. (See ANSI/AAMI/ISO 14971:2000.)

3.11 life cycle model: Framework containing the processes, activities, and tasks involved in the development, operation, and maintenance of a software product, spanning the life of the system from the definition of its requirements to the termination of its use. (See ISO/IEC 12207:1995.)

3.12 misuse: Any use of a device, whether malicious or not, that is not within the parameters of the intended use.

3.13 mitigative measures: Risk control measures meant to reduce the severity of a hazardous situation.

3.14 preventive measures: Risk control measures that prevent causes from contributing to hazardous situations.

3.15 protective measures: Actions taken to reduce the risk of hazards and ultimately the risk of harm. These actions may be preventive, corrective, or mitigative. Although the term *mitigation* is sometimes used as a synonym for protective measure, *mitigation* has a more specific meaning as indicated above. *Protective measures* or *risk control measures* are the generic terms used to describe the steps taken to reduce risk.

3.16 reliability: Predictable and consistent performance of the software under conditions specified in the design basis. This top-level attribute is important to safety because it decreases the likelihood that faults causing unsuccessful operation will be introduced into the source code during implementation. (See IEEE 610:1990.)

3.17 residual risk: Risk remaining after protective measures have been taken. (See ANSI/AAMI/ISO 14971:2000.)

3.18 risk: Combination of the probability of occurrence of harm and the severity of that harm. (See ANSI/AAMI/ISO 14971:2000.)

3.19 risk analysis: Systematic use of available information to identify hazards and to estimate the risk. (See ANSI/AAMI/ISO 14971:2000.)

3.20 risk assessment: Overall process comprising a risk analysis and a risk evaluation. (See ANSI/AAMI/ISO 14971:2000.)

3.21 risk control: Process through which decisions are reached and protective measures are implemented for reducing risks to, or maintaining risks within, specified levels. (See ANSI/AAMI/ISO 14971:2000.)

3.22 risk control measure: Methods used to reduce the severity or probability of harm.

3.23 risk evaluation: Judgment, on the basis of risk analysis, of whether a risk that is acceptable has been achieved in a given context, given the current values of society. (See ANSI/AAMI/ISO 14971:2000.)

3.24 risk management: Systematic application of management policies, procedures, and practices to the tasks of analyzing, evaluating, and controlling risk. (See ANSI/AAMI/ISO 14971:2000.)

3.25 robustness: Capability of the safety system software to operate in an acceptable manner under abnormal conditions or events. This top-level attribute is important to safety because it enhances the capability of the software to handle exception conditions, recover from internal failures, and prevent propagation of errors arising from unusual circumstances (not all of which may have been fully defined in the design basis). (See IEEE 610:1990.)

3.26 safety-related software: Software that, if defective, could result in a hazardous situation.

3.27 safe state: Device modes that are free from unacceptable risk.

3.28 safety: Freedom from unacceptable risk. (See ANSI/AAMI/ISO 14971:2000.)

3.29 safety-related requirements: Requirements that, if not met, could result in a hazardous situation.

3.30 severity: Measure of the possible consequences of a hazard. (See ANSI/AAMI/ISO 14971:2000.)

3.31 software item: Any identifiable part of a software product. (See ISO 9000-3:1997.)

3.32 software unit: Software item that is not subdivided into other items for the purposes of configuration management or testing.

3.33 system: Term used in software standards such as ISO/IEC 12207:1995 to include the computer hardware on which the software will run. In general, *system* refers to the entire system and not just to software, electronics, mechanics, fluidics, or other specific components. In the context of this report, the term *medical device* is used to refer to the system of which software is a part.

3.34 systematic failure: Design errors that do not occur randomly and for which a probability of occurrence cannot be calculated. (See ANSI/AAMI/ISO 14971:2000, Annex E 4.[2])

Discussion of definitions

In the ANSI/AAMI/ISO 14971:2000 taxonomy, there is *harm*, with an emphasis on clinical mishap and effect; there are *hazards*, which represent potential sources of harm; and there are *causes* or *contributing factors* for the hazards.

This definition of *hazard* contributes to a great deal of confusion when compiling a list of known or foreseeable hazards. This confusion arises, in part, because the definition is ambiguous: almost every state or event within the device system, given certain conditions, could be construed as a potential source of harm. The distinction, then, between a hazard and the events resulting in a hazard can become very subjective. For example, if the device has a component insulation failure, resulting in an electrical short, resulting in an electrical shock, resulting in cardiac arrhythmia, resulting in fibrillation or death, which of these events is the potential source of *harm* (i.e., the *hazard*), and which is the cause of the hazard? Similarly, if a device software component performs an incorrect calculation, resulting in an incorrect electrical stimulation, resulting in cardiac arrhythmia, which event is the potential source of harm, and which is the cause of the hazard? The ambiguity of the definition results in hazard lists that often vary widely between manufacturers of similar devices.

What constitutes a hazard depends on how the designers abstract the boundaries of the device or system, as shown in Figure 1. It can also depend on the intended use of the device, and even how the designers define a *hazard*. It would be useful to regulators, auditors, and manufacturers if these boundaries were unambiguously standardized. ANSI/AAMI/ISO 14971:2000 begins to address this issue by providing lists of clinical hazards and classes of hazardous conditions in its annexes.

---

[2] A random failure is associated with something that works but over time fails because of lifetime issues. Software does not wear out. It fails under the same conditions whenever they occur.

The diagram of Figure 1 provides a model of the medical device *hazard–cause* continuum. The device may affect the patient, user, and service domains by means of direct or indirect (environmental) interfaces in such a manner as to cause harm. Clinical and device-level hazards listed in the diagram generally map to the ANSI/AAMI/ISO 14971:2000 annexes and may serve as a starting point for establishing hazards.

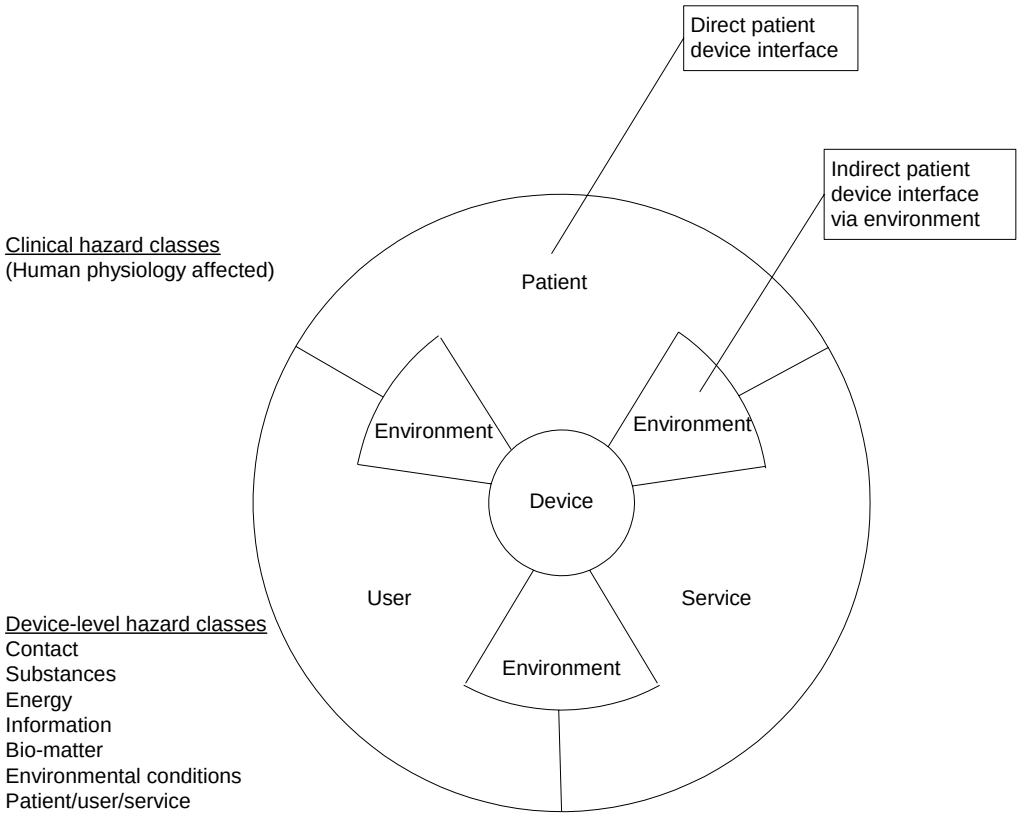

Figure 1—Hazard cause continuum

In this report, an attempt is made to use the terms in a less ambiguous manner. It is believed that these refinements are consistent with the spirit of ISO/IEC definitions and will help resolve some of the confusion experienced by medical device developers.

The term *hazard*, therefore, is used to refer to any broadly characterized means, mode, or manner by which a medical device (or, more generally, a system being analyzed) might cause harm. The classification of hazards is highly subjective and purely pragmatic. The fundamental test for classifying hazards is whether the proposed classification scheme provides insights that are useful for analyzing risk. For example, many hazards involve exposure to harmful amounts of energy, provision of incorrect diagnostic information, or provision of incorrect therapy.

A *hazardous event* is any occurrence of a hazard. Whether harm results from a given hazardous event depends on the degree to which hazardous preconditions are present, the precise timing of contributory events, and the effectiveness of any risk control measures designed into the system.

A cause of a hazard is any set of events or circumstances, the combination of which might reasonably be expected to result in a hazardous event. A given hazard might have one, several, or many possible causes (*contributing factors*). Additionally, hazards might be caused by logical combinations of precursor causes or, indirectly, by random failures.

If we apply these definitions to the example cited earlier, electrical shock is most properly identified as the hazard. The insulation failure, together with the circumstance (e.g., shock, vibration, or handling) that caused the exposed conductors to become shorted together, constitutes a possible cause of the hazard. One might also identify other entirely different causes of electrical shock (misassembled components, software algorithm error, operator error, etc.).

The medical conditions of cardiac arrhythmia and fibrillation are possible consequences (i.e., harms) of a hazardous event involving electrical shock, and death is one possible outcome.

This report uses the term *risk control measure* for the actions taken to prevent or minimize the probability of a hazardous situation from occurring, to reduce its severity if it does occur, or to do both. This term can be used to cover what some engineers and regulators refer to as "mitigations" or "safeguards." This term is chosen both for consistency with ANSI/AAMI/ISO 14971:2000 and because of its broader focus on anything that reduces either the probability or the severity of a hazard.

4 Perspective 1: Basic concepts of medical device software risk management

ANSI/AAMI/ISO 14971:2000, *Medical devices—Application of risk management to medical devices*, recognized worldwide by regulators, is widely acknowledged as the principal standard to use when performing medical device risk management. ANSI/AAMI SW68:2001, *Medical device software—Software life cycle processes*, also recognized by the U.S. Food and Drug Administration (FDA), makes a normative reference to ANSI/AAMI/ISO 14971:2000 requiring its use. The organization and content of these two standards provides the foundation for this TIR.

ANSI/AAMI/ISO 14971:2000 provides a framework and taxonomy for *medical device* risk management but does not provide details or explanations of its requirements in the context of software development. ANSI/AAMI SW68:2001 focuses on software development processes with specific references to risk management activities within each development process. However, ANSI/AAMI SW68:2001 does not provide detailed software risk management methods or fully explain how software risk management integrates into overall *medical device* risk management. This report attempts to address these details.

This section attempts to identify and explain key concepts relevant to reducing the risk of medical device software failures leading to unacceptable health risks.

4.1 Medical device risk management

The successful development of a safe and effective medical device requires the integration of a diverse set of knowledge and experience of personnel from a variety of disciplines such as clinicians, human factors engineers, biomedical engineers, electrical engineers, mechanical engineers, and software engineers. Experts from each domain will look at risk management with a different perspective, which is based on their particular experience. The goal of medical device risk management is to integrate each of these perspectives in such a manner that the device developed is as safe and effective as can be reasonably expected. Key to the success of this effort is a clear understanding, by all involved, of the device's intended use. With this understanding, the domain experts can begin to reason together about how each can best contribute to the overall safety and effectiveness of the device. One hopes it is self-evident that if these experts do not communicate effectively, the likelihood of the device performing as intended will be diminished.

By establishing the device's intended use, designers can begin to work on functional system requirements and their associated system boundaries, such as the device interface (human factors), hardware, software, and "networked" accessories. As these system boundaries begin to take shape, a basis is established for risk management processes. With this understanding, decisions can be made regarding how best to implement a risk control measure (e.g., hardware, software, information, or some combination of these).

4.1.1 Software risk cannot be managed effectively in isolation

Software can be a part of a medical device, an accessory to a medical device, or a medical device itself. Adequate medical device software risk management cannot be performed effectively by analyzing software alone, even in the case in which stand-alone software is considered a medical device. This type of device software runs on one or more computer and operating system platforms. The potential failure modes of these platforms, their configuration management, and the user environment must also be considered in order to perform adequate risk management.

On the one hand, software risk management cannot be performed in isolation from overall medical device risk management, while on the other hand, some special focus and knowledge are needed to adequately accomplish the software aspects of risk management.

Closer study of the purposes of software risk management activities reveals that the *system* is mentioned as often as the *software*. One cannot identify device *harms* by examining only software. Similarly, *hazards* may result from software events (causes) as well as hardware events, user events, or other nonsoftware events. Similarly, the protective measures—or risk control measures—may be designed into the software, or they may be designed into the hardware, documentation, or other nonsoftware element of the system.

Since this report focuses on software, it may seem as though software risk management can be a stand-alone process. To isolate software risk management from *medical device* (system) risk management would result in an incomplete device analysis, with a myopic view of the system, and a likelihood of lost safety requirements. The

software risk management activities are simply that part of system-level risk management that is related to software causes of—and software risk control measures for—medical device hazards.

4.1.2 Software input is an important part of device risk management

Although this report focuses on software risk management, there are many opportunities for software designers to contribute to the overall safety of the medical device during the early stages of device design. One should not wait to consider software until after key decisions about the design of medical devices are made. Doing so could result in missed opportunities for device-level risk control measures or inadequate understanding of risk during critical early requirement and design decisions.

By participating in the medical device design process, the software engineer can contribute to safety-related decisions concerning software risks and risk control measures as the design evolves. Additionally, the software engineer can encourage a safety-related decision involving the hardware–software partitioning and other aspects of the device design, functionality, and intended-use environment and computer platform.

Examples of software-related questions one could ask at the medical device design level include:

— Is a particular hazard best controlled using redundant hardware as opposed to software checks?

— What is the safe or safest state for this device?

— What are the special considerations for the power-up and power-down of this device? What is the hardware designer expecting the software to do during these times, and what power-up self tests are expected of the software?

— If nonvolatile storage is being used, what hardware interfaces are present, or should be present, to allow the software to properly manage nonvolatile storage when the device is powered down and powered up?

4.2 Software risk management

From a process perspective, software risk management for medical devices can be seen as the application of ANSI/AAMI/ISO 14971:2000 risk management requirements to the development of software. Software can be a part of a medical device, an accessory to a medical device, or a medical device itself.

One can assume that software cannot directly cause harm (it causes harm through hardware or provision of misleading information to users). Therefore, one should identify hazards (at the medical device level) and then look for potential software failures that could cause the hazards in isolation or through a chain of events or combination of conditions.

Where the software is itself a medical device intended to be distributed for use on commercial computer and operating systems, it is sometimes unclear where to draw the boundaries for software risk management. The fact that the hardware and operating system are not packaged with the software is no reason for the medical device designer to ignore hazards that could result from faulty platforms or poor administrative control and maintenance of these platforms by the users (e.g., security, virus checking, configuration management).

An important part of the software design and risk management process is that potential causes for hazards be considered in the intended-use environment. For instance, one may restrict the platforms to those that meet certain specifications included in the medical device labeling. Additionally, one might build checks into the software for acceptable platforms and for detection of platform failures.

While software aspects of risk management cannot be effectively performed in isolation from overall medical device risk management,[3] there are activities that may be best performed by software engineers as an integral part of the software life cycle. There are also elements of software risk management[4] that require more focus and different handling than that provided in ANSI/AAMI/ISO 14971:2000 for medical device risk management overall. We refer to this aspect of medical device risk management simply as *medical device software risk management*. It is important to stress that even the software aspects of risk management need to focus on the risk of harm—not just risk of software failure—in order to be effective.

---

[3] Because of the interdependence of hardware failures, software failures, and hardware and software risk control measures.

[4] For instance, all software failures are systematic, not random (as many types of hardware failures or breakdowns are), and their probability cannot be accurately estimated. Therefore, the way in which the probability component of risk is applied to software is quite different.

When one is discussing the potential for software to cause a hazardous situation, it is useful to develop a framework to support the discussion. Figure 2 illustrates such a framework in simple terms. First, we consider a system-level view of the software. From this view, we can see that hazards could be caused by software. Likewise, hazards could potentially be prevented, corrected, or mitigated by software.

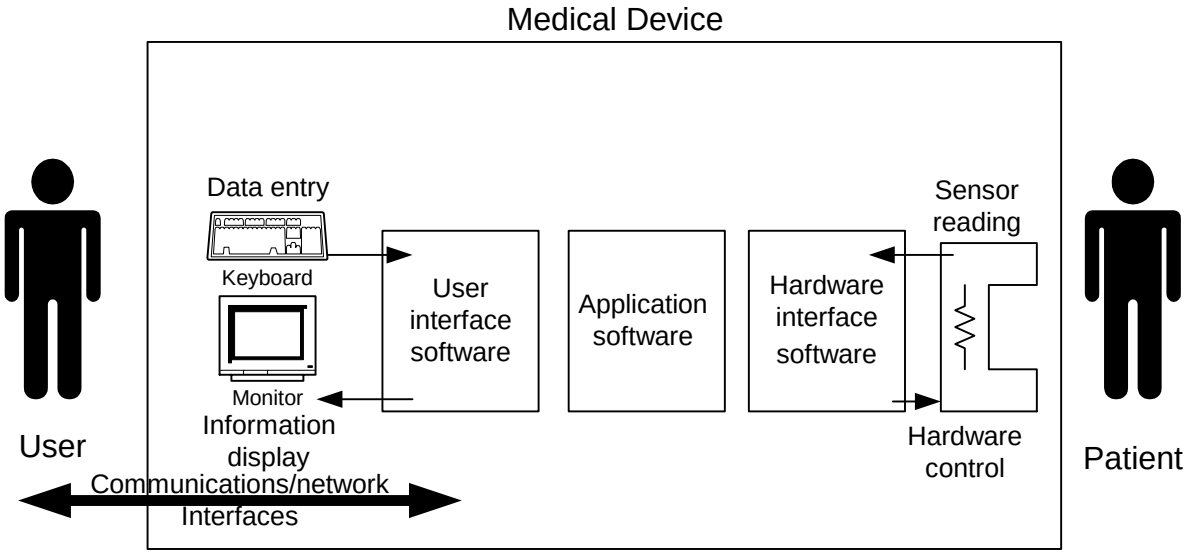

Figure 2—Software risk management context diagram

For example, in a patient-monitoring device that provides clinicians with vital signs data, the software responsible for displaying the vital signs data on the instrument could have a defect that causes the wrong value to be displayed. This defect creates a hazardous situation. If we assume the operator acts on this information, patient harm could occur either through improper treatment or delay in treatment. In this example, software is the cause of the hazardous situation. As another example, consider that the pressure in the cuff used for noninvasive blood pressure measurements could be monitored with a pressure transducer by software. If the software detects a pressure greater than a predetermined safety threshold, the software could take some protective action, thus mitigating the hazard of failed hardware that could result in uncontrolled pressurization of the cuff. In this example, the software is a protective measure for the system.

From a system-level view, we understand that software, in and of itself, cannot cause harm. However, that software, when controlling a hardware device in contact with the patient (depicted on the right side of Figure 2), or when reporting diagnostic information or treatment information (depicted on the left side of the figure), through a communication interface to another device or system (not shown), or through some environmental condition (not shown), could contribute to a hazardous situation if no other risk control measures are in place. Software may be used to implement risk control measures for hardware failures, so defects in this protective software can also affect safety.

Additionally, we need to identify software defects that could cause a hazard (or be in a causal chain that could result in a hazardous situation).

There are several distinctions in types of software functionality that are useful in software risk management:

— software intended to perform functions to implement clinical requirements for the intended use of the device (e.g., software calculating a radiation dosage);[5]

— software intended to implement a risk control measure; and

— other software.

Figure 3 depicts how these and several other distinctions should be used to identify software that is safety related.

