MEDICAL DESIGN BRIEFS
EBOOK

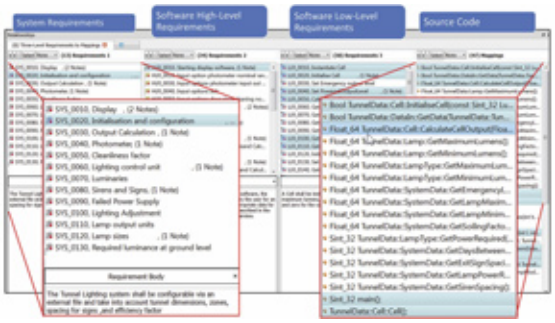
Standards Guide to Design Validation and Safe Medical Device Software

Contents

6

9

12



ABOUT

Design validation is one of the most important aspects of the design and development process for medical devices. In this *Medical Design Briefs* Ebook, learn about the standards and regulatory requirements that are involved in the design validation process. Equally critical are medical design software test standards like IEC 62304, and knowing how to adhere to today's medical device software design specifications.

ON THE COVER

In the article, "*Implementing IEC 62304 for Safe and Effective Medical Device Software, Part 1*," the development of detailed requirements and associated design of medical devices specified by IEC 62304 is examined, culminating in a detailed software design in accordance with clause 5.4 of the standard. Read more on page 8.

Design Validation and Regulatory Requirements

Making the Right Product the First Time

Design validation is one of the most important aspects of the design and development process for medical devices. It is at this stage that the medical device manufacturer confirms that the device that was designed is the right product that meets the needs of the user. Successful design validation requires a thorough understanding of the user needs. Some of the questions that need to be answered during design validation are:

- Does the device work for the user?
- Does it meet the specified user needs?
- Does the device work in the user's environment?
- Is the device's usability easy, clear, obvious, and evident?
- Is the device safe and effective for both the user and the patient?

Design validation has been the number one citation in FDA warning letters for design controls from 2011 to 2015. Six common categories for the design validation warning letter citations are:

1. Not conducting any design validations to ensure finished device meets the intended use and end-user needs.
2. Not using production units, lots, batches, or their equivalents.
3. Not conducting design validations under defined operating conditions, under simulated, or under actual use conditions.
4. Not conducting design validations when changes were made to a device released for production, sale, and use.
5. Not providing rationale for the decision not to perform design validation.
6. Not selecting individuals who represent actual users.

What is Design Validation?

Design Validation is defined in 21 CFR Part 820 – Quality System Regulations as "establishing by objective evidence that device specifications conform with user needs and intended use(s)."[1] Design validation can occur during the development process, before the device has been released for production, sale, and use and, when changes are made to a device that has been released for production, sale, and use.

Figure 1 illustrates where design validation is conducted in the design and development process. The design and development process starts with the identification of user needs. The user needs are translated into the design requirements (design inputs). The user needs and the design inputs are the foundation for developing the right design and subsequently verifying and validating the design. Incomplete, inadequate, or incorrect user needs and design inputs will result in the design of the wrong product, wasting enormous amounts of resources, time, and money.

Design validation is a requirement for design and development in the U.S. FDA regulation 21 CFR Part 820, and the global international standard ISO 13485:2003/ISO 13485:2016.[1–3] In addition to design performance and functionality requirements, human factors and usability are critical in design validation studies. The risks associated with the design (hardware, software, user-interface, and usability) should be identified and managed to ensure safety and effectiveness. Risk management for medical devices is conducted using the international standard ISO 14971.[4] Usability engineering requires the use of the standard IEC 62366.[5] This standard points to several aspects of ISO 14971 regarding the management of use-related hazards and risks. ANSI/AAMI HE75, though not a voluntary standard, provides in-depth information on human factors engineering and design guidelines for medical devices.[6]

To Validate or Not to Validate?

Design validation is a key requirement when changes to a device are made after it has been released for production, sale, and use. According to 21 CFR Part 820.30 (i), ISO 13485:2003 Clause 7.3.1, and ISO 13485:2016 Clause 7.3.1, design changes shall be verified and validated as appropriate, and, reviewed and approved before implementation.[1–3] Rationale should be provided when design validation is not conducted. Several companies have been cited for not conducting design validation on a change to an existing, marketed product. A

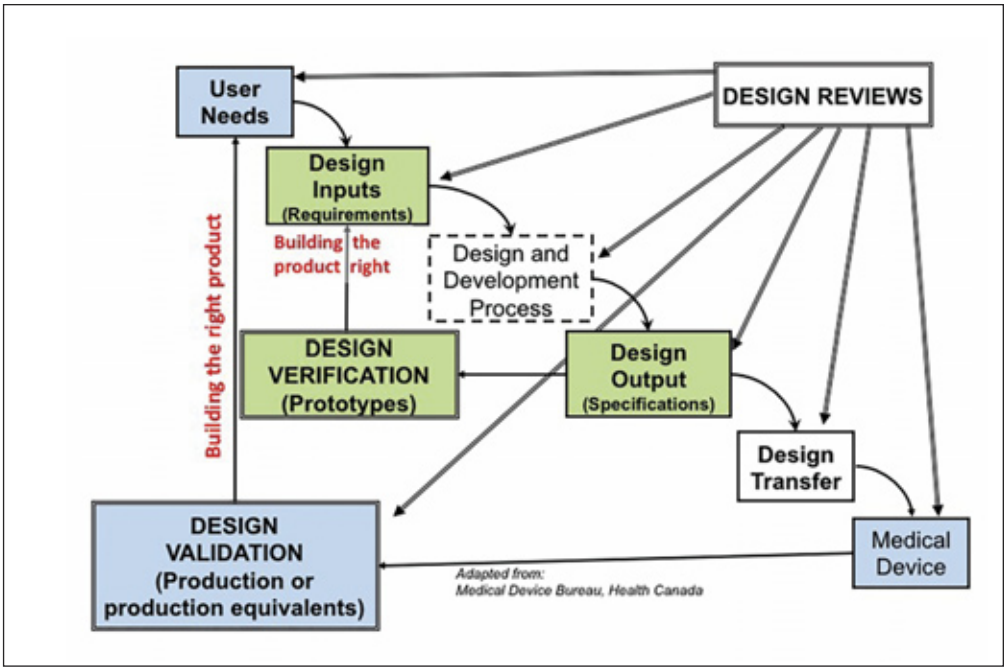


Fig. 1 – The design and development process for medical devices.

common reason used is that design validations cost time, money, and resources and could affect sales and revenue.

Instructions for use (IFU) is a prime example where companies rationalize that a simple verification of the change in an IFU is sufficient. Not validating this change with the user (i.e., conducting a design validation ensuring that the updated and revised instructions are evident and effective) can lead to use errors and potential harm or injury to the patient. Not validating a simple change in a device component shape, color, or feel can result in user confusion and use errors. Changes to software in a device are especially prone to adverse events in the field if the software is not validated adequately.

The proper due diligence in evaluating whether or not design validation is required is critical to patient safety and device effectiveness.

Design Verification vs. Design Validation

Many companies use the term design V&V for design verification and design validation, intentionally implying that the two activities are equivalent and can be done together. *Design verification* and *design validation* are two distinctly different terms and have significantly different requirements. Table 1 illustrates the differences between the two activities.

Things to consider for design validation – Several reports over the years by regulatory bodies indicate that as many as one-third of medical device failures and adverse events that involve use of medical devices point to failures of device use rather than failure of the device itself. This has often resulted in suboptimal medical treatment, injuries, and even patient deaths. In many cases the redesign of the device-user interface improved usability and reduced patient harm and injury.

Usability is defined as the "characteristic of the user interface that facilitates use and thereby establishes effectiveness, efficiency, and user satisfaction in the intended use environment." (IEC-62366).[5] All aspects of usability can result in safe and effective use or can result in use error leading to an unsafe, ineffective device. Factors that influence the outcome of either correct use or use error include the use environment, the user, the device, its user interface, and the IFU and training associated with the device (see Figure 2).

Design Validation should consider the following nine criteria:

1. User-friendly device designs and operations that are self-evident and mistake-proof.
2. Device safety, effectiveness, functionality, and performance.
3. The range of intended user population(s). These include the range of users that have different physical characteristics and capabilities (size, height, dexterity, flexibility, functional reach ranges, vision, hearing, tactile sensitivity, etc.), cultural backgrounds and languages, learning capabilities, and emotional and cognitive capabilities. The level of knowledge and experience and training can also influence how well a user is able to interact with a particular medical device.
4. The intended patient population(s), some of which include neonates, children, young adults, adults, and the elderly.
5. The use environments, which can range from operating rooms (ORs), emergency rooms (ERs), to standard hospital rooms, clinics, and homes. The clinical environment is a complex system of medical and support personnel and patients. They can house a large number of different medical devices and supporting equipment. The environments in a clinical setting are well controlled compared with the environment of home. Factors like temperature, humidity, noise, vibration, compatibility with other devices, lighting, space, electrical and electromagnetic interference, radio-frequency interference, and atmospheric pressure should be built into the design validation.
6. Clear, understandable IFUs that can be easily followed and remembered when users return after a period of time to use the device again. Methods of instructions should be geared toward specific user populations. Electronic instructions (videos) might be more effective for a certain demographic of users versus physical user manuals. In addition, methods of writing the instructions — i.e. pictorial versus straight text — can also affect usability.
7. Effectiveness of use, which can be measured by the number of steps done correctly divided by the total number of steps. This can point to use errors and the severity of those errors during use.
8. Efficiency of use, which can be measured by the total time taken to complete the tasks versus a targeted goal.
9. User satisfaction and acceptability of use for each task and the overall usability and operation of the device can be measured using questionnaires with a Likert-type scale and subsequently analyzing the data.

