B L A C K W O O D D E S I G N S

Quality Software Design

SOFTWARE DESIGN GUIDES

AUTHOR RANDALL MAAS

OVERVIEW This guide describes the design of high quality software for embedded systems. The intent is to promote well-founded, justified designs and confidence in their operation. It provides guides, checklists and templates.

BENEFITS
Improve the quality of source code: its maintainability, testability, *etc.*
Prevent potential defects
Smoother, shorter design / release cycles
Better software products

TEMPLATES
Design documentation templates
Design review checklists

Software Risk Analysis Templates
Bug reporting template
Coding Style guides for C and Java
Bug defect type classification
Code review checklists
Code quality rubric

FILE: G:\My Documents\BD4 SW Design Documents\guide-quality software\BD4 Quality Software Guide.doc

RANDALL MAAS has spent decades in Washington and Minnesota. He consults in embedded systems development, especially medical devices. Before that he did a lot of other things… like everyone else in the software industry. He is also interested in geophysical models, formal semantics, model theory and compilers.

You can contact him at randym@randym.name.

LinkedIn: http://www.linkedin.com/pub/randall-maas/9/838/8b1

Preface

This guide aims to provide relevant tools to support creating quality software. It tries to do so in a manner that the reader may apply to their projects. Why create such a thing? As a consultant who has seen many client development organizations, I've found that few have the material that I present here. None has any guidelines on good software designs, design reviews and hazard analysis of software. Many lack coding style guide, code review guidance, and bug reporting standards. If they do have code guidelines, it is sparse, and could do so much more to improve quality.

This is a guide will only cover the quality of software design and the workmanship of source code. It does not cover:

- Writing software requirements
- Testing of the software
- Debugging the software
- Project and development management
- Planning, scheduling or budgeting

1. ORGANIZATION OF THIS DOCUMENT

This guide is written in 3 parts, with the broadest up front, and the most specific or esoteric toward the rear.

- CHAPTER 1: PREFACE. This chapter describes the other chapters.

PART I: SPECIFICATIONS.

- CHAPTER 2: OVERVIEW OF SOFTWARE DESIGN QUALITY. Introduces what is meant by quality.
- CHAPTER 3: PROCESS.
- CHAPTER 4: REQUIREMENTS CHECKLISTS. This chapter provides checklists for reviewing requirements.
- CHAPTER 5: SOFTWARE RISK ANALYSIS.

PART II: SOFTWARE DESIGN & DOCUMENTATION. This part provides guides for software design and its documentation

- CHAPTER 6: DESIGN OVERVIEW & WRITING TIPS.
- CHAPTER 7: GUIDELINES FOR HIGH-LEVEL DESIGNS. Provides guidelines for high-level designs (e.g. architectures).
- CHAPTER 8: SOFTWARE ARCHITECTURE RISK ANALYSIS.
- CHAPTER 9: GUIDELINES FOR DETAILED DESIGNS. Provides guidelines for detailed designs (e.g. major subsystems or "stacks").

- CHAPTER 10: PROTOCOL DOCUMENTATION TEMPLATE. Provides a guide for protocol documentation.
- CHAPTER 11: SOFTWARE MODULE DOCUMENTATION TEMPLATE. Provides a guide for detailed design documentation of a module.
- CHAPTER 12: GUIDELINES FOR MODULE DESIGNS. Provides guidelines for low-level module designs.
- CHAPTER 13: DESIGN REVIEWS CHECKLISTS. Provides checklists for reviewing a design.
- CHAPTER 14: SOFTWARE DETAILED DESIGN RISK ANALYSIS. Describes reviewing software for hazard analysis.

PART III: SOURCE CODE CRAFTSMANSHIP. This part provides source code workmanship guides

- CHAPTER 15: OVERVIEW OF SOURCE CODE WORKMANSHIP. Provides TBD.
- CHAPTER 16: C CODING STYLE. This chapter outlines the style used for C source code.
- CHAPTER 17: JAVA CODING STYLE. This chapter outlines the style used for Java source code.
- CHAPTER 18: CODE INSPECTION & REVIEWS. Describes code reviews.
- CHAPTER 19: CODE INSPECTION & REVIEWS CHECKLISTS. Provides checklists for reviewing source code.
- CHAPTER 20: BUG MANAGEMENT. Describes the approaches to managing bugs

APPENDICES: The appendices provides extra material

- APPENDIX A: ABBREVIATIONS, ACRONYMS, & GLOSSARY. This appendix provides a gloss of terms, abbreviations, and acronyms.
- APPENDIX B: PRODUCT STANDARDS. This appendix provides supplemental information on standards and how product standards are organized
- APPENDIX C: BUG REPORTING TEMPLATE. This appendix provides a template (and guidelines) for reporting bugs
- APPENDIX D: TYPES OF DEFECTS. This appendix provides a classification of different kinds of software defects that are typically encountered.
- APPENDIX E: CODE COMPLETE REQUIREMENTS REVIEW CHECKLISTS. This appendix reproduces checklists from *Code Complete, 2nd Ed* that are relevant to requirements reviews.
- APPENDIX F: CODE COMPLETE DESIGN REVIEW CHECKLISTS. This appendix reproduces checklists from *Code Complete, 2nd Ed* that are relevant to design reviews.
- APPENDIX G: DESIGN REVIEW RUBRIC. This appendix provides rubrics relevant in assessing the design and its documentation.
- APPENDIX H: FLOATING POINT PRECISION. This appendix recaps the limits of floating point precision.
- APPENDIX I: CODE COMPLETE CODE REVIEW CHECKLISTS. This appendix reproduces checklists from *Code Complete, 2nd Ed* that are relevant to code reviews.
- APPENDIX J: SOFTWARE REVIEW RUBRIC. This appendix provides rubrics relevant in assessing software workmanship.

REFERENCES AND RESOURCES. This provides further reading and referenced documents.

"The project development people seemed to be a special breed of programmers whose incomprehensibility was matched only by their desire to document in a level of detail that baffled the minds of ordinary folk."
– NSA Cryptolog, 1979 June

[This page is intentionally left blank for purposes of double-sided printing]

PART I

Specifications

This first part provides guides for software design and its documentation

- OVERVIEW OF SOFTWARE DESIGN QUALITY. Introduces what is meant by quality.
- PROCESS
- REQUIREMENTS CHECKLISTS. This chapter provides checklists for reviewing requirements.
- SOFTWARE RISK ANALYSIS.

[This page is intentionally left blank for purposes of double-sided printing]

"The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging of the specification."
– Fred Brooks

CHAPTER 2

Overview of Software Design Quality

This chapter promotes good software quality:

- Software quality overview
- Where do bugs come from?
- How quality software can be achieved
- A tip on staffing

2. OVERVIEW

Software lives as part of a system within a product. Typical embedded software can be described as fit into a hierarchy of systems and subsystems:

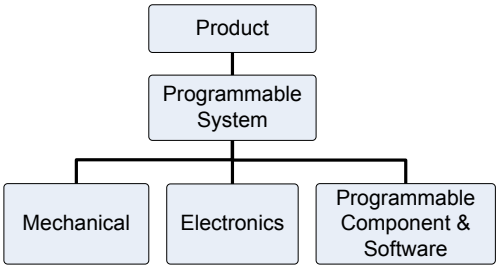

Figure 1: *The hierarchy of systems & subsystems*

There is the "final" *product* itself, with a portion – sometimes a large portion, sometimes a small portion – that is the *programmable system*. This system has mechanical and electronic subsystems, as well as the *programmable component* (usually a microcontroller) that is executing the software that will be discussing through this guidebook.

The diagram below synopsizes the levels of abstraction in the normative software development process. Guidance documents help the work to be performed be done quickly, and with appropriate craftsmanship. The tests and reviews help catch errors and improve the construction of the software.

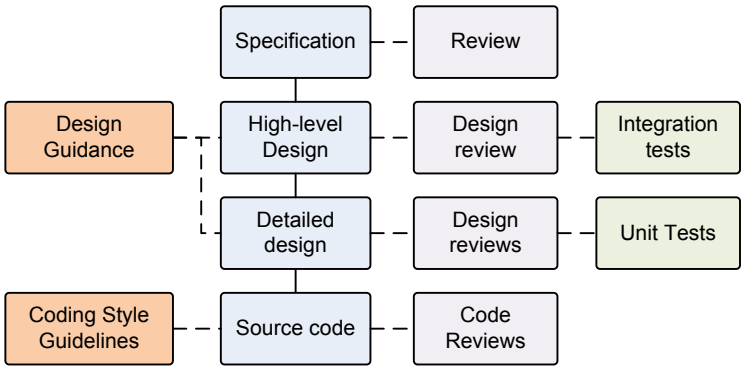

Figure 2: *Levels of abstraction in development process*

Review checklists & rubrics should be a dual (twin) to the coding style. Everything in one should be in the other.

3. SOFTWARE QUALITY OVERVIEW

It may be helpful to provide a brief overview of what "software quality" is. ISO/IEC 25010 model of software quality is one useful way to organize the areas of quality:

Quality factor	Quality Criteria
Functionality	Completeness, Correctness, Appropriateness
Performance & Efficiency	Time behavior, Resource utilization, Capacity
Compatibility	Interoperability
Usability	Appropriateness, Recognisability, Learnability, Operability, User error protection, Aesthetics, Accessibility
Reliability	Maturity, Availability, Fault tolerance, Recoverability
Security	Confidentiality, Integrity, Non-repudiation, Accountability, Authenticity
Maintainability	Analyzability, Modifiability, Modularity, Reusability, Testability
Portability	Adaptability, Installability, Replaceability

Table 1: *ISO/IEC 25010 model of software quality*

McCall's model is another way to organize the areas of quality. It maps each top-level area of quality to a more specific quality.

Quality factor	Quality Criteria
Correctness	Traceability, Completeness, Consistency
Reliability	Consistency, Accuracy, Error tolerance
Efficiency	Execution efficiency, Storage efficiency
Integrity	Access control, Access audit
Usability	Operability, Training, Communicativeness
Maintainability	Simplicity, Conciseness, Self-descriptiveness, Modularity
Testability	Simplicity, Instrumentation, Self-descriptiveness, Modularity
Flexibility	Simplicity, Expandability, Generality, Modularity

Table 2: *McCall model of software quality*

Portability	Simplicity, Software system independence, Machine independence
Reusability	Simplicity, Generality, Modularity, Software system independence, Machine independence
Interoperability	Modularity, Communications commonality, Data commonality

These same metrics apply to the *programmable system,* and perhaps the product overall.

3.1. WHERE DO BUGS & DEFECTS COME FROM?

Where do the bugs & defects come from?

- The wrong requirements – that the product and programmable system was designed to the wrong set of rules.
- Operation action and input – inconsistent settings, out of range entries, and so forth. These errors indicate insufficient requirements about the constraints on the user interface.
- Poor design – a design is unsound, an algorithm has too high of computational complexity, bottlenecks & contention for resources, prioritization issues, etc.
- Edge case circumstances, such as race conditions and overloading of processing resources.
- Programmer mistakes, such as language mistakes, or incorrect of use of hardware – use of disabled peripherals, bad parameters, index out of range, hardware exceptions, divide by zero, and the like. These are often in the form of "exceptions" and "assert" failures.
- Hardware components may have shifted values; connections break.
- Environmental conditions – such as a component being used out of its operating range, a low battery, and so forth.

It is important to note: the software can perform with high quality, and the programmable system low quality. This can come from the wrong requirements, at any level.