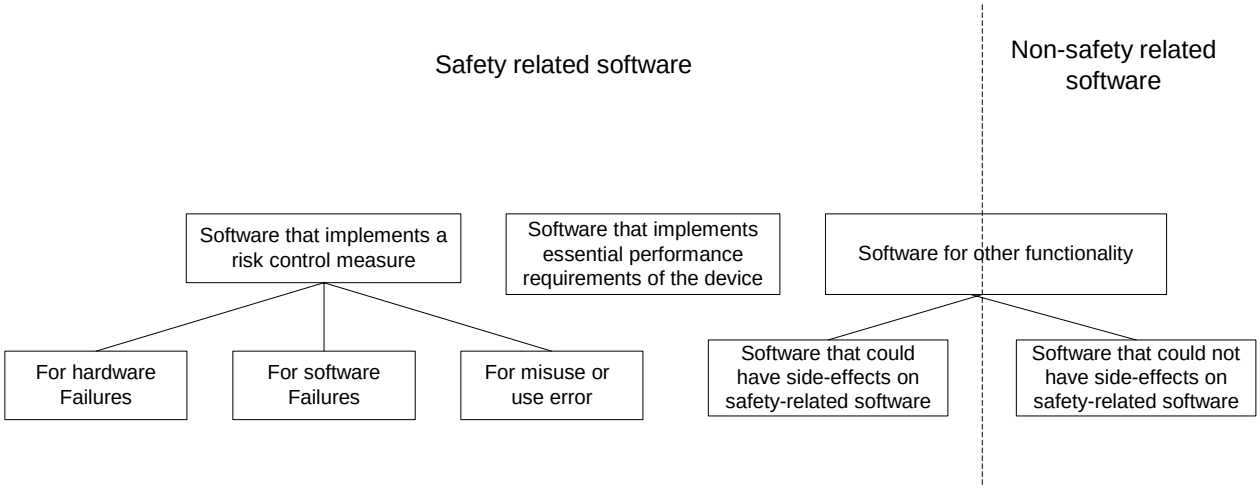

Figure 3—Types of functionality

Failure modes of all types of software components related to safety and effectiveness should be analyzed to determine appropriate risk control measures. If we look strictly from a functional perspective, it is easy to miss portions of the software that are safety related since some software is safety-related only from the perspective of side effects or defects that could inadvertently affect other safety-related software. This type of functionality is indicated on the right side of Figure 3.[6]

An identification of safety-related software should be done from both logical (i.e., functional) and physical (i.e., component) perspectives. From a functional or logical perspective, some software requirements may be directly related to safety and effectiveness, and others may not. From a physical or component perspective, some software components may be directly related to safety in terms of their purpose, others may be indirectly related in terms of potential predictable or unpredictable side effects, and still others may be unrelated and unable to affect safety if isolated by system design (e.g., running on a separate central processing unit (CPU) with a totally independent and protected memory and disk space). The results of this analysis allow identification and tracking of safety-related code or modules or components as well as detailed analysis of possible defects that could result in hazards.

From a physical perspective, it can be useful to assign risk or hazard severity ratings to software items or modules to distinguish highly critical components (e.g., those that could result in death or serious injury if they are defective) from components that could not affect safety. Doing so can serve as a basis for greater rigor and focus in verification, risk control, and configuration management activities for more critical components. If this is done, side effects need to be carefully considered, and the less critical components must be rated the same as any more critical components they could affect.

Off-the-shelf software used in a medical device could fall into one or more of the boxes in Figure 3. When it does, it must be handled accordingly without the presumption that such software is outside the scope of risk management or that risk control measures are not possible for such software.

---

[5] This is sometimes referred to in IEC 60513:1994 as *essential performance*.

[6] If there is no medical device-level isolation of software components, any defective component could affect other safety-related components and must itself be considered safety related.

We must note that software components could be identified initially as safety related. Subsequently, through certain risk control measures or design choices, the residual risk could be reduced to a level that the same software components could be treated as less critical or, in some cases, even as non-safety related (if the risk control measure entirely eliminates its potential hazard). Properly performed risk management results in reducing safety-related software to the smallest possible subset through isolation and inherently safe design.

A key aspect of software risk analysis is identifying potential causes of hazards. In this context, cause refers to software errors of commission or omission. This task may not be a simple one, because for each identified hazard there can be a single cause or multiple causes. Causes may exist singly or in multiple causal chains. These causal chains can be coupled or uncoupled or the result of a temporal condition. A simple causal chain example might consist of a buffer pointer overflow, which corrupts data in the next memory location, which results in a reasonable data value, but not for the current patient displayed on a screen, and which is acted on by the user. The concept of causal chains can be useful in determining effective and efficient risk control measures, as discussed in the next section and elsewhere in this report. Figure 4 illustrates this concept.[7]

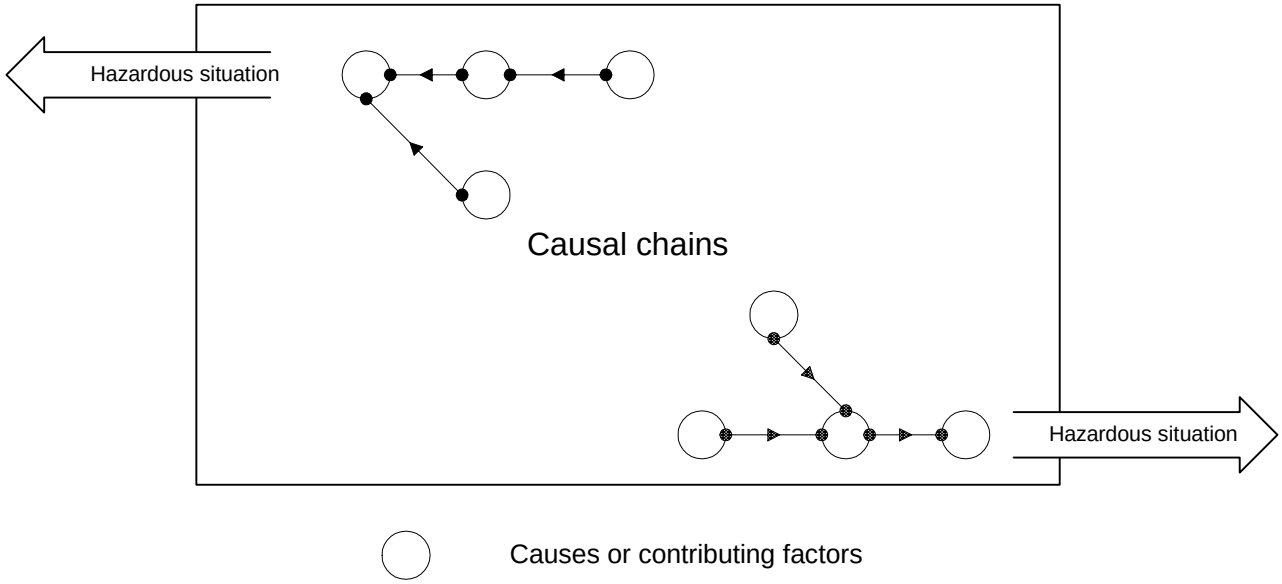

Figure 4—Causal chains

Direct versus indirect causes

When identifying potential software causes of hazards, one shall focus attention on software directly related to the clinical functionality of the device (e.g., an algorithm that calculates glucose levels in blood). Such software, if implemented incorrectly, can directly cause a device hazard. Examples of such “direct causes” include incorrectly implemented algorithms, incorrectly presented information, errors in information obtained from sensors, errors in information sent to actuators, and others. It is equally important to consider software causes that could indirectly result in device hazards. Examples of such “indirect causes” include uninitialized pointers, memory corruption, stack overflow, race conditions, and others. Although indirect causes for hazards may generally be unpredictable, appropriate attempts should be made to detect their occurrence and control any resulting hazards.

Annex A provides examples of specific causes based on categorization of typical medical device software functionality and relevant risk control measures that could be considered. Annex B provides examples of indirect causes for hazards attributable to unpredictable side effects and relevant risk control measures that could be considered. Neither of the tables in the annexes is exhaustive, and they should not be used as an exhaustive checklist. The tables are intended to trigger discussion and to serve both as an aid to identifying relevant causes and as a basis for challenging the thoroughness of the results of existing risk management processes.

[7] The intersection at a common node could be an “AND” or “OR” condition, and as many possible sequences and necessary and sufficient conditions could apply.

4.3 Risk control

The purposes of medical device software risk management are to

— avoid, prevent, or mitigate the conditions under which software use, misuse, or defects could lead to a hazardous situation;

— reduce the severity of the hazard if such an event should occur; and

— minimize the probability of a hazardous situation caused by misuse or defects.

To accomplish this, software-specific aspects of risk management should address the following:[8]

— The analysis of the clinical or other possible intended use[9] of the device to identify clinical hazards and related software functionality that provides clinical information or control commands related to safety and effectiveness.

— The identification of potential software causes (including causes that are part of a chain of events attributable to hardware failures or user errors) that could contribute to a hazardous situation. This analysis should include failures in software functionality directly related to safety and side effects from other software, hardware, or user errors.[10]

— The identification of software components that implement or could affect functionality related to safety or to the implementation of risk control measures—sometimes referred to as *safety-critical* or *safety-related* components.

— The identification of hardware, software, and labeling risk control measures to prevent software errors from resulting in a hazardous situation or to reduce the probability of such errors.

— An evaluation of residual risk after risk control measures are identified and determination of whether the risk is acceptable given the intended use and target population of the device.

— The verification of risk control measures and safety-related software components.

Information on the medical device evolves gradually as the risk management process is applied incrementally and iteratively over the development life cycle. The earlier in the development life cycle potential causes of hazards can be identified, the more likely inherently safe design choices can be made. Causes of hazards discovered late in the development life cycle are not likely to result in the most effective or efficient risk control measures. Section 6 of this report discusses risk management activities in relation to the software life cycle and provides examples of activities of risk management to consider at each stage of development.

Early in a project life cycle, from an analysis standpoint, it is often useful to consider the first and last points of potential risk control for each hazard. These will be referred to as the “first point of software control” (FPOSC) and “last point of software control” (LPOSC). In Figure 5, the first and last points of control are at the points of interface to the hardware on the left and right sides of the box.

---

[8] Note that these areas are relevant both during initial design and development and during maintenance as each change is considered and implemented.

[9] A device may be intended for a specific use for a specific target population, but if it could easily be used beyond this intended use, then relevant risks and risk control measures should be considered.

[10] For instance, an uninitialized pointer in a module that results in overwriting of critical data or hardware memory failures that corrupt code or data.

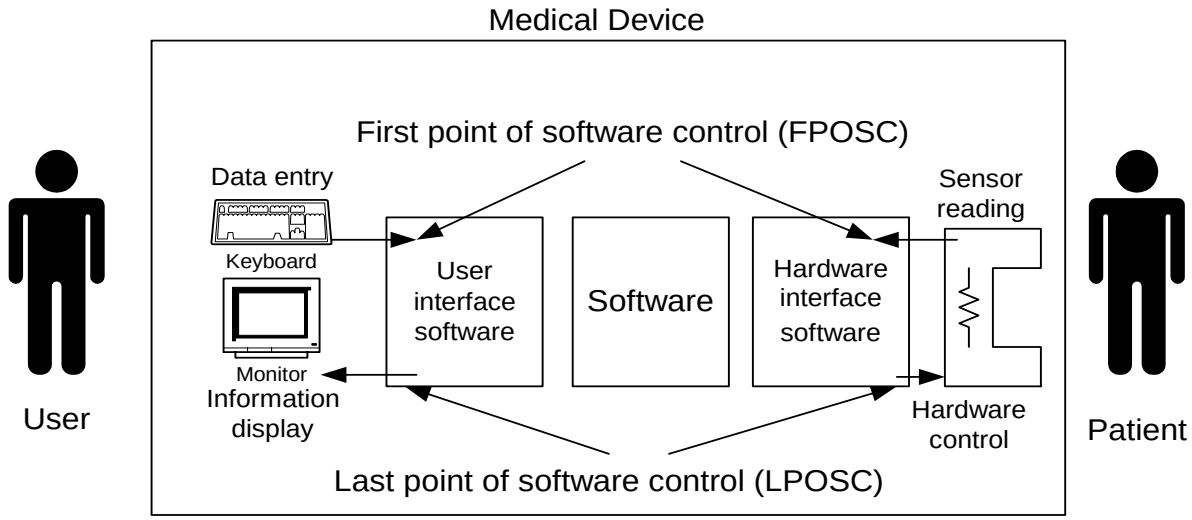

Figure 5—First and last points of control context diagram

A determination of the LPOSC versus the FPOSC in causal chains is highly subjective and purely pragmatic.[11] However, considering the device design in this context may provide insights that are useful for analyzing risk and identifying the best risk control measures.

A more detailed view of the concept of causal chains of events and first and last points of software control is provided in Figure 6. In this figure, each of the bubbles represents an error that is a potential cause. The chains indicate that multiple chains could create a hazardous situation. Generally, several complementary hazard identification methods are needed to help identify nonobvious causes and causal chains. If there is an adequate risk control measure at the end of the causal chain (LPOSC) that prevents the hazardous situation, then the individual links in the chain might not require separate risk control. For example, several links in the causal chain might represent erroneous states resulting from bad data elsewhere in the system. If data integrity can be ensured (FPOSC), then there may be no practical reason to put guard conditions around each possible intermediate error state.

[11] For instance, one might define an LPOSC for an infusion pump as the software that sends the commands to the pump motor, and an FPOSC might be at the user interface for entry of the infusion rate.

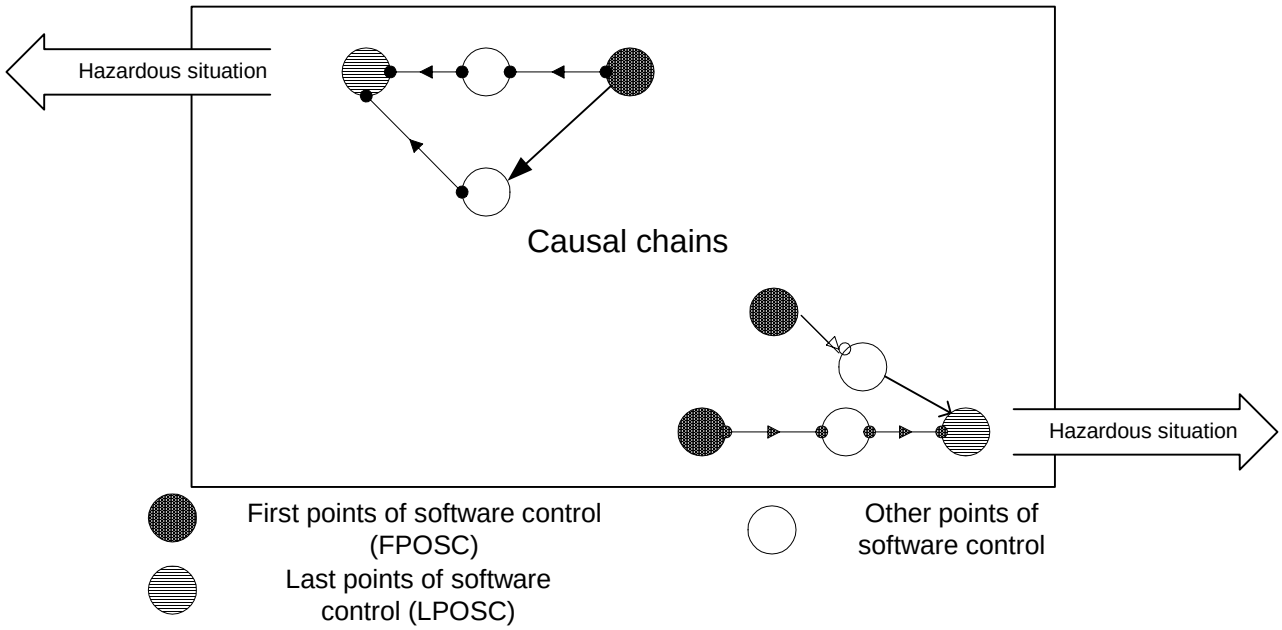

Figure 6—First and last points of software control in causal chains

The chains in the lower right corner indicate another one of many possible variations in causal chains leading to a different hazardous situation. In this example, several erroneous independent causal chains could result in a fault state. Here, an LPOSC could be effective, whereas several FPOSCs would be needed otherwise.

If one considers causal chains, the FPOSC, and the LPOSC, some benefit may be derived in determining the most effective and least complex and expensive risk control measures. In many cases, cost-effective risk control measures can be implemented to break the hazardous causal chain at the beginning or end of the chain. In other cases, the risk control measures merely reduce the probability or severity of the occurrence of an erroneous state. Therefore, multiple risk control measures at multiple points in each causal chain are often necessary.

Risk control measures implemented at either the LPOSC or outside the software through hardware or labeling are often important in minimizing risk, especially if some software defect or failure mode that could lead to hazards through a sequence of events has not been identified and adequately mitigated.

In cases in which very effective risk control measures can be defined at the last points of control, it may be easier to defend the safety and effectiveness of the software than to demonstrate that every possible cause or causal chain has been identified and mitigated. Such risk control measures may also be more cost effective than implementing specific risk control measures for each possible contributing factor. Additionally, risk control measures at the last points of control may be the best way to address indirect causes that have unpredictable effects and to deal with situations in which full analysis of all possible sequences cannot be feasibly analyzed.

Consider a medical device that controls fluid pressure. If the pressure exceeds a certain limit, injury could occur. The software implements a complex control algorithm, including a variety of error checks and cross-checks for possible software or hardware faults that could result in exceeding the limit. Additionally, separate software (possibly on a separate CPU card or memory space) monitors a pressure sensor at the point closest to the patient. If the pressure reaches the limit, this software shuts down the pump (assuming shutting down the pump did not represent a hazardous situation). Such a measure would limit the risk of injury at the LPOSC, whether or not all possible causes and chains of events had been mitigated.

Although risk control measures implemented at the last point of software or system control can be very effective, they are often not foolproof. Generally, it is best to implement risk control measures for one or more of the contributing factors in a causal chain as well as at the LPOSC. These earlier measures reduce risk should the LPOSC measures not be foolproof, whereas the LPOSC measures serve as a safety net to reduce risk should some potential causes for hazards be missed or if existing risk control measures fail.

In addition to the LPOSC, attention to the FPOSC is important. Implementing risk control measures at the FPOSC can limit the number of combinations of potential causes resulting in hazards. The FPOSC concept is especially relevant to the prevention of erroneous information that could subsequently lead to a hazard. For example, an algorithm may be correct and may work properly in determining a test result. If, however, various raw data acquired

through sensors are erroneous, or if the user inputs parameters or options improperly, the result could still be wrong. Examples of FPOSC risk control measures include consistency, context, fault, or range checks on sensor information or user interface error checks on information entered by operators.[12] In some situations, LPOSC risk control measures are more effective, and in others, FPOSC measures are more effective. In many cases, combinations of LPOSC, FPOSC, and intermediate risk control measures are needed. Suitable risk control measures depend on the device's intended use, available technology, and software design constraints.

While the concept of first and last points of control can be useful, risk analysis and control at those points alone are usually not adequate. Even if risk control measures at those points seem adequate, other risk control measures may be worthwhile to further reduce the risk or as a "belt and suspenders" approach in case the evaluation of adequacy is not foolproof.

When designing risk control measures, one must recognize that they can add complexity. That is, more risk control functionality could result in more defects. When establishing risk control measures, one should carefully weigh the consequences of additional design complexity.

4.3.1 Risk control through development assurance levels

Many times during the risk management process, one will face tradeoffs between adding more design risk control measures and relying on process confidence measures for the software in question. Put simply, if it can be established that there is a high level of confidence in the ability of the software to perform its intended function reliably, and if the severity of the relevant hazards is not high, then more focus can be put on other sections of software for which the confidence level is not so high or for which the potential hazards are greatest.

One such method for establishing confidence is what is sometimes referred to as the development assurance levels (DAL) concept. A DAL could be considered a measure of the rigor of the development process used to develop the software or some other characteristic that would indicate higher reliability. Presumably, software components with a higher DAL are more reliable than components with a lower DAL.

This concept could be used for making requirements, as well as architecture and design decisions, regarding risk control measures. For a given project or analysis, more critical components implementing essential performance requirements or risk control measures might require higher DALs. Conversely, lower risk software components might have reduced development and verification process requirements. However, in the case of a commercial off-the-shelf software (COTS) component, or software of unknown pedigree (SOUP) from prior projects or free public code libraries, one may have no knowledge of the process used in its development or verification. When such software is used for a highly critical device, additional design risk control measures (e.g., in the custom code that calls the COTS component) for unexpected failure conditions may be needed to achieve a satisfactory postmitigation risk level.

4.3.2 Achieving development assurance level requirements

If the method for risk reduction using DAL requirements is used,[13] a DAL scheme appropriate to the software being developed must be defined. There are many examples of DAL schemes available to use as guides during development, but rarely would they be applicable as written. Schemes can be created based on the developer's current quality assurance practices and risk management techniques.

How DAL levels are defined will vary greatly depending on the nature of the system. However, any effective scheme will require classifying steps of rigor in the essential areas of the software quality process, including verification and validation.

Consider two extreme examples:

1) Software application with 32 kilobytes of embedded custom code; no COTS operating system; software quality assurance measures throughout the life cycle (requirements, design and code reviews, multiple levels of formal testing, etc.); and highly stable validated compiler and libraries. This application could be a critical implantable device.

2) Complex software system or device with multiple interfaces (including user interfaces), off-the-shelf operating system, third-party libraries, and legacy code integrated. It is impossible to determine the complete life cycle control of all components of this system.

Clearly, no general set of rules is going to apply in both of these scenarios.

---

[12] This does not intend to assert that all possible operator errors could be detected.

[13] DEF STAN 00-55, 00-56:1997 presents the most detailed scheme for determining DAL by qualitative methods.

Other considerations when developing a DAL scheme for your organization might include:

— Staff competency—skills, qualifications, experience, and training. (Who develops the software?)

— Methods—suitability of the specification, design, coding, and testing methods. (What is the process of development?)

— Rigor, formality, and scope of reviews and inspections. (How much static analysis is performed?)