Identifying User Needs

Successful design validation can only occur if the user needs and intended use have been well defined at the very beginning of the design and development phase. One of the major reasons for product

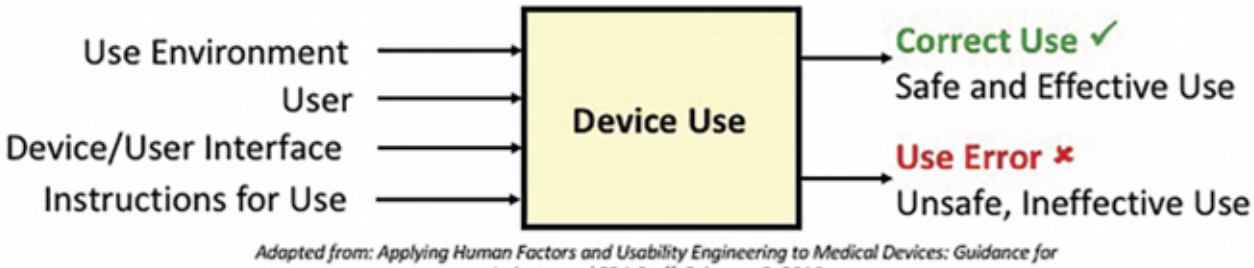


Fig. 2 – Factors affecting usability of medical devices.

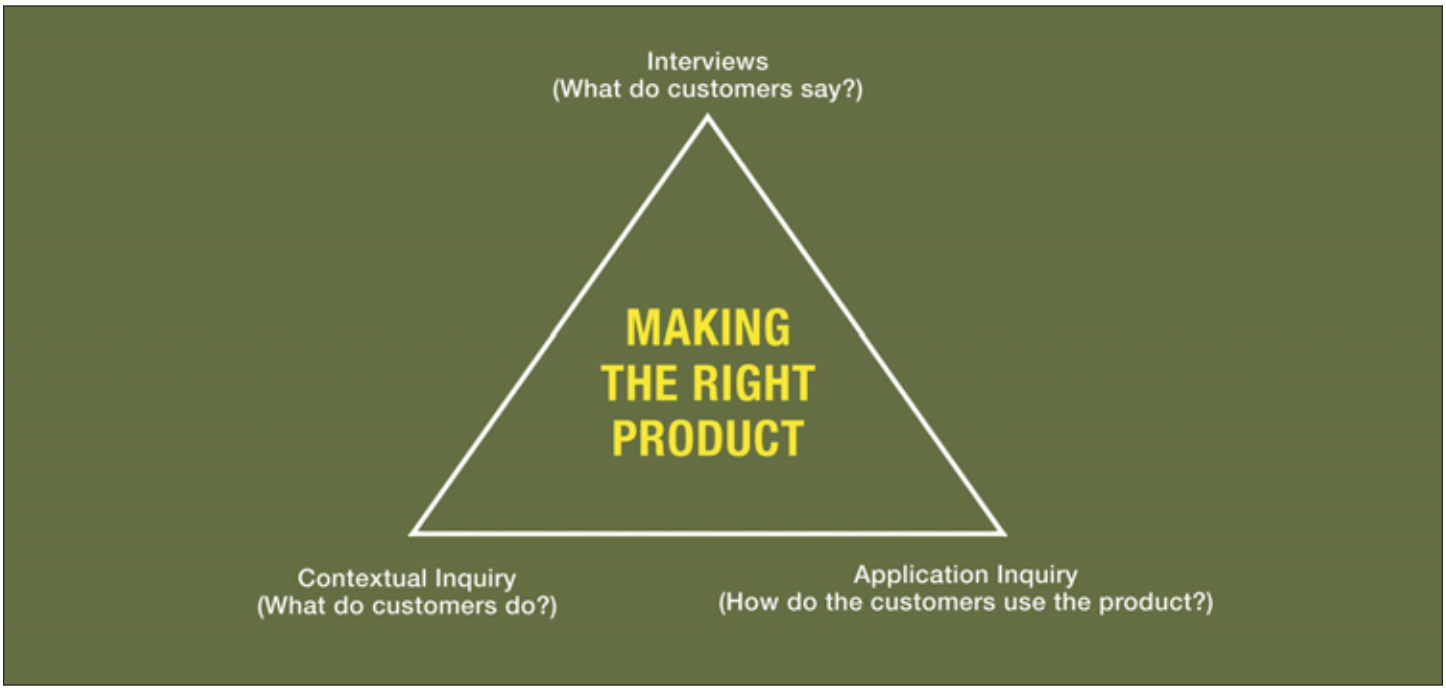


Fig. 3 – Methods of identifying user needs.

failures over the years has been the inability of devices to meet the user and market needs.

Those companies that have honed their skills in effective market intelligence and "product usability intelligence" demonstrate higher success rates in both new product launches and lower incidences of use errors and device failures in the field. In order to design, manufacture, and sell the right product, companies should develop methods and processes to listen to the customers and users, know what the users do, and understand how they use the product (see Figure 3).

Listen to the user and customer – Several tools can be used to identify user needs. Voice of the customer (VOC) methods can involve interviews, focus groups, surveys, and initial prototype usability evaluations. It is important to know how users verbalize their needs with respect to what is said and what is unspoken. Facilitating the discussions and asking open-ended questions can further elicit valuable information. Listening skills are crucial to the effectiveness of understanding what the users are saying and communicating.

Know the environment of the user – Contextual inquiry is a specific type of interview for collecting information from users. It is usually done on a one-to-one basis. Interviewees are interviewed in their work environment, when doing their tasks, with as little interference from the interviewer as possible. Information and data gathered during such interviews should be unbiased.

Observe how the user uses the product – Observing how the users will use the product within their use environment can also provide valuable information with respect to usability (either ease of use or difficulty of use), potential use errors, and indications of potential safety and effectiveness defects of the product. The device-user interface needs can also be identified.

Using cross-functional groups that include marketing, regulatory affairs, engineering and clinical/medical to participate in the identification and documentation of user needs will ensure that all the different perspectives are captured in the process.

Conclusion

Common best practices to obtain the user needs and develop the right product are:

- Seek user input early and often.
- Partner with strategic users and customers.
- Refine the design through the design and development process.
- Ensure that usability, human factors, safety, and effectiveness are addressed.
- Identify and address use errors and device functionality and performance failures.
- Plan design validation in the early stages of the development once user needs have been identified.
- Conduct design validation with the right users, in their use environments, complying with all applicable regulatory requirements.

Successful design validation predicates identifying the right user needs, designing the right product, identifying and addressing all risks, and verifying that the product meets all requirements. Designing, manufacturing, and selling the right product can lead to customer and user satisfaction, increased sales and revenue, and high-quality safe and effective products.

References

1. 21 CFR Part 820 – Quality System Regulation 7 October, 1996.
2. ISO 13485:2003 Medical devices — Quality management systems — Requirements for regulatory purposes.
3. ISO 13485:2016 Medical devices — Quality management systems — Requirements for regulatory purposes.
4. ISO 14971:2007 and ISO 14971:2012 Medical Devices – Application of risk management to medical devices.
5. IEC 62366-1:2015 Medical Devices Part 1 – Application of usability engineering to medical devices.
6. ANSI/AAMI HE75:2009 Human Factors Engineering – Design of Medical Devices.

Vinny Sastri, Ph.D., is president and founder of Winovia LLC, Chicago, IL.

How to Simplify Medical Device Software Integration and Certification

Medical device manufacturers face a balancing act between time to market and certification efforts.

Medical device manufacturers operate in a challenging environment filled with stringent regulatory requirements and industry pressures. With a rise in mainstream competitors from the consumer electronics space and an uptake of touch-screen interfaces and wireless connectivity, medical device manufacturers must develop increasingly complex devices in timelines that are more typical of consumer-grade electronics, but difficult to meet in a regulated industry.

So, how can the lives of medical device developers be simplified? Let's start with a look at the software integration process.

You begin by building the software architecture for your medical device. You've locked in on an operating system (OS), you have a good idea about the cybersecurity requirements, and you've completed the safety design. So you purchase a reference hardware board, download an evaluation copy of the OS you have chosen, and begin preliminary proof-of-concept benchmarks. Only, the OS doesn't have the drivers for the Wi-Fi device you selected. Or the version of the OS doesn't line up with the version of the Wi-Fi driver. Or you're restricted to a specific Bluetooth hardware module, as it supports the profile that you require for your medical device, but your hardware platform does not support that module. What should be a simple integration of OS, device drivers, and hardware modules instead becomes a major amount of work. And you still need to build in a hazard and risk analysis, a failure analysis, a safety case. And you haven't even begun to code your app.