3.2. HOW QUALITY SOFTWARE CAN BE ACHIEVED

Steps to quality software include recognizing that

- It is an acquired, disciplined art.
- It requires practice, diligence and assessment
- Organizations must teach how to write quality code.
- The organization must value quality software in order for the individual to value it
- The development organization has a culture of accountability and commitment
- There is encouragement for respectful, frank, rational conversations about failures
- Information, activities and agreements are explicitly communicated (rather than tacit and assumed)

3.3. TESTING

Testing

- Has an important role in quality
- Most often removes the "easy" and frequent bugs
- Won't find subtle timing bugs and edge cases. It can help regression test to ensure that specific occurrences do not recur
- Doesn't improve workmanship

4. A TIP ON STAFFING

This guidebook generally does not address development process – plans, schedules, sequencing, staffing, and so on. However, here are some opinionated tips:

1. Assign leadership to those who care about the quality. In any organization, there is a leader somewhere who capitates the quality – even if there is a leaders who drives a minimum quality standard. It doesn't matter if the quality is something aesthetic (like being stylish & usable), or a process quality (like being maintainable and traceable), or other quality.
2. Work with people who value the development artifacts they are creating and the processes they work in. For instance, my experience has been that people who dislike writing or reading documentation will create poor documentation and the hate shines thru.
3. Encourage *gracious professionalism*[1] where the staff is fiercely driven, seeks mutual gain, are intensely respectful and kind
4. Reduce stress. Faux urgency and cranking up the time pressure is a common managerial technique in too many places. Meeting regular shipment schedules or quality goals is a long marathon.

In short, *care* and *drive* (or *passion, internal motivation*, *pride*).

5. REFERENCES AND RESOURCES

ISO/IED FDIS 25010:2011, *"Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models"* 2011

IEEE Std 730-2014, IEEE *Standard for Software Quality Assurance Processes*, 2014

IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*

[1] Coined by Dr Woodie Flowers, registered trademark of FIRST

CHAPTER 3

Process

This chapter describes the software development process:

- Process, specifications, and requirements
- The role of standards & certification
- System engineering
- Development plan
- Risk analysis
- Testing, Verification, Validation, and Testing

6. PROCESS

A process is how – implicitly or explicitly – an organization achieves a goal. Explicit processes decompose the steps of what an organization may do (or must do or should do), spelling out the activities and artifacts (more importantly information to be captured in the artifacts). Rigorous processes attempt to assure that

- the project will succeed, *project assurance*
- that the schedule will be reasonably met,
- the cost of development is acceptable,
- the product is acceptable & performs as intended *design assurance*
- the product does not pose an unacceptable risk of harm
- the product is well made
- the product can be kept in use / operation for a time period, including revising and maintaining the product.

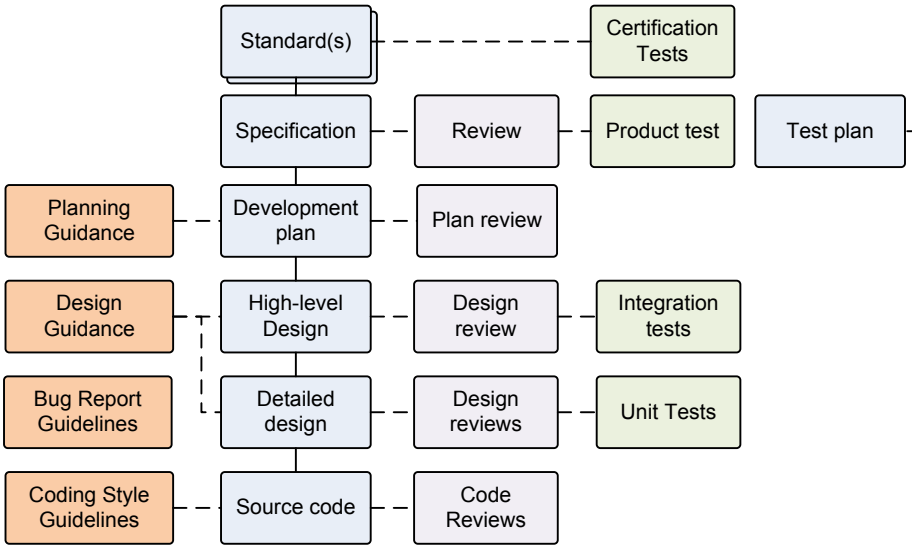

Figure 3: *Levels of abstraction in development process*

The motivation to use rigorous processes is to more directly check that the quality (especially the safety related quality) is done right. As opposed to being satisfied by other aspects of the development and concluding that quality is acceptable.

A design should be thoughtfully worked out, drilling down from the high-level specifications to the more specialized specifications, and designs.[2] Ideally – and depending on the rigor – each should be assessed for appropriateness, matching the products intent and requirements. Once a module's design has been approved, its source code may created in earnest.[3]

The process should call out (and provide) workmanship guidelines, style guides, standards, and evaluation rubrics used to craft the source code; this is often done in the development plan. One goal of the guides is to provide direction to producing clear code, with a low barrier to understanding and evaluation. The following chapters provide reference guides.

The source code should be reviewed (and otherwise inspected) against those guides, and designs. The purpose of reviewing the work is to examine quality of construction – it is not an evaluation of the engineers, and it is more identifying defects.

6.1. THE DIFFERENT TYPES OF SPECIFICATION DOCUMENTS

The documents – or portions of documents – discussed here include:

high-level specification

A *high-level specification* is a finite set of requirements specification, e.g. system specification, customer inputs, marketing inputs, etc.

requirements specification

A *requirements specification* is a set of requirements, and clear text explaining or justifying the requirements. A justification may base the requirement in other documents, such as research, standards, regulations or other laws.

---

[2] Designing of a "lower" layer can begin (and often does) based on the anticipated top-level design, and norms for the lower layer. Its completion is dependent the top-level design being settled.

[3] Not all reviews or designs must be complete before implementations begin, except in the most stringent of processes. Modules built in an investigatory (or as a short-term shim) fashion are useful but should be considered in an "as-is" or draft state, until they have been revised to match the design, workmanship rules, and so in.

- A *requirement* defines what an item must do, and often is presented as text in a special form. *requirement*
- A *customer requirement* is a requirement in any of the top-level documents, but especially in the customer (or user) requirements specification. *customer requirement*
- A *comment* is text, usually to provide context, clarify or explain the requirement(s). *comment*
- An *identifier* can refer to product, specific version of the product, a document, requirement, test, external document, or comment. In practice this is so important that each item is given a label. *identifier*

A *design document* explains the design of a product, with a justification how it addresses safety and other concerns. *design document*

Test specifications describe a set of tests intended to check that the product meets it requirements. The test specifications define: *test specification*

- A set of *test requirements* that define what tests a product must pass. *test requirements*
- A set of *test procedures* that carry out the test requirement and test the product *test procedure*
- A mapping of a test requirement to a set of requirements that it tests. {note: this may be covered in the trace below.}

A *test report* is a set of outcomes: <*test id, product id, result*> describing how a product performed under test. (The performance may vary with versions of the product) *test report*

A *trace matrix* is used to identify requirements in a higher level specification that are not carried thru to lower requirements specifications and designs; and (in stringent cases) identify features of the design without requirements, and requirements in lower documents that are not drive by requirements at a higher level. Logically it defines two functions, forming a directed acyclic graph: *trace matrix*

- It maps a requirement to the set of requirements that it directly descends (or dervices) from
- It maps a requirement to a set of requirements that directly or indirectly descends from it.

6.2. CRITICAL THINKING

Quality oriented – and especially safety oriented – processes apply analysis and reasoning to further improve the product being developed. All processes try to the address what/why/where/when/how questions, by identifying where the information is or comes from:

What are we making?

1. The high level specification

How do we know that we have the right (product) specification(s)?

1. Standards
2. Stakeholder reviews
3. Customer feedback (e.g. voice of customer)
4. Hazard analysis
5. Usability studies
6. Field tests

How do we know that the product meets the specification(s)?

1. Verification activities of the system and subsystem
2. Validation activities of the product

Why are we confident that product is well made and safe?

1. Reviews of specifications and design
2. Analysis of the specification for key qualities, esp. safety
3. Verification & validation, testing

How do we know if a part of a higher-level specification was missed when making a lower-level (more specific) specification?

1. Tracing
2. Validation & validation, testing

How do we know what to do?

1. Specifications
2. Development plans
3. Guidelines, e.g. coding style guides, design guides
4. Development protocols & work instructions

Why the product was designed and made this way?

1. Specifications
2. Guidelines, e.g. coding style guides, design guides
3. Design documentation
4. Design reviews

and so on

7. THE ROLE OF CERTIFYING STANDARDS

Product certification – specifically the standards being certified against – may drive software quality. Standards approach software quality as necessary to achieve product quality, especially safety and security. To simplify (and over generalize), such standards have specifications that address the following areas of software quality:

- *Risk management*, including analysis, assessment and control of the risks
- The process and artifacts, and how they will be stored and updated. These include a *software development lifecycle* (SDLC) and *quality management systems* (QMS)
- *Techniques* to be applied in the software design and implementation
- *Tests* and characterizations to be applied.

Some important examples of the safety-facing standards are:

- IEC 61508: *Functional safety of electrical/electronic/programmable electronic safety-related systems* (Part 3 deals with software and Part 7 with specific techniques)
- IEC 60730: *Automatic Electrical Controls*. (Annex H deals with software)
- ANSI/IEC 62304:2006 *Medical Device Software – Software Lifecycle Processes*

- DO-178C, *Software Considerations in Airborne Systems and Equipment Certification*
- IEEC Std 7-4.3.2 2010 *IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations*

7.1. IEC 61508 AND DERIVATIVE STANDARDS (E.G. IEC 60730)

IEC 61508

IEC 61508's has many process facing areas, over a complete safety life cycle. It mandates

- A specific safety management approach, parallel to the development of primary functionality. This produces a set of *software safety requirements*.
- A specific risk management approach, including a risk assessment and analysis approach that is far more strenuous than the art in many fields. (And was when it was introduced).
- A software development lifecycle[4], with several activities to be performed, and several artifacts to be produced.
- A mandate and guidance to apply very specific & detailed software design and implementation techniques, depending on the classification of software. Most of the techniques had been documented at least two decades prior to the first version of the version (1998-2000); all were documented at least decade prior. Most, however, were not in common use outside of niche applications.

see Appendix B for more

Several IEC standards adapt IEC 61508 (a *basic safety publication*) for an industry segment, a groups of products (a *group safety publication*), or specific applications (*product publications*). The IEC group safety publications may normatively reference the IEC 61508 standard (that is, mandate its use), or choose to incorporate the relevant portions into the narrower standard, with some modifications. The product publications are specific standards targeting requirements of specific categories of products or applications. These specific standards often modify the group standard, reducing the stringency in some areas.

IEC 60730

IEC 60730-1 incorporates much of IEC 61508's software requirements (but not the risk assessment system) for home appliances. This includes the production of software safety requirements. The IEC 60730-2-xyz standards specify requirements for various types of appliances. IEC 60730 divides functionality (including software function) into three categories of safety:

see Appendix B for other classifications

- Class A are the functions that are not relied upon for safety
- Class B are the functions that directly (or indirectly) prevent unsafe operation
- Class C are the functions that directly (or indirectly) prevent special hazards (such as explosion).

IEC 60335

IEC 60335 follows the same pattern: 60335-1 incorporates most (but not all) of IEC 60730 software requirements. The IEC 60335-2-xyz standards specify requirements for various types of appliances.

This guidebook has been structured in such a manner to directly support software development under these standards. This includes not just software design & implementation, but the artifacts: requirements, design, and documentation.

[4] Modern software development lifecycle can be found in IEEE Std 12207 (ISO/IEC 12207).

7.2. ANSI/IEC 62304

ANSI/IEC 62304

ANSI/IEC 62304 is a software development lifecycle document, and it is organized in the classic "v-model" fashion. It mandates a variety of artifacts and activities in the software development. It works intimately with a separate risk management process, and quality management system.