— Tools—quality of tools such as compilers and configuration management tools. (What tools are used during development of software?)

4.4 Integration of risk management in the software life cycle

Effective software risk analysis and risk management cannot be accomplished in any single meeting or activity. Risk cannot be effectively minimized at the end of the product development cycle by retroactively preparing a software hazard analysis. For software risk management to be implemented properly, a focus on hazard identification, risk evaluation, and risk control must be integrated into each phase and relevant activity of the software development life cycle. Additionally, risk management activities are assumed to be incremental and iterative as new risk-related information becomes available (regardless of the type of software life cycle model used).

To ensure that software risk management is adequately considered in the risk management process, one should establish plans or procedures to address aspects specific to evaluating and controlling software risks. Such plans should include defining how software will be developed (e.g., coding conventions and analysis tools such as syntax checkers and code analyzers) and how the risk management process will be addressed in the software development life cycle model to be used (recognizing that the activities and approaches are somewhat different at each stage of development). For example, early in the software management analysis process, analysis cannot be as complete as it can be later in the life cycle when more information is available. This situation is also discussed in section 6 in the context of life cycle planning and process implementation required by ANSI/AAMI SW68:2001.

An iterative risk management process requires each activity and phase of the software life cycle to include risk analysis, risk evaluation, and risk control. More specifically, it requires initial identification or subsequent refinement of the following:

— Hazards

— As many causes for the hazards as possible, given the level of information available within the development activity

— Risk control measures that should be implemented for potential software failures

— Estimation of risk and residual risk after risk control

— The acceptability of residual risks of software-related hazards with the risk control measures in place

— Repetition of the previous steps until residual risk is acceptable

As the project progresses, more information becomes available and existing information evolves (i.e., requirements documents, architecture documents, design documents, and the software itself), and the basic steps of the risk management process are repeated, making use of the new details of the product's design and implementation.

It is also important to recognize that, as risk control measures are designed and implemented, the severity of a hazard of a single cause or causal chain may decrease. The need for further analysis and risk control for these specific causes may be reduced or eliminated. For example, early in a project one might identify that an incorrect calculation might result in issuing commands to move a motor too fast, which could result in crushing a body part. A risk control measure might have been implemented at the device level in hardware through motors that are self-limiting to an inherently safe range, thus preventing any harm regardless of software commands. Thus, there is a difference in the pre- and post-risk control rating. Although the software still needs to issue proper commands for the device to be effective, further risk control can be commensurate with the associated residual risk.

Traceability is not unique to software risk management activities, but it is essential for the engineer to verify that all medical device risks trace to some method of risk control measure and that these measures have been implemented and verified. As the product definition, design, implementation, and test designs evolve, the trace is checked and updated to ensure that risk control measures are not inadvertently missed.

Some form of traceability is required as a key aspect of the process to ensure that

— all requirements related to safety and effectiveness, and all associated software and nonsoftware risk control measures are implemented;

— all software components that can affect safety or effectiveness of the device directly or indirectly have been identified;

— all requirements, risk control measures, and software components related to safety and effectiveness directly or indirectly under a range of operating conditions have been verified; and

— maintenance changes can be effectively analyzed for risk.

At the end of the software development process, traceability provides an ability to demonstrate that

— relevant hazards have been identified,

— software requirements essential to clinical performance have been identified,

— causes for hazards have been identified,

— risk control measures have been implemented for the causes and hazards identified,

— software components that can affect safety and effectiveness have been verified under a range of conditions on recent versions of the software and hardware,

— risk control measures have been verified under a range of conditions and on recent versions, and

— maintenance changes have been adequately analyzed for their effect on safety and effectiveness.

4.4.1 Risk management is also essential for software maintenance

Software changes can affect existing risk control measures, introduce new causes for hazards, or even introduce new clinical hazards. It is essential that an effective risk management process adequately address maintenance activities as well as changes during initial development. This concept is discussed in more detail in other sections of this report, but it cannot be overemphasized. There are usually many maintenance releases of software for each medical device, and often the personnel performing the maintenance are not involved in the original development and risk management work.

Hazard analysis for software changes applies at a macro level during the maintenance activity to assess the risk impact of proposed changes to the software for a new release of the device software. The same methodology applies to all activities within the software development process. As product requirements, design details, or implementation evolve during development, each activity must assess (through risk analysis) whether the proposed changes have

— created new hazards,

— introduced new causes for existing hazards,

— subverted previously designed risk control measures,

— introduced new failure modes for which new risk control measures might be needed, or

— introduced new defects that could affect safety-related software.

If the changes have introduced new risks, then new control measures must be designed into the product to reduce the risk to an acceptable level.

Section 6 of this report discusses some of the software risk management aspects of each primary life cycle process defined in ANSI/AAMI SW68:2001.

4.5 Common confusion regarding software risk management

Ensuring software safety requires a variety of activities throughout the product development life cycle. Reliability techniques such as formal methods for failure analysis are not complete risk analysis methods. It is also important to recognize that reliability and safety, although often related, are not identical attributes. A life cycle process that focuses on reliability may not achieve maximum safety.

Software risk analysis is often confused with applying a single failure analysis technique such as fault tree analysis (FTA) or failure mode and effects analysis (FMEA) at a single phase of the software development life cycle. Although those techniques can play a role in software risk management, that role is only one of the requirements of a thorough

software risk management process for a medical device. Taken individually, these techniques are generally not adequate to identify potential hazards, software causes for hazards, and optimal risk control measures.

Performing risk analysis at only one stage of the medical device and software development life cycles does not ensure acceptable risk that is as low as reasonably practicable. A variety of activities and methods are needed at each stage. Adequate risk analysis requires an incremental and iterative risk management process with varying techniques used at various stages. Different techniques apply to this process at different life cycle activities. For example, a software FMEA would not be possible until some knowledge of the design was available, whereas hazard identification could be performed on the basis of the intended use of the device much earlier in the process, and some architectural risk control measures could be identified by performing a preliminary system-level failure analysis before actual software design.

What about firmware, microcode, and programmable logic?

In complex electronics devices, one now encounters hardware with large logical state machines, highly embedded microcode, and many complex configuration modes based on software programming. Many failures of these classes of hardware have the same systematic error characteristics as software faults. Additionally, the random faults that can propagate to system errors are often so complex (embedded inside highly integrated devices, for example) that they cannot be easily identified or their probability quantified.

The techniques and methodologies discussed in this report can be applied to these devices. If the logic contained can be reduced to a finite state machine with a small number of defined outputs for all inputs, then exhaustive testing may be possible. Otherwise, the function must be approached in a similar manner as one would approach a software system or subsystem. In addition, there is usually no guarantee that the outputs of a programmable logic device will be defined for an internal failure of the device. Therefore, the premitigation hazard analysis should address the possibility of the outputs being in any state.

Careful, methodical analysis of these devices should be conducted either as part of the software risk analysis or part of an FMEA or FTA for the hardware subsystem.

5 Perspective 2: Software considerations in medical device risk management

This section provides information on the medical device risk management process defined in ANSI/AAMI/ISO 14971:2000 as it relates to software. The intent is to emphasize and clarify the software considerations for each aspect of the general risk management process of ANSI/AAMI/ISO 14971:2000 without implying that software risk management can be done effectively in isolation from overall medical device risk management.

This section of this report is organized using the major elements of the risk management process identified in Figure 1 of ANSI/AAMI/ISO 14971:2000. These are:

— Risk analysis

— Risk evaluation

— Risk control

— Postproduction information

An additional section addresses the Risk Management Report referred to in ANSI/AAMI/ISO 14971:2000.

Each subsection below

— summarizes the intent of each ANSI/AAMI/ISO 14971:2000 element,

— discusses the intent from a software perspective, and

— identifies some potential pitfalls from a software perspective.

Subsections discuss issues of particular importance in software risk management. Short, useful examples are provided to further explain the approach and activities involved.

5.1 Risk analysis

ANSI/AAMI/ISO 14971:2000 defines *risk analysis* as the “systematic use of available information to identify hazards and to estimate the risk.” Sometimes, the terms *hazard analysis* and *risk analysis* are used interchangeably, but an important distinction exists. Hazard analysis only identifies hazards and the foreseeable sequence of events and/or circumstances resulting in the hazard (causes). Risk analysis adds a level of harm (severity), and a likelihood of

occurrence estimation process (which must also take into account the identification and assessment of environmental conditions along with duration of exposure). Thus, hazard analysis is a subset of risk analysis.

14971 requirement

The risk management process shall include risk analysis as an element of the process.

Software perspective

As described in ANSI/AAMI/ISO 14971:2000, *risk analysis* is a term used to encompass three distinct activities: identifying the intended use, identifying the known or foreseeable hazards (and their causes), and estimating the risk of each hazard. Subsequent subsections in this report address each of these activities in a software context. One must recognize that for risk analysis to be effective, it must be performed as an integral part of the entire software development process—not as one or two discrete events. This is because hazard and failure mode information accrue over the entire software development life cycle process and need to be considered at each stage of design.

Because it is not possible to quantitatively estimate the probability of software defects that could contribute to hazardous situations, and because software does not fail randomly in use as a result of wear and tear, the focus of software aspects of risk analysis is on identification of potential software functionality and defects that could result in hazardous situations—not on estimating probability.

Pitfalls

— The application of unrealistic low probability estimates to software failures results in unrealistic risk ratings leading to inappropriate risk control measures.

— Software features are added without performing risk analysis to determine if new hazards or causes have also been added to the device or if existing risk control measures have been compromised (either during initial development or after release as part of maintenance).

— The medical device risk analysis process defines only system- and hardware-level aspects, and neither adequately addresses the relationship of software to adequate risk analysis nor requires specific consideration of software defects as potential causes for hazards.

— The rigor of the risk analysis and software development life cycle procedures is not commensurate with the potential harm of the medical device.

5.1.1 Risk analysis procedure

14971 requirement

The approach to, and results of, the risk analysis shall be documented.

Software perspective

Plans or procedures should address aspects specific to evaluating and controlling software risks in order to ensure that software is adequately considered in the risk analysis. This task should include defining how the risk management process will be addressed in the software development life cycle. The severity of the worst-case hazard is a primary input in determining the level of rigor of the risk management and software development processes. The information provided in the following subsections is intended to help identify software-specific aspects of an effective risk management process. Additionally, software aspects of the risk analysis should be identifiable in the resulting documentation and should include both software used to implement risk control measures for hardware failures and software causes for hazards and their associated risk control measures.

The best risk control measures are often those that can be implemented as part of the initial design. Therefore, it is important for the risk analysis plans and procedures to address related activities at each stage of the device and software life cycles and recognize that the activities and approaches vary among different stages. Note that early software risk analysis cannot be performed effectively by using the same methods used later in the life cycle when more information is available.

Pitfalls

— The risk analysis process defines only system- and hardware-level aspects. Software is only addressed where it implements risk control measures for hardware failures.

— The rigor of the risk analysis and software development life cycle procedures is not commensurate with the potential harm of the medical device.

— Software is considered part of risk analysis only late in the product development life cycle.

5.1.2 Intended-use or intended-purpose identification

14971 requirement

Describe the intended use or intended purpose and any foreseeable misuse.

Software perspective

Knowledge of the intended use and foreseeable misuse of a medical device is a key factor in determining the overall potential for harm. The role of software in a device and the potential for it to fail in a way that could make it a contributing factor to a hazardous situation determines its risk significance.

There are several classes of intended use that have a major effect on the overall approach to software design and choice of risk control measures. As an example, one could classify devices on the basis of intended purpose as:

— Life supporting

— Therapeutic

— Diagnostic or monitoring

Each of these intended-use classes could be further subdivided. Even at this high level of abstraction, however, one can see that a control measure for a high-risk failure could be quite different for each class of device. For instance, a high-risk failure mode for a diagnostic device could be presentation of an incorrect result. On the one hand, a software risk control measure that halts operation immediately if any suspect operation or corruption is detected could be appropriate. For a life supporting device, on the other hand, continued software operation to reach a safe state would be more appropriate.

Actual potential for harm depends not only on the intended clinical purpose of the device but also on the possible harm the device could cause a patient or clinician should its hardware or software fail. The following are examples in which the device's potential for harm from an intended-purpose perspective seems less severe than it may actually be:

— A device intended for minor cosmetic repair of skin—if the design of the device includes a power source or mechanical device that is capable of severe injury to a patient because of misuse or failure of hardware components or software logic, the device could cause significant harm.

— A laser device for correcting vision—if all goes well, the intent is not life supporting or life saving, but if the device is not designed properly, a failure mode could cause blindness.

— A minimally invasive device for sample collection or monitoring—if the device is reusable and the software controls a cleaning and disinfection cycle, and if the algorithm is not implemented correctly or a fault is not detected properly, disease can be transmitted from patient to patient, possibly causing death.

Clinical intended-use requirements are a precursor and primary input to risk analysis. A thorough identification of clinical use and users allows identification of possible harm and a preliminary hazard analysis early in the software development process.

Some medical device software may be intended to run on multiple platforms (i.e., hardware, operating systems), whereas others are delivered on platforms provided and controlled by manufacturers. Issues such as these are often a market or clinical requirement and need to be identified early to ensure adequate management of associated risks.

Consideration should also be given to any communications interfaces between the device and other medical devices. The device may be used as an accessory to other devices through interfaces, and there may be different safety issues associated with the information being passed between the interfaces. For example, a pharmacy system may communicate to a handheld personal digital assistant (PDA) to provide nurses prescription dosage information. If the PDA has a wireless interface to an infusion pump, the safety significance and associated characteristics of the software on the PDA may lead to different requirements and risk control measures for the PDA or for the infusion pump software itself.

Although each medical device has an intended use for defined target populations (e.g., adults, terminally ill patients, infants) the potential for misuse—intentional or otherwise—should also be considered.

For example, a medical image system that archives images for nondiagnostic purposes may store such images at lower resolutions or without adequate integrity protections for diagnostic use; yet, it may interface with other systems used for diagnosis or have displays in areas where they could be easily confused for diagnostic displays. This

possibility should be considered in risk analysis to ensure that appropriate risk control measures are addressed (such as on-screen labeling indicating the image is not for diagnostic purposes).

Pitfalls

— Considering only a subset of user environments and potential computer system platforms

— Not considering the platform evolution or need for security or other COTS patches

— Inadequately considering the misuse and user error resulting in potential hazards and, therefore, not identifying their corresponding risk control measures

5.1.3 Identification of known or foreseeable hazards

14971 requirement

A list of hazards and foreseeable sequences of events that result in a hazardous situation shall be compiled for both normal and fault conditions.

Software perspective

Hazard identification is a prerequisite to risk estimation, evaluation, and control. The analysis of hazards forms a basis for device safety requirements and is integral to the device and software development life cycle processes. When hazards are identified early in product development and in early stages of the software development life cycle, better software risk control measures can be designed into the software.

For personnel to adequately identify potential hazards, clinical use of the device must be well understood in general and for each type of user, user environment, and target population. It is, therefore, imperative to establish and maintain communications with the relevant domain experts involved in the design of the device.

Medical device hazard identification should include indirect as well as direct hazards. Indirect hazards include potential harm that could result from display of erroneous clinical information, delay of treatment, inadvertent destruction of blood or tissue samples where the sampling procedure might itself present patient risk or discomfort, and other factors related more to the effectiveness and essential performance of the device than to direct harm. An example of a direct hazard would be exposure to excessive radiation or physical harm from motors moving components that contact patients or that could release contaminated fluids or excessive medication.

In addition to their concerns over direct harm, some regulatory authorities such as the U.S. FDA are concerned about the ability of the device to perform correctly. Thus, from a regulatory perspective, “effectiveness” or “essential performance” must be considered in addition to direct harm when identifying hazards and determining risk.

Each hazard for which software is a potential cause should be identified in the risk management file for inclusion in further risk management activities. Early assumptions about the likelihood or severity of the hazard or risk control measures should not preclude hazards from being identified, because the coupling of hazardous conditions discovered in later analyses may become apparent. As the design of the device evolves, one needs to consider if functionality defined for software is creating new hazardous situations or additional causes for previously identified hazards.[14]

For each identified hazard, a list of software-related causes should be developed. A device may have multiple hazards associated with it. Likewise, each hazard may have multiple causes. The goal should be to identify software causes (or combinations of events) that could result in each hazard.

The hazards and software causes should then be traced throughout the software development process to ensure that requirements, design, code, and testing accounts for any associated risk control measures, whether implemented in software, hardware, or labeling.

For the purposes of this document, a *software-related hazard* is defined as any hazard that could have software as a cause. For software-related hazards, the hazard analysis documentation should identify traceability between the hazard and the software causes. This documentation should be recorded in the risk management file.

The search for potential causes for hazardous situations is complex. There can be a variety of necessary and sufficient contributing factors for each hazard. It can be useful to consider chains of events to help in subsequent

[14] For instance, adding a calculated value to a chemistry analyzer on the basis of two measured blood tests, adding control commands to a medical image management device to use the interface to X-ray equipment to move the X-ray head, or changing the design to use dynamic memory rather than static memory allocation could introduce new defects.

identification of risk control measures. An understanding of the chains of events that can lead to, or are necessary conditions for, a hazardous situation can sometimes help one identify certain links in the chain that, if controlled, could reduce the risk. It can also help one identify the LPOSC in a chain—which, if controlled, could eliminate the need to implement controls for every possible contributing factor. Many times, it is possible to implement hardware or software risk control measures at the LPOSC that break the causal chains that would otherwise result in a hazardous situation (see Figure 6).

5.1.3.1 Methods of identification

A variety of methods can be used to identify potential causes for hazards, such as fault tree analysis, failure mode and effects analysis, and others.[15] In general, different methods will be used during different phases of the development life cycle as more becomes known about the medical device design, software design, and intended use of the final medical device product.

The process of identifying hazards and their potential causes can become arduous because, while many of these hazards and causes are immediately obvious, additional less obvious hazards, causes, and contributing factors are sometimes very difficult to identify. Often, the process can begin as a kind of free-association process—just letting one's thoughts dwell on all aspects of the software and what problems could occur. A nagging problem with this hazard generation process is that one is never sure that the list is complete. Therefore, a process should be developed and followed that will make use of all available knowledge, such as corporate product history, device industry literature, relevant standards, vendor-provided bug lists and release notes (if such off-the-shelf software is used), and relevant hazards identified in other industries where safety is critical.

Early in the life cycle, an initial list of device-level hazards should be generated that does not take into account specific causes of those hazards. No method is available to demonstrate the completeness of a generated list of hazards and their causes. It is possible that no matter how much rigor is applied to the hazard identification process, some hazards and their causes will not be identified. These latent hazards and their causes could result in harm. Therefore, development of the hazard list should not omit hazards and potential software causes because they seem to be unlikely or because historical precedence is missing. Hazard analysis is a "cornerstone" for continuous and effective evaluation of device safety.

It is important to be aware that the identification of hazards and their causes is a multifaceted activity requiring a variety of skills and perspectives. Design engineers, manufacturing and product service personnel, and clinicians all have unique perspectives. Input from multiple sources of knowledge, such as physiology, pharmacology, life sciences, physical sciences, human factors, and relevant engineering disciplines, is necessary, in addition to software technical knowledge, to ensure a comprehensive identification of hazards and potential causes.

Historically, one of the glaring omissions in some medical device risk analyses has been the failure to identify software errors as possible causes for hazards. Moreover, if such errors are identified, little detail or differentiation in the potential software errors is provided.

Pitfalls

- Using FMEA or FTA methodologies as if they alone suffice for adequate risk management
- Performing FMEA or FTAs for hardware and software in isolation
- Ignoring a whole class of hazards and causes such as
  - Indirect causes—software errors that have unpredictable effects
  - Errors in software logic used as risk control measures for hardware failures
  - Errors in software logic for the intended clinical purpose of the device (such as algorithms for results calculations)
  - Failures of software platforms—operating systems, libraries, off-the-shelf software
  - Failures of computer components and peripherals
  - Malware (e.g., viruses, denial of service attacks)
  - Failures of communications interfaces

[15] These methods are only one part of risk management and are not sufficient on their own.

— Human factors

5.1.3.2 Direct causes versus indirect causes

When identifying causes for hazards, it is easiest to focus on software *directly* related to the essential performance of the device (e.g., an algorithm that calculates glucose levels in blood) and the direct causes[16] for related hazards. It is important, however, to also consider software errors that could result in unpredictable effects and, therefore, *indirectly* cause one or more device hazards. Examples of such indirect causes are uninitialized pointers, memory corruption, stack overflow, race condition, and others (refer to Annex B for other examples). Even though the effects are unpredictable, appropriate risk control measures still need to be identified for indirect causes.

Figure 7 is similar to an earlier diagram, Figure 6, illustrating the concept of LPOSC but has an additional set of bubbles to represent causes that cannot be predictably tied to specific chains of events. The figure emphasizes that indirect causes can affect the software at many unpredictable points, including interfering with risk control measures at any point in the causal chains. For this reason, among others, focusing exclusively on first and last points of software control is not sufficient to ensure safety.