Managing the process of designing software components is just one of the challenges of building a medical device.

Clearly, the effort of assembling all of the underlying software components for a medical device can be as sizeable, and as fraught with risk, as the actual development of your specific application. The number of technology suppliers that you, the developer, must coordinate with in order to build complete functionality is a major challenge.

So what can help alleviate some of this pain? To begin with, the choice of OS can improve this whole (square peg, round hole) process. Furthermore, the right OS can be an even bigger boon to compliance validation than to the overall development/test cycle, which is just a small component of the overall product delivery cycle. In other words, the compliance efforts for your new device can take more time than actual development. So, while it's important to reduce the time for coding, testing, and building, it's just as important to determine how the compliance process can be made easier, faster, and cheaper. When evaluating an OS, you need to examine how it can help or hinder your product development and certification process.

When selecting an OS, the fundamental choice comes down to open source or commercial-off-the-shelf (COTS). But before we go into detail on OS selection, we must introduce the medical compliance standard known as IEC 62304. It has significant bearing on the choice of OS.

The choice of operating system has a lot of ramifications on IEC 62304 compliance.

IEC 62304

IEC 62304 is a standard that has been endorsed under medical device-related directives by the FDA in the US and by the Directorate-General for Health and Consumers in the EU, enabling manufacturers to follow good development practices and to produce high-quality software for the medical community.

IEC 62304 forms an important part of the OS discussion. Agencies, such as the FDA, evaluate devices as a whole and not their discrete parts, so it is to a manufacturer's advantage to use an OS that has a history of use in devices that comply with regulatory requirements. Also, using an OS that has already been proven to comply with a design standard, such as IEC 62304, can significantly reduce the effort, cost, and time to achieve certification.

IEC 62304 uses phraseology such as Software Of Unknown Provenance, or SOUP, which is software that a manufacturer uses in a medical device but has not explicitly developed for that device. IEC 62304 assumes that commercial or open source software will be used and it offers two definitions of SOUP, which can be software not developed for a medical device or that has unavailable or inadequate records of its development processes. IEC 62304 does not prohibit using SOUP in a medical device and, in fact, several clauses in the standard make the assumption that SOUP will be used. So, then, the question isn't whether we're allowed to use COTS software or SOUP in medical device software, but how we decide whether a particular COTS or SOUP item is appropriate for the medical device in question, and how we validate whether the item supports the functional safety requirements for the device.

The difficulty with using open source in functionally safe systems like a medical device is that the processes for open source development are neither clearly defined nor well documented. This is precisely what concerns IEC 62304 compliance. With open source, you typically don't know how the software was designed, coded, or verified; attempting to validate functional safety claims without this knowledge can be an onerous undertaking.

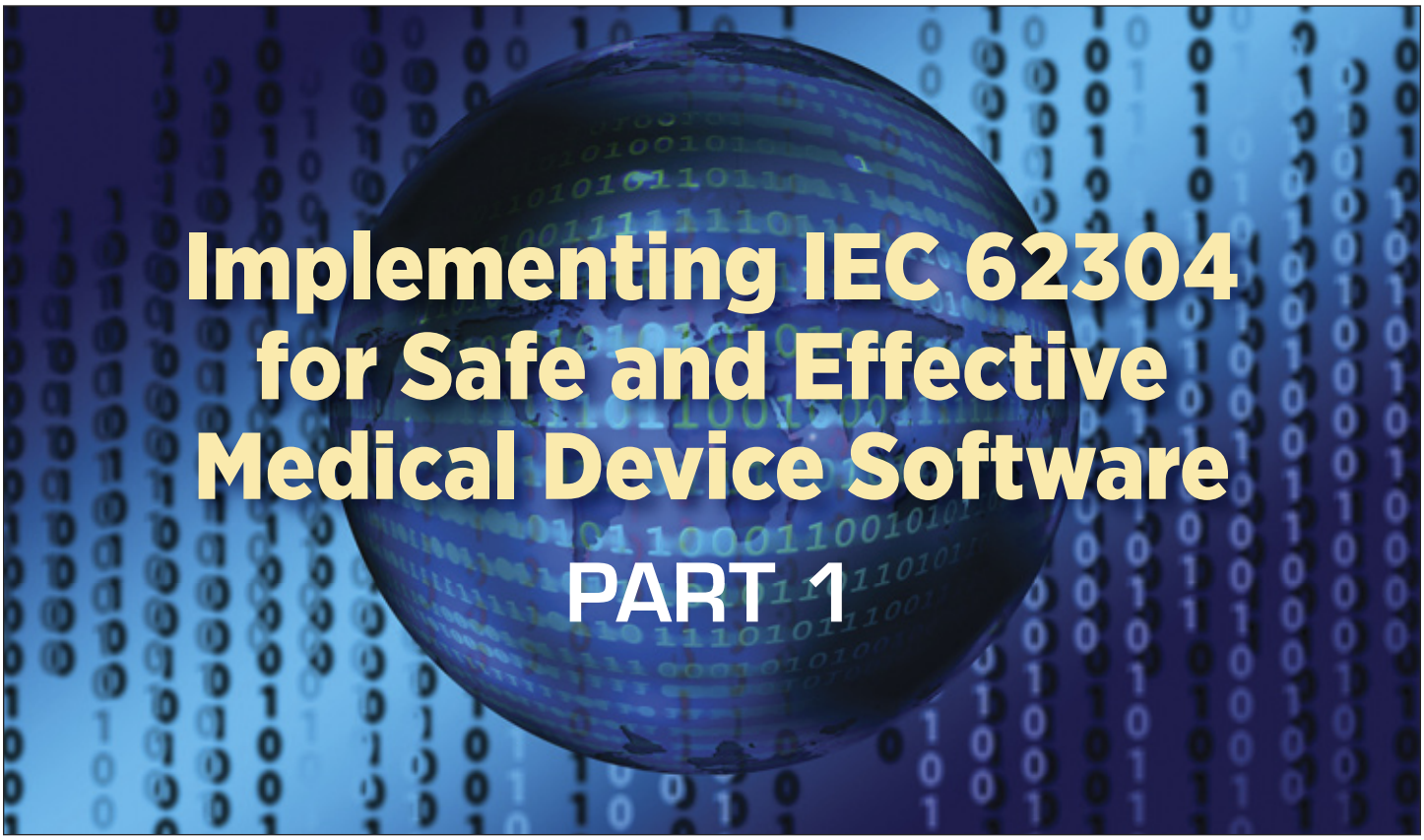
IEC 62304 requires that all dead code, or code not directly related to the functionality of the medical device, be removed. For an open source OS, this requires extensive examination of the source code and implementation. Even if the dead code can be extracted without disrupting the confidence from use derived from years of active service of this software, IEC 62304 also requires the device manufacturer to maintain the code for as long at the device is in service—meaning that the development team needs to monitor and examine all patches to the open source project to determine if the patches are relevant to the medical device. And, if the patches are relevant, the patches must then be ported to the modified source tree, which has all this dead code removed.

IEC 62304 accounts for the inclusion of off-the-shelf components, like the OS. It's not expected that the device manufacturer also be an OS company.

If the ultimate goal of a medical device manufacturer is to get the device certified, then a COTS OS offers the path of least resistance. But remember, all operating systems are not created equal. To be truly appropriate for being used in medical devices, the OS vendor's software must have a well-documented quality management system. And at some point, the software must undergo a complete fault-tree analysis. This will be difficult to do on behalf of the vendor because the device manufacturer won't know the architecture and design of the OS, so knowing if the vendor has performed this fault-tree analysis will be very relevant. Having access to the fault-tree analysis can greatly reduce the work for the device manufacturer, similarly so for the software being accompanied by a safety manual that supports these claims. If the COTS OS is accompanied by a safety manual, the job of building safety case becomes much easier.

Again, the choice of OS is exactly that—a choice. A criterion in this selection, when arriving at a choice, should be how that OS helps or hinders with the compliance of a medical device. An OS that offers complete pre-integrated components like the Wi-Fi and Bluetooth drivers, the Qt graphics packages, clock synchronization libraries and processes, etc., all integrated and version-aligned, will suit timelines better than one lacking these traits. Better timelines mean faster to market, faster to revenue, and reduced risk.

This article was writtten by Chris Ault, Senior Product Manager, QNX Software Systems, Ottawa, Ontario, Canada.



Implementing IEC 62304 for Safe and Effective Medical Device Software

PART 1

FDA's introduction to its rules for medical device regulation states: "Medical devices are classified into Class I, II, and III. Regulatory control increases from Class I to Class III. The device classification regulation defines the regulatory requirements for a general device type. Most Class I devices are exempt from Premarket Notification 510(k); most Class II devices require Premarket Notification 510(k); and most Class III devices require Premarket Approval." [1]

Given that such a definition encompasses a large majority of medical products other than drugs, it is small wonder that medical device software now permeates a huge range of diagnostic and delivery systems. The reliability of the embedded software used in these devices and the risk associated with it has become a vital concern.