Like IEC 60730, it divides software into three categories of safety:

- Class A are the functions that pose no risk of injury
- Class B are the functions that pose a "non-serious" risk of injury
- Class C are the functions that could result in death or serious injury

It mandates a formal development processes, including checkpoints with formal reviews and signoffs by key personnel, assuring successful completion of all criteria.

This guidebook has been structured in such a manner to directly support software development under these standards. This includes not just software design & implementation, but the artifacts: requirements, design, and documentation.

Note: ANSI/IEC 62304 is meant to work with a risk management approach, but – unlike IEC 61508 – it is expected to be provided separately. It also expects to work with a separated defined quality management system.

7.3. A SIDE NOTE ON THE ECONOMIC BENEFIT TO DEVELOPMENT

Vendors have developed support for these software functions, as these functions are employed in a many product markets. Their support is in the form of certified microcontroller self test libraries, and application notes giving guidance on how to meet these standards (especially using their libraries).

This standardization also provides a means of identifying the skills and experience needed, and thus able to find expert workers.

7.4. THE SAFETY ELEMENTS

The standards atomize – with respect to behaviour and element of electronics and software – the product functions & requirements into:

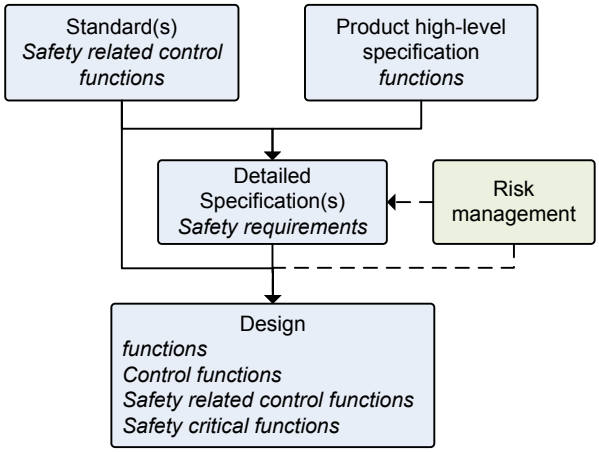

Figure 4: *Where key functions & requirements are identified in the process*

function

The high-level specification of the product defines the intended, primary *function* of the product. The function is its role or purpose, and the operations that it is intended to perform.

control function

The standards identify *control functions*[5] that are to be provided by the product and its design. The standards categorize functions along three axes:

1. Whether or not it is a control function relevant to safety (earlier this was rated as type A, B, or C);
2. Whether or not the function is critical to the operation of a control function used in safety
3. Whether or not software is responsible (at least in part) for the function

This becomes:

safety-related control function

S*afety-related control functions* are a type of control function that prevents unsafe conditions and/or allows the operator to use the equipment in the intended, safe manner. In IEC 60730 *Type B* control functions prevent unsafe state; *Type C* prevents special harms. The product specifications and design often expand the number of control functions, and elaborate their specific operation.

safety-critical function

The *safety critical functions* are those functions that, should they fail, present a hazardous situation. This may be because they impair the ability for safety-related control function to fulfill its specification. The standards impose a variety of software functions to "self-check" that the microcontroller (or other programmable element) is functioning sufficiently to carry out the other functions. A safety-related control function is often (but may not be) a safety critical function, but not all safety critical functions are safety-related control functions.

When software is used to realize (i.e. implement) a safety-related control function, or a safety critical function, the standards impose a many requirements on the design and behaviour of the software (and supporting electronics). This is a very good thing for quality, and this guidebook is intended to help address these.

safety requirements
software safety requirements

The product and subsystem specifications are to provide a detailed set of *safety requirements*, which specify in detail the functional behaviour of the product, and each of those safety-related control functions and safety-critical functions. This is true for the functions implemented by software. The *software safety requirements* are to provide added requirements that address:

- potential faults in the software as well as the programmable element (e.g. the microcontroller) and the electronics,
- construction techniques of the software to prevent or mitigate software flaws

The motivation to use a rigorous process is to more directly check that the safety related behaviour is done right. As opposed to being satisfied by other aspects of the development and concluding that safety is acceptable.

[5] Function(s) can have types (or roles) such as control, filter, protection, monitoring, test, conversion, limiting, distribution, isolation, protection, and so on.

8. SOFTWARE DEVELOPMENT PLAN

A development plan should be put into place before the software is created. The development plan typically includes:[6]

- Names
- Location of artifacts and sources
- Tools and key components
- Workmanship guides and how the workmanship will be evaluated. This includes a *coding style guide*, which identifies a good, restricted subset of a programming language that is acceptable to use.
- Steps that will be done in the development process, such as reviews and risk analysis
- How changes to the software will be managed. What is the source code repository? Is commit approval required from a module owner? (e.g. the owner evaluates the appropriateness of the changes to their area of the code base.)
- How issues, bugs and so on are tracked, prioritized, and dispositioned. Example templates for bug reporting can be found Appendix C and categorization of the defect in Appendix D.

Software development plans are about being organized to succeed, and to keep succeeding for a long time. Most projects (e.g. those lasting a few months with a small number of people) do not need to spell out all of the potential elements; the ones listed above are often sufficient.

9. RISK ANALYSIS

UCRL-ID-122514

At regular steps, an analysis is performed to double check that the safety control functions, safety requirements, and design are provide a acceptably safe product. The objective of these analyses is "to identify and correct deficiencies and to provide information on the necessary safeguards."

hazard analysis

A *hazard analysis* is a process that is performed on the *product*, its specifications, functions, and design.

harms

- It identifies a set of potential *harms* that the product (or its use) presents

severity
severity class

- It maps a harm to *severity* or *severity class*

hazards
hazard class

- It identifies a set of *hazards* or *hazard classes* that are potential sources of harm

likelihood

- It maps a hazard or hazard class to *likelihood* or frequency that it may occur

risk acceptability level
risk acceptability matrix

- It maps the combined severity of harm and likelihood of occurrence to an *acceptability level*. This is done using an accepted rubric, most often a *risk acceptability matrix.*

The acceptability level is used to prioritize changes to the specifications and design. The changes must be made until there are no unacceptable risks presented, and that the cumulative (overall) risks presented is at an acceptable level. The changes often included added functions (such as tests of the hardware or operating conditions), tighter conditions on existing requirements, added requirements, and the like.

[6] A development plan includes much more, related project assurance, process, management, staffing, etc.

A *risk analysis* follows the same pattern, checking that the specification, functions and design of a subsystem for the risks that the subsystem will present a hazard. A *software risk analysis* is what the software may contribute to risk or control of the product risks.

risk analysis
software risk analysis

- Each risk analysis builds an upon earlier risk analysis
- Each type of analysis may produce a different, but related, form of output
- Each produces a summation of hazards (and risks), any identified rework, and mandates for tests for Verification & Validation activities.

9.1. INPUTS AT EACH STAGE OF SOFTWARE RISK ANALYSIS

Software is analyzed at several stages of development to assess how it will impact products risk. The table below summarizes the inputs to each of the software risk analysis:

***Table 3:** Inputs for each kind of risk analysis*

Requirements risk analysis	**Architecture risk analysis**	**Detailed Design risk analysis**	**Source code analysis**
Product Preliminary Hazards list	*Product Preliminary Hazards list*	*Product Preliminary Hazards list*	*Product Preliminary Hazards list*
Product Risk analysis	*Product Risk analysis*	*Product Risk analysis*	*Product Risk analysis*
Programmable system requirements	*Programmable system requirements*	*Programmable system requirements*	*Programmable system requirements*
Programmable system description	*Programmable system description*	*Programmable system description*	*Programmable system description*
Software requirements	*Software requirements*	*Software requirements*	*Software requirements*
	Software requirements risk analysis	*Software requirements risk analysis*	*Software requirements risk analysis*
	Software architecture description	*Software architecture description*	*Software architecture description*
		Software architecture risk analysis	*Software architecture risk analysis*
		Software design description	*Software design description*
			Software design risk analysis
			Coding style guide
			Source code

The risk analysis of the source code itself is covered by specialized code reviews.

10. TERMS RELATED TO TESTING, VERIFICATION, AND VALIDATION

A *fault* is a system or subsystem deviating from its specification, e.g. not meeting one or more of its functional requirement.

A *failure* is not providing service to the user, e.g. not meeting user requirement, often a user non-functional requirement.

Verification[7] is set of activities that include

- Testing the item against its specifications.
- Inspecting and review the items standards, specifications, design, and construction

Validation includes verification of the item, *and* activities that include

- Testing the item against the *higher-level* (such as the product's) specifications.
- Inspecting and review the items against the *higher-level* (such as the product's) standards, specifications, design, and construction
- Testing the item against use cases
- Performing field trials, usability studies
- Evaluating customer feedback.

11. REFERENCES AND RESOURCES

DO-178C, *Software Considerations in Airborne Systems and Equipment Certification,* RTCA, Inc. 2012 Jan 5

> This is a particularly stringent standard. It seeks to ensure that not only ensure that all requirements and functions (from the top on down) are carried thru and test… it also seeks proof that no element of software, function, or requirement is present unless it traces all the way back to the top.

RTCA/DO-254, *Design Assurance Guidance for Airborne Electronic Hardware*, RTCA, Inc. 2000 Apr 19

IEC 61508: *Functional safety of electrical/electronic/programmable electronic safety-related systems* 2010

> Part 3 deals with software and Part 7 with specific techniques. The software and electronics techniques to check the programmable element function are well founded, providing a good explanation of their motive, approach, and many references for each.
>
> The testing portion is a bit of a muddle.

IEC 60730: *Automatic Electrical Controls*, 2010

> Annex H deals with software

UL 1998, *Standard for safety – Software in Programmable Components*

11.1. RISK MANAGEMENT

UCRL-ID-122514, Lawrence, J Dennis *"Software Safety Hazard Analysis"* Rev 2, U.S. Nuclear Regulatory Commission, 1995-October

ISO 14971:2007, *Medical devices – Application of risk management to medical devices*

EN ISO 14971:2012, Medical devices. *Application of risk management to medical devices*

> This standard is for the European market; the earlier one is used rest of the world

[7] As there are many muddled definitions of verification and validation, I am using definitions that are compatible the FDA guidance, DO-178C, and DO-254

Speer, Jon *"The Definitive Guide to ISO 14971 Risk Management for Medical Devices"* Greenlight Guru, October 5, 2015
https://www.greenlight.guru/blog/iso-14971-risk-management

A clear introduction to the concepts and steps, with some elegant diagrams.

11.2. DEVELOPMENT LIFECYCLE

ANSI/IEC 62304:2006 *Medical Device Software – Software Lifecycle Processes*

This is a well written standard on the development life-cycle.

ATR-2011(8404)-11, Marvin C. Gechman, Suellen Eslinger, "*The Elements of an Effective Software Development Plan: Software Development Process Guidebook*" 2011-Nov 11, Aerospace Corporation, Prepared for: Space and Missile Systems Center, Air Force Space Command
http://www.dtic.mil/dtic/tr/fulltext/u2/a559395.pdf

The above guide is particularly rigorous and intended for long-lived project (e.g. two decades) with large & changing hierarchies of many people working for many different organizations (thus many organizational boundaries), across a geographic area, and wide range of organizational roles and backgrounds. The SDP is creating an institution for the development & maintenance.

ISO/IEC/IEEE 12207:2017(E) "*Systems and software engineering – Software life cycle processes"*

This standard is a successor to J-STD-016, which is a successor to MIL-STD-498, which is a successor to DOD-STD-2167A and DOD-STD7935A. (And that only dates to the 1980s!) It "does not prescribe a specific software life cycle model, development methodology, method, modelling approach, or technique."