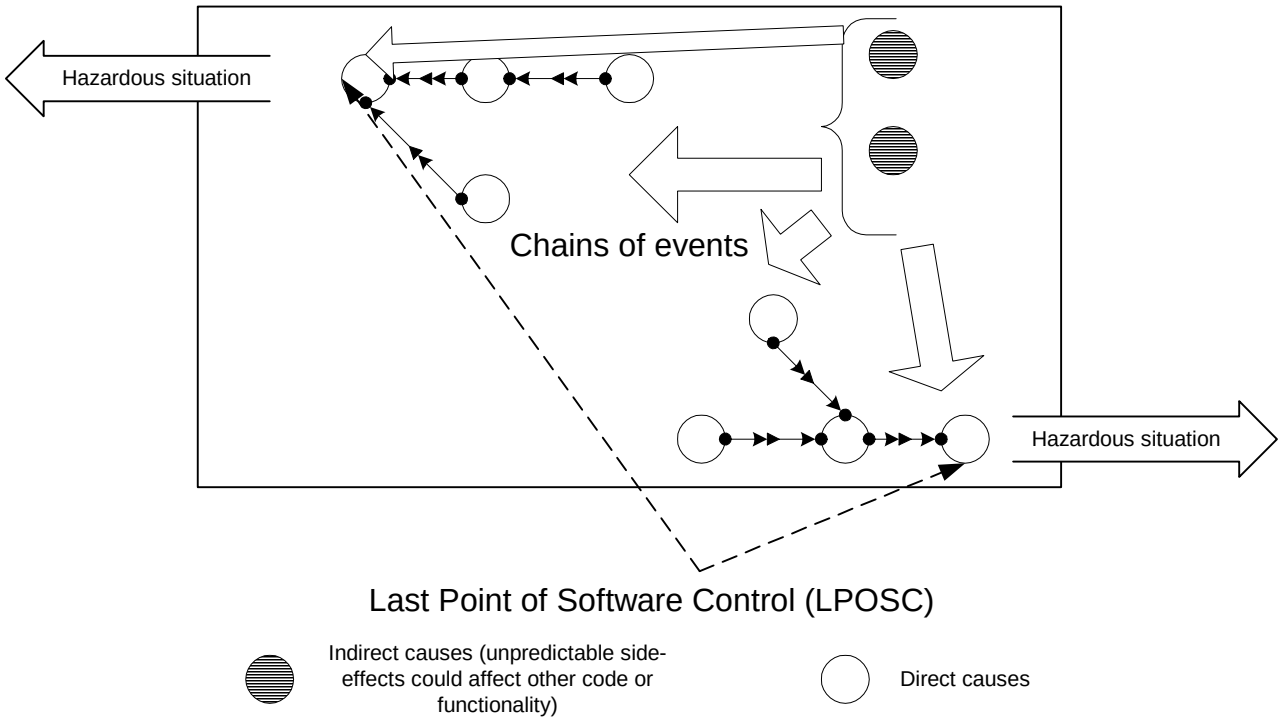

Figure 7—Direct and indirect causes

When one is identifying risk control measures for indirect causes, it is generally unproductive to attempt to trace every possible hazardous situation to each indirect cause. Generally, it is better to identify types of indirect causes and then identify detection mechanisms and risk control measures (e.g., checksums to detect data corruption of critical data, safe shutdown, or user notification of such events) for each type of cause to minimize risk should it occur.

Pitfalls

Approaching cause identification with the presumption that

— software defects will affect the functionality only in a specific component and will not have side effects on other code or data;

[16] *Direct causes* are defects in software whose functionality is clearly related to the clinical functionality of the device and that lead to one of the device hazards. An example is a defect in an algorithm for calculating a test result.

— software will work properly;

— potential indirect causes are too numerous and unpredictable for identification, detection, or risk control;

— causes and risk control measures at the first and last points of software control alone are always sufficient risk management; and

— causes can be identified without involvement of experienced software engineers.

5.1.4 Estimation of the risks for each hazard

14971 requirement

Estimate the risk for each hazard in both normal and fault conditions. *Risk* is considered to be the combination of the severity of harm and the probability of its occurrence.

Software perspective

Software failures are systematic rather than random in nature, and, therefore, their probability of occurrence cannot be determined using traditional statistical methods. As such, risk estimation for software products should be focused on the severity of the hazard resulting from failure, *assuming* that the failure will occur. Probability can sometimes be applied in a qualitative manner to evaluate risk control measures. Subjective rankings of probability could also be assigned on the basis of clinical knowledge to distinguish failures that a clinician would be likely to detect from those that a clinician would not be likely to detect and that, thus, would be more likely to cause harm.

The role of software in the medical device can also have a significant effect on both the probability that a software failure will result in a hazardous situation and the potential severity of a software failure (e.g., risk is lowered by hardware safeguards or users who are likely to detect a failure before harm would occur). The evaluation of the residual risk can take such factors into account.

5.1.4.1 Probability

Software perspective

Annex E of ANSI/AAMI/ISO 14971:2000 distinguishes *systematic failures* from *random failures* and asserts that software failures are systematic. Numerical estimates of probability cannot be determined for systematic failures. Some hardware failures are also systematic in nature (such as use of an incorrectly rated fuse or improper raw material), but many are not (such as normal wear and tear and lifetime failures). When dealing with software faults, one is dealing exclusively with systematic failures whose probability cannot generally be accurately estimated and for which there is little consensus on acceptable methods for estimation. In contrast, with hardware faults, a key focus is often analyzing the mean time between failures (MTBFs) of component parts and assigning probabilities.

It is inadvisable to use subjective estimates of the probability of a systematic (nonrandom) software failure. However, although probability of a software failure may not be estimated quantitatively, many risk control measures do not entirely eliminate the possibility of a software failure causing a hazardous situation—some simply reduce the probability that such a failure would lead to a hazardous situation to acceptable levels.[17] Therefore, probability can be used in a relative sense when evaluating alternative risk control measures or when performing pre- and post-risk control measure evaluations. It should also be noted that medical device and software testing do not prevent hazards. Rather, they attempt to detect failures that could lead to hazards. Because medical device software is too complex to test exhaustively, testing can be viewed as a method to lower the probability of harm. However, testing does not meet most definitions of a risk control measure and is insufficient alone.

In risk analysis, the probability of individual failures is often used to help determine risk, but it is really the probability of a hazardous situation or harm that is of most value. There are situations in which the probability of individual failures cannot be estimated but the probability of harm could be estimated. For many types of devices and possible failures, normal clinical practice (including physician use of patient condition contraindications) can be considered in estimation of the probability or severity of actual harm. Although one would not want to rely exclusively on clinical practice (because it varies and is outside the device designers' control) as providing adequate risk control, it certainly affects the probability and severity of harm and can be a useful factor in determining whether acceptable risk has been achieved.

[17] For instance, a checksum for memory corruption or as part of communications protocols does not guarantee that any possible corruption would be detected. Rather, it would detect the vast majority of such corruption and, thus, may lower the risk to acceptable levels.

Another aspect to consider is the size of the target population. If the device will be widely distributed, traditional statistics and confidence levels can be misleading. Society and regulatory authorities may not consider the failure rate but only the actual number of deaths or injuries when evaluating safety events.

Some risk assignment methods include probability in a manner that multiplies (or weights in some other manner) probability and severity to end up with a risk rating. A cutoff point is often chosen for the risk rating, below which risk control measures need not be considered. If such a scheme is used, and if probability assignments are faulty, the result can be inadequate risk control. To prevent this problem, especially when systematic software failures are included in the same analysis as random hardware failures, one can set a severity override rating. If severity of any potential failures is higher than a specified level, then risk control must be considered regardless of the probability and resulting risk rating. This method helps prevent overreliance on probability estimation methods in determining acceptable risk control. Generally, for medical device software, the severity override rating should be set very low to encourage consideration of software risk control measures (because some may be very simple and inexpensive if considered early in design) and to maximize focus on designing the safest software possible.

In summary, software risk estimation should focus primarily on hazard severity and the relative likelihood of harm if a failure should occur,[18] rather than on attempts to estimate the probability of each possible software failure.

Pitfalls

— Using subjective probability of a software defect to decide that risk control is not required

— Assuming single point of failure concepts apply to software systematic design issues and sequences of events

— Assuming that testing—which cannot be exhaustive—reduces the probability of a particular failure to zero

5.1.4.2 Severity

Software perspective

The purpose of defining and assigning different levels of severity is to ensure focus on those hazards and their contributing factors that can result in the most harm. There are many possible severity scales defined in standards and regulatory guidance. No single scale is best in every situation. The primary requirement is that the scale facilitate differentiation and focus. If a scale of severity levels is chosen and most or all hazards are at one level, then perhaps the scale is not appropriate. If a scale lumps highly critical hazards (that could result in death) with less severe hazards, then it is probably not appropriate. In the end, a severity scale is adequate if

— its use ensures focus on the most severe hazards throughout development, and

— it does not lead to the dismissal of certain risk control measures for the most severe hazards simply because the same measures cannot (either for technical or cost reasons) be implemented for less severe hazards.

On the other hand, a severity scale may be deemed inefficient if significant effort is spent assigning subtle severity levels with no corresponding difference in the risk control measures implemented for each level.

Generally, the severity scale used for software is predetermined by whatever scale is chosen for the medical device overall. Generally, such scales, if adequate for the device, are also adequate for the software. However, it is important for those involved in software to be aware of the clinical perspective and target population throughout the risk management and software development process. For example, the potential benefits of a device may outweigh certain risks for some target populations and not for others—greater risk of death may be acceptable in a target population of elderly, terminally ill patients, but not in a population of healthy infants.

Because probability is difficult to establish for software failures, the focus in software risk management is on identifying potential software failures, the associated severity of any resulting hazard, and risk control measures that are appropriate given the severity of the hazard. In other words, the adequacy of the risk control measure is determined primarily based on the severity of the hazard rather than on the probability of occurrence, as discussed in section 5.1.4.1.

Severity can also be used as a guide for test planning, with the most rigorous testing under the widest range of normal, abnormal, and stress conditions being focused on software components whose failures are potential causes for the most severe hazards. Additionally, severity can be a guide in determining the focus for regression testing when software changes are made.

---

[18] This approach helps provide distinctions between hazards of the same severity, thus allowing greater focus on those with higher probability of actual harm.

The worst-case severity of any hazardous situation to which a software failure could contribute can also be useful in its own right. This concept is sometimes referred to as the software's overall *level of concern*[19] for a given device.

The worst-case severity can be used as an input to determine the rigor of the overall software risk management, development, and maintenance processes for individual subsystems or modules. In general, one would implement more rigorous software risk management, development, validation, configuration management, maintenance, and other related processes for software in devices that could cause death than for software in devices where the worst possible outcome is minor injury.

In determining the worst-case severity of a software failure, note that the role of software in the device could result in the software being much less hazardous than one might expect by looking only at the intended use of the device overall. For instance, a laser used for surgery might have the potential for significant harm, but the software in the device might not actually control the laser output or might be unable to initiate output above a safe level.

One also needs to recognize the evolving and iterative nature of risk management when determining software's level of concern. Where hardware risk control measures are used (which may not be decided immediately at the beginning of a project), software measure can be less critical, because software failures might no longer lead to hazardous situations, and the software could be treated accordingly. This software's level of concern can be accounted for by identifying pre- and post-design risk and risk control. Even where this is accounted for, one should ensure that hardware design and risk control measures that are used to lower software risk are not inadvertently eliminated in subsequent design steps or as part of product maintenance or cost reduction.

Pitfalls

— Assuming, based on functionality, that certain subsystems or modules will not be safety related without considering the potential for unexpected side effects

— Assigning severity without adequate clinical knowledge of the effect of hazards on all potential users and target populations

— Assigning low severities on the basis of assumptions that clinicians will detect failures or erroneous information

— Assigning low severities on the basis of assumptions that all users will follow device labeling and manuals exactly or without inadvertent error

— Assuming some planned risk control measure for a hazard as part of assigning initial severity—if the assumption is wrong, the low initial severity may result in lack of adequate risk control being identified later

— Using only the potential for direct patient harm to identify severity without considering indirect use of information provided by the software to the user, delay of treatment, and other factors related to effectiveness and essential performance of the medical device

— Assuming that a clinician would always cross-check information provided by the software or could detect misinformation, assigning a low severity, and therefore not implementing other risk control measures

5.2 Risk evaluation

14971 requirement

For each identified hazard, use criteria identified in a risk management plan or procedures to determine if risk reduction is needed.

Software perspective

Before making software risk control decisions, evaluate the need for software risk control measures for each hazard identified. This is the premitigation stage. Here, risk evaluation identifies which hazards require risk control measures and which do not;[20] it does not identify whether planned risk control measures are adequate.[21] Recognize that mistakes made at this point can result in no risk control measures being considered in areas that could contribute to

[19] The U.S. FDA uses three levels of concern for software in a medical device as a guideline, with the highest indicating risk of death or serious injury (major, moderate, minor).

[20] Many believe that attempts should be made to identify risk control measures for all hazards—no matter how minor—because the risk control measure may be easy and inexpensive if identified early in design.

[21] Note that it may be obvious at this point that certain risks cannot be adequately controlled with software and that medical device-level changes are required (e.g., to add hardware redundancy or limits of operation).

hazards. This is a good point for quality assurance staff and system engineers to aggressively challenge software assumptions.

If you use a risk control rating scheme to assign risk numbers, it is common to create a table that depicts the combination of overall risk (probability and severity) and severity override ratings that require risk control.

Identify which hazards have potential software causes and which do not. Software aspects of risk management apply only to those hazards for which the software can be a contributing factor or where software may be used to implement risk control measures for hardware failures or user error.

There are often hazards to which software is not a contributing factor. These hazards are addressed in other parts of overall device risk management efforts. Even so, for those hazards that software cannot cause, it can be useful to document rationales to prevent future system-level changes that could result in omissions in software risk control. Then, the remaining subset of hazards is the starting point for identifying risk control measures for software. For example, software involved in motor control may not require risk control if failure of the software to properly control motion or limit speed or torque would not result in a hazardous situation or have a detrimental effect on the effectiveness of the device because of the inherent safe design of the hardware (e.g., self-limiting power source).

This initial risk evaluation requires knowledge of clinical use, possible misuse, and system-level medical device hazards, and cannot be done effectively by exclusively focusing on software alone. Use of interdisciplinary independent reviews with clinical representatives should be considered.

Pitfalls

— Eliminating a hazard from software consideration because of hardware characteristics and later changing or removing the hardware involved in a way that makes software a possible contributing factor, but not considering additional risk control measures for the software[22]

— Not considering a potential software defect as a contributing factor to a hazard because it is assumed that the software would work as intended or that testing would catch all bugs

ALARP

One important risk management concept from IEC 60601-1-4:2000 and Annex E of the revised ANSI/AAMI/ISO 14971:2000 is the goal of minimizing risk “as low as reasonably practicable” (ALARP). Although this concept is well explained in these standards, it is worth highlighting the concept’s intent and relationship to software risk management.

The ALARP concept suggests three regions of risk: intolerable, ALARP, and broadly acceptable. *Intolerable* is where additional risk control is required or the device is unacceptable. *Broadly acceptable* is where the risk is so low that no further risk control is necessary. The *ALARP* concept is not simply that one needs to be in the ALARP or broadly acceptable regions. Rather, one needs to reduce the risk to the lowest level practicable. Therefore, even when one determines that risk has been brought within the ALARP region (either in this initial risk evaluation or later after some risk control is defined), one continues to evaluate and implement risk control measures to reduce the risk further wherever practicable. What is reasonably practicable depends on the technology, clinical expectations, medical benefit and target population of the intended use, and medical economics at the time.[23]

Although the standards indicate that if risk is in the broadly acceptable region, risk control need not be actively pursued, there are a number of reasons to consider further risk reduction, particularly for software:

— Software risk control measures implemented in the design process are, in some cases, very inexpensive or free if considered early in the life cycle. (There is often no additional “parts” cost per device.)

— For clinical, regulatory, and business reasons, the safer and more effective a device, the better.

— The dividing lines between the three risk regions, although conceptually clear, are somewhat arbitrary and subjective in practice, and they shift over time. Hence, the greater the safety margin, the better.

— Risk is a combination of probability and severity, and it is difficult to estimate probability for software failures. Thus, avoiding risk control measures because of low but subjective probability estimates can be dangerous.

---

[22] This was a key issue in one of the most infamous medical device software failures—the Theracs radiation device where hardware safeguards were removed.

[23] Note that acceptable risk generally decreases over the years because of technology, changes in clinical practice, and societal pressures.

— Risk control fosters a general mindset of staying focused on clinical effect and striving for minimum risk.

5.3 Risk control

14971 requirement

Appropriate risk control measures that reduce risk to acceptable levels are identified, implemented, and verified.

5.3.1 Options analysis

Once hazards and causes are identified, corresponding risk control measures can then be identified. This stage is where the real value of risk analysis becomes apparent—in facilitating the identification of risk control measures that will lower the risk of the medical device.

ANSI/AAMI/ISO 14971:2000 identifies three classes of risk control measures. In order of importance, these are:[24]

— Inherent safety by design

— Protective measures

— Information for safety

These three classes are not mutually exclusive and may be used together.

Inherent safety by design means that design decisions have been made that either preclude or reduce the likelihood of certain causes or hazardous conditions. *Protective measures* are design decisions that detect potentially hazardous situations and either prevent them from resulting in a hazard or reduce the likelihood that harm will result. *Information for safety* is information communicated to users through labeling, user manuals, screen displays, or training. This information helps the users to avoid harming themselves or the patient by promoting proper interaction with the device and proper use of any clinical data displayed by the device.

Software perspective

It is important for software engineers to understand these distinctions and strive for the best possible type of risk control measure in each case. To do this effectively, software engineers must implement software risk management early in the device development process and software development life cycle.

It is also worthwhile to use examples to better understand each of the three classes of risk control defined in ANSI/AAMI/ISO 14971:2000 from a software perspective. Two examples are described below, and Table 1 displays examples of risk control measures of each class:

— Example 1: Software controls a motor that controls closure of a cover on a device. One potential hazard could be physical harm to a patient's finger if it is caught between the cover and the frame of the device.

— Example 2: Software controls radiation therapy dosages. A potential cause for overdose is identified as a dynamic memory allocation failure.

Table 1—Examples of risk control measures

	Inherent safety by design	Protective measures	Information for safety
Example 1	The size of the motor is selected so that closing the device cover could not injure a patient's finger even if the motor is at its maximum torque.	A software-controlled sensor reverses or stops the motor if the motor's power draw exceeds a specified limit.	A label on the cover reads, "Do not touch while in operation."
Example 2	Use of dynamic memory management is avoided by using only static structures.	The radiation output is terminated if the software error detection algorithm detects a memory allocation failure.	Instructions direct the user to reboot the device before each therapy session.

[24] In addition to the three classes of risk control measures listed, some believe that process measures can be classified as valid risk control measures. This is a topic of significant debate, especially in light of the detrimental historical tendency to list testing as the only risk control measure for software failures. There is strong consensus, however, that process measures are beneficial when considered in combination with other types of risk control measures and if defined in detail.

Note that information for safety can be in the form of screen displays[25] and online help, as well as device labels and user, service, and administrator manuals.

It is often impossible to identify individual risk control measures for every possible software defect or line of code that could contribute to a hazard. A better approach is to work at a higher level of abstraction such as considering all possible types of defects and focusing the greatest attention on the most critical components.

It is sometimes possible to implement hardware or software risk control measures at the first and last points of control (see section 4.4 and Figures 5 and 6) that break the causal chains that would otherwise result in a hazardous situation. In Figure 4, the closer one can break a specific chain of events to the actual edge of the box where the hazardous output exists, the less likely that any earlier causes in the chain that may occur would result in a hazard. Although it is not always possible to implement adequate risk control measures at the last point of control (close to the edge of the box in the diagram), there are effectiveness and efficiency reasons to attempt to do so.

To efficiently implement appropriate risk control measures for software, one must give careful consideration to the product development and software life cycles. Some types of risk control measures are very easy to implement early in design and are impossible or very costly to implement later in development. If software is not carefully considered from a risk management perspective early in the product development process, hardware decisions may be made that inadvertently place excessive reliance on proper software operation for the safety of the medical device. Section 6 of this report discusses life cycle aspects of software risk management and provides examples of the kinds of risk control measures best considered during specific software development stages and activities.

For software risk management, it can also be helpful to consider, in addition to the three classes of risk control measures described above, the categories of risk control measures described in the following subsections. Consideration of each category helps ensure the comprehensiveness of the risk control measures identified and ensures proper identification of software components that are related to safety.

5.3.1.1 Hardware risk control measures implemented in software

This category includes software that monitors or responds to hardware failures (protective measure) to ensure a safe response or shutdown in the event of a hardware failure that could otherwise lead to harm. An example is software that monitors the position of a needle and stops operation or retracts the needle when it detects a hardware sensor failure.

5.3.1.2 Software risk control measures for user errors

This category includes software that checks user inputs for validity and displays errors or requests further confirmation for inputs that are invalid or questionable (protective measure). Examples include simple data entry error checks for valid expiration dates or patient age and software that checks that the user inserted a test strip properly.

5.3.1.3 Risk control measures for direct causes

This category includes software or hardware used to ensure that specific safety-related software functionality that is traceable to specific hazards operates properly. Examples of inherent safety by design are hardware limits for motor torque or radiation output or duration. Examples of protective measures include algorithm error checking, detection of out-of-range conditions, alarms, security levels to restrict use, and redundant cross-checks. Examples of information for safety are display of user warnings and alarms. Other examples of direct causes based on categorization of typical medical device software functionality and relevant risk control measures are provided in Annex A.