In response to that, the functional safety standard IEC 62304, "Medical device software – Software life cycle processes," has emerged as an internationally recognized mechanism for the demonstration of compliance

Software Documentation				
Clause	**Subclauses**	**Class A**	**Class B**	**Class C**
Software development plan	5.1.1, 5.1.2, 5.1.3, 5.1.6, 5.1.7, 5.1.8, 5.1.9	X	X	X
	5.1.5, 5.1.10, 5.1.11		X	X
	5.1.4			X
Software requirements specification		X	X	X
	5.2.3		X	X
Software architecture	5.3.1, 5.3.2, 5.3.3, 5.3.4, 5.3.6		X	X
	5.3.5			X
Software detailed design	5.4.1		X	X
	5.4.2, 5.4.3, 5.4.4			X
Software unit implementation and verification	5.5.1	X	X	X
	5.5.2, 5.5.3, 5.5.5		X	X
	5.5.4			X
Software integration and integration testing	All requirements		X	X
Software system testing	All requirements		X	X
Software release	5.8.4	X	X	X
	5.8.1, 5.8.2, 5.8.3, 5.8.5, 5.8.6, 5.8.7, 5.8.8		X	X
Software maintenance process	6.1, 6.2.1, 6.2.2, 6.2.4, 6.2.5	X	X	X
	6.2.3		X	X
Software risk management process	7.1, 7.2, 7.3, 7.4.2, 7.4.3		X	X
	7.4.1	X	X	X

Table 1. Summary of which software safety classes are assigned to each requirement, highlighting clause 5.4.2 as an example.

with the relevant local legal requirements. In practice, for all but the most trivial applications, compliance with IEC 62304 can only be demonstrated efficiently with a comprehensive suite of automated tools.[2,3]

Part 1 of this article examines the development of detailed requirements and associated design of medical devices specified by IEC 62304, culminating in a detailed software design in accordance with clause 5.4 of the standard and shown in context in Figure 1.

Released in 2006, the IEC 62304 standard provides a framework of software development life cycle processes with activities and tasks necessary for the safe design and maintenance of medical device software. The processes, activities, and tasks described in clause 5 establish a common framework for medical device software life cycle processes that can be understood and shared within and between teams working on a project (see Figure 1).

IEC 62304 applies to the development and maintenance of medical device software when:

- Software is itself a medical device.
- Software is used as a component, part, or accessory of a medical device.
- Software is used in the production of a medical device.

The IEC 62304 standard expects the manufacturer to assign a safety class to the software system as a whole, based on its potential to create a hazard that could result in an injury to the user, the patient, or other people.

There are three software safety classifications, as follows:

- Class A: No injury or damage to health is possible.
- Class B: Nonserious injury is possible.
- Class C: Death or serious injury is possible.

The classification assigned to a project has a tremendous impact on the code development process from planning, developing, testing, and verification through to release and beyond. It is therefore in the interest of medical device manufacturers to get it right the first time and avoid expensive and time-consuming rework.

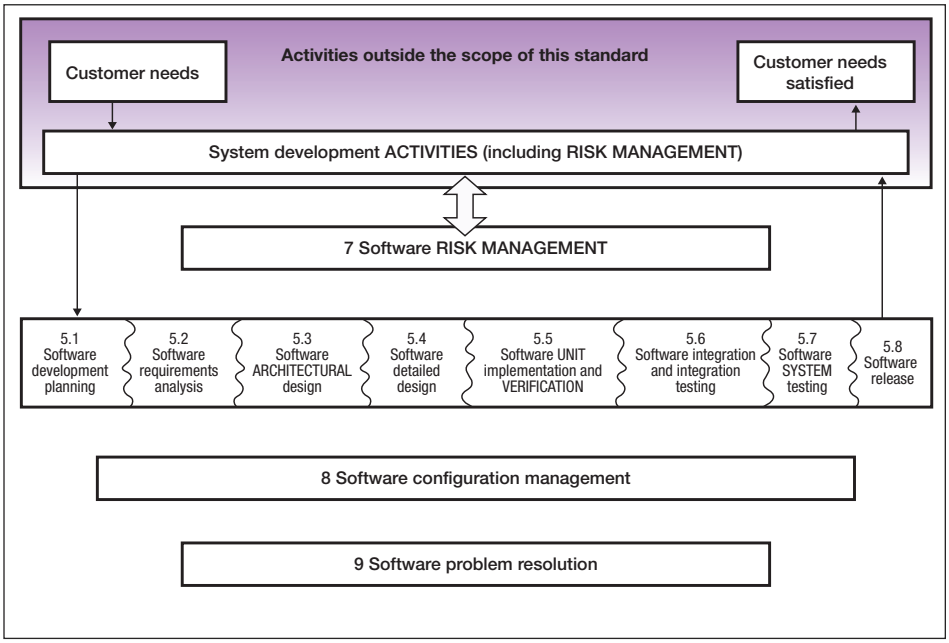


Fig. 1 – Overview of software development processes and activities.

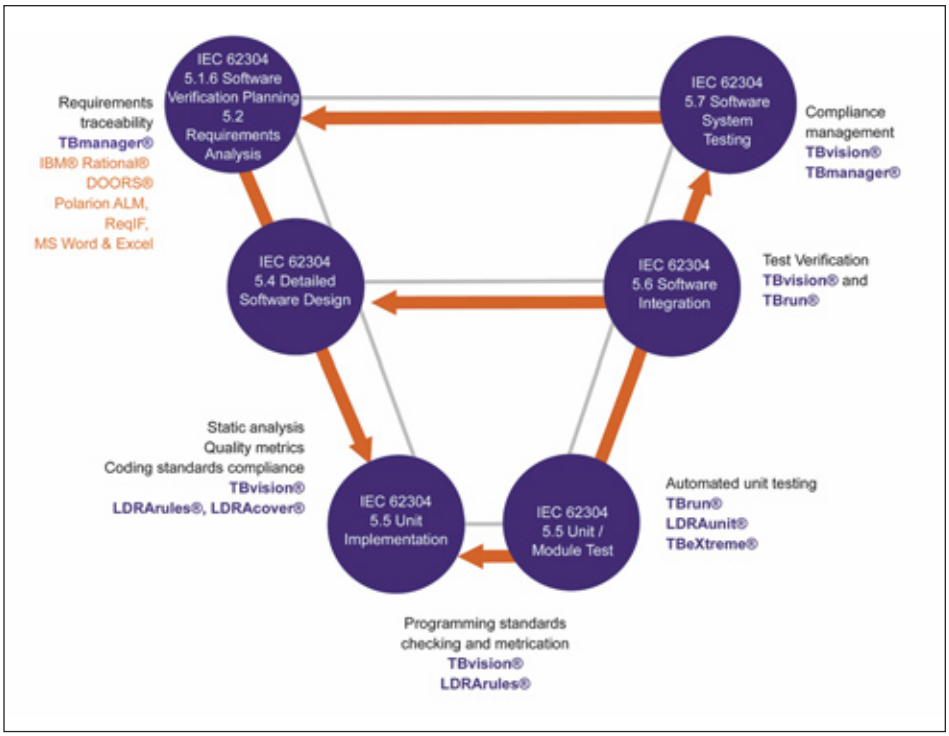


Fig. 2 – Mapping the capabilities of a comprehensive tool suite to the guidelines of IEC 62304 with the LDRA tool suite.

In practice, any company developing medical device software will carry out verification, integration, and system testing on all software regardless of the safety classification, but the depth to which each of those activities is performed varies considerably. Table 1 is based on table A1 of the standard and gives an overview of what is involved.

For example, subclass 5.4.2 of the standard states that "The MANUFACTURER shall develop and document a detailed design for each SOFTWARE UNIT of the SOFTWARE ITEM." Reference to the table shows that it applies only to Class C code and that it is not required for classes B and C.