ISO/IEC/IEEE 15288:2015, *Systems and software engineering – System life cycle processes*

Wikipedia, *Software development process*
https://en.wikipedia.org/wiki/Software_development_process

Provides a history of the different contributions to software development processes

11.3. QUALITY MANAGEMENT, TEST

FDA, "*Design Control Guidance for Medical Device Manufacturers,"* 1997 March 11

IEEE Std 1012-2004 - *IEEE Standard for Software Verification and Validation*. 2005. doi:10.1109/IEEESTD.2005.96278. ISBN 978-0-7381-4642-3.

ISO/IED FDIS 25010:2011, *"Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models"* 2011

ISO/IEC/IEEE 29119, *Software Testing Standard*

ISO/IEC/IEEE 29119-1: *Concepts & Definitions*, 2013 September

ISO/IEC/IEEE 29119-2: *Test Processes*, 2013 September

ISO/IEC/IEEE 29119-3: *Test Documentation*, 2013 September

ISO/IEC/IEEE 29119-4: *Test Techniques*, 2015 December

ISO/IEC/IEEE 29119-5: *Keyword Driven Testing*, 2016 November

ISO/IEC 90003 *Software engineering – Guidelines for the application of ISO 9001:2008 to computer software*

CHAPTER 4

Requirements Checklists

This chapter provides a requirements review checklist

12. OVERVIEW OF WELL WRITTEN REQUIREMENTS

The presentation of a requirement in the text should include:

- *Clear demarcation* of the requirement. For instance, place the requirement on an indented line, by itself.
- A means to *uniquely identify* or refer to the requirement. It is important to be able to identify the requirement be discussed. The requirement will be referred to in other documents, trouble tracking, etc.
- A brief *summary* of the requirement and its purpose or intent.
- *The actor* who carries out or meets the requirement. The actors should be defined earlier in the section or the document.
- *What* the actor is to do
- *Time bounds*: how fast, how long, how soon it act or when, etc
- What value and bounds
- *Rationale*, the description of the requirements role, purpose, motivation, and/or intent must be clear and readable

12.1. PROPERTIES OF A GOOD REQUIREMENT

A well-written requirement exhibits the following characteristics:

- Complete – contains sufficient detail to guide the work of the developer & tester
- Correct – error free, as defined by source material, stakeholders & subject matter experts
- Concise – contains just the needed information, succinctly and easy to understand
- Consistent – does not conflict with any other requirement
- Unambiguous – must have sufficient detail to distinguish from undesired behaviour. includes diagrams, tables, and other elements to enhance understanding

- Verifiable (or testable) – when it can be proved that the requirement was correctly implemented
- Feasible – there is at least one design and implementation for it.
- Necessary – it is traced to a need expressed by customer, user, stakeholder;
- Traceable – can be traced to and from other designs, tests, usage models, etc. These improves impact assessment, schedule/effort estimation, coverage analysis scope management/prioritization

13. REQUIREMENTS REVIEW CHECKLIST

See also

- Appendix E for the *Code Complete* Requirements Review check lists

Names:

- ☐ Are the names clear and well chosen? Do the names convey their intent? Are they relevant to their functionality?
- ☐ Do they use a good group / naming convention (e.g. related items should be grouped by name)
- ☐ Is the name format consistent?
- ☐ Names only employ alphanumeric characteristics?
- ☐ Are there typos in the names?

13.1. ARE THE PROPERTIES, STATES AND ACTIONS WELL DEFINED?

- ☐ Is a definition duplicated?
- ☐ Is a property defined multiple different times.. but defined differently?
- ☐ Are the definitions complete?
  - Are all instances and kinds defined – or some missing?
  - Are there undefined (i.e., referred to, but not defined) nouns, properties, verbs?
  - Are events referred to but not defined?
- ☐ Are they consistent?
- ☐ Are the properties something that the system can measure or otherwise detect?
- ☐ Are the instances something that the system can identify or otherwise distinguish?
- ☐ Is a state not needed? Is it unused by any state classification, action, event, or requirement?
- ☐ Is a property not needed? Is it unused by any state classification, action, event, or requirement?
- ☐ Are the properties something that the system can detect?
- ☐ Are the events something that the system can detect?

13.2. REQUIREMENTS REVIEW

Reviewing requirements should look to identify:

- ☐ Are the requirements organized in a logical and accessible way?
- ☐ Is the requirement clearly demarcated?
- ☐ Does the requirement have a clear and fixed identifier? Is the identifier unique?
- ☐ Is the description supporting the requirement clear? Is it sufficient to support the requirement?
- ☐ Is the requirement too wordy? A requirement should be concise, containing just the needed information.
- ☐ Does the requirement use the proper modal auxiliaries?
- ☐ Does the requirement have the right conditions? The ubiquitous form of requirement is rare. Look for missing triggers and other conditions on the requirement.
- ☐ Are the time-critical features, functions and behaviours identified? Is the timing criteria specified?
- ☐ Is there requirement declarative? Or is the requirement an attempt to repackage an existing implementation with imperative statements? These are bad.
- ☐ Does the requirement conflict with any other requirement? Is its use of conditions (e.g. thresholds) consistent with the other requirements?
- ☐ Is the action to carry out clear? Is the action well defined within the rest of the specification?
- ☐ Are the actions something that can be accomplished?
- ☐ Duplicated requirements?
- ☐ Ambiguity. Can the requirement be interpreted different ways? Is there sufficient detail to distinguish from undesired behaviour?
- ☐ Is the requirement vague or ambiguous in any way? Pronouns, demonstratives, and indexicals often introduce ambiguity.
- ☐ Is the requirement specifying a single action.. or many? A requirement should specify only a single action.
- ☐ Complexity. Is the requirement over specified, too complex?
- ☐ Requirements that are too expensive, burdensome, impractical or impossible
- ☐ Are the requirements ones that fit the practical use with customer wants/needs/etc?
- ☐ Is the requirement unnecessary? Does it lack a trace to a need expressed by customer, user, or stakeholder? Is each requirement traceable to a customer that requires it?
- ☐ Check for consistency and sufficient definition
- ☐ Does the requirement have errors, such as misstating bounds, or conditions in the source material, or from other stakeholders or subject matter experts?
- ☐ Are there missing requirements? Is there a lack of sufficient detail to guide the work?

13.3. ARE THE REQUIREMENTS TESTABLE?

- ☐ Are the triggers something that the system can detect?
- ☐ Is the action or result of the requirement observable? Can it be measured?
- ☐ Are the quality requirements measurable?
- ☐ Is the requirement time bound? Is there a clear time bounds between the condition or trigger, and the action?

- ☐ Is the requirement untestable? Is there a direct means of stating how to test that the requirement was correctly implemented?
- ☐ Is the actor to carry out or meet the requirement clear? Is the actor well-defined within the rest of the specification?
- ☐ Are the actions testable? Is their outcome testable?
- ☐ Is the requirement bounded? Or is the actor allowed to do the requirement at the end of the universe?

13.4. THE LEADS REVIEW REQUIREMENTS

- ☐ Are they complete? Are requirements or definitions missing? Are there undefined nouns, properties, verbs?
- ☐ Are they consistent?
- ☐ Are they doable?

CHAPTER 5

Software Risk Analysis

This chapter provides an initial template for software risk analysis.

14. SOFTWARE REQUIREMENTS RISK ANALYSIS

The outputs of a software requirements risk analysis include:

- A table mapping risks to the requirements that address it. *This table may have been produced by another activity and is only referenced in the output*
- A list of software risks, acceptability level, and their disposition
- A criticality level for each hazard that can be affected by software
- Recommended changes to the software requirements specification, programmable system architecture, etc. For example, actions required of the software to prevent or mitigate the identified risks.
- Recommended Verification & Validation activities, especially tests

The steps of a software requirements risk analysis include:

1. Identify the requirements that address each product hazards
2. Examine the risks of errors with values
3. Examine the risks of message capacity
4. Examine the risks of timing issues
5. Examine the risks of software functionality
6. Examine the risks of software robustness
7. Examine the risks of software critical sections
8. Examine the risks of unauthorized use
9. Recommendations for rework

14.1. STEP 1: IDENTIFY THE REQUIREMENTS THAT ADDRESS PRODUCT HAZARDS

Go thru each of the identified product hazards, and list the software requirements that address it.

14.2. STEP 2: EXAMINE ALL VALUES FOR ACCURACY

Identify all the elements of the system – sensor 1, sensor 2, actuator, motor, calculations, operator inputs, operator outputs, each parameter received, etc. For each of these elements, create a copy of **Table 4** (below) and populate it with an analysis with respect to the requirements. Strike inapplicable conditions, and add other identified conditions.

***Table 4:** Value accuracy risks*

Condition	Hazard, likelihood & severity
the value is off by 5% of the actual value	
the value is stuck at all zeroes	
the value is stuck at all ones	
the value is stuck at some other value	
the value is too low; the value/ result is below minimum range	
the value is within range, but wrong; with calculation, e.g. the formula or equation is wrong	
the physical units are incorrect	
the value is incorrect *(for non-numerical values)*	
the value type, or format size is wrong	

In reviewing each condition, identify the least acceptable risk for each applicable condition. Other documents (i.e. the product safety risk analysis) are responsible for the identifying the set of possible hazards and their severity. **Table 5** provides an example likelihood levels; **Table 6** provides an example mapping of severity & likelihood pair to risk acceptability.

***Table 5:** Hazard probability levels based on Mil-Std 882*

Likelihood	Estimate of Probability
Frequent	Likely to occur on in the life of an item, with a probability of occurrence greater than 10^{-1} in that life.
Probable	Will occur several times in the life of an item, with a probability of occurrence less than 10^{-1} by greater than 10^{-2} in that life
Occasional	Likely to occur sometime in the life of an item, with a probability of occurrence less than 10^{-2} but greater than 10^{-3} in that life.
Remote	Unlikely, but possible to occur in the life of an item, with a probability of occurrence less than 10^{-3} but greater than 10^{-4} in that life.
Improbable	So unlikely, it can be assumed occurrence may not be experienced, with a probability of occurrence of less than 10^{-4} in that life.

***Table 6:** An example risk acceptability matrix determining risk acceptability*

	Catastrophic	Critical	Marginal	Negligible
Frequent	high	high	high	medium
Probable	high	high	medium	low
Occasional	high	high	medium	low
Remote	high	medium	low	low
Improbable	medium	low	low	low

14.3. STEP 3: EXAMINE THE MESSAGES CAPACITY

This step examines the ability of the software to achieve its objectives within the hardware constraints.

Identify all the messaging elements of the system – I^2C sensor, task 1, user input, etc. For each of these elements, create a copy of **Table 7** (below) and populate it with respect to the requirements. Strike inapplicable conditions, and add other identified conditions.

Table 7: *Message capacity risks*

Condition	Hazard, likelihood & severity
message is smaller than state minimum	
message is larger than stated maximum	
message size is erratic	
messages arrive faster than stated maximum (e.g. response time)	
messages arrive slower than stated minimum (e.g. response time)	
message contents are incorrect, but plausible	
message contents are obviously scrambled	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

14.4. STEP 4: EXAMINE THE CONSEQUENCES OF TIMING ISSUES

This step examines the ability of the software to achieve its objectives within the hardware constraints.

Identify all the input elements of the system – button #1, frequency input, I^2C sensor, task 1, user input, etc. This list should include elements those receive messages, and send messages. For each of these elements, create a copy of **Table 8** (below) and populate it with respect to the requirements. Strike inapplicable conditions, and add other identified conditions.