5.3.1.4 Risk control measures for indirect causes

This category includes software or hardware that protects against indirect cause failures that could have unpredictable results that could contribute to one or more hazards. An example of inherent safety by design is running safety-critical software on a separate CPU with protected memory so that defects in other components would not be able to contribute to certain software hazards via unintended side effects on the safety-critical code. Examples of protective measures include checksums on critical data checked each time the data is used, background monitoring of critical timing, virus detection software, hardware watchdog timers to detect software hanging, or hardware-implemented memory error detection. Other examples of possible indirect causes and relevant risk control measures are provided in Annex B.

---

[25] Examples would be audible and visible error flags or warnings or requests for user confirmation of critical actions.

Note that it is not generally productive to attempt to trace every potential indirect cause to every potential hazard through every possible sequence of events. If a certain type of defect can have unpredictable effects on parts of the software that could affect safety or effectiveness, then this potential should be identified and the focus should be simply to identify a risk control strategy and specific risk control measures (in hardware or software). It can be more efficient to create a simple table of indirect causes and their risk control measures rather than to try to incorporate each indirect cause into tables or traces for each sequence of events and each hazard.

5.3.2 Implementation of risk control measures

Once risk control measures are identified, they need to be implemented and their effectiveness verified.

Software perspective

Risk control measures should be traceable to the hazards to which they relate, the requirements and design specifications in which they are defined, the software components in which they are implemented, and the test cases that verify their effectiveness.

Verification that the risk control measures have been properly implemented and are adequate may be difficult, but it is still essential for software. Key aspects to consider include

1) traceability to ensure that all safety-related software items and units are identified and all safety-related functionality is specified, implemented, and tested in all relevant versions and variants (e.g., for different platforms, languages, or device models) of the software;

2) greater rigor and coverage when testing risk control measures, including testing them under a wide range of abnormal and stress conditions; and

3) focus on regression-testing risk control measures and safety-related functionality whenever any changes are made, even if those changes are not expected to affect safety.

Pitfalls

— Risk control measures are verified under normal or limited conditions, but not under a wide range of abnormal and stress conditions.

— Software or data used to implement a risk control measure is not protected from access by other software, so the potential for hazardous side effects is high.

— Risk control measures are only verified on one operating platform or program variant.

— Some risk control measures are not actually verified because of the difficulty of forcing their occurrence (e.g., memory failure, race condition, data corruption, stack overflow).

5.3.3 Residual risk evaluation

An evaluation is performed to determine if the risk control measures reduce the risk to an acceptable level. If not, then risk control measures (hardware, software, or labeling) are identified or the determination of acceptable risk must be revisited, on the basis of the potential benefits of the device. The benefit derived by the patient must always be considered, along with the risk of using a medical device. Hence, the need for protection from risk caused by the medical device differs depending on the patient. These varying protection needs must be considered when determining the acceptability of residual risk.

In some cases, risk control measures deemed adequate when initially identified are found to be inadequate during verification. When this situation occurs, the residual risk should be reexamined, and if that risk is unacceptable, better risk control measures should be identified and implemented.

Software perspective

Given the difficulty in estimating the probability of software defects, residual risk evaluation usually focuses on attempts to determine if all causes for hazards that have been identified have risk control measures that are likely to work properly. These software activities are often most related to reviews of traceability and coverage and the adequacy of verification activities and results.

Defect information collected during these activities is often useful if bug severity ratings associated with safety are tracked and evaluated. Bugs with safety consequences can be evaluated to determine whether they were identified in the risk analysis and whether identified risk control measures would suffice for these bugs once implemented.

Pitfall

— Assuming that all safety-related bugs are found during development and that testing ensures that the software will work properly in the field

5.3.4 Other generated hazards

Examine the identified risk control measures to determine if those measures introduce any new hazards. If they do, then analyze whether additional risk control measures are needed or whether these risk control measures are inappropriate.

Software perspective

An example would be implementing a background memory check diagnostic to detect memory corruption or failure. If this diagnostic is implemented improperly, another asynchronous task could read memory locations that have been temporarily written with a diagnostic test pattern and misinterpret this reading as actual diagnostic values. If this possibility is not considered, then a potential cause (albeit in a risk control measure) for a hazard could be missed.

Pitfall

— Implementation of a risk control measure that makes the software design significantly more complex. This complexity increases the potential for additional software defects or may introduce new hazards.

5.4 Risk management report

This TIR does not define any specific documentation requirements. It does provide information that may aid in meeting the documentation requirements of other standards or regulations.

Software aspects of risk management can be documented to conform to ANSI/AAMI/ISO 14971:2000 requirements for the results of the risk management process. A key requirement of ANSI/AAMI/ISO 14971:2000 for the risk management report (whether a single document, multiple documents, or a document in electronic form) is that each hazard must be traceable to its causes, risk control measures, their verification, and their post-risk control risk assessment.

Software perspective

Clearly, this traceability should include software risk control measures for users, hardware or software causes for hazards, software components that implement risk control measures, and software testing that verifies the proper function of those components.

The current focus in medical device risk management is on the process that identifies potential hazards, the failure modes that could contribute to these hazards, and appropriate risk control measures. When one is dealing with random hardware failures, there is often a clear concept of the granularity of the analysis and control based on the physical components with which one is dealing. There is also some degree of confidence in the independence of many of the components (although sometimes this sense of confidence is proven false).

When one is dealing with software, it is not clear at what level of granularity to define a component, and the potential for side effects between components is often high. It is not possible in any but the smallest of programs to consider each programming statement to be a component, and there is no inherent independence from statement to statement in a single software module or even across different modules in a single memory space. Additionally, the adequacy of a software risk control measure may not be as obvious as for some hardware risk control measures.[26]

For these and several other reasons, it can be useful to think in terms of developing a “safety case” as a summary of the result of the risk management process. A safety case is a requirement in many safety standards. Explicit safety cases are required for military systems, the offshore oil industry, rail transport, and the nuclear industry. A safety case can be defined as, “A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.”[27]

One’s ability to make a safety case and the confidence one can have in the safety case are directly related to the adequacy of the risk analysis and risk controls implemented. However, the articulation of a safety case can be different from documentation of a traditional risk analysis. A safety case could use the results of the risk management

[26] For instance, an electrical shock hazard may be eliminated through use of a low power DC power source, or physical harm may be eliminated through use of a low torque motor in some cases. Is encapsulation of critical data as clear or effective?

[27] From a research article, “A methodology for safety case development,” by Bishop and Bloomfield of Adelard London.

process to articulate why the software is safe enough for its intended use and why it meets all regulatory requirements (and could do so in the relevant regulatory terminology). Reviewers (whether internal to the organization or outside regulators[28]) may find it quicker and easier to reach a similar conclusion by this method than by examining detailed risk analysis tables and voluminous documentation developed incrementally and iteratively throughout the development life cycle. Of course, one must be attentive to the need for adequate documented evidence to support the safety case in a regulated environment.

A safety case can also be used to articulate why use of off-the-shelf software is appropriate.

One could view a safety case as a risk management or residual risk summary with references to more detailed documentation for supporting information. It could also include cross-references to demonstrate specification and test coverage for all risk control measures.

If safety cases are developed, it is important to recognize that they are summaries and should not replace detailed risk analysis.

Pitfalls

— Omission of supportable rationale for the acceptability of the final residual risk

— Inadequate explanation of severity or probability rating schemes and associated risk calculation

— Inadequate traceability to demonstrate that risk control measures have been implemented and verified

5.5 Postproduction information

14971 requirement

Procedures shall be established to review and evaluate information after field release. It is especially important to determine if any unrecognized hazards are present, if the estimated risks are no longer acceptable, or if the original risk assessment is otherwise invalidated.

Software perspective

Software risk management should continue after product release both for changes and for evaluation of field information from use of the device. This section of ANSI/AAMI/ISO 14971:2000 addresses the evaluation of field information. Refer to section 6.9 of the life cycle portion of this report for more information on overall risk management during software maintenance.

Field complaints and field failures should be analyzed not only to determine if specific fixes are needed but also to determine if assumptions in the risk analysis may have been incorrect. For instance, excessive problems attributable to use errors could indicate that human factors-related risk control measures[29] were not effective or that assumptions about the skill of certain types of intended users were inappropriate. If safety and effectiveness are involved, it cannot be assumed that because the software performs according to specification it is acceptable. If the user interface design is confusing to users and they make mistakes that relate to safety or effectiveness, then the design should be further evaluated. Simply assuming that events caused by user error cannot be reduced (in probability or severity) through software changes is inappropriate.

It is also important to collect and assess severity information, in terms of both the symptoms of anomalies reported after release and their underlying root causes determined after investigation. In some cases, a reported anomaly may seem trivial, but on investigation it may become clear that the underlying defect could cause a hazardous event.

The approach to assessing and assigning severities to a reported field problem must be carefully considered. Occurrence of an anomaly that could lead to a hazardous situation does not always lead to a hazardous event or actual harm (because of human intervention, other safeguards, or luck). Information distinguishing between possible and real harm should also be collected for subsequent analysis.

Many software defect tracking schemes use a severity scale that is unrelated to safety. For instance, the worst severity might be a crash in many common schemes. Yet, for many medical devices, a crash or inability to obtain a result may not be unsafe. It is useful to establish defect severity ratings that can clearly convey safety ramifications for

---

[28] Although the U.S. FDA's Quality System Regulation, ANSI/AAMI/ISO 13485:2003, and ISO 9001 require risk analysis, the intent is to ensure safety and effectiveness. When performing premarket reviews, the FDA is attempting to evaluate safety and effectiveness and not just ensuring that a risk management process was formalized and followed.

[29] For instance, alarm or error indications may not be adequately visible or audible for the actual users in their actual environment.

reporting trends and conducting analysis, as well as for properly prioritizing corrective action. Note that this information can be useful during development as well as postproduction.

Pitfalls

— Ignoring potentially hazardous field events caused by user error when additional risk control measures could be introduced

— Assuming initial probability and severity estimates are accurate with no evaluation of field information

— Not considering reported misuse when assessing needs for additional risk control measures

— Not considering a reported unanticipated use for which the implemented risk control measures may be inadequate; for example, an *in vitro* diagnostic for an HIV test was intended for individual use but is then used for screening the public blood supply

6 Perspective 3: Software risk management within a software life cycle

The primary goal of software risk management activities is to ensure that software's contribution to or mitigation of device risk is addressed in a disciplined manner. This means that software causes or contributing factors to a hazardous situation are identified and evaluated, and appropriate risk control measures are implemented and verified. In particular, these activities help ensure that

— software does not cause or contribute to a hazardous situation,

— software does not fail to prevent a hazardous situation when expected to do so,

— software does not fail to detect a hazardous situation when expected to do so, and

— software does not fail to take corrective action for a hazardous situation when expected to do so.

Just as software risk management cannot be performed in isolation from overall medical device risk management, software risk management activities cannot be effectively performed unless they are integrated into the software development life cycle and considered in each software development and maintenance activity.

This section takes the concepts of ANSI/AAMI/ISO 14971:2000 and applies them to medical device software life cycle processes and activities. The software risk management activities are organized and discussed in this section for each of the two primary life cycle processes—development and maintenance—and their associated activities, as defined by ANSI/AAMI SW68:2001.

Although this report is focused on software risk management, there are many opportunities for software designers to contribute to the overall safety of the medical device during system design. By participating in the system design process, the software designer can contribute to safety decisions concerning hardware–software partitioning and other aspects of the device design.

6.1 Risk management–life cycle integration

Risk management activities are best accomplished as complementary design and development activities during each phase of a software development life cycle. Although causes for hazards may be uncovered throughout the software development life cycle, their mitigation is best accomplished early in the design process, particularly in the case of risk control measures that rely on architectural decisions or require hardware. Device safety may be compromised when retrospective analysis, "on the fly" coding changes, or documentation changes occur late in the development life cycle.

Pitfalls

— Risk management activities are not established in software plans and life cycle processes.

— Software risk management activities are not related to overall medical device risk management activities.

— Software hazard analysis occurs only at one phase in the life cycle.

— Software developers and testers are not trained or experienced in risk management.

— Software risk management is assumed to be covered by general risk management activities.

Table 2 lists the development activities of ANSI/AAMI SW68:2001 and associated software risk management activities. Note that this table is not intended to represent a strict, sequential waterfall life cycle.

Table 2—Life cycle–risk management grid

Activity	Risk analysis	Risk evaluation	Risk control
Development process			
Process implementation	— Plan how risk management will be integrated into the software life cycle and how software will be considered in system risk management activities. — Identify key decision points and ensure that appropriate risk activities occur before milestone transitions. — Identify personnel to participate in risk management activities and ensure appropriate skill and training.		
Requirements analysis	— Analyze intended use and users to identify hazards. — Define potential hazards in relationship to clinicians, patients, and service personnel, and anyone else who comes into contact with the device. — Identify software requirements related to safety. — Take into account product stages such as installation and assembly, training, use, upgrade, and maintenance. — Initially identify defects or classes of defects that could result in hazards.	— Analyze severity of potential hazards related to software. — Determine whether software risk control measures alone would be adequate or whether hardware risk control measures are necessary or desirable and feasible. — Evaluate severity and probability of single and multiple point failures.	— Identify software risk control measures for hardware failures and user error. — Initially identify software risk control measures for software failures that could affect design. — Identify software to increase detectability, reduce severity, or reduce probability of a hazardous situation. — Review for new potential hazards. — Identify (or label) risk controls as “safety requirements” for traceability purposes.
Architectural design	— Identify critical data and components and classes of defects that could lead to hazards, paying special attention to indirect causes. — Identify associated hazards. — Identify interfaces, what is communicated, and when it is communicated. — Evaluate performance criteria and limitations.	— Reevaluate acceptability of software’s allocated role from a risk perspective. — Evaluate susceptibility of control measures to be affected by functionality not related to safety.	— Identify architectural risk control measures to isolate critical components and prevent or detect specific and indirect causes for hazards, paying special attention to providing any appropriate redundancy. — Identify hazards associated with redundancy. — Identify global methods for detection and control.

Table 2 *(continued)*

Activity	Risk analysis	Risk evaluation	Risk control
Detailed design and coding	— Identify additional potential causes for hazards. — Assume data, coding, and transmission errors. — Assume hardware failures.	— Reevaluate adequacy of risk control measures. — Determine separation of safety-related and non-safety-related code.	— Implement specific risk control measures and defensive design/ programming practices. — Perform trace and coverage analysis to ensure that safety-related requirements and risk control measures are implemented. — Determine whether unspecified functionality has been implemented.
Unit, integration, system, and validation testing	— Identify additional potential causes for hazards. — Evaluate each test failure for similar code implementations.	— Reevaluate adequacy of risk control measures by challenging risk control measures under a range of conditions and by testing with representative users in representative environments.	— Verify safety-related requirements and risk control measures under a range of conditions across all platforms. — Do regression testing of risk control measures before final release. — Perform trace and coverage analysis to ensure that safety-related requirements and risk control measures are implemented and tested.
Software release	— Identify configuration management plan, including configuration items and interdependencies.		— Verify that proper versions of custom and COTS software are released. — Verify that build environment is under configuration control.
Maintenance process			
Process implementation	— Plan how risk management will be performed for changes, enhancements, and fixes and how field use information will be monitored and analyzed to assess adequacy of risk control and opportunities for additional risk reduction.		
Problem and modification analysis	— Analyze field problems to identify additional hazards and causes for hazards.	— Reevaluate risk ratings and adequacy of risk control measures.	— Analyze field problems to identify additional risk control measures needed or modifications to existing measures.

Table 2 *(continued)*

Activity	Risk analysis	Risk evaluation	Risk control
Modification implementation	— Similar to development, but with a focus on the effect of changes to — affect existing risk control measures, — introduce new causes for hazards, and — introduce new intended-use functionality that introduces new hazards. — Regression testing of safety-related code.		

6.2 ANSI/AAMI SW68:2001 development process

ANSI/AAMI SW68:2001 defines development as one of only two primary processes.[30] Within this process, a number of activities are required. The following subsections correspond to each of the activities defined in ANSI/AAMI SW68:2001, and risk management aspects of each of these activities are discussed. As in section 5, examples of pitfalls are also provided. For certain activities, recognized techniques exist that are applicable to the activity. Those techniques will be mentioned in context; however, detailed information about the techniques is beyond the scope of this section.

The remainder of this section discusses risk management aspects of the activities of the development process defined in ANSI/AAMI SW68:2001, including

— process implementation,

— software requirements analysis,

— software architectural design,

— software-detailed design,

— software coding and testing,

— software integration and testing,

— software system testing,

— software validation, and

— software release.

6.2.1 Process implementation

SW68 requirement

ANSI/AAMI SW68:2001 does not define a specific life cycle. It does require formal planning and implementation of each required process and activity in establishing a life cycle. It refers to such planning as “process implementation.” It also specifically requires definition of a software hazard management process and mentions the establishment of risk and safety information and analysis during many development activities.

Risk management perspective

When establishing the device risk management process, aspects unique to software risk management should be considered, such as safe coding standards, verification methods (e.g., formal proofs, peer reviews, walk-throughs, and simulations), use of syntactic and logic checkers, and other unique software issues raised in earlier chapters.

Software risk management activities should be addressed for each stage of device development in plans, procedures, or training, as appropriate.

Pitfalls

— Software risk is not managed in a disciplined manner.

[30] It defines a number of other supporting processes not addressed explicitly in this report such as documentation and problem resolution.

— Traceability of safety decisions is not established.

— Separate software and hardware risk analyses are planned without proper coordination at the system level.

— Plans do not identify specific risk management activities at each stage of the life cycle.

6.2.2 Software requirements analysis

SW68 requirements

The developer shall establish software requirements, including safety requirements and risk control measures for hardware failures and software defects. The requirements shall include safety-related functionality dependent on off-the-shelf software. Each safety requirement implemented in software shall be uniquely identified and traceable to specific risk control measures.

Risk management perspective

Software requirements evolve in concert with system requirements. These software requirements are analyzed throughout the device life cycle as new information becomes available or changes and enhancements are required. Software safety requirements may address device function related to

— the clinical use of the device (e.g., an algorithm to calculate a dosage or test result),

— risk control measures for hardware failures (e.g., detection of sensor fault) or user errors (e.g., range checking of data entry),

— risk control measures for software defects (e.g., data corruption or algorithm error), and

— human factors conditions (e.g., misleading or illogical user interface or display colors).

Analysis of software safety requirements should explicitly cover[31]

— the clinical or possible intended use[32] of the device and associated clinical information (presented, calculated, or sensed) and control commands (to actuators) related to safety and effectiveness;

— software conditions that could be a contributing factor to a hazardous situation—including as part of a chain of events attributable to hardware failures or user errors—or side effects from other software, hardware, or user errors;[33]

— hardware, software, and labeling risk control measures to prevent software faults from leading to a hazardous situation or to reduce the probability that such faults could lead to a hazardous situation;

— residual risk and risk acceptability, given the intended use and target population of the device after risk control measures are established; and

— any requirements needed to allow adequate verification of risk control measures and clinical function.

Software requirements that are risk control measures themselves need to be evaluated to determine whether they have introduced other causes of hazards or coupling effects that could initiate undesired chains of events.

Annex A provides a list of software functional areas, examples of causes associated with these functional areas, and questions to consider. Some of these examples are relevant to requirements analysis, while others are relevant to architectural or detailed design.

The most important risk management outcome from performing software requirements analysis is the establishment of safety-related software requirements. These requirements may define

— essential performance (e.g., device algorithms, data limits, therapy sequencing) or

---

[31] Note that these areas are relevant both during initial design and development and during maintenance as each change is considered and implemented.

[32] A device may be intended for a specific use for a specific target population, but if it could easily be used under other circumstances, then relevant risks and risk control measures should be considered.

[33] For instance, (a) an uninitialized pointer in a module resulting in the overwriting of critical data and (b) hardware memory failures that corrupt code or data.

— software functions that implement risk control measures (e.g., detect or mitigate possible failures, such as detecting a hardware failure or a user error).

Determining all safety-related requirements can be difficult. As a starting point, one can analyze the device's clinical utility and intended use. Analyses of the device's hazards and defined safe states may yield safety-related software requirements that, given a similar hazardous condition, might result in quite different risk control measures, depending on the intended uses of the device. For example, a device designed to provide radiation therapy for cancer treatment is intended to be used in a hospital environment under the control of a trained user. On detecting that the device settings have been corrupted, the software may place the device in a *safe state*, which has been defined as a state preventing therapy, until specific user intervention has occurred. Conversely, stopping therapy for some types of devices might be fatal. Thus, the clinical use of the device needs to be well understood to determine the appropriate software safety requirements. Device manufacturers must ensure that all safety-related software requirements identified in the system requirements, system hazard analysis, and system design trace to the software requirements. It is also prudent to document the rationale for safety decisions.

Table 3 presents different types of analyses that may help the designer better identify safety-related software requirements.