IEC 62304 is essentially an amalgam of existing best practices in medical device software engineering, and the functional safety principles recommended by the more generic functional safety standard IEC 61508, which has been used as a basis for industry specific interpretations in a host of sectors as diverse as the rail industry, the process industries, and earth moving equipment manufacture.[4]

A process-wide, proven tool suite such as the LDRA tool suite, has been shown to help ensure compliance to such software safety standards (in addition to security standards) by automating both the analysis of

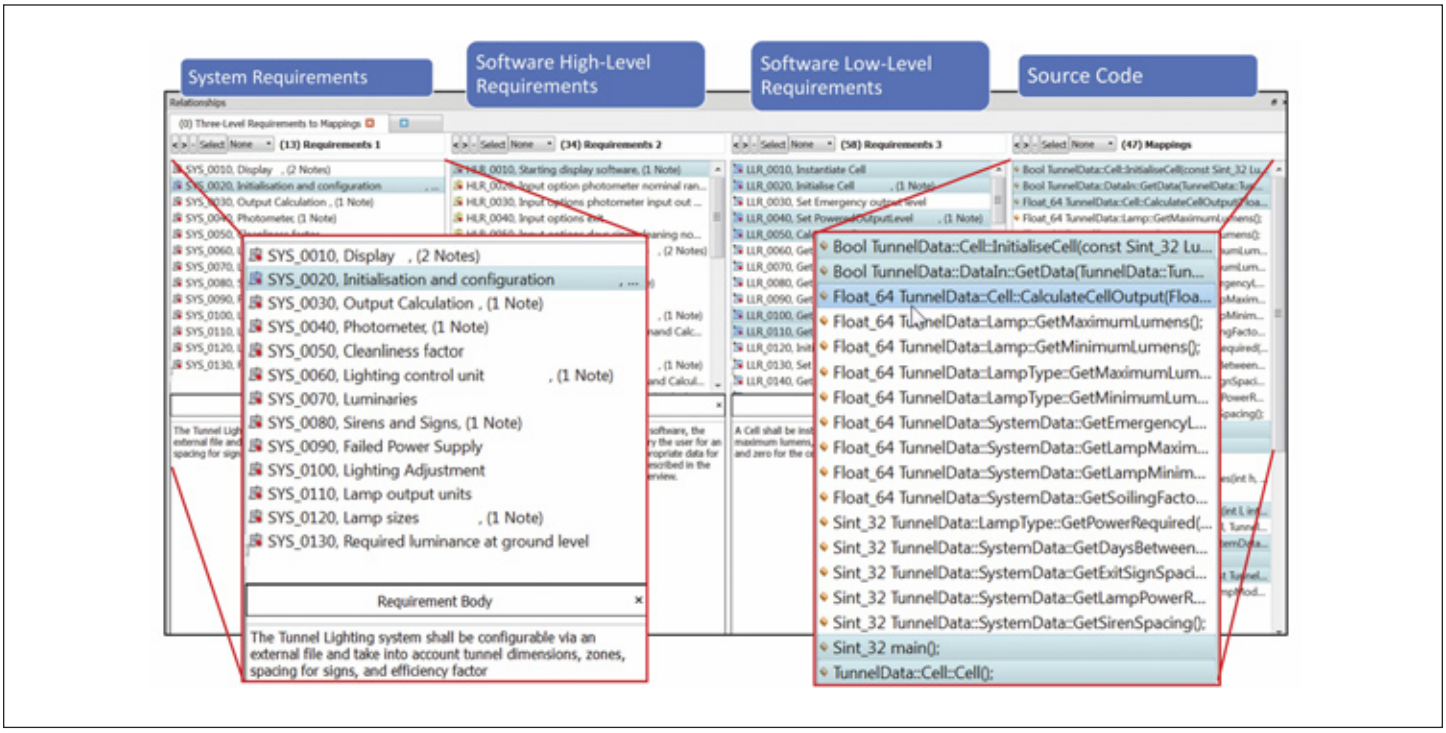


Fig. 3 – Automating requirements traceability with the TBmanager component of the LDRA tool suite.

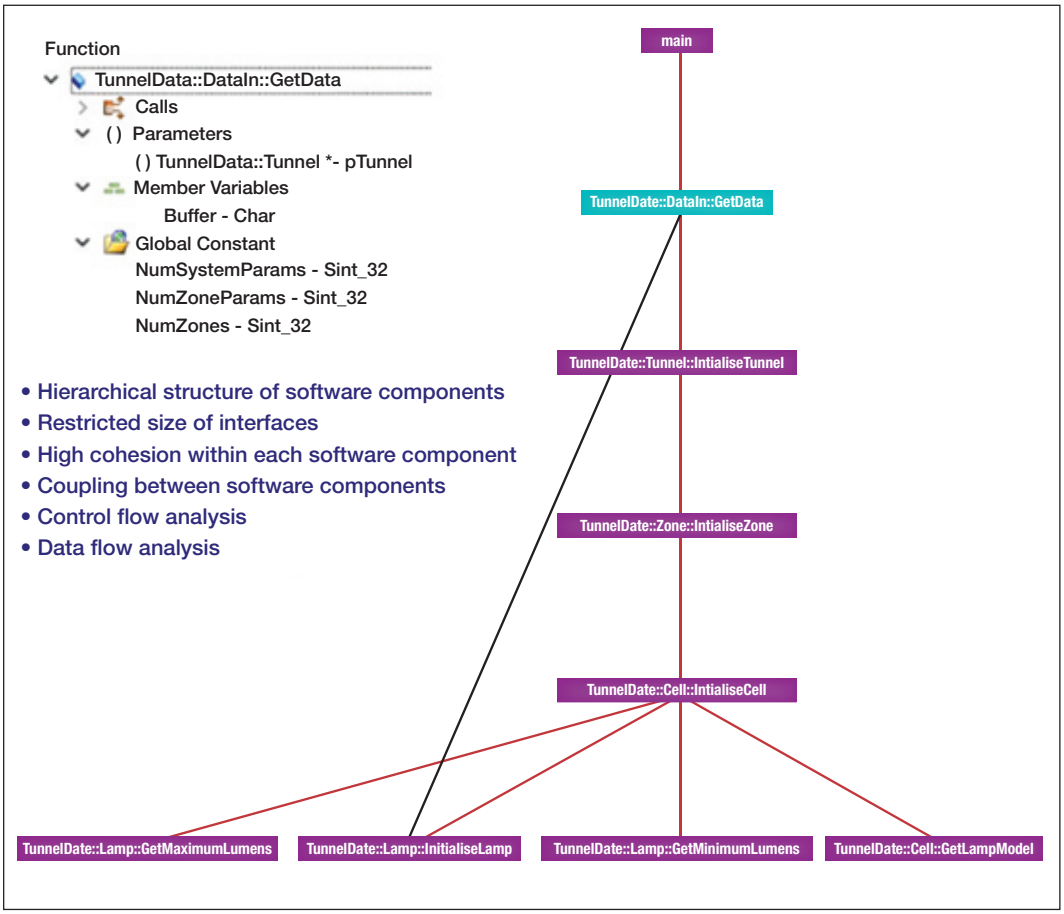


Fig. 4 – Diagrammatic representations of control and data flow generated from source code by the LDRA tool suite aid verification of software architectural and detailed design.

the code from a software quality perspective as well as the required validation and verification work. Equally important, the tool suite enables life cycle transparency and traceability into and throughout the development and verification activities, facilitating audits from both internal and external entities. It is therefore a logical choice for working in accordance with IEC 62304.

The V diagram in Figure 2 illustrates how the LDRA tool suite can help through the software development process described by IEC 62304. The tools also provide critical assistance through the software maintenance process (clause 6) and the risk management process (clause 7). Clause 5 of IEC 62304 details the software development process through eight stages ending in release. Notice that the elements of Clause 5 map to those in Figure 1 and Table 1.

5.1 Software Development Planning

Following the steps listed in Table 1 — and the V diagram in Figure 2 — the first objective is to plan the tasks needed for development of the software in order to reduce

risks as well as to communicate procedures and goals to members of the development team. This helps ensure that system quality requirements for the medical device software are understood and met by all team members and can be verified within the project.

Development planning applies to all classes of devices. It involves the creation of the plan with reference to the system design and the definition of system requirements, and the selection and definition of the development standards, methods, and tools to be used. There should be a defined procedure for integration and integration testing. A format for documentation should be specified along with plans for configuration management and verification.

Thorough software development planning is vital to success. The foundations for an efficient development cycle can be established by using tools that can facilitate structured requirements definition, such that those requirements can be confirmed as met by means of automated document (or artifact) generation. IEC 62304 dictates that medical device software must include appropriate risk control measures in its system requirements.

The preparation of a mechanism to demonstrate that the requirements have been met will involve the development of detailed plans. A prominent example would be the software verification plan to include tasks to be performed during software verification and their assignment to specific resources.

5.2 Software Requirements Analysis

Once each system requirement has been identified in such a way as to make it possible to demonstrate traceability right from the system requirement through to software system testing, and to show this risk control measures have been accounted for, the next step is to derive and document the software requirements from the system requirements.

Achieving a format that lends itself to bi-directional traceability will help to achieve compliance with the standard. Bigger projects, perhaps with contributors in geographically diverse locations, are likely to benefit from an application life cycle management tool such as IBM® Rational® DOORS®, Siemens® Polarion® PLM®, or more generally, similar tools offering support for standard Requirements Interchange Formats. Smaller projects can cope admirably with carefully worded Microsoft® Word® or Microsoft Excel® documents, written to facilitate links up and down the V model.[5–7]

Requirements rarely remain unchanged throughout the lifetime of a project, and that can turn the maintenance of a traceability matrix into an administrative nightmare. A requirements traceability tool alleviates this concern by automatically maintaining the connections between the requirements, development, and testing artifacts and activities. Any changes in the associated documents or software code are automatically highlighted such that any tests required to be revisited can be dealt with accordingly (see Figure 3).

5.3 Software Architectural Design

The software architectural design activity requires the manufacturer to define the major structural components of the software, their externally visible properties, and the relationships between them. Any software component behavior that can affect other components should be described in the software architecture, such that all software requirements can be implemented by the specified software items. This is generally verified by technical evaluation.