Table 8: *Timing capacity risks*

Condition	Hazard, likelihood & severity
input signal fails to arrive	
input signal occurs too soon	
input signal occurs too late	
input signal occurs unexpectedly	
input signal occurs at a higher rate than stated maximum	
input signal occurs at a slower rate than stated minimum	
system behavior is not deterministic	
output signal fails to arrive at actuator	
output signal arrives too soon	
output signal arrives too late	
output signal arrives unexpectedly	
insufficient time allowed for operator action	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

14.5. STEP 5: EXAMINE SOFTWARE FUNCTION

This step examines the ability of the software to carry out its functions.

Identify all the functions of the system; that is, the operations which must be carried out by the software. For each of these elements, create a copy of **Table 9** (below) and populate it with respect to the requirements. Strike inapplicable conditions, and add other identified conditions.

Table 9: *Software function risks*

Condition	Hazard, likelihood & severity
Function is not carried out as specified (for each mode of operation)	
Function preconditions or initialization are not performed properly before being performed	
Function executes when trigger conditions are not satisfied	
Trigger conditions are satisfied but function fails to execute	
Function continues to execute after termination conditions are satisfied	
Termination conditions are not satisfied but function terminates	
Function terminates before necessary actions, calculations, events, etc. are completed	
Function is executed in incorrect operating mode	
Function uses incorrect inputs	
Function produces incorrect outputs	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

14.6. STEP 6: EXAMINE SOFTWARE ROBUSTNESS RISKS

This step examines the ability of the software to function correctly in the presence of invalid inputs, stress conditions, or some violations of assumptions in its specification.

Create a copy of **Table 10** (below) and populate it with respect to the requirements. Strike inapplicable conditions, and add other identified conditions.

Table 10: *Software robustness risks*

Condition	Hazard, likelihood & severity
Software fails in the presence of unexpected input signal/data	
Software fails in the presence of incorrect input signal/data	
Software fails when anomalous conditions occur	
Software fails to recover itself when required	
Software fails during message, timing or event overload	
Software fails when messages are missed	
Software does not degrade gracefully when required (e.g. crashes instead)	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

14.7. STEP 7: EXAMINE SOFTWARE CRITICAL SECTIONS RISKS

This step examines the ability of the system to perform the functions that address or control risks.

Create a copy of **Table 11** (below) and populate it with respect to the requirements. Strike inapplicable conditions, and add other identified conditions.

Condition	Hazard, likelihood & severity
Software causes system to move to a hazardous state	
Software fails to move system from hazardous to risk-addressed state	
Software fails to initiate moving to a risk-addressed when required to do so	
Software fails to recognize hazardous state	

Table 11: *Software critical sections risks*

In reviewing each condition, identify the least acceptable risk for each applicable condition.

14.8. STEP 8: UNAUTHORIZED USE RISKS

Create a copy of **Table 12** (below) and populate it with respect to the requirements:

Condition	Hazard, likelihood & severity
Unauthorized person has access to software system	
Unauthorized changes have been made to software	
Unauthorized changes have been made to system data	

Table 12: *Unauthorized use risks*

In reviewing each condition, identify the least acceptable risk for each applicable condition.

14.9. STEP 9: RECOMMENDATIONS FOR REWORK

Summarize each of the identified conditions with an unacceptable risk level. These items items mandate rework, further analysis, and/or Verification & Validation activities.

15. REFERENCE DOCUMENTS

MIL-STD-882E *"Standard Practice System Safety"* 2012 May 11

NASA-GB-8719.13, *NASA Software Safety Guidebook*, NASA 2004-3-31

NASA-STD-8719.12, *NASA Software Safety Standard,* Rev C 2013-5-7

UCRL-ID-122514, J Dennis Lawrence, *Software Safety Hazard Analysis* Rev 2, U.S. Nuclear Regulatory Commission, 1995-October

---

"A good engineer tries to get something not to work – that is, after getting it working, the good engineer tries to find its limits and make sure they are well-understood and acceptable."
– Michael Covington

[This page is intentionally left blank for purposes of double-sided printing]

PART II

Software Design & Documentation

This part provides guides for software design and its documentation

- OVERVIEW & WRITING TIPS.
- OVERVIEW OF SOFTWARE DESIGN.
- GUIDELINES FOR HIGH-LEVEL DESIGNS. Provides guidelines for high-level designs (e.g. architectures).
- SOFTWARE ARCHITECTURE RISK ANALYSIS.
- GUIDELINES FOR DETAILED DESIGNS. Provides guidelines for detailed designs (e.g. major subsystems or "stacks").
- PROTOCOL DOCUMENTATION TEMPLATE. Provides a guide for protocol documentation.
- SOFTWARE MODULE DOCUMENTATION TEMPLATE. Provides a guide for detailed design documentation of a module.
- GUIDELINES FOR MODULE DESIGNS. Provides guidelines for low-level module design.
- DESIGN REVIEWS CHECKLISTS. Provides checklists for reviewing a design.
- SOFTWARE DETAILED DESIGN RISK ANALYSIS REVIEWS. Describes reviewing software for risk analysis.

[This page is intentionally left blank for purposes of double-sided printing]

CHAPTER 6

Design Overview & Writing Tips

This chapter describes the recommended approach for design documentation

- The role and characteristic of design documentation
- Organization of the documentation

16. THE ROLE AND CHARACTERISTICS OF DESIGN DOCUMENTATION

This chapter describes my recommendations for writing design documentation. The role of documentation is to

- Provide assurance to an outside reviewer – who is without the tacit knowledge that the developing team and organization will share – that the product is well-craft and suitable for its intended purpose, and will achieve the safety & quality requirements;
- Communicate with future software development, and test teammates; and to reduce the puzzles and mysteries when handed a completed software implementation with the expectation to make it work/modify it/test it;
- Drive clarity of thought on the part of the designers; experience has repeatedly shown that if it can't be explained clearly, it isn't understood. A lack of understanding impairs product quality, and creates project risk (thrashing).

The design document…

- Establishes the shape of the software modules
- Shows how the design addresses the software requirements and other specifications
- Provides a mental map of the design, making the design understandable.

Characteristics of a good design description include:

- A straightforward mapping to the implementation
- Mixing visuals and text to explain the concepts in alternate ways
- Scoping diagrams so that the amount to hold in the readers head is small

The requirements at the design stage should provide a clear enough view to allow the high-level design to be crafted.

16.1. TIPS ON THE WRITING PROCESS

This section focuses on presenting the design as much as it does on tips for crafting a design. I've found that most engineers dislike documentation and defending in detail their designs.[8] That's a pity, since the documentation is a necessary skill in quality software domains (such as safety critical products), and an important one to project success. Some tips:

1. Read & study good examples of design, and design documentation. These can be found in books like McKusick (2004), and Kehan (1987). Other examples might be found in application notes, and past projects
2. Use templates and writing guides. They provide the scope and main outline, reducing the burden of how to organize the documentation.
3. Plan on the design in stages of completeness: a preliminary version of the design before the development begins in earnest, revisions during design discussions, and a finished design at the end of the project.
4. Take the writing in small, doable pieces. Write the document in a series of drafts, targeting only a few pages a day. Revise the draft, and repeat.
5. Start with the areas that you know what to write; don't necessarily worry about starting with lower-level design documents if that is what you know. They won't be committed to yet, but the information and experience will help write the upper layers.
6. Then work down from the top – or up from the bottom – in a vertical slice relevant to what you do know. Add in organizational material (such as outlines for the section) or an expository explanation, and keep moving. Use stubs – such as "TBD" – for specific values, names or other references that you do not know yet.

McKusick, Marshall Kirk and George V. Neville-Neil. The Design and Implementation of the FreeBSD Operating System, *1st Edition Addison-Wesley Professional; 1st edition, 2004*

Kenah, Lawrence; Ruth Goldenberg. VAX/VMS Internals and Data Structures: Version 5.2, *Digital Press, 1987*

16.2. AUDIENCE

The audience for the design documentation includes:

- The certifying body
- The development team
- Management, such as project manager
- The regulatory affairs department (i.e. the design history file)
- Release engineering
- The software and system test group
- Technical publications

17. DOCUMENTATION ORGANIZATION

One of the first steps in the developing the documentation is to pick a style or organization for the documentation. This will help layout (block) the overall documentation, and provide not just the structure, but the start of an outline and size of the work to accomplish the scope.

[8] I've also found that the best designs, the highest quality ones, were produced quickly by designers who will *love* talking with others about their designs, and crave to find others who will appreciate it.

An expansive, large system might be best served with documentation that subdivides the design into several large portions with a mid-level design, and then breaks into detailed design of the individual components.

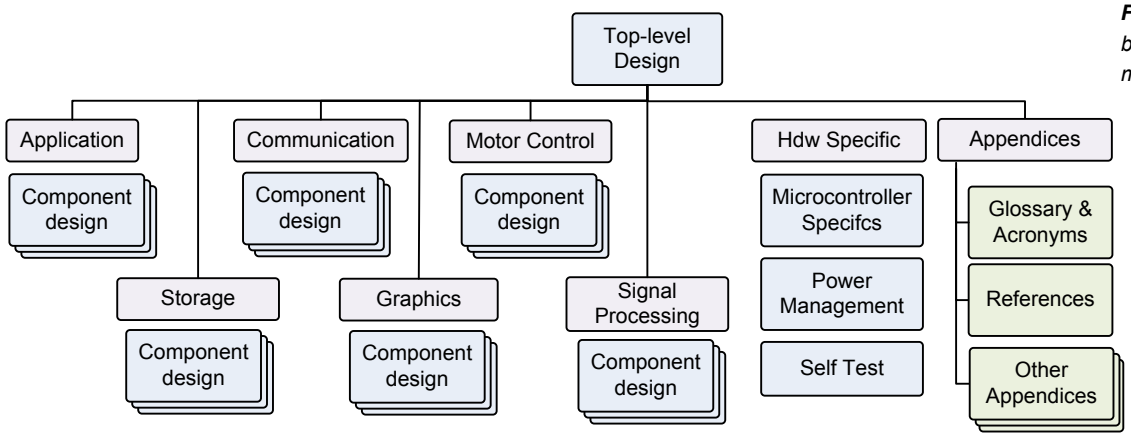

Figure 5: *Structure of a broad design with moderate-fan out*

A moderate, small system might be best served with documentation that introduces the high-level design and then breaks into detailed design of the (many) individual components.

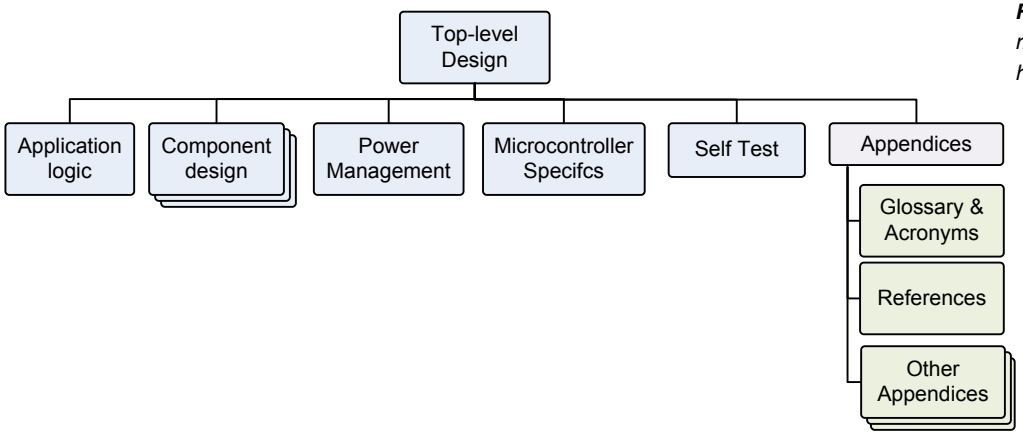

Figure 6: *Structure of a mid-size design, with high-fan out*

Note: Major portions of the structure may be mandated by the standards the product is being certified against; and/or by the developing institute's processes. Many, for example, mandate the presence of the references and glossary, but that they be in the front of the document.