Table 3—Types of analyses

Analysis	Description and examples
System safety analysis	— Review system documents for safety requirements, including system risk analysis, system hazards, and intended use. — Review functionality that provides any clinical information or is directly related to safety and effectiveness.
Prototype analysis	— Develop prototypes of safety-critical functionality to determine concept feasibility and to enhance understanding and completeness of requirements. — Analyze the review and feedback obtained from the prototype. This type of analysis is usually more fruitful than analysis of written requirements. Often, prototypes make the review and feedback process more fruitful because they are virtually nonambiguous, a problem often found with written requirements.[a]
Operational analysis	— Determine the "safe" states of the device or software. — Determine the required fault reaction time (maximum allowable time between the occurrence of a potentially hazardous situation and when the problem is detected by the software). — Examine dependency on date and time. — Examine system boundaries for effects on external interfaces and other system components. — Investigate the effect of viruses or other inadvertently introduced software. — Investigate dependency on operating system security updates. — Determine power-up, power-down, power-loss, and recovery requirements.
Human factors analysis	— Solicit the opinions of human factors engineers and clinical experts on the completeness and correctness of the software safety requirements, including those for alarms, training, and labeling.

Table 3 *(continued)*

Analysis	Description and examples
Interface analysis	— Determine the requirements for user and multiuser accessibility, system security features, and network and web access in light of the user's proficiency and intended uses. Analyze these requirements for potential hazards and potential failure modes that could contribute to these hazards, and identify risk control measures. — Analyze the requirements for user interfaces (e.g., data or screen, control types, screen-to-screen relationships, menu items, critical control locations, colors, sizes) and ensure that they are documented in the software requirements specification. — Define the hardware–software interfaces, and identify faults that could contribute to hazards. Identify risk control measures for these hazards.
Use analysis	— Invite typical users and subject matter experts to review functional and performance requirements, prototypes, and user interface design concepts. — Investigate validity checks for user input and user-initiated actions.
Standards analysis	— Determine the domestic and international, regulatory, and product safety or performance standards that apply to the device, plus those aspects that are relevant to software. — Determine if any additional standards are appropriate (e.g., user interface standards).
Historical product analysis	— Analyze prior product field history to identify additional hazards and contributing factors that should be considered (e.g., complaint records, service records, recalls, adverse event reports, and on-label and off-label use).
Hardware analysis	— Understand the software's role in hardware failure mitigation (e.g., investigate fail-safe conditions in which software may be a mitigating requirement). Include this functionality in the software requirements document.
Performance analysis	— Determine hardware and software performance limitations. — Evaluate stress conditions and peak loads.

[a] Ensure that code implemented for a medical device follows the appropriate software development process.

6.2.3 Software architectural design

SW68 requirement

The developer shall transform the requirements into an architecture that describes the software's structure, identifies the software items, and defines interfaces between internal as well as external software items.

Risk management perspective

Software architecture defines the software system in terms of computational components and interactions among those components. It is typically specified during the system design process, and it is often tightly coupled to the hardware platform or operating system on which it is implemented. It can be a monolithic architecture (for example, an infusion pump controller built around a single microprocessor), or it can be a heterogeneous architecture (such as a linear accelerator built around several microprocessors; there might be one software architecture for the device console, one for the device controller, and one for the safety system). Integral to the architectural design should be the incorporation of architectural features to support risk control. It is important to identify potential risks and their causes (including possible software failure modes and classes of programming errors) as early in the design process as possible because system-level design decisions can have a dramatic effect on risk. A specific focus on risk management aspects should be required during design brainstorming and reviews.

Software safety architectures may have similarities to functional architectures, but the requirements on which they are based are markedly different. For example, if a requirement exists to accept a three-character ASCII input value, the functional architecture does just that; the safety architecture might check to see if the three characters were valid for

the requirement. The appropriate software system safety architecture is determined on the basis of a thorough risk assessment that includes considering types of direct and indirect causes of hazards.

Annex A provides a sample list of software functional areas, sample causes associated with the functional area, and questions to consider. Some of these questions may be relevant to requirements analysis, and others relevant to architectural or detailed design. Annex B provides a sample list of indirect system-wide causes and potential risk control measures to consider during architectural or detailed design.

There are three fundamental architectural design strategies for reducing risk. These strategies may be used alone or in combination. In order of preference, the three strategies are

— inherently safe design,

— fault tolerance, and

— protective measures.

6.2.3.1 Inherently safe design architectures

A hazard associated with a software control function might be avoided, for example, by using hardware to implement the function. Similarly, a hazard associated with a hardware function (wear, fatigue) might be avoided by using software.

Sometimes a hazard can be completely avoided by making a high-level design decision. For instance, from a hardware perspective, use of batteries as a power source in place of AC power could eliminate the risk of electrocution. Sometimes a whole class of programming errors that could lead to a hazard can be eliminated by making high-level design decisions. For instance, memory leaks can be avoided by using only static data structures.

6.2.3.2 Fault-tolerant architectures

For mission-critical systems, such as life-supporting devices, medical device failure poses a hazard to patients or users. High-reliability components and fault-tolerant architectures may improve the inherent safety of the device by limiting risk associated with device failure.

Fault-tolerant design is a very common approach to improving device dependability. This approach aims to reduce the likelihood of medical device failure, thereby minimizing risk, by using some form of redundancy in the design. The objective of fault-tolerant design is to ensure that the safety-related systems will continue to operate in the presence of component faults.

There are many techniques for achieving redundancy. For example, the redundant elements may be interchangeable, or they may use diversity in their design (e.g., independently-developed algorithms and code for cross-checking critical sensory input) to minimize the likelihood of a common failure mode. Another consideration is whether the redundant elements actively participate in the normal operation of the device or whether they remain in standby until needed. Some redundancy schemes use majority voting or similar combinatorial operations to select the outputs from redundant elements. Other medical device architectures place the redundant elements in hot standby, cold standby, or even a supervisory role in which the redundant element intervenes or is activated only in the event of failure of the primary element.

Software redundancy can be implemented using various methods. Some examples are provided in Table 4.[34]

[34] “Safety modeling” (Pradhan, 1996).

Table 4—Various methods of software redundancy

Information redundancy	Time redundancy	Software redundancy
— Parity codes	— Transient fault detection	— Consistency checks
— Duplication codes	— Permanent fault detection	— Capability checks
— Checksums		— Self-checking programming
— Cyclic codes		— N-version programming diversity
— Arithmetic codes		— Recovery blocks
— Berger codes		
— Hamming error-correction codes		
— Self-checking concepts		

6.2.3.3 Protective measures—Architecture

In many cases, it is not practical to avoid all hazards by inherently safe design or to implement fault tolerance for all potential failures. In those cases, protective measures are the next best approach to managing a potential hazard. These measures typically operate by detecting a potentially hazardous situation and either intervening automatically to mitigate consequences or generating an alarm so that the user may intervene.

For example, a therapeutic X-ray system may have an interlock system using software logic or hardware that shuts down the X-ray generator if any door to the facility is opened. The interlock function has no role in delivering the therapy. Its sole purpose is to mitigate the hazard of unintentional radiation exposure.

To meet an acceptable level of risk, risk control measures should reduce the severity of the potential for harm or reduce the probability of the occurrence of the event. Each additional control measure increases the chance of design error and may introduce new hazards or potential contributing factors into the device. Therefore, control measures should be as simple and straightforward as possible and must always be subjected to a new risk assessment.

In some cases (i.e., where loss of device functionality does not pose a hazard), safety may be achieved at the expense of the mission. For example, a failure of a laboratory blood analyzer to provide a result may not in some cases be hazardous, but providing an incorrect result could be hazardous. In this example, shutting down the analyzer when defensive programming checks indicate unexpected faults, rather than continuing to operate, reduces risks. In a *fail-safe* architecture, a system or component fault or other hazardous condition may lead to a loss of function, but in a manner that preserves the safety of operators and patients. In a *fail-operational* system, the system may continue to operate safely, but with degraded performance (e.g., reduced capacity or slower response time). These techniques may also be applied on a function-by-function basis in complex devices, which leads to the concept of *graceful degradation*.

Protective measures can be characterized as *active* or *passive*. *Active* measures require execution time to perform their protective function (e.g., run-time checks, built-in tests, and built-in warnings). Built-in tests common to embedded systems include Read only memory (ROM) cyclic redundancy check (CRC)or checksum tests, redundant storage and cross-checks of critical variables, stack checks, and program checks. *Passive* protective measures can be thought of as properties of the system or software architecture such as partitioning, encapsulation, static data storage, or filling unused ROM with known values. These measures do not require execution time to perform their protective function. Many user interface design techniques, such as logical grouping and color coding of controls, constitute passive protective measures as well.

6.2.3.4 Combination of architectures

Sometimes a single architecture will not be sufficient to achieve the necessary risk reduction. In such a case, various architectures can be combined. In fact, many practical risk control architectures defy categorization into the neat classification scheme outlined above but instead combine elements of two or three categories. Thus, it may not be beneficial to expend a lot of energy on characterizing the risk control architectures one chooses to use. Rather, it is important to be certain that the risk management process has addressed the risk in a reasonable manner.

For example, the effectiveness of protective measures such as warning indicators and error messages may rely on the user having adequate training or prior information so as to respond correctly. It is interesting to note that, while

inherently safe design and fault tolerance strive to eliminate the user from the risk equation, protective measures and user information approaches tend to rely on human factor considerations and the user.

If combined measures are used, they must not be mutually exclusive, and they must all be available at the time the cause of a hazard occurs. Care must be taken that fault reaction times do not add up to an unacceptable delay.

6.2.3.5 Commercial off-the-shelf software considerations

Use of COTS components (or any software of unknown pedigree, commercial or not) is often determined during the system design. The higher the potential hazards of the medical device, the more closely potential failure modes of COTS components shall be examined and the more carefully risk control measures shall be designed. COTS components cannot generally be modified to incorporate new risk control measures, nor is there adequate internal design information available to identify all of the potential hazards caused by COTS components. Therefore, the system and software architecture must be designed to provide any risk control measures needed to monitor or isolate COTS components to prevent them from causing hazards if they fail.

When COTS components are included in the architecture, care must be taken to prevent compromising device integrity. Such a situation may call for the introduction of “wrappers” or a middleware architecture. The middleware may (a) prevent the invocation of prescribed features of the operating system, (b) perform logical checks to ensure that correct information is transferred between the application programming interface (API) and the device software, or (c) provide additional information needed by the device.

Another critical issue related to COTS is the use of commercial operating and communication systems. An architecture must be established that permits changes (e.g., stability, security) to the software platform on the basis of a thorough risk assessment without compromising safety. Such a risk assessment must include an analysis of the frequency of changes necessary to ensure device safety integrity, such as installing network security patches.

Pitfalls

— Missing inherently safe design risk control measures by not considering risk analysis and control when defining software architecture

— Assuming that testing will make ineffective architectures adequately safe

— Failing to identify safety-related aspects of the architecture, resulting in unknown safety risks when these architectural elements are subsequently changed or eliminated

6.2.4 Software detailed design

SW68 requirement

The developer shall define testable software units and their detailed design, including interfaces between units and with external components.

Risk management perspective

During detailed design, there is opportunity to plan for making the various software units safe individually and collectively before the code has been written and becomes much more difficult to change. Here, safety requirements, typically expressed in abstract terms, are made concrete. Examples include inherently safe design decisions such as not using global variables, or partitioning the processing of a complex algorithm into a number of simple units.

Detailed design activities should

— define implementation specifics for risk control measures,

— include analysis and reviews of the effectiveness of the risk control measures implemented,

— identify additional potential causes of hazards,

— identify and document any additional risk control measures needed,

— document safety decisions,

— document and assess assumptions,

— identify, for traceability, all software components that are related to safety and effectiveness, and

— reevaluate the architecture to ensure that it is adequate to achieve safety objectives.

During detailed design and coding, it is important to assume that defects will occur and ask, “What defensive risk control measures could be used to mitigate potential faults or reduce their probability of occurrence?”

Considering types of direct and indirect causes for hazards is helpful in identifying risk control measures that can be incorporated into the detailed design.

Pitfalls

— Focusing only on handling normal cases and assuming that interfaces and parameters passed between components will be correct, rather than incorporating multiple levels of error checking

— Not considering identification of potential software failures that could lead to hazards and associated risk control measures during detailed design brainstorming as well as subsequent reviews

— Ignoring systematic indirect software failure causes (see Annex B) in risk management activities

6.2.5 Code and unit test

SW68 requirement

Software units are established and approaches are defined for verifying each unit. Software code and test procedures are verified.

Risk management perspective

It is important to continue to evaluate both the adequacy of the risk control measures and the completeness of the hazards and causes identified in previous activities. During coding, it is not uncommon to identify additional causes that could lead to hazards or to identify weaknesses in risk control measure designs.

Annex B provides a list of sample types of indirect causes to consider, associated risk control measures, and verification techniques.

There are many opportunities to introduce errors and faults that result in safety-related issues when translating a design into code. The standards’ framework, on which this report is based, is relatively silent in regard to good coding practices in general and safe coding practices in particular. Although this section makes no attempt to address all of the various coding language nuances, it does serve to identify some safe coding practices common to the various coding languages and (integrated) development or prototyping environments.

Coding

Good programming practices should be defined and implemented to either prevent defects or detect them before they trigger a chain of events that could lead to a hazardous situation. Because software systems are so complex, it is almost impossible to identify every possible cause or chain of events. Therefore, it is important from a risk management perspective for defensive programming to be broadly implemented in all components that could affect code (directly or indirectly) related to the safety or effectiveness of the device.[35] Code inspections should challenge both the risk control measures and the completeness of identified causes of hazards.

Safe coding syntax

For every coding language, there is a coding syntax that is safer than other coding syntax. For example, in the C or C++ language, the statement, *if (a == NULL)*, is not as safe as, *if (NULL == a)*. This is because if the programmer inadvertently types “=” instead of “==”, the variable a, in the first case, will be assigned the value NULL, which could result in incorrect or unpredictable program execution. In the second case, the compiler could be expected to issue a syntax error, causing the programmer to look at the statement and realize that the operator “==” was intended.

Simplicity

Code that is difficult to understand is difficult to test and to modify. In many cases, those responsible for modifying and retesting the code in a maintenance activity are not the same people who first wrote the code. The maintenance programmers may not know the assumptions made by the original programmers. Complex code tends to have more errors than simple code. The more complex the code is, the more likely that errors will be introduced when the code is modified at a later time. Wherever possible, programmers should use the simplest programming expression that

[35] One should not treat issues enumerated in this section as exhaustive. Literature on the specific language or development environment should be sought out to understand how issues raised in this section apply and to understand how other issues may be equally relevant.

accomplishes the task. For example, if an algorithm requires multiplying a value by two, the programmer should not shift bits in the value left in an attempt to optimize the operation. Modern compilers will do this automatically. As a matter of fact, programmers' attempts to optimize the code with unusual sequences of operations may defeat the compiler's own optimization strategies.

Using appropriate coding and commenting styles (documentation) in a disciplined manner can improve code safety indirectly by encouraging the programmer to verify statements in the code with design requirements. Similarly, those charged with making changes to the code (maintenance) can more easily verify design requirements before making changes.

Defensive techniques

There are certain simple, proactive techniques that a programmer can use to facilitate data-checking schemes (robustness), program flow stability, and retrospective failure analysis; for example,

— preloading known values in stacks, queues, and buffers;

— initializing variables before use (especially during power-up and reset scenarios);

— initializing memory (as appropriate);

— using defined constants instead of hard-coding numerical values within an expression;

— minimizing the scope and number of global variables; and

— ensuring that each variable should have one and only one distinct purpose.

Other simple practices can help detect errors early in the coding cycle. These practices include setting compiler options to report all detected errors and investigate all compiler warnings and using syntax checkers (e.g., *lint*), tools that check for memory leaks, and other analysis tools that can find common coding errors.

Safe coding practices

The following list enumerates some safe coding practices that may help reduce the likelihood of defects that could lead to hazards:

a) Implement predictable memory utilization techniques such as

   — minimizing (or avoiding) dynamic memory allocation;

   — minimizing memory paging and swapping by locking pages needed by time-sensitive, safety-related functions in memory;

   — minimizing (or avoiding) recursive function calls;

   — using boundary checking for memory-related functions; and

   — verifying proper array indexing and all pointer usage.

b) Implement program control flow that is predictable through techniques such as

   — maximizing modularity and component independence;

   — minimizing control flow complexity;

   — using single entry and exit points for subroutines, functions, or methods;

   — minimizing component interface ambiguities;

   — using strong data typing;

   — addressing precision and accuracy needs;

   — using proper language rules for order of precedence of arithmetic, logical, and functional operators, simplifying whenever possible, and considering possible ambiguities that could result in errors;

   — avoiding functions, procedures, and macros with side effects;

   — separating assignment from evaluation;

— handling properly the program (debugging) instrumentation;

— controlling class library size;

— minimizing (or avoiding) the use of dynamic binding; and

— minimizing (or avoiding) operator overloading.

c) Implement predictable timing behavior through techniques such as

— avoiding use of recursive routines,

— avoiding use of dynamic memory allocation,

— avoiding unbounded waits for hardware or software responses,

— understanding the maximum execution time of all library and operating system calls, and

— analyzing the code to ensure that the processor utilization is adequate and that all tasks meet their schedulability requirements (timing deadlines).

d) Obtain predictable mathematical or logical results through techniques such as

— using a default case in every switch statement;

— avoiding the use of mixed-sign arithmetic, or at least being aware of how the coding language handles an operation on signed and unsigned variables;

— understanding the effects of rounding, truncation, and changes in precision in mathematical operations; and

— avoiding operating on very large and very small numbers in the same expression.

Robustness considerations

Care should be taken when coding for robustness because such coding tends to add complexity to the system. This added complexity increases the probability that additional errors will be introduced into the code. When programmers are coding for robustness, they should consider the following issues.

At a minimum, data should be checked for appropriate data type, acceptable ranges, and reasonableness, thereby ensuring that correct data is not rejected.

The use of diverse coding schemes can be an effective means of implementing redundancy. However, diverse methods do *not* eliminate the possibility of common-mode failures, and they add complexity to program flow, which can complicate safety concerns.

Diverse coding schemes can include any or all of the following methods:

— Diverse design (code, version)

— Diverse data

— Temporal diversity

The use of exception handling can be a sort of double-edged sword. On the one hand, “catching” exceptions can facilitate bringing a failing system into a known safe state. On the other hand, (a) unintended consequences are possible, such as unwinding the stack and thereby creating a situation in which resources allocated before the exception might not be freed, or (b) aborting a program, which could mean that other programs might not be able to acquire the resource, thereby resulting in a resource leak.

A common hardware mechanism used to enhance robustness is a watchdog timer. If the code does not update the watchdog within a set time, the system is assumed to be in an indeterminate state, and appropriate actions are taken (usually resetting the processor). In the absence of a hardware watchdog timer, or in conjunction with it, the code can also initiate hardware actions to preserve the overall system robustness. The code can monitor the state of the system and, when it detects a serious problem, initiate a nonmaskable interrupt to bring the system to a known safe state.

Unit testing

Adequately challenging risk control measures at higher levels of testing is often difficult. Therefore, in general testing, it is important for unit and white box testing to focus on challenging risk control measures to the greatest extent possible.

Unit testing typically emphasizes verifying that code works as intended for an expected use at some well-defined component boundary.

Sufficient resources should be allocated to verifying safety-related code. This effort is a bit contrary in nature, because the objective is to get the component to fail and then verify that the code implements the appropriate action. It is important to have this assurance at the unit-testing stage because, during the integration process, component interactions and system states become much more complex. This complexity leads to difficulties in developing test cases that exercise all safety-related code at the component level.

Engineering analysis

In general, dynamic testing techniques are unreliable in finding indirect causes of hazards and verifying their associated risk control measures. For this reason, safe coding practices are important. These indirect software causes of hazards can be detected only by focusing on the implementation details of a program as they interact within the entire system. An examination of the software system in its entirety, looking for the particular causes of specific problems, is required. Because an exhaustive analysis of every detail in a program and its interaction with all other items in the code is generally impossible, analysis is focused on specific areas that are known to cause problems.

Dynamic data structures such as those described below are particularly problematic to verify and are routinely the source of safety-related problems:

— Stack depth—Determine the worst-case stack depth and ensure that adequate stack space has been allocated.

— Heap usage—If a heap is used, ensure that adequate heap space is available under worst-case conditions. Ensure that when the system is used for an extended period of time, heap fragmentation does not cause memory allocation requests to fail.

— Watchdog usage—Locate all places in the code that reset the watchdog counter, and ensure that the watchdog will not time out under all correctly operating scenarios.

— Shared resource usage—Analyze the use of all data (and hardware resources) used in independent sections of the code to ensure that adequate protection measures have been applied to shared resources to prevent access conflicts (race conditions).

— Queue and buffer size—Use best practices available in determining queue and buffer sizes (queuing theory). Verify that queue and buffer space is not being exhausted. For example, check that some percentage of initialized queue or buffer memory maintains its state.

Multitasking systems that lock shared resources may have errors that are very difficult to detect. The following analyses may be required:

— Priority inversion—Use priority-inheritance and other design techniques to ensure that unbounded priority inversion can never occur.

— Deadlock—Ensure that deadlock can never occur as a result of locking shared resources. The four conditions for deadlock are

  1) mutual exclusion,

  2) nonpreemption,

  3) wait-for condition, and

  4) circular wait.

— Reentrancy—Locate all functions that are called by more than one independent section and verify that they have been designed to be *reentrant* (called again before the previous invocation has returned).

Systems that have hard real-time requirements must be analyzed to ensure that they always meet their time constraints under all conditions. The following *timing analyses* may be relevant:

— Processor utilization—Using the frequency of the periodic tasks and the least time between recurrence of all asynchronous tasks and events, ensure that the processor utilization is within appropriate limits.