Since the software architecture is based on the requirements, developing the architecture means defining the interfaces between the software items that will implement the requirements. Many of these software elements will be created by the project, but some may be brought in from other projects even in the form of third-party libraries. Such code must be shown to comply with the project's functional and performance requirements.

In addition, some architectural elements may need to be segregated for risk management purposes, such that their connections to the other elements then become part of the overall architecture. Tools can help in many elements of the architectural design process, including requirements specification, design model traceability, and verification.

If a model-based approach is taken to software architectural design — for example, using MathWorks® Simulink®, IBM Rational Rhapsody®, or ANSYS® SCADE — then a tool suite that is integrated with the chosen modelling tools will make the analysis of generate code and traceability to the models far more seamless.[8–10]

5.4 Software Detailed Design

Once requirements and architecture are defined and verified, detailed design can be built on this foundation. Detailed design specifies algorithms, data representations, and interfaces between different software units and data structures. Because implementation depends on detailed design, it is necessary to verify the detailed design before the activity is complete, generally by means of a technical evaluation of the detailed design as a whole, and verification of each software unit and its interfaces.

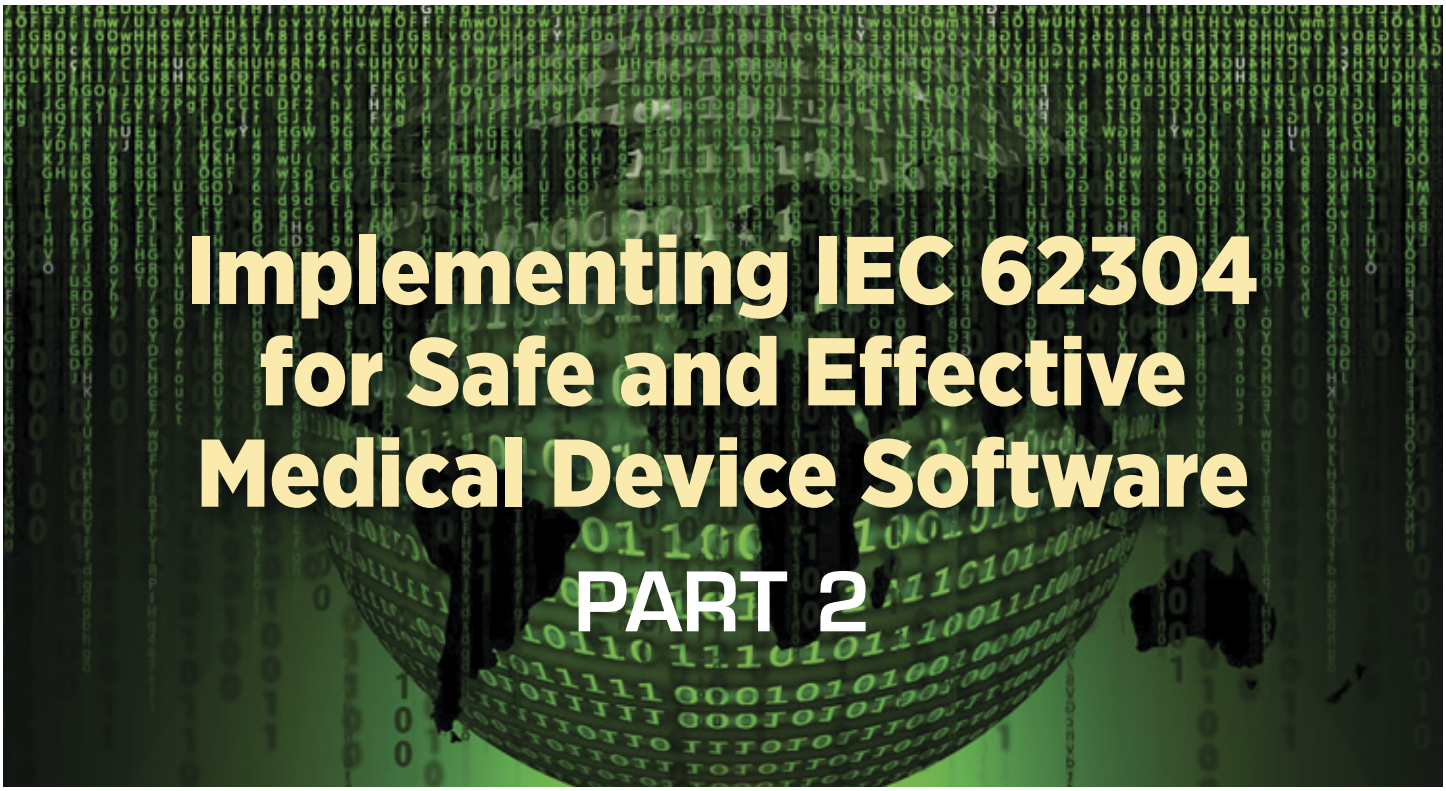
Later in the development cycle, a tool suite can help by generating graphical artifacts suited to the review of the implemented design by means of walkthroughs or inspections. One approach is to prototype the software architecture in an appropriate programming language, which can also help to find any anomalies in the design. Graphical artifacts like call graphs and flow graphs, are well suited for use in the review of the implemented design by visual inspection (see Figure 4).

Part 2 of this article will look at the second part of the project life cycle involving the implementation of the now-established design in the form of source code. It will consider how static analysis can help to ensure that coding rules are adhered to, helping to minimize errors in the developed application. It will discuss how dynamic analysis in the form of system and unit tests show that the design has been implemented correctly and completely, and that the software has been properly exercised during test and prior to release. Finally, it will outline how the influence of IEC 62304 extends to the maintenance and modification of the product once out in the field.

References

1. https://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview
2. IEC 62304 International Standard Medical device software – Software life cycle processes Edition 1 2006-05
3. http://www.imdrf.org/docs/imdrf/final/procedural/imdrf-proc151002-medical-device-software-n35.pdf
4. IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems
5. http://www03.ibm.com/software/product/en/ratidoor
6. https://polarion.plm.automation.siemens.com/
7. http://www.omg.org/spec/ReqIF/
8. https://uk.mathworks.com/products/simulink.html
9. http://www03.ibm.com/software/products/en/ratirhapfami
10. http://www.ansys.com/products/embedded-software/ansys-scade-suite

This article was written by Mark Pitchford, Technical Specialist for LDRA (Wirral, UK).



Implementing IEC 62304 for Safe and Effective Medical Device Software

PART 2

The reliability of the embedded software used in medical devices and the risk associated with it has become a vital concern. IEC 62304, "Medical device software – Software life cycle processes," has thus emerged as an internationally recognized mechanism for the demonstration of compliance with relevant local requirements. Part 1 of this article examined the development of detailed requirements and associated design of medical devices specified by IEC 62304, culminating in a detailed software design in accordance with clause 5.4 of the standard and shown in context in Figure 1. Part 2 details the process applicable to the implementation of that design in source code, the verification and validation of that code, and the ongoing maintenance of the system after release.

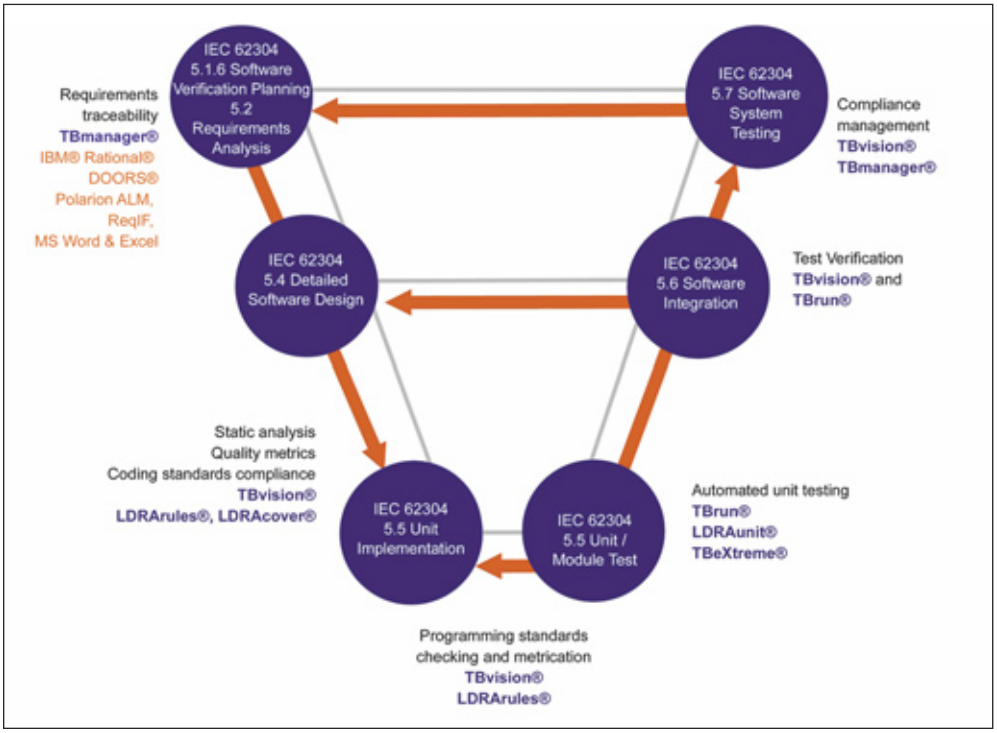


Fig. 1 – Mapping the capabilities of a comprehensive tool suite to the guidelines of IEC 62304 with the LDRA tool suite.