17.1. TERMS AND PHRASES TO EMPLOY

Once the broad structure is selected, begin thinking about the terms and phrases that are and will be used in the project, and your approach in the documentation:

- What terms and phrases will be used?
- Which will not be used in the documentation?
- What additional terms and phrases should be given a translation to the project – a mapping to the terms and phrases used in the design documentation?

The standards (to which the product may be certified against), and the specifications will already be employing a stock of terms and phrases. The design doesn't necessarily need to

use them (and it isn't always warranted). In that case, the design document should provide a definition of what those terms and phrases are in within the project.

The terms and phrases used should make sense for the project and design. They could come from

1. The requirements and other specifications for the product
2. The jargon used within the rest of the organization, or team
3. Other conventions, such as the industry jargon.

The design should address the terms and phrases of the standards and specifications that are not otherwise used in the design documentation. This can be achieved by providing a mapping of these terms and phrases used in the standards and specifications to those in the design.

17.1.1 Tips for getting the definitions for standards terms

Most standards provide a glossary of the terms and phrases that they use. However, the definitions within a standard can sometimes be unclear, confusing or otherwise not helpful. Fortunately there are resources that can be used to gather variations of the definition to help clarify the term or phrase.

The IEEE provides a glossary of terms in IEEE Std 610.12-1990.

There are two search-tool resources than can be used to look up the definitions across IEC standards:

- http://www.electropedia.org/
- http://std.iec.ch/glossary

And for ISO standards:

- https://www.iso.org/obp/ui

17.2. APPENDICES

The design documentation often includes several appendices. The ones described here

- Acronyms and Glossary
- References, Resources and Suggested Reading

The ones described in the software development plan:

- Configuration of the compiler, linker and similar tools
- Configuration or settings of the analysis tools and similar
- The files used in the project
- The configuration of the software. This is often divided into application and board-specific configurations.

17.2.1 The Acronyms and Glossary

I recommend that the list of acronyms and glossary be in the rear of the documentation. In some development protocols it is preferred that they be in the front of the documentation.

In this appendix, define all acronyms, terms and phrases. We have all seen documents that include definitions for simple, common items (such as a LED), while not defining specialized

items (such as “adaptive linear filter,” or “hybrid turboencabulator”) referred to heavily in a document. Don’t do this.

I recommend that the expansion of acronyms to its words be presented in a separate list from the definition of terms and phrases.

This appendix “living,” which is to say, they acronyms, terms and phrases will continue to expand and be added to throughout the development. In good documentation the glossary can be extensive.

Tip: The acronyms and glossary are well suited for reuse in many projects. Make a stock document with the most common terms and potted definitions to be included in each project.

17.2.2 The References, Resources and Suggested Reading

I recommend that the references be in the rear of the documentation, excluding (perhaps) the list of standards that are inputs to the design. In some development protocols it is preferred that they be in the front of the documentation.

In this references appendix the list should include data sheets, industry and legal standards, communication protocols, etc. Include a designator for each document. Use this through the remainder of this specification to refer to the document.

17.2.3 Files

The detailed design often includes an appendix listing the files. How much to list really depends on the stringency of development. Exhaustively listing the files is simply no joy. I’m still a fan of listing and getting on the names, or at least most. In every case, the list should include

- Files with strange names
- Files with particular importance

The list should *not* include temporary or generated files. Separate out the object files, assembly listings, temporary files, etc. unless there is good reason to keep them.

Should describe what each file is and its role.

Can use groupings, folders and names to help organize the names.

17.3. REUSING DESIGNS AND DOCUMENTATION

Some observations on design and documentation reuse:

- Software libraries are one way to reuse designs. However, the design, creation, and support of a library is a development effort in and of itself, with many factors that impair success;
- The high-level design ideas are readily reused; so are specific low-level modules (e.g. digital input, output, analog conversion, etc.).
- Good design practices facilitate easier reuse.

Some approaches and techniques to help promote reuse of designs:

- Divide the documentation in pieces that can be reused
- Provide a segment of time (e.g. at the end of the project) for reviewing and identifying reusable sections.

- Identify, during design reviews, areas where prior design *should*[9] have been reused (but was not).

Existing designs are (mostly) worked out; their reuse can accelerate a project schedule – *if* the design is appropriate to the project. Such design must be stored in manner so that it is accessible, easily found, and readily reusable. Each successive project may contribute to the collection of reusable design pieces. I recommend:

1. Break out each chapter into its own file.
2. Create an overall structuring document for the design documentation.

To merge these into a single design document for release with a project:

1. Make a copy of that overall structuring document
2. Insert each of the files for the chapter

When it comes time to do another project, the chapters of the past projects serve as a starting point for documentation reuse. The levels of documentation & design reuse:

1. Verbatim: The chapter is picked up and used without changes.
2. The chapter is copied and modified it to adapt it to the project.
3. A template document is reused where the structure is used largely unchanged, but the contents are customized for the new project. This avoids re-inventing the structure, and while using a fill-in-the-blank approach.

18. REFERENCES AND RESOURCES

Alred, Gerald, Charles Brusaw, Walter Oliu; *Handbook of Technical Writing*, 10nth Ed, St Martin's Press, 2011

IEEE Std 610.12-1990, *IEEE Standard Glossary of Software engineering Terminology,* 1990.

[9] Note the emphasis is on *should*, not *could.* This step is fraught with politics in some offices, which will undermine quality

CHAPTER 7

High-Level Design Template

This chapter is my template for a high-level design description

19. BASIC OUTLINE

The structure of top level design changes the most between projects. Initially the design is skeletal, and fleshed out over time. The following is an outline for a design description:

1. Synopsis
2. Other front matter
    a. Related documents and specifications (documents that are part of the product)
3. Design overview
4. Appendices:
    a. Glossary, acronyms
    b. References, resources, suggested reading

The appendices can be expanded upon as the design is developed:

- Compiler Configuration, flags, etc.
- Analysis tool (e.g. LINT, MISRA C checks, etc.) configuration including which checks are enabled and disabled.
- Linker configuration & Linker scripts
- The software configuration settings.
- Files employed in the software.

19.1. SYNOPSIS AND FRONT MATTER

THE SYNOPSIS. A one or two paragraph synopsis of what the software's role in the product is.

THE RELATED DOCUMENTS & SPECIFICATIONS. This is a list of internal organization & project standards, and design specifications, with a designator for each document. The designator is to be used through the remainder of this specification to refer to the document.

19.2. THE GLOSSARY, REFERENCES

Next, prepare place holders for the common elements, such as the acronyms, glossary of terms, and references. These are "living," which is to say, they will continue to expand and be added to throughout the development. In good documentation the glossary can be extensive.

Note: I recommend the following be in the rear of the documentation, along with all of the other supplemental information.

THE ACRONYM AND GLOSSARY TABLES. Define all acronyms, terms and phrases. We have all seen documents that include definitions for simple, common items (such as a LED), while not defining specialized items (such as "adaptive linear filter," or "hybrid turboencabulator") referred to heavily in a document. Don't do this.

I recommend that the expansion of acronyms to its words be presented in a separate list from the glossary definition of terms and phrases.

Tip: The acronyms and glossary are well suited for reuse in many projects. Make a stock document with the most common terms and potted definitions to be included in each project.

THE REFERENCES, RESOURCES, SUGGESTED READING. The documents to list include data sheets, industry and legal standards, communication protocols, etc. Include a designator for each document. Use this through the remainder of this specification to refer to the document.

19.3. DESIGN OVERVIEW

Describe the role and responsibility of the software. Include the functions and features that it is responsible for (that is, the functions and features that the software is responsible for providing).

Include a diagram summarizing the software design, with the major sections and their interconnections. This may include a reference to external elements that it controls or depends on. The diagram should include a description introducing to how inputs are turned into outputs. It should show the basic structure of the signal flow (both control signals and TBD signals). Ideally it would also provide context, and show key external elements (where relevant).

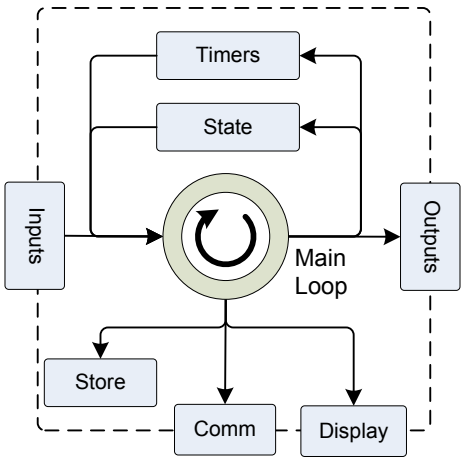

Figure 7: *Basic flow structure of the software*

Provide a description of the main elements of the software design:

Element	Description
element 1	Description of the element
...	
element n	Description of the element

***Table 13:** The software design elements*

The external elements are:

External Element	Description
element 1	Description of the element
...	
element n	Description of the element

***Table 14:** The external elements*

Note: this diagram (architecture) is often stylized and reused across products as a platform or design style. Requirements may be written against an abstract model based on it.

Other sections to include

1. Detailed block diagram of the software organization. This should include the IO, communication, power management, sensors, drivers, control loops and other subsections that will be described in detail in the rest of the document.
2. Major modules and module prefixes
3. Storage and data integrity
4. Communication and data integrity
5. Time keeping
6. Sensors, the signal chain and other inputs to the microcontrollers
7. Safety model, Self-check / self-protect functions, watchdog, prioritization
8. Power management
9. Configuration

Is an RTOS used? That provides a specific kind of structure and breakdown.

19.4. PARTITIONING INTO A TWO PROCESSOR MODEL

Consider separating the more stringent functions (such as safety critical functions) from the main – but less stringent functions – by placing them into a separate processor.

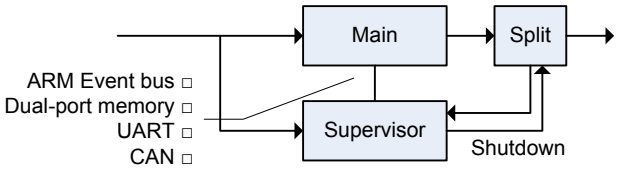

***Figure 8:** Processor with a supervisor processor*

The second processor monitors condition and places the system into a safe state in the event that the main processor or system conditions leave a well-defined safe operating region (within some abstract space).

20. DIVISION INTO MODULES

There should be a solid discussion of how the software is structured and implemented in a modular manner. This design approach breaks the development down into manageable chunks. It also supports *unit testing* of the software. Some example text (plus a supplemental diagram)

The software system has 5 major module groupings, based on the kind of work they do, or information they organize:

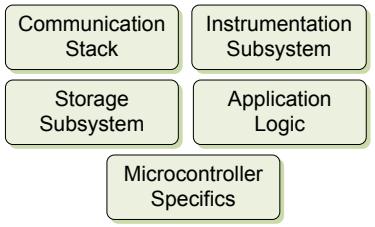

Figure 9: *Major functionality groups*

This is only if the diagram did not include them. It usually does not. I don't get much out of this but it is better to include a diagram than to omit one.

The modules

Group	Description
application logic	The logic specific to the application and its requirements.
communication	This group provides the communication stack to send and receive information remotely.
instrumentation	This includes functions for gathering signals and applying the control logic
microcontroller specifics	This includes the drivers and chip-specific software, helping improve portability by supporting the designs uses of alternative microcontroller.
storage subsystem	This logs relevant information, and configuration information. Critical data is store in a manner to prevent data loss if there is a loss of power.