— Interrupt latency—Determine the maximum interrupt latencies, and ensure that they are acceptable.

— Scheduling—Form the worst-case schedule of all interrupts, tasks, and events, and ensure that it is achievable.

— Timing constraints—Analyze all interrupts, tasks, and events, and determine whether they meet all of their timing constraints. These time constraints are

  1) event response time,

  2) task completion time, and

  3) periodic task jitter in release and completion times.

Pitfalls

— Believing that the best coding or testing process, practices, tools, or employees can make up for a poor, inherently unsafe, or overly complex design

— Using inexperienced developers for critical code development

— Failing to define and require specific defensive programming practices

— Relying on dynamic testing exclusively, without performing code inspections or static code analysis, especially on critical components

— Lacking control over compilers and other development tools that result in inadvertent code changes

— Deviating from the design without understanding the relevance of the design requirements to risk management

— Running unit tests on critical components only once, early in the development process, without repeating them as part of regression testing

— Focusing testing exclusively on dynamic, black box, system-level techniques and not performing static and dynamic white box verification

— Relying on programmers to detect all significant defects in their own code

6.2.6 Integration, system, and validation testing

This section addresses several sections of ANSI/AAMI SW68:2001:

— Software integration and testing

— Software system testing

— Software validation

SW68 requirements

Integration, system, and validation plans shall be established for all software, including off-the-shelf software. Regression testing will be performed if incremental methods are used and when changes are made. Tests shall address exceptional, stress, error-handling, and worst-case conditions and shall include testing for specific intended uses, types of users, and intended platforms and environments.

Risk management perspective

When planning and executing integration, system, and validation testing, one should allocate proper resources to verifying (software) risk control measures and other safety-related code under a range of normal, abnormal, and stress conditions. Similar attention should be given to any regression testing performed. Some degree of ad hoc and user testing should be performed to help identify potential causes of hazards that may have been overlooked.

Safety-related code components should be tested under a wide range of conditions, not just under a single condition. It is often under abnormal or stress conditions that failures occur. If one has to apportion resources, the most critical safety-related code should receive the most testing under the broadest range of conditions.

When personnel develop integration and test plans and specific test cases, all too often they place emphasis on verifying conformance to functional requirements under conditions of normal use. It is important that they consider and plan for the verification of risk control methods as well. Software risk control methods may range from several lines of code expressed as, for example, “throw–catch” logic, sprinkled throughout a software unit, to complex interrupt handlers, to software units using redundancy techniques, to hardware–software interfaces (firmware). This kind of software system (safety) diversity found in today’s hybrid medical systems requires equally complex test cases to verify that risk control designs will perform as intended. The complexities and couplings brought about by integrating this diverse code can make the development of risk control test cases a daunting task, consuming considerable development resources. Even so, all risk control methods should be verified, if practical. Any relevant failure history (from similar devices or other industry sectors) should be incorporated into these plans. The goal is to expose errors, omissions, and unexpected results that could cause hazards.

Many methods are available to facilitate assurance that risk control methods are likely to perform as intended, some more resource-intensive than others. No single method is sufficient. Some of these methods are identified in Table 5.

Table 5—Methods to facilitate assurance that risk control methods are likely to perform as intended

Static analysis	Dynamic testing	Modeling
— Walk-throughs[a]	— Functional testing	— Environmental modeling
— Design reviews	— Timing and memory tests	— Timing simulation
— Sneak circuit analysis	— Boundary value analysis	— Use case and user workflows
	— Performance testing	
	— Stress testing	
	— Statistical testing	
	— Error guessing	
	— Thread-based testing	
	— Use-based testing	
	— Cluster testing	

[a] Establish traceability among risk control methods, test cases, and test case results.

Testing should account for a variety of types of tests (e.g., stress, boundary, timing, power failure, fault, and COTS failure) to ensure that safety-related software is tested under an adequate range of conditions rather than focusing exclusively on requirement-based testing.

Testing should be performed to account for all hardware and software platforms (versions and variants). Testing should also include or simulate actual users in their intended environment to ensure that workflows tested are representative of real usage sequences and to verify safety-related human factor design decisions.

Traceability should be reviewed to ensure that all safety-related tests have been executed and that these tests cover all safety-related components, including COTS, requirements, and risk control measures.

Any tools related to testing or configuration management should be controlled and qualified, or the testing will be considered unreliable for safety-related components and requirements.

Testing driven by risk analysis and knowledge of risk control measures can result not only in a safer system, but also in testing that is more efficient with less unnecessary redundancy (particularly in regression testing).

Pitfalls

— Not using risk analysis information to plan testing or for training testers

— Depending on testing as a risk control measure even though 100 % testing is impossible

— Not creating the system or software failure modes and defects as part of testing to verify risk control measures

— Using automated tools for testing that are not qualified or validated and controlled, and relying on the results

— Failing to properly analyze the code to detect errors that testing cannot reveal

6.2.7 Software release

SW68 requirements

ANSI/AAMI SW68:2001 requires the following as part of the software release activity:

— Software testing must be completed, and the results must be evaluated.

— The versions of software must be documented, and all software documentation must be completed.

— The procedure and environment used to create the released software must be documented.

— Software, source code, and documentation must be archived for the life of the device.

— Procedures to handle, protect, store, label, package, and deliver the software must be established.

Risk management perspective

When determining that all software testing has been completed and the results evaluated, one should adopt a risk-based perspective that focuses on:

— Verification that all software-implemented risk control measures are traceable, from the risk–hazard analysis definition of the control measure to the test or tests that verify the correct implementation of that control measure.

— Verification that all tests that trace back to risk control measures (i.e., safety-related tests) have been executed with acceptable results.

— Analysis of all defects in safety-related tests that are deferred for correction. One should verify that the defects do not represent a possible cause for a hazard or are of sufficiently low risk as to justify the deferral of their correction.

— Analysis of all defects in non-safety-related tests that are deferred for evaluation. Verify that those defects do not represent new causes for hazards that were not previously identified.

— If a number of generations of regression tests were compounded for the set of final test results, analysis of the version spread for the safety-related tests. One should verify the validity of test results for versions other than the final release version and investigate whether any changes were made in the general “vicinity” of the safety-related code that might suggest reexecution of the tests for that safety-related code.

Pitfalls

— Failure to bring the version of released documentation into agreement with the released code; such failure will mislead the development and test team on future releases of the product and could lead to overlooking hazards, their causes, or their “implied” control measures that are not fully documented

— Failure to involve those with adequate clinical knowledge in the evaluation of deferred anomalies

— Evaluation of the significance of deferred anomalies solely on the basis of the functional symptom detected rather than on a complete root cause analysis to determine all potential side effects under a range of conditions

— Failure to clearly present rationales for deferring correction of safety-related issues

Documentation and control of the procedures, environment, and tools used to build the software have relevance to risk management because any unexpected changes to the software attributable to these factors represent potential causes for hazards. Mistakes in the build process or changes in tools used in the build process could result in unintended software changes. Any assumptions about test results from prior versions are not valid if changes to the software cannot be controlled because of changes in the build procedure or environment.

Pitfalls

— Hidden environment settings that are not controlled and that can result in unpredictable builds

— Inadequate control of versions of any tools, especially compilers—Simple reliance on a version printed on the label of the box in which the tool came or on a high-level version displayed without revision and patch level can be inadequate. Internet-based automatic updates to tools and operating systems make it difficult to know, let alone document, the versions of tools and operating system components. This problem has become complex enough that it is becoming increasingly common for device manufacturers to “freeze” an entire development

station at the end of a project to guarantee that they can have an identical starting point for future development work on a product.

— Lack of control of the documentation environment when managing changes in the software development environment—Inaccurate documentation and traceability could lead to the loss of links and risk control measures or to failure to adequately verify safety-related code.

Archiving the source code, development environment, and documentation files is important for many reasons. One is to ensure that all of the information needed to perform a full failure investigation is available for as long as that version of software is in the field (regardless of the current released version) should field safety problems arise. Although business software vendors may withdraw support for obsolete versions, medical device manufacturers need to be able to perform investigations of field safety issues and perform timely corrections for as long as the device is in use. Copies of the tools that were used to create these files will also need to be archived to make use of the source code, development environment, and document files in the future. Be especially careful to archive any off-the-shelf packages that are embedded into the product and libraries that may be packaged with the compiler, assembler, or linker.

Pitfalls

— Overlooking the fact that once a third party is no longer distributing a specific COTS version, those versions will not be available for failure investigations and field corrections—Medical device users often do not update COTS software to current versions and use their devices for many years.

— Losing the ability to create the specific software version because the specific tool and version of the tool (such as a compiler) were not archived

— Not taking into account that the life of the device might well be longer than the life of your current archive media—As old archive media are replaced with new, one should plan a migration path for old archives onto the new media.

Labeling of software both on displays and on packaging can allow the user to play a role in ensuring that proper versions of software are executing and that patches are applied. Doing so facilitates resolution of field safety issues and permits faster and more effective root cause analysis, workarounds, and corrective action.

Devices with visible user interfaces should have an identifiable release or version identifier displayed on power-up or through a simple user selection. The same version identifier should be displayed or labeled on whatever medium the software is delivered (diskette, CD-ROM, NVRAM, etc.). Device users and service personnel should be able to identify device software versions easily. This labeling is especially important for software safety if defects are identified after the product is released to the field or if installation of incompatible versions of different components is possible.

A configuration management plan should be put into place to identify which versions of subsystem software and hardware components have been tested together. When feasible, software interlocks in the device, which automatically check for compatible versions of subsystem software, are helpful.

Pitfalls

— Vague version requirements such as “latest version” or “Rev. 2.0 or higher” for off-the-shelf software and platform software—Potential effects of differences cannot be predicted and may introduce new causes for hazards or defeat implemented risk control measures, depending on system and software design.

— Omission or lack of control of versions of any programmable parts’ firmware in the configuration management or labeling procedures—Software-readable versioning is preferred for automatic testing and lockout of invalid versions. Parts labeling should be required for any programmable part not downloaded by the device software and not readable by the device software.

— Inadequate formalization and quality control of administrative aspects of copying, labeling, and shipping

6.3 ANSI/AAMI SW68 maintenance process

ANSI/AAMI SW68:2001 defines maintenance as one of only two primary processes.[36] Within this process, a number of activities are required by ANSI/AAMI SW68:2001. The following subsections correspond to each of the

[36] Note that ANSI/AAMI SW68:2001 also defines a number of other supporting processes, such as documentation and problem resolution, that are not addressed explicitly in this report.

maintenance activities defined in ANSI/AAMI SW68:2001, and the risk management aspects of each of these activities are discussed. As in section 5, sample pitfalls are also provided.

6.3.1 Process implementation

SW68 requirement

ANSI/AAMI SW68:2001 does not define a specific maintenance life cycle. It does require formal planning and implementation of each maintenance activity. Note that the development process automatically applies to modifications unless a specific maintenance process is defined for implementation of modifications.

Risk management perspective

The approach to risk management for corrective, adaptive, and perfective changes should be formally defined, including defining how the design baseline risk management information will be used as input and how it will be updated. The approach should specify that reported field problems receive prompt investigation to determine the root cause because analysis of just the reported symptom may not reflect the true risk of the underlying defect.

Pitfalls

— Establishing a maintenance process that does not have a clear approach to risk management of changes

— Establishing risk management for changes that addresses only the intended functionality of a change and not affected components and their associated risks

6.3.2 Problem and modification analysis and implementation

This section addresses two activities—analysis and implementation—listed separately in ANSI/AAMI SW68:2001.

SW68 requirements

Risk management perspective

Software changes occur after device release for many reasons. This phase, which can be referred to as the postproduction phase of the device life cycle, focuses on software changes associated with error correction, adaptations required because of changes in the software environment or platform, and changes because of newly identified customer requirements related to the device.

When field problems occur or when device improvements are desired, it is common to implement them in software rather than in hardware because of the perceived speed of implementation and distribution, as well as the negligible effect on product unit cost. Such perceptions overlook issues of software complexity and side effects that can affect safety.

Effectively analyzing a problem, developing a correction, and assessing implementation side effects require experienced personnel. Inexperienced or new staff members may not be able to determine subtle component (logic) interrelationships, assumptions, or environmental constraints (e.g., clinical use, target population, or social expectations). Many organizations allocate their most experienced personnel to new device designs, failing to recognize the importance of having experienced personnel implementing device changes.

Pitfalls

— Assuming that a small functional change cannot affect safety

— Expanding the use of the medical device to new target populations, new disease indications, new types of users (e.g., nurses instead of surgeons), or new platforms without reexamining the existing risk control measures and the appropriateness of the user interface

— Setting problem resolution resource priorities on the basis of the reported symptoms of field problems without determining root causes and potential side effects

— Relying too much on software changes to correct hardware problems

— Using inexperienced staff members, who will not understand nuances, to maintain code

6.3.2.1 Risk management of changes

The specific activities that precede any software release depend on the type of change and the extent to which the safety integrity of the software is affected. Sufficient regression analysis and testing should be conducted to

demonstrate that portions of the software not involved in the change were not adversely affected. This analysis and testing is in addition to testing that evaluates the correctness of the implemented changes.

Some of the specific activities to be conducted when changes are made are as follows:[37]

- Review the trace between the system hazard analysis and the software hazard analysis to ensure that any software changes made during postproduction do not violate the safety integrity originally designed into the system.
- Evaluate if the change introduces any new unintended uses, target populations, or hazards.
- Evaluate if new causes for hazards are being introduced.
- Evaluate if any existing risk control measures could be compromised.
- Evaluate if any new risk control measures are needed.
- Evaluate if changes to seemingly innocuous functionality could have side effects on safety-related code.
- Include in the documentation for software changes the reason for the change, the solution, and the activities that prove that the changes are effective and adequate.
- Revise the software and system risk management documentation as needed to incorporate new hazards, causes, or risk control measures identified.
- Include in the software plan the activities necessary and commensurate with the severity of the changes.
- Make the software changes go through the same elements of the process used for developing the software in the first place, although some activities may be abbreviated if appropriate. This abbreviated process, however, should be driven by a regression analysis on the software modifications and the nature of the changes. All verification and validation activities should be driven by the results of the regression analysis.
- Evaluate all system and software changes to ensure that
  - no new hazards are introduced,
  - existing safety requirements and their implementation are not compromised with the changes,
  - the changes made to the software do not affect system-level requirements, and
  - the right people are involved when reviewing the changes, and they double-check the intended use requirements when evaluating the changes.
- Apply software configuration controls to ensure that correct revisions are tested and released.

Some or all of the phase activities discussed earlier may apply depending on the nature and criticality of the changes.

In many cases, regression testing should include reexecution of tests to verify that the changes have not affected critical risk control measures and highly critical code. This reexecution should be done under a range of conditions and often can be adequately performed only by including relevant white box-based unit or integration tests as well as system-level tests.

Pitfalls

- Software designed for nonclinical purposes (e.g., billing) contains clinical data, which is subsequently distributed for clinical purposes without appropriate risk management
- Inadequate regression testing of safety-related requirements and risk control measures based on assumptions that the changes will not have unintended side effects
- Inadequate communication of the safety significance of a change to those responsible (including users) for distribution and installation of the update
- A maintenance process lacking adequate rigor for the number and magnitude of the changes over time and the risks involved

[37] Although described in the postproduction section, these activities are also appropriate for changes during development.

— Inadequate consideration of the effect of software changes on user documentation and labeling

6.3.2.2 Types of software changes during postproduction

Changes made to correct errors and faults in the software are *corrective maintenance*. Changes made to the software to improve the performance, maintainability, or functionality of the software system are *perfective maintenance*. Software changes to make the software system usable in a changed environment are *adaptive maintenance*. Risk considerations vary with the type of maintenance (or device evolution) to be performed as discussed below.

6.3.2.3 Adaptive maintenance

For a device containing software, any changes made to the hardware, accessories, operating system, COTS software, patches, compiler upgrades, tools, development, or operating environment may affect the safety of the original design.

For example, allowing the software to be run under a new operating system without examining the failure modes and vendor bug lists for the new operating system could lead to additional failure modes that could result in a hazardous situation, unless additional risk control measures are implemented.

6.3.2.4 Perfective maintenance

When new software functionality is added, the system and software risk management information should be updated in all relevant design documentation. Any new hazards, causes, risk control measures, or needed changes to activities should be incorporated.

An example would be the addition of a new feature to a point-of-care blood analyzer on an infusion pump. The new feature would automatically adjust the infusion rate on the basis of blood analysis information transmitted over the interface. Such a change would have major ramifications that could compromise the adequacy of existing risk control measures and even the overall design requirements.

6.3.2.5 Corrective maintenance

A software change to implement a fix should be analyzed for its effect on system and software safety requirements. Changes should not compromise the safety integrity of the original design.

For example, a bug fix to solve a problem with the speed of the communications interface speed could inadvertently eliminate a risk control measure such as a message checksum.

7 Perspective 4: Soft factors in software risk management

A major part of this report deals with the formal aspects of software development and risk management processes and includes specific lists of potential causes and risk control measures to prevent software from leading to a hazardous situation.

As with software development and quality assurance techniques in general, it is recognized that what are often referred to as “soft factors” (such as domain knowledge, team dynamics, and programming experience) can be of equal or greater importance than methodologies and process controls. Although it is difficult to measure soft factors, their contribution to device safety is generally recognized as being extremely important. The next few paragraphs are intended to identify key soft factors in ensuring software safety.[38]

7.1 Intended-use and domain knowledge

It is important that the right cross-functional resources, and domain experts, are brought to participate early in the design process to help establish the various means, modes, and circumstances of device use that can result in user

[38] The comments related to experience of various types in the following paragraphs are not intended to imply that every member of the team involved in software development and verification for a medical device must be experienced in all of these areas. They do, however, intend to identify that the team overall should have these experiences and that the members of the team with experience in certain areas should be encouraged and allowed to participate in the aspects of the development where their experience might contribute to catching potential software safety hazards. There are often teams with an adequate mix of experience, but each of the individuals is focused on his or her particular area of the medical device, and so the experience exists in the team but is not used across the team. There are other instances in which teams do not have adequate experience and perhaps it is not possible to acquire full-time team members with the experience needed. In such cases, appropriate actions include use of experts from other groups (within the company or external to the company). This action would provide appropriate experience in key reviews throughout the project to focus on those things most likely to be missed by the team members, given their particular mix of experience. Such an action should be factored into the project in quality planning.

or patient harm. Such resources include those experienced with the intended or foreseeable use or misuse of the device, designers familiar with the medical device environment, safety engineers, verification and validation experts, and manufacturing experts. It is essential that, as a group, this team have a full understanding of the potential intended-use (clinical) harm, the associated hazardous situations, and the role that software could play in creating a hazardous situation. The involvement of domain experts, and the degree and type of their interaction with the software engineering and test staff both formally and informally, can have a profound effect on the software risk management process and the ultimate safety of the device. By sharing their domain-specific knowledge, all of these experts have the opportunity to proactively assess their contribution to the device development from a safety perspective. This is equally true in a postmarket analysis context as well.

It is this knowledge that allows each team member to focus on the aspects of his or her tasks that have the strongest likelihood of contributing to a hazard. Without this knowledge and awareness, individuals may focus only on particular functional attributes of the software without addressing safety attributes of the software. Although the formal risk management process is intended to uncover all hazards and their causes, the fact is that the process is not foolproof for complex software systems. Informing team members will increase the chances of uncovering risk-related anomalies as they go about doing their daily development activities. Further, it is this knowledge that helps ensure that test planning—particularly regression testing—is focused on those areas most likely to contribute to hazards or to introduce side effects that can contribute to other hazards.

Knowledge of intended use and direct and indirect clinical hazards can be increased through training and expert input on a given project. However, clinical experience in development of similar devices provides an awareness that can help ensure that significant hazards, causes, and their risk control measures are identified during the formal hazard analysis and risk control activities.

7.2 Team dynamics

Team dynamics is a soft factor that relates to the size of the team involved in the definition, development, and testing of the software and to the team structure, culture, and communication among the team members. National and international software development life cycle standards, including ANSI/AAMI SW68:2001, attempt to address the importance of development team communication by requiring disciplined processes and consistency in design documentation. Although these disciplined processes help with communication, it is often informal communications and team member buy-in to the risk management process that contribute to each team member's understanding of his or her contribution to comprehensive risk assessment and control. Team dynamics (such as paired programming), culture (such as safe coding practices), and team continuity throughout the life cycle can make a critical difference in achieving safe device software.

7.3 Management

Management commitment and culture is a soft factor in terms of establishing relative priorities and incentives for the different project objectives and establishing the way in which those objectives are communicated and enforced. Most projects have a mix of objectives, including schedule deadlines, quality level, feature set, and—almost silently stated—safety. If safety is not communicated as a primary objective in the software development process by management, the likelihood of developing safe software is diminished. The way in which management communicates objectives, evaluates staff members, provides feedback, and sets priorities is critical to ensuring the development of safe software. For example, does management establish a positive environment for identifying safety issues and problems even if they are discovered late in the design life cycle or outside the formal process? Is a culture of naive optimism fostered with regard to safety? Although most individuals will not intentionally implement unsafe software, regardless of management schedule or cost pressure, developers, testers, and others involved in requirements analysis are likely to spend less time in analysis, design, coding, and testing when under direct or extreme schedules. Such an environment will result in missing key hazards, the causes for these hazards, and the required risk control measures. In other words, when rushed, staff members will proceed without being aware that they may be creating unsafe software. When there are known software defects that are unsafe, individuals and management are likely to take a firm stand, regardless of schedule pressure. But when schedule pressure prevents staff members from spending adequate time on risk management-related activities, a false sense of optimism results, and unsafe software is released.