5.5 Software Unit Implementation and Verification

Next comes the process of translating the detailed design into source code. To consistently achieve the desirable code characteristics, coding standards should be used to specify a preferred coding style, aid understandability, apply language usage rules or restrictions, and manage complexity. The code for each unit should be verified using a static analysis tool to ensure that it complies in a timely and cost-effective manner.

Verification tools largely offer support for a range of coding standards such as MISRA C and C++, JSF++ AV, HIS, CERT C, and CWE. The better tools will be able to confirm adherence to a very high percentage of the rules dictated by each standard, whereas more lightweight tools will be unable to detect the subtler violations. More comprehensive solutions will also support

the creation of and adherence to in-house standards from both user-defined and industry standard rule sets.

IEC 62304 also states that the manufacturer shall establish strategies, methods, and procedures for verifying each software unit. Where verification is done by testing, the test procedures shall be evaluated for correctness. Among the acceptance criteria are considerations such as the verification of the proper event sequence, data and control flow, fault handling, memory management and initialization of variables, memory overflow detection, and checking of all software boundary conditions.

Unit test tools often provide a graphical user interface for unit test specification, which is used to create tests according to the defined specification and to present a list of all defined test cases with appropriate pass/fail status. The ability to automatically present the graphical presentation of control flow graphs, and to create test harnesses, stub functions, and cover for missing member or global variables means that unit test execution and interpretation becomes a much quicker and easier process, requiring a minimum of specialist knowledge. By extending the process to the automatic generation of test vectors, the tools provide a straightforward means to analyze boundary values without creating each test case manually. Test sequences and test cases are retained so that they can be repeated (regression tested), and the results compared with those generated when they were first created.

Thorough verification also requires static and dynamic data and control flow analysis. Static data flow analysis produces a cross reference table of variables, which documents their type and where they are utilized within the source files or system under test. It also provides details of data flow anomalies, procedure interface analysis, and data flow standards violations.

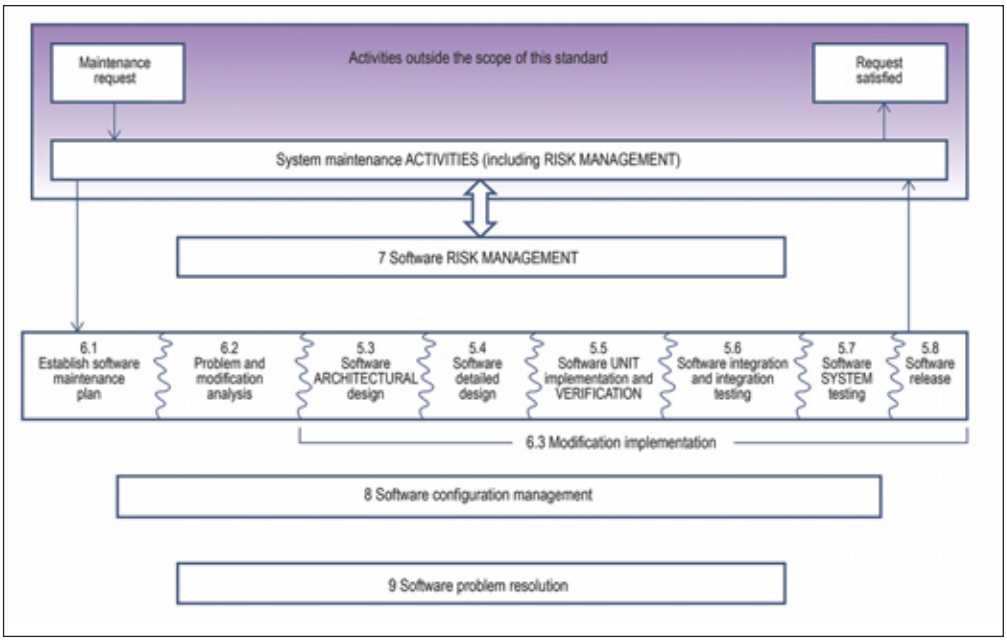


Fig. 2 – Overview of software maintenance processes and activities.

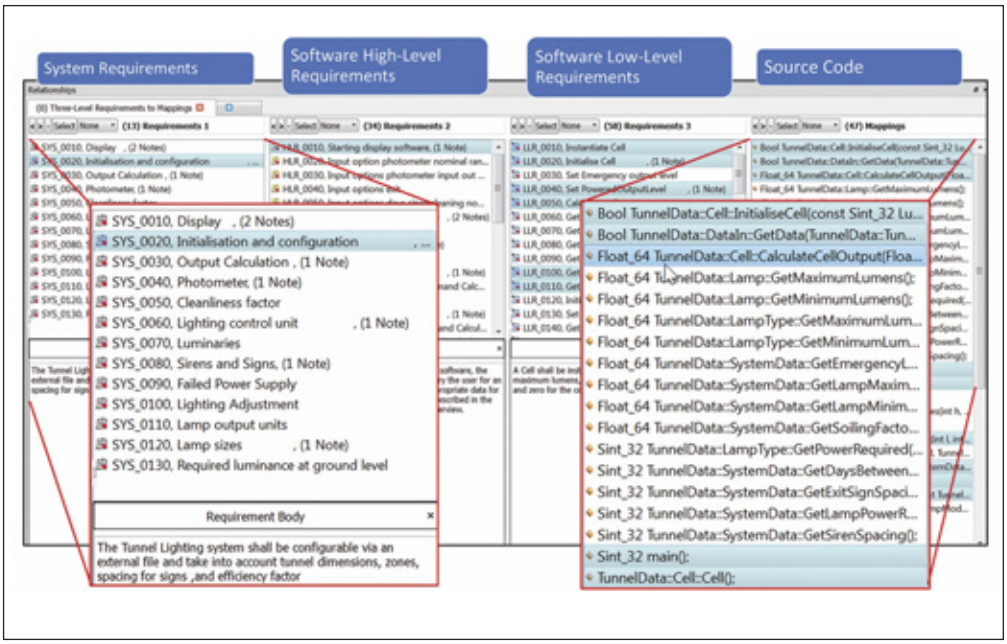


Fig. 3 – Identifying the impact of requirements change with the TBmanager component of the LDRA tool suite.

Dynamic data flow analysis builds on that accumulated knowledge, mapping coverage information onto each variable entry in the table for current and combined datasets and populating flow graphs to illustrate the control flow of the unit under test.

5.6 Software Integration and Integration Testing

Software integration testing focuses on the transfer of data and control across a software module's internal interfaces and external interfaces such as those associated with medical device hardware, operating systems, and third-party software applications and libraries. This activity requires the manufacturer to plan and execute integration of software units into ever larger aggregated software items, ultimately verifying that the resulting integrated system behaves as intended.

Integration testing can also be used to demonstrate program behavior at the boundaries of its input and output domains and confirms program responses to invalid, unexpected, and special inputs. The program's actions are revealed when given combinations of inputs or unexpected sequences of inputs are received, or when defined timing requirements are violated. The test requirements in the plan should include, as appropriate, the types of

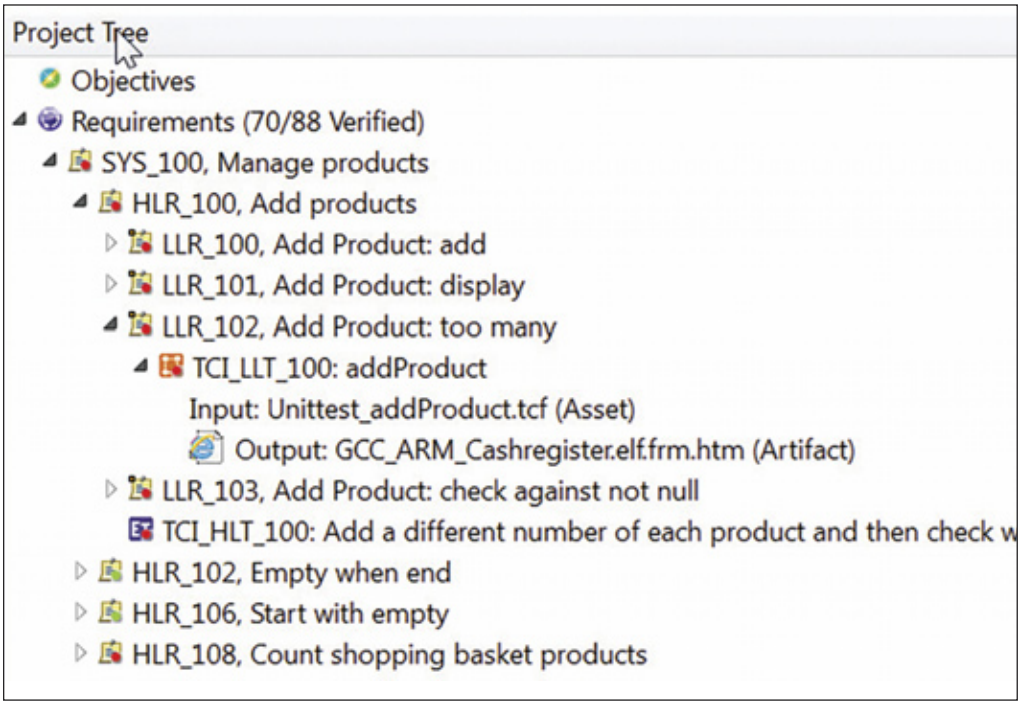


Fig. 4 – Showing functions requiring retest with the TBmanager component of the LDRA tool suite.