Table 15: *The functionality groups*

20.1. CRITICAL ELEMENTS

Highlight or emphasize elements that are "critical" and need special precaution. Some elements that may be critical include:

- Elements that are necessary to achieve the safety functions of the product
- Elements mandated by the standards (being certified against) as critical
- Elements that are depended up by those elements, or are necessary to prevent systematic faults of any critical element.

The critical elements should get extra review steps and have more documentation. While not all critical elements may be known in the first pass of the design, experienced designers will be able to anticipate many that will be.

Separate the software into different categories:

- The stringently defined area that is focused on addressing the functions critical to the safety requirements in those standards. The risk management, process, techniques and testing are most focused on this category of software
- The other elements of software that whose functionality does not present a safety risk (since the above category is responsible for that function).

This separation allows the other elements to be construction in a less stringent manner. For instance, a high-degree of assurance may not be tractable or even meaningfully definable (in the present state of the art).

20.2. MODULE PREFIXES

{It is arguable whether this information should in the high-level design, or in the appendices. I find it helpful as guidance to the development}

Each module has a separate prefix. The table below describes the prefixes employed for the modules

Table 16: *Summary of module prefixes*

Prefix	Module
AIn	The analog input module, including ADC sampled values, etc.
AOut	The analog output procedures
App	Application procedures and application specific logic
BSP	Board specific package related procedures
DIn	The digital inputs are GPIO logic signals.
IIR	Infinite impulse response filters
Poly	Polynomial correction of signals
Time	Time-keeping related
Tmr	Timer related
UART	UART, a hardware serial interface

20.3. SOURCE CODE CONFIGURATION FILE(S)

The firmware is configurable, allowing changes in the electronics design and specific features of the application. The settings for the other three configuration files are described in appendix TBD.

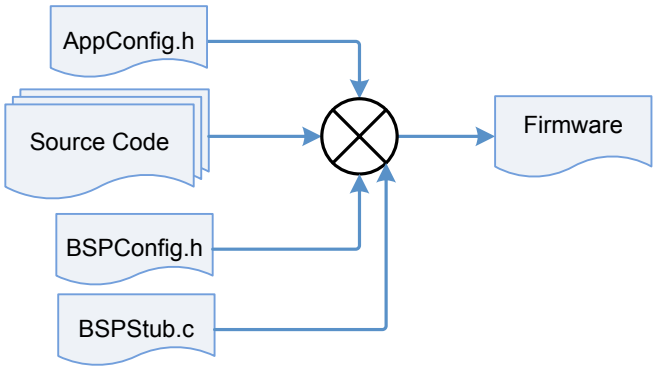

Figure 10: *The configuration of the production firmware*

The BSPStub.c provides the linkages the microcontroller register, such as the digital input and output data registers. (These differ between microcontroller families, and sometimes within them; but every microcontroller has some form of these registers). This also provides resource sizing and buffers related to these inputs.

THE CONFIGURATION HEADER FILES are used to enable (or disable) features, and size the remaining resources. The features that can be enabled have a control macro suffixed with _EN. For example to enable, feature XYZ:

```
#define XYZ_EN (1)
```

Alternatively, to disable it:

```
#define XYZ_EN (0)
```

The board specific defines are located in a file called BSPConfig.h. The application or framework features are located in a file called AppConfig.h.

21. REFERENCES AND RESOURCES

DI-IPSC- 81432A, *Data Item Description: System/Subsystem Design Description* (SSDD), 1999 Aug 10
http://everyspec.com/DATA-ITEM-DESC-DIDs/DI-IPSC/DI-IPSC-81432A_3766/

DI-IPSC-81435A, *Data Item Description: Software Design Description* (SDD), 1999 Dec 15
http://everyspec.com/DATA-ITEM-DESC-DIDs/DI-IPSC/DI-IPSC-81435A_3747/

ISO/IEC/IEEE 42010:2011, *Systems and software engineering — Architecture description*. Note: this supersedes IEEE Std 1471-2000

CHAPTER 8

Software Architecture Risk Analysis

This chapter provides an initial template for software architecture risk analysis.

22. SOFTWARE ARCHITECTURE RISK ANALYSIS

The outputs of a software architecture risk analysis include:

- A table mapping the software requirements to the architecture element that addresses it. *This table may have been produced by another activity and is only referenced in the output.*
- A list of software risks, acceptability level, and their disposition
- A criticality level for each hazard that can be affected by software
- Recommended changes to the software architecture, software requirements specification, programmable system architecture, etc. For example, actions required of the software to prevent or mitigate the identified risks.
- Recommended Verification & Validation activities, especially tests

The steps of a software architecture risk analysis include:

1. Identify the architecture element that addresses each requirement. *This may have been produced by another activity and is only referenced in the output.*
2. Examine the risks of errors with values
3. Examine the risks of message capacity
4. Examine the risks of timing issues
5. Examine the risks of software functionality
6. Examine the risks of software robustness
7. Examine the risks of software critical sections
8. Examine the risks of unauthorized use
9. Recommendations for rework

22.1. STEP 1: IDENTIFY THE ARCHITECTURE ELEMENTS THAT ADDRESS EACH REQUIREMENT

Go thru each of the software requirements and list the architecture elements that address it.

22.2. STEP 2: EXAMINE ALL VALUES FOR ACCURACY

Identify all the elements of the architecture – sensor 1, sensor 2, actuator, motor, calculations, operator inputs, operator outputs, each parameter received, etc. For each of these elements, create a copy of **Table 4** ("*Value accuracy risks*") and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition.

22.3. STEP 3: EXAMINE THE MESSAGES CAPACITY

This step examines the ability of the software to achieve its objectives within the hardware constraints.

Identify all the messaging elements of the system – I^2C sensor, task 1, user input, etc. For each of these elements, create a copy of **Table 7** ("*Message capacity risks"*) and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition. Strike inapplicable conditions, and add other identified conditions.

22.4. STEP 4: EXAMINE THE CONSEQUENCES OF TIMING ISSUES

This step examines the ability of the software to achieve its objectives within the hardware constraints.

Identify all the input elements of the system – button #1, frequency input, I^2C sensor, task 1, user input, etc. This list should include elements those receive messages, and send messages. For each of these elements, create a copy of **Table 17** (below) and populate it with respect to the architecture. Strike inapplicable conditions, and add other identified conditions.

Table 17: *Timing capacity risks*

Condition	Hazard, likelihood & severity
input signal fails to arrive	
input signal occurs too soon	
input signal occurs too late	
input signal occurs unexpectedly	
input signal occurs at a higher rate than stated maximum	
input signal occurs at a slower rate than stated minimum	
system behavior is not deterministic	
output signal fails to arrive at actuator	
output signal arrives too soon	
output signal arrives too late	
output signal arrives unexpectedly	
processing occurs in an incorrect sequence	
code enters non-terminating loop	
deadlock occurs	
interrupt loses data	
interrupt loses control information	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

22.5. STEP 5: EXAMINE SOFTWARE FUNCTION

This step examines the ability of the software to carry out its functions.

Identify all the functions of the system; that is, the operations which must be carried out by the software. For each of these elements, create a copy of **Table 18** (below) and populate it with respect to the architecture. Strike inapplicable conditions, and add other identified conditions.

Table 18: *Software function risks*

Condition	Hazard, likelihood & severity
Function is not carried out as specified (for each mode of operation)	
Function preconditions or initialization are not performed properly before being performed	
Function executes when trigger conditions are not satisfied	
Trigger conditions are satisfied but function fails to execute	
Function continues to execute after termination conditions are satisfied	
Termination conditions are not satisfied but function terminates	
Function terminates before necessary actions, calculations, events, etc. are completed	
Hardware or software failure is not reported to operator	
Software fails to detect inappropriate operation action	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

22.6. STEP 6: EXAMINE SOFTWARE ROBUSTNESS RISKS

This step examines the ability of the software to function correctly in the presence of invalid inputs, stress conditions, or some violations of assumptions in its specification.

Create a copy of **Table 10** ("*Software robustness risks"*) and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition. Strike inapplicable conditions, and add other identified conditions.

22.7. STEP 7: EXAMINE SOFTWARE CRITICAL SECTIONS RISKS

This step examines the ability of the system to perform the functions that address or control risks.

Create a copy of **Table 11** ("*Software critical section risks"*) and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition. Strike inapplicable conditions, and add other identified conditions.

22.8. STEP 8: UNAUTHORIZED USE RISKS

Create a copy of **Table 12** ("*Unauthorized use risks"*) and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition. Strike inapplicable conditions, and add other identified conditions.

22.9. STEP 10: RECOMMENDATIONS FOR REWORK

Summarize each of the identified conditions with unacceptable risk levels (e.g. of "medium" or "high"). These items mandate rework, further analysis, and/or Verification & Validation activities.

CHAPTER 9

Detailed Design

This chapter is my detailed design tips.

- Diagrams and design decomposition into modules
- Organization of the modules
- Examples of common subsystem designs
- Firmware and subsystem test support

23. DIAGRAMS AND DESIGN DECOMPOSITION INTO MODULES

Detailed design begins with several elements of the architecture and expands upon them. The architecture introduced, perhaps with diagrams, structural and connective elements. The detailed design breaks out the design into major areas and then into modules (with specific function) for that area. This is classic structured decomposition.

The detailed design description should include several diagrams to explain design:

- Structural network diagram to show flow thru the functional units
- Stratified diagram of modules, to show the level of abstraction and logical dependency

23.1. STRUCTURAL NETWORK DIAGRAM

Include a structural diagram summarizing the software design, with the major structural elements and their interconnections. This may include a reference to external elements that it controls or depends on. The diagram should include a description introducing to how inputs are turned into outputs. It should show the basic structure of the signal flow (both control signals and primary signals). Ideally the diagram would also provide context, and show key external elements (where relevant).

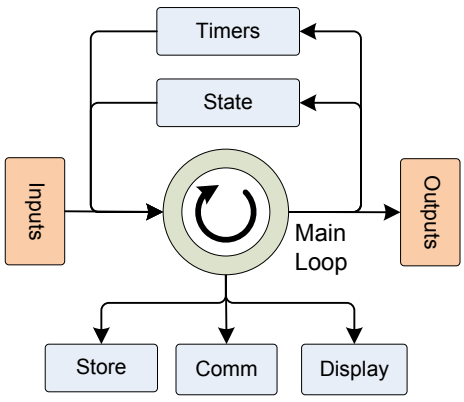

Figure 11: *Basic structure diagram of the software*

This type of diagram should provide a mental map of the design

- Make the design understandable
- The modules are functionality (or actors) and are represented as boxes
- The links show the connectivity and flow of signal/info/data

Provide a description of the main elements of the structured network diagram:

Element	Description
element 1	Description of the element
...	
element n	Description of the element

***Table 19:** The structural diagram elements*

Example of the structural elements may include:

- Libraries
- Functionality built into libraries
- Code layers
- Threads / processes / tasks

The external elements are:

External Element	Description
element 1	Description of the element
...	
element n	Description of the element

***Table 20:** The external elements*

23.2. DESIGN CRITERIA FOR MODULES

Each module should perform a distinct task or provide a distinct function. A module should be "thin":

- It should be easy to define its input-output behaviour; it should perform specific, limited functions
- Most modules should have no internal state or 'memory' – memory not in the sense of RAM usage, but in the sense that it's functioning or output is dependent on past inputs and/or outputs)
- The module should use few (if any) timers
- The modules should be isolated, manage their state/memory and have low complexity

23.3. THE SIGNAL/DATA

The connective links should be described (and annotated) to provide information about:

- The signal/info/data and the mechanism of representing the signal/info/data (format/encoding/structures/other object)

- The mechanism of the link: is the link shared variables? a message queue? semaphore(s)?
- The mechanism of transporting the represented data over the link

Example connective elements include:

- Variables
- Buffers
- Queues
- Mailboxes
- Semaphores

The later when used in a multitasking environment

Family of related modules, usually connected by signal flow

Divide up into sufficient detail. Modules, names of each element, names of actors, other

23.4. STRATIFIED DIAGRAM OF MODULES

The stratified diagram is organized into layers with the lowest closest to mechanics; successively higher layers present more abstraction. This form of diagram shows much less of the structure, and little of the connectivity. The signal & control flows are up and down; both the inputs and outputs are at the bottom. The flow is not as clear as with a structural diagram in this regard. This form of diagram usually shows the dependency – what module depends on another's function to deliver its own.