7.4 Programming experience and attitude

Programming and testing experience and attitude represent another soft factor. Inexperienced developers and testers often believe that software will function correctly, but if it does not, any problems will be detected during testing. Inexperienced staff members may not have learned the many ways in which software can fail. They may not realize that the requirements themselves (on which the software is based) could be wrong, and they may not know how small a percentage of the software logic testing can actually cover. So they have a more optimistic view of potential outcome. Also, there is often a tendency to emphasize functionality without making a distinction regarding safety.

Developers should consider the effects of an error, the possibility of similar defects elsewhere in the software, and the best approaches to fix the defect even when under schedule pressure.

On the one hand, experienced staff members may be more realistic because of their past design or defect experiences. On the other hand, inexperienced staff may be more objective in some cases. Therefore, a mix of experienced and inexperienced staff members is needed, with the experienced team members providing mentoring and oversight and the inexperienced team members challenging past assumptions.

Either through experience with past failures or because of an inherently objective or skeptical attitude, staff members who approach development and testing with a belief that things can and will fail in unexpected ways are a necessity for performing effective risk management.

Often, management assigns junior-level developers to postproduction (maintenance) activities. Changes made to software can have unknown side effects. Experienced and knowledgeable developers should be assigned to exercise oversight of those activities.

7.5 Technical knowledge

Software development, testing experience, and technical knowledge of the specific software platform, language, development tools, third-party software, and device environment are essential for identifying failure modes and their side effects within the risk management process. Without such knowledge and experience, potential causes for hazards could be missed, and methods of controlling associated risks could be omitted.

Annex A

Direct causes sample table

This table is not exhaustive but is an aid to the thought process used for developing safe and effective software. It lists functional areas of software often related to hazards and provides examples of failures that are potential causes for hazards. It also provides example questions to ask during software development that can help lead to improved risk control

Some of the information in this table may not apply to all medical device software. Their relevancy for specific medical devices depends on the intended use of the device, the system-level design of the device, the role of the software in the device, and other factors. This table is intended as a starting point.

Software functional area	Sample causes for hazards	Questions to ask
Alarms and alerts		
Priority	— Lower-priority alarms mask the display or audible output of higher priority alarms. — Critical alarm does not “latch.”	— Do specifications identify how the system reacts to multiple alarm conditions? — Are there multiple levels of alarms? — Do higher levels override the audio for lower levels? — Should any of the alarms latch until the user can acknowledge the alarm?
Protective measures	— Protective actions are not clear for each alarm condition or alarm category. — Safe termination of protective actions once alarm clears is “not specified.”	— Does the protective action create a usability problem (i.e., can the user safely navigate from the protective action)?
Shutdown/failsafe/recovery	— Fail-safe action is not adequate. — Fail-safe action creates new hazards.	— Is fail-safe action appropriate for intended use? — Have clinical staff members reviewed fail-safe scenarios? — Is fail-safe state apparent to the user?
User interface	— Crowded or poor user interface masks “real” alarm condition. — Actions to take in response to the alarm are not clear.	— Have clinical staff members reviewed protective measures for usability?
Log	— Persistent error occurring is signaling a pending failure. — Logged alarms are associated with wrong patient.	— Are detected errors logged? — Is log large enough? — Is log storage reliable? — How is log cleared? — Is the user aware when log is cleared?

Software functional area	Sample causes for hazards	Questions to ask
Audible	— Background noise suppresses alarm sound. — Alarm volume is so loud that operators find alternative means to disable alarm. — Audio system has failed, but user is unaware of failure.	— Has intended-use environment been considered for audible alarm design? — Have users been involved in developing requirements for user interface design? — How is audio system verified per power-up or per patient?
Critical power cycle states		
Data integrity	— Nonvolatile writes are in progress at shutdown. — Critical parameters are not preserved to resume treatment on power cycle. — Critical parameters are inadvertently overwritten by latent software defect during operation.	— What happens to a memory write that is in progress when power is lost? — Is the software made aware of impending power loss? — Is nonvolatile storage verified on power-up? — Are critical parameters checked before use?
Reset	— Failure to resynchronize with component occurs after unexpected reset. — Persistent resets go undetected but could be signaling a pending failure.	— Is reset being used as a risk control measure? — Will I/O (on/off) control be compromised during reset cycle? — Is user made aware of resets?
Recovery	— Device is unavailable for intended use while power-on initializations are occurring. — Nonvolatile failures occur at power-up—what should the user do? — User is not aware that critical setting is restored to factory defaults.	— Is recovery time a safety issue? — Is availability of device a safety issue? — How is nonvolatile storage affected by fail-safe protective measures?
Power modes	— Audible functions or other critical user interface functions are not available during low-power states. — Transition to low-power states inadvertently disables critical interrupt.	— Are any risk control measures compromised during low-power modes? — Has software recovery from low-power modes been considered as a possible start-up state for activities?
Critical user controls and usability		
Adjustment, navigation, and selection	— User interface software process handles value change, but control process never gets new value.	— Is user notified if adjustment is made to a new value but never selected or confirmed? — Should the parameter being adjusted require a two-step operation for change?

Software functional area	Sample causes for hazards	Questions to ask
Data entry	— User inputs out-of-range value. — User inputs in-range but unintended value.	— Should user be prompted for confirmation? — Does software check data entry for validity? — Does software require supervisory user log-in to confirm highly critical inputs or error overrides?
Accessibility	— Shutoff control is “buried.” — Touchscreen controls do not function with surgical gloves.	— How many “layers” must user navigate to access safety-related function?
User interface design	— Alarm “automatically” causes display to switch screens. — Error conditions are color coded. — User cannot determine alarming condition.	— Have all automatic screen switches been evaluated? — How will color blind operator interpret error message? — Have users been involved in developing requirements for user interface design?
Display		
Diagnostic images	— Orientation reversal — Patient misassociation	— Is there a technique being used to ensure correct orientation of image? — How are images associated with patient?
Diagnostic waveforms	— Improper “display” filter — Aliasing — Distortion — Scaling errors — Timebase errors — Loss of compression	— What frequency content is required for display? — Have clinical staff members reviewed that requirement? — Has display filter been fully characterized (i.e., what is rejected and what is passed over full range of inputs)?
Hardware controls		
Algorithms Motor control	— Integral windup — Aliasing — Timing — Overflow — Porting error	— What is the sampling rate? — If there is proportional integral derivative control, is integral gain limited? — Has algorithm been characterized over the full variation of manufactured hardware? — If there is feedback control, what checks are made to the validity of the feedback signal? — Have all data types been evaluated for the microprocessor and compiler in use?

Software functional area	Sample causes for hazards	Questions to ask
Applying energy to distinguish defibrillator from X-ray	— All “gates” are not checked initially and continuously during therapy. — Safety system has failed, but user is not aware.	— Are all assertions continuously verified on a scheduled basis? — Could a “common mode” error exist in therapy drive software and safety monitor software? — Are safety monitors verified per power-up or per patient?
Discrete	— Bit is struck. — Changes in bit go undetected because of polling interval.	— Does software detect that discrete is stuck (never changes)? — Has polling rate been discussed with system or hardware engineer?
Calibration or self-test, range checks, and assay calibration (software-specific calibration)	— Poor instructions for use leads user to set up device incorrectly for calibration, resulting in erroneous calibration constant. — Auto-zeroing operation done with nonzero signal (i.e., unexpected pressure in cuff or in line) or force on transducer.	— Does software perform a reasonableness or validity check of calibration values (i.e., slope or offset)? — Is user aware of auto-cal or auto-zero?
Hardware fault detection	— Hardware fault is detectable but never reported to user, and device continues to be used in this condition. — Hardware fault occurs after power-up, and software only checks for hardware fault at power-up.	— Are all hardware faults reported to user? — Should hardware fault be checked at power-up, before each treatment or session, or on a continuous basis such as once per second?
Self-cleaning	— User aborts cleaning or disinfection process midcycle.	— Does software enforce completion of cycle? — Can software detection of incomplete cleaning or disinfection cycle be defeated?
Fluid delivery	— Improper calibration. — All “gates” are not checked initially and continuously during therapy.	— Are all assertions continuously verified on a scheduled basis? — Can safety systems be defeated (i.e., pump operated without tube in safety clamp)?
Life support	— Safe state is never defined, and software inadvertently “returns from main.” — Out of many shutdown paths, one does not disable interrupts. — No backups for life-support functions.	— Have safe states been defined and analyzed thoroughly, including effect of delay of treatment and safe shutdown sequences on the range of target populations (e.g., adults, neonates)? — Can software support a “limp home” mode and inform user of situation?

Software functional area	Sample causes for hazards	Questions to ask
Monitoring		
Decision	— Common mode error occurs in monitoring software. — Race condition causes incorrect decision result.	— Have therapy drive and therapy monitor software been developed independently? — Has software design eliminated or minimized possibility for race condition for this decision point?
Deactivation	— Control system is unaware that monitoring system has shut down subsystem. — Networked system logging parameter is deactivated at device.	— Is control subsystem aware of monitor subsystem actions? — How are deactivated parameters communicated to user or networked systems?
Display	— Displayed value is not being updated, but user is unaware. — Display writes are performed at two or more priority levels.	— How is user made aware of “frozen” display? — Is video “context” saved before preemption?
Measurement	— Data acquisition timing or sampling rate is wrong.	— Is sampling rate appropriate for frequency content of signal? — Is the measurement value stored in consistent units throughout software layers?
Interfaces		
Bad argument passing	— Function passes value in microliters, but driver expects value in milliliters. — Bad pointer is passed. — Pointer to volatile memory is passed, and values are lost before processing.	— Does each software function verify passed parameters? — Does software language support more robust type checking? — Is software designed with consistent units for values throughout the software package? — Are arguments modified at higher-priority processing layer?
Network	— Software enters endless loop, waiting on response from host computer. — Devices on network are given identical “names,” resulting in data misassociation. — Networking data processing dominates CPU cycles, starving safety or intended-use functions.	— Has software been designed to tolerate any physical network connection condition? — Can remote connection bring system “to its knees” by repeatedly sending commands or bogus data? — Does device check that the network names are not already in use?
Data		
Clinical information system	— System accesses wrong patient record and display does not make it apparent. — System stores data from patient in wrong archive.	— Can there be display of multiple independent identifiers to put the user in the loop of detecting mix-ups? — Can critical identifiers be embedded with actual data as a cross-check?

Software functional area	Sample causes for hazards	Questions to ask
Reports	— Report provides incorrect data or identifies it in the wrong sequence or without units.	— What reports will be used for clinical purposes? — What is the severity of the hazard if the data is incorrect? — How likely is it that a clinician would notice the problem?
Databases	— Data is corrupted because of side effects from system-level failures or COTS components.	— How can data corruption be detected before data is used? — Can this be done with each use instead of only at start-up?
Diagnostics		
Algorithms	— Algorithm error due to invalid sensory or user inputs or logic error	— Can in process diagnostics or redundant cross checking be implemented to detect such errors?
Decision-making diagnosing software	— Artifact detection indication suppresses asystole indication on the display.	— Has the hierarchy of alarm indications been thoroughly reviewed by software engineers and clinical staff members?
Data reduction	— Arithmetic precision errors result in invalid result. — Algorithm uses or displays incorrect units.	— What arithmetic precision is required? — How should mathematical formulas be coded to ensure adequate precision?
Automated preventive maintenance	— Background diagnostic modifies data temporarily but while application code is retrieving the data for actual use. — Background diagnostic interferes with proper timing.	— Are application processes locked out during diagnostics at appropriate times? — Are diagnostics locked out during critical timed cycles?
Security		
Configuration Options	— Protection of access to critical configuration parameters or data is lacking or inadequate.	— What data is critical and should not be modifiable by the user or should require supervisory authorization to do so? — Is an audit trail needed?
Functional Access	— Protection of access to controls of therapy or instrument operation is lacking or inadequate.	— Should operators be required to log in before operation? — Can patients inadvertently operate the device?
Interface Access	— Protection from data and commands submitted through communications interfaces or networks is lacking or inadequate.	— What should be allowed remotely? — Should assumed controls at the remote systems be relied on, and, if so, why?

Software functional area	Sample causes for hazards	Questions to ask
Performance		
Capacity/load/ response time	— Critical timing is affected during peak load. — Sequence of transactions, inputs, and outputs is affected. — Motor control is affected under peak system loads.	— During peak load or when capacity limits are reached, will data or timing be lost or affected in undetectable ways? — Will inputs and outputs be queued up in a correct deterministic sequence under peak loads? — Have critical functionality and risk control measures been tested under these stress conditions? — Have risk control measures been implemented to detect limits? — Can interrupts be set to segregate critical time-constrained functionality from other functionality?

Annex B

Indirect causes and risk control measures table (failures due to unpredictable behaviors)

This table is not meant to be comprehensive, but is a guide to the thought process used for developing safe and effective software. The table provides examples of failures that are potential causes for hazards, plus possible risk control measures to consider. Some causes or risk control measures may not apply to all medical device software. Their relevancy for specific medical devices depends on the intended use of the device, the system-level design of the device, the role of the software in the device, and other factors. This table is intended as a starting point; additional indirect causes and risk control measures identified for a particular application should by no means be excluded.

Common testing of requirements-based systems is often ineffective at identifying indirect causes or verifying technical risk control measures associated with them. The columns on the right of the table identify the type of static or dynamic verification methods that might be appropriate for each indirect cause.

		Verification types—Analysis: Static/Dynamic/Timing		
			Unit test	
			Inspection	
Indirect causes	**Risk control measures**			
Arithmetic				
Division by zero	Use run-time error traps and defensive coding.	✦	✦	***D***
Arithmetic errors	Use defensive coding, parentheses to force precedence, simple expressions, and complete specification of input data; and avoid mixed-sign expressions.	✦	✦	
Numeric overflow and underflow	Use range checks and floating-point data representations.	✦	✦	***D***
Floating-point rounding	Use robust algorithms.	✦		
Incorrect algorithms	Implement design review and simple and straightforward algorithms.	✦	✦	
Improper range or bounds checking	Use defensive coding.	✦	✦	***S***
Off-by-one	Use defensive coding.	✦	✦	
Hardware-related				
EEPROM usage—long access times, wear-out	Use special access modes (page/burst mode), write only when data change, cache writes; and update EEPROM only when power is lost.	✦		***T***
CPU and hardware failures	Implement power-on CPU check, program image CRC check, RAM test, clock check, watchdog check, nonvolatile storage check, timeouts and reasonability checks on hardware responses, and short-to-power/gnd checks on sensors; and test sensor response with known signal.		✦	
Noise	Debounce digital inputs; filter analog inputs; and ensure that all interrupts, used and unused, have interrupt service routines (ISRs).			
Peripheral interface anomalies	Provide start-up delays for ADC/DAC; verify that timing and other interface requirements are always met; and conduct reasonability checks.	✦		***T***

		Verification types—Analysis: Static/Dynamic/Timing		
			Unit test	
			Inspection	
Indirect causes	**Risk control measures**			
Timing				
Race conditions	Identify and protect (lock) shared resources during updates; implement a single, non-reentrant process to handle all accesses to the shared resource; and conduct shared resource analyses.			***S***
Missed time deadlines	Specify timing requirements; use appropriate real-time design algorithms; eliminate priority-inversion and deadlock issues by design; avoid nondeterministic timing constructs; and verify all timing assumptions in completed code.			***T***
Missed interrupts	Implement simple interrupt architecture (fewest priorities) and ensure that ISRs are short and fast and that disabling of interrupts is infrequent and of short duration; and use an appropriate real-time design.			***T***
Excessive jitter in outputs	Ensure that ISRs are short and fast; avoid nondeterministic timing constructs; use an appropriate real-time design; and update all outputs at beginning of periodic task or in high-priority ISR.			***T***
Watchdog timeouts	Determine longest time between watchdog timer resets and set timeout appropriately; and set timeout to longest period possible while still avoiding hazardous situation.			***T***
Moding				
Abnormal termination	Use exit traps, run-time error traps, appropriate watchdog timer design, validity checks on all program inputs, and power-up self-test; remove all “debugging” code and other nonproduction features from program before release; and ensure that “special” modes (service, manufacturing) cannot be entered inadvertently.			***D***
Power-loss, recovery, and sequencing problems	Perform power-up checks for CPU, program image CRC, RAM, clock, watchdog, nonvolatile storage, peripherals, etc.; use an appropriate state design; initialization of time-averaged values, and reinitialization of peripherals; store or recover system state from nonvolatile storage; and use external voltage monitor and reset circuitry.			
Start-up/shutdown anomalies	Perform power-up checks (see above); implement proper initialization of peripherals and data; ensure appropriate nonvolatile memory usage; and ensure appropriate state design.			
Entry and exit of low-power modes	Implement appropriate interrupt design.	✦		***T***
Data issues				
Data corruption	Shadow RAM, block CRCs or checksums; encapsulate data access through functions; minimize global data; keep data structures simple; be aware of how compiler aligns data in structures; avoid casts (see also “Errant pointers” and “Use of intermediate data” below).			***S***
Resource contention issues	Conduct shared resource analysis (see also “Race conditions” above).			
Errant pointers	Use defensive coding; test for validity before dereferencing; use a strongly typed language; minimize use of pointers; and avoid pointer casts.	✦	✦	***S***

		Verification types—Analysis: Static/Dynamic/Timing		
				Unit test
			Inspection	
Indirect causes	**Risk control measures**			
Data conversion errors: typecasting, scaling	Avoid typecasts and use floating-point representations.	✦	✦	***S***
Incorrect initialization	Preinitialize time-averaged variables; and clear all data memory to "0" at power-up.	✦	✦	***S***
Averaged data out of range	Ensure that enough samples are taken before calculating average (especially at power-up); or preinitialize average to a known (or last) good value.	✦		
Rollovers	Conduct reasonability checks.	✦	✦	
Volatile data	Verify that volatile storage class is used for all data changed by hardware, ISRs, or different priority tasks.	✦		***S***
Unintended aliasing	Sample data at least two times faster than the largest frequency component of the signal (Nyquest criteria); and limit the bandwidth of the signal.			
Use of intermediate data[39]	Ensure that any data that is assumed to be synchronous in time is updated all at once; and conduct shared resource analysis.	✦		
Interface issues				
Failure to update display	Conduct continuous instead of event-driven update.			
Human factors—misuse	Use logs for reconstructing scenarios, context-sensitive help, and simple user interface design.			
Network issues (e.g., multiuser)	Perform load testing.			***T***
Hardware–software configuration errors and wrong drivers	Implement software development process and configuration management tools.			
Bad patches and updates	Program image CRC and version check at power-up; check protocol revisions; and observe expiration dates.			
Off-the-shelf failure modes: hangs or does not return, locks, interrupts too long, etc.	Examine off-the-shelf software vendor errata; ensure a robust design (e.g., timeouts on all blocking calls); lock memory pages that are used frequently or used by ISRs; and use only the off-the-shelf software features required and remove all others.	✦		***T***
Malware	Implement virus checkers, firewalls, intrusion detection, denial of service, and fail-safe state.			
Browser or web incompatibility	Integrate version checking at start-up; and conduct compatibility testing.			
Miscellaneous				
Memory leaks	Avoid dynamic allocation of memory.	✦	✦	***D***
System deadlocks	Use simple locking strategies (process can only have one resource locked at a given time); and conduct deadlock analysis.			***S***

[39] A sequence of calculations should never be performed on a global (or shared) variable when the calculation can be interrupted or preempted. Instead, perform all of the calculations on a temporary variable and update the global variable with a single, uninterruptible instruction.

		Verification types—Analysis: Static/Dynamic/Timing		
				Unit test
			Inspection	
Indirect causes	**Risk control measures**			
Reentrancy	Ensure that all functions, including those in third-party libraries, that are called from interrupts (or multiple tasks of different priorities) are designed to be reentrants.			***D***
Stack overflow	Use run-time stack guards, high-water marks, and stack analysis.			***S***
Logic errors and syntax errors	Use source code analysis tool (such as lint) or maximum compiler warning level, and conduct dual diversity and cross-checks at critical control points.	✦	✦	***S***
Infinite loops	Use loop counters, loop time-outs, and watchdog timer.	✦	✦	
Code corruption	Ensure that power-up and run-time program image CRC check.			
Dead code	If dead code not removed, insert an error check that will alarm or perform a safe shutdown if dead code (in custom or for off-the-shelf software components) begins to execute.			***D***
Incorrect conditional code	Ensure that conditional compilation is used appropriately and only when necessary.	✦		
Unintended macro side effects	Use parentheses around all macro parameters.			***S***
Resource depletion	Conduct stack, heap, and timing analyses.			***T***
Incorrect alarm and alert prioritization	Perform stress testing.			
Unauthorized features (“gold plating,” “back doors,” etc.)	Implement requirement and design reviews, and trace matrices.			
Incorrect order of operations and precedence	Perform “bread-crumbing” and call sequence monitoring.			
Safe state	Use an independent monitor.			