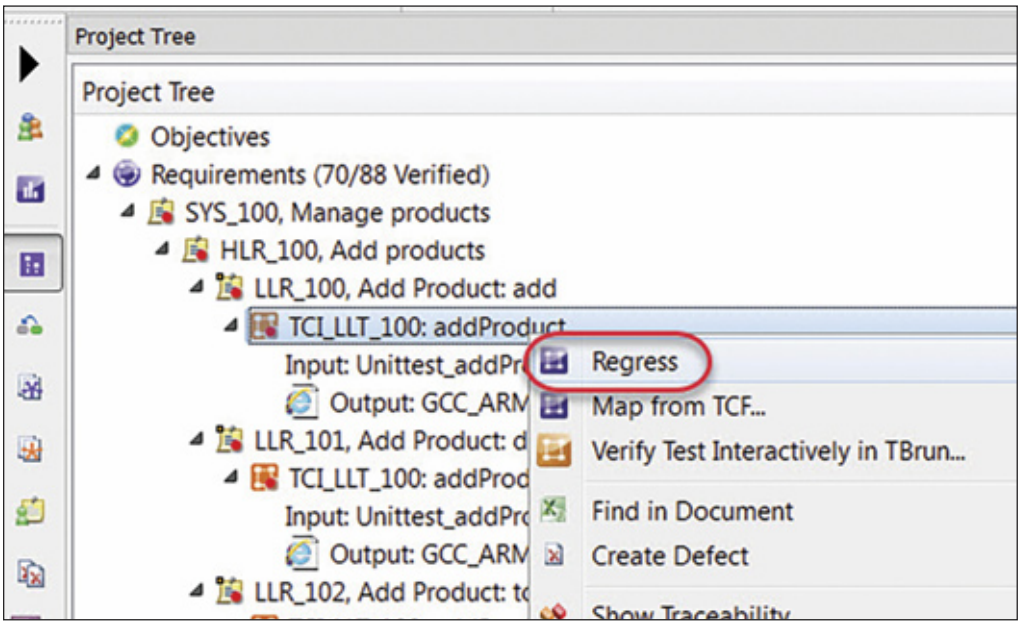


Fig. 5 – Regression testing reruns unit tests to show that the functionality they describe still holds true, using TBmanager and TBrun components of the LDRA tool suite.

white box testing and black box testing to be performed as part of integration testing.

Testing tools need to have the capability to provide dynamic structural coverage analysis, both at system test level and at unit test level, which is a mechanism to show which parts of the code base have been exercised during testing. The coverage data derived from unit and system test can be combined to provide the most effective way of working for the particular needs of a development project. A common approach is to operate them in tandem, so that, for instance, coverage can be generated for most of the source code through a dynamic system test and complemented using unit tests to exercise defensive code and other aspects.

Testing of changed source code can again be performed using regression testing. It is advisable to validate test cases as a matter of course and perhaps automatically, to ensure that any changed code has not affected proven functionality elsewhere.

The requirements for structural coverage metrics like statement, branch, condition, procedure/function call, and data flow coverage varies depending on device classification, and all are provided by both the unit test and system test facilities within tools such as the LDRA tool suite.

5.7 Software System Testing

Testing at the system level requires the manufacturer to verify the software's functionality by verifying that the requirements for the software have been successfully implemented. Software system testing demonstrates that the specified functionality exists in the system as it will be deployed, and that performance of the program is as specified.

Software system testing is in many ways the ultimate integration test, and comments relating to integration testing still apply here. Functional (black box) testing with no code coverage can also be performed although it might be desirable to use white box methods to more efficiently accomplish certain tests, initiate stress conditions or faults, or increase code coverage of the qualification tests.

Software system testing tests the integrated software and can be performed either in a simulated environment, on actual target hardware, or on the implemented medical device.

Software System Maintenance

These devices require maintenance when out in the field, and that also involves risks that must be tracked, managed, and mitigated in accordance with the processes and procedures laid out in the standard, as Figure 2 illustrates.

A high percentage of medical device software defects are introduced after product release, and many are related to the application of inappropriate software updates and upgrades. For that reason, the software maintenance process is considered to be as important as the software development process. Clause 6 of the IEC 62304 standard spells out details for the software mainte-

nance process, which has much in common with the software development process.

Subclause 6.3.1 states that the manufacturer shall apply the software development process as mentioned in clause 5 (5.3, 5.4, 5.5, 5.6, 5.7, and 5.8), or an established maintenance process to implement the modifications. Either way, it is important to manage changes in accordance with subclause 7.4.1, which requires that changes be analyzed to prevent introducing any harmful causes that could contribute to a hazardous situation. Furthermore, those changes are to be assessed to determine whether additional software risk control measures are required.

Given the similarity of the required processes before and after release, and assuming successful deployment of high assurance software in the first place, it makes sense to continue to apply the same tool chain as was used in development.

Establishing a software maintenance plan requires the manufacturer to create or identify procedures for implementing maintenance activities and tasks. To implement corrective actions, control changes during maintenance, and manage the release of revised software, the manufacturer should document and resolve reported problems and requests from users, as well as manage upgrades and associated modifications to the functionality of the medical device software.

The impact of each modification can therefore vary enormously, from a minor correction to the migration of an application to a new environment or platform. In each case, the objective is to modify the released medical device software while preserving its integrity. Where the changes require new development work, they should be managed in accordance with clauses from the body of the standard as well as those specific to maintenance.

Just like the requirements specified in the development of the device, achieving a format that lends itself to bidirectional traceability helps to achieve compliance with the standard. Bigger projects, perhaps with contributors in geographically diverse locations, are likely to benefit from an application life cycle management tool such as IBM® Rational® DOORS®, Siemens® Polarion® PLM®, or more generally, similar tools offering support for standard Requirements Interchange Formats.[1–3] Smaller projects can cope admirably with carefully worded Microsoft® Word® or Microsoft® Excel® documents, written to facilitate links between phases the development life cycle, whatever model is applied.

Identifying Necessary Retest

Many software modifications will require changes to the existing software functionality — perhaps with regard to additional utilities in the software. In such circumstances, it is important to ensure that any changes made or additions to the software do not adversely affect the existing code.

A requirements traceability tool such as LDRA TBmanager® helps alleviate this concern by automatically maintaining the connections between the requirements, development, and testing artifacts and activities. In the example in Figure 1, a change is proposed to the system-level requirement "Installation and configuration." The traceability established at development time between requirements, code, and tests mean that the tool can show which parts of the code are impacted by the proposed change, as highlighted in the example.

The existing code as launched should also have undergone quality control measures such as static analysis to assess whether coding standards have been met, unit tests to confirm functionality of each code module, and dynamic coverage analysis to show that all parts of the code have been exercised.

Figure 2 shows a display from a requirements traceability tool. In this example, a system has been subject to a change request for the "Add products" requirement. Those parts of the system that are potentially affected by the change are easily identified by means of a red dot, whereas unaffected functions carry a green dot.

In the example, there is a test case file (tcf) associated with each of four low-level requirements: "add", "display", "too many," and "check against not null." Those files retain the test vectors associated with each of these low-level requirements, meaning that they can all be rerun at the touch of a button and their functionality confirmed.

Of course, some tests will require changes to reflect the new functionality dictated by the updated requirement, but in many cases, it is enough to confirm that things work as they did previously. Automating the process in this way means that doing so requires minimal effort.

Development process artifacts such as coding standards compliance reports and code coverage reports are also retained and associated with requirements, so that as the tests are rerun (regression tested), the artifacts are regenerated to go with them (see Figure 3).

Conclusion

A software functional safety standard such as that prescribed by IEC 62304 with its many sections, clauses, and subclauses may at first seem intimidating. However, once broken down into digestible pieces, its guiding principles offer sound guidance in the establishment of a high-quality software development process, not only leading up to initial product release but also into maintenance and beyond.

Such a process is paramount for the assurance of true reliability and quality — and above all — the safety and effectiveness of medical devices. When used with a complementary and comprehensive suite of tools for analysis and testing, it can smooth the way for development teams to work together to effectively complete large projects with confidence in their quality.

References

1. IBM® Rational® DOORS®, http://www-03.ibm.com/software/products/en/ratidoor
2. Siemens® Polarion® PLM®, https://polarion.plm.automation.siemens.com/
3. Object Management Group, Requirements Interchange Format™, http://www.omg.org/spec/ReqIF/

This article was written by Mark Pitchford, Technical Specialist for LDRA, Wirral, UK.