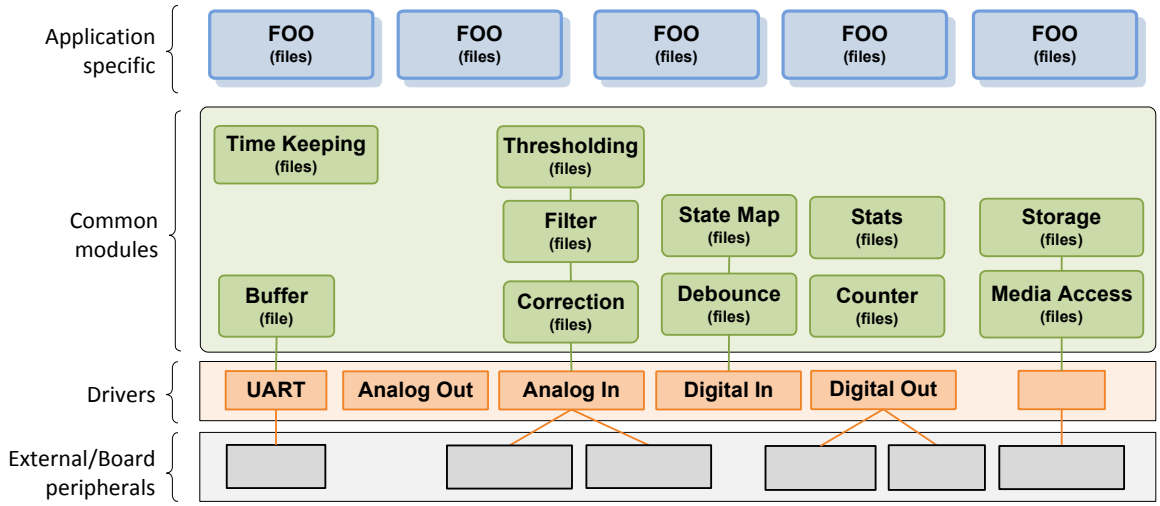

Figure 12: *Basic stratified diagram of the software modules*

Most of the lower layers are specific to the hardware… and thus limit portability of the application to other hardware. However, they may be used in other projects. The upper layers are often specific to the application and/or product… and so they may are less likely to be reused as well.

24. ORGANIZATION

This section outlines the organization of the modules and implementation

- File system layout
- File grouping for a module's implementation
- Configuration points

24.1. FILE SYSTEM LAYOUT

Note: The software development plan typically provides the information in this section.

The directory structure is a set of nested folders. Some folders are shared between projects; some are unique to this project. The top-level folders are:

Folder	Description
doc	The documentation for this project
Components	The vendor documentation for the components used in this project or evaluated for it
src	The project source code
Release	The released application image, suitable for download to the unit

Table 21: *Top-level Folders in the project file directory*

Within the project source code, the folders might be:

Folder	Description
CSP	Holds core specific modules
IO	Holds support for non-communication input put, such as GPIO, PWM, etc.
Power	Holds support for measuring the battery level
Sensors	Support for sensors such as accelerometers, and temperature sensors
STM32	Holds STM32 microcontroller specifics
win	Windows specific files

Table 22: *Source code folders in the project file directory*

24.2. FILE GROUPING FOR A MODULE'S IMPLEMENTATION

A module may implemented by one or more source files.

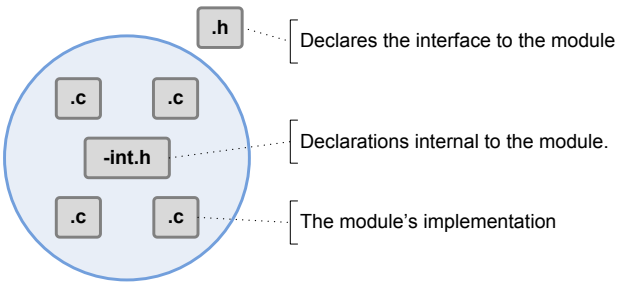

Figure 13: *How .h and .c files related to a module*

- One or more `.c` files that implement the module – it is better to break down a module into groups of relatively short files rather than one large file a thousand lines or longer.
- The module may have other `.h` files (suffixed as `-int.h`) that are for use only within the module. These should not contain information intended to be used for the whole system
- A module has one (or more) `.h` files that declares the procedures, variables, and macros that other modules may use. This file should not have 'internal' only information; that is it should not include information that other modules *should not use*.

24.3. CONFIGURATION POINTS

Build Configuration, in .h files

- Application specific build configuration
- Board specific build configuration
- Chip specific build configuration

25. REFERENCE SUBSYSTEM DETAILED DESIGNS

Standard Functionality of Common Modules:

- RTOS threading
- Instrumentation control loops
- Storage IO stacks
- Communication IO stacks
- Storage system type designs
- Motor control type designs
- Platform
- Application
- Board
- Chip specific
- Core specific

Specific fields often settle on a structure/schema which may be reused for a domain-specific class of problems. This simplifies the process and can offer advantages:

- Solid underlying theory
- Well-design test cases or easy to define in/out test cases
- Solid reference model

Use specific computing strategies; design architecture that does these kind of functions well.

Many modules are known (at least known in 80% detail). MCU peripherals, key/common troped elements

In an embedded system the design must divide the work the work between:

- Interrupts
- DMA and peripherals
- Threads (aka tasks), in the use of an RTOS or other OS.

25.1. RTOS AS AN ORGANIZING TOOL AND TO PROTECT TIME

Is an RTOS used? An RTOS provides a specific kind of structure and breakdown into threads that communicate. Major areas and COTS/SOUP are given their own thread.

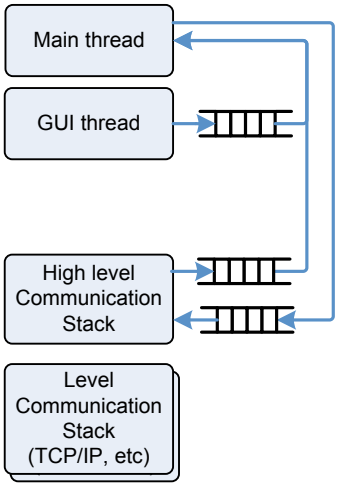

Figure 14: *Basic Separation into threads.*

The GUI and communication stacks are separated out into their own threads.

- Not safety critical, and may be unreliable or inconsistent.
- Must be isolated in time and space from the rest of the system. By placing these into their own thread – prioritized appropriately – the other functions (especially critical functions) can proceed even if the GUI & communication is "slow."

This is true of many other COTS/SOUP subsystems.

The device drivers may be separated out into one or more threads as well. This is common if there is a communication interface (including a communication interface to storage peripherals)

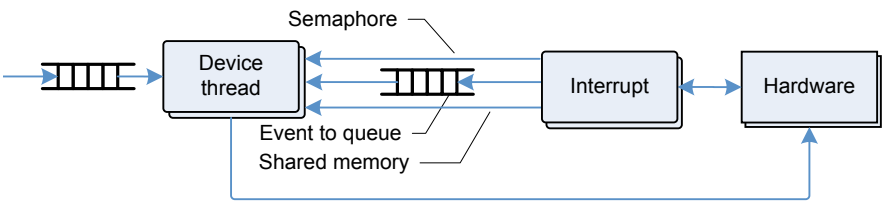

Figure 15: *Separation into threads & interrupts to drive hardware*

The interrupt service routine communicates with its thread by posting a semaphore (usually) or other similar event object.

A thread has the following basic structure

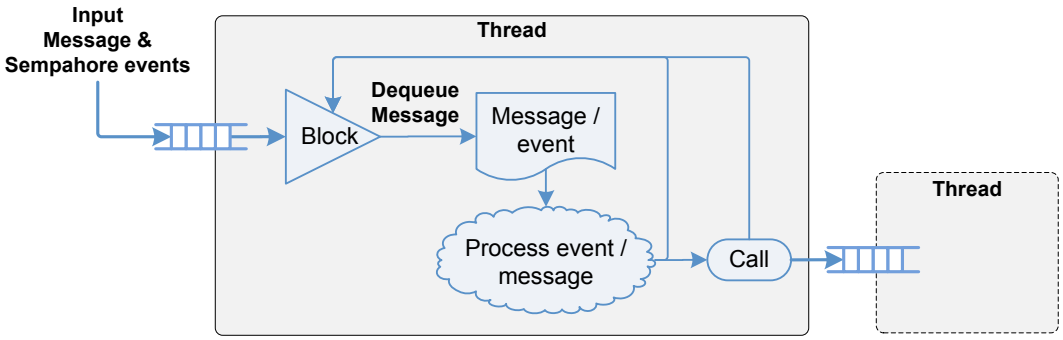

Figure 16: *Basic thread structure*

Threads typically have an input message queue. The thread blocks on semaphore and message queue events. When it wakes, it dequeues the event, takes action and goes back to sleep. It may post semaphores and messages to other threads, possibly indirectly as a result of framework/library/system calls.

Note: queues – message queues, IO queues, and so on – must be bounded in size.

25.2. INSTRUMENTATION

This section gives a template for instrumentation subsystems. Such a design typically looks like:

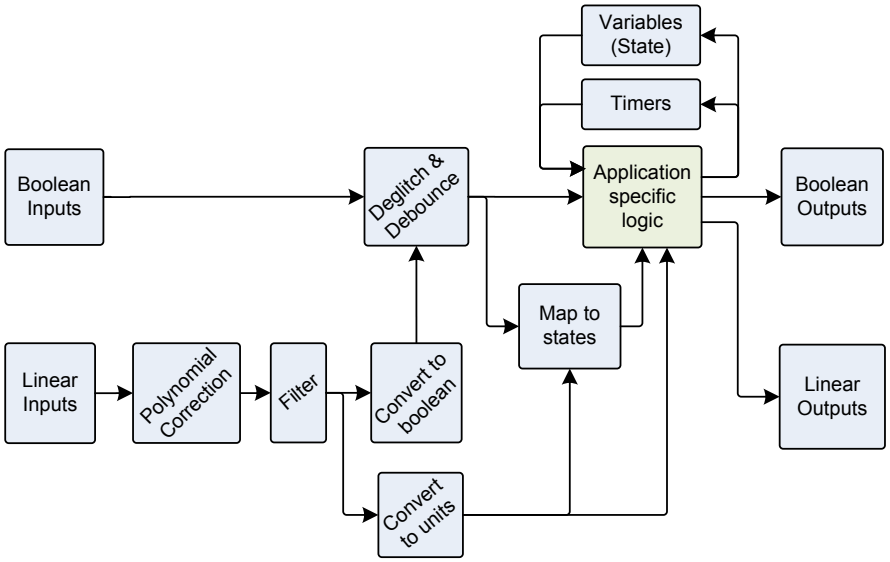

Figure 17: *Typical instrumentation structural diagram*

Each of the modules is "simple" input / output, so there is rarely any need to have them be in separate tasks/threads, and queues between them. The work of each can be done in a low, bounded amount of time. A task can run a loop like so to drive the inputs and outputs to each of the modules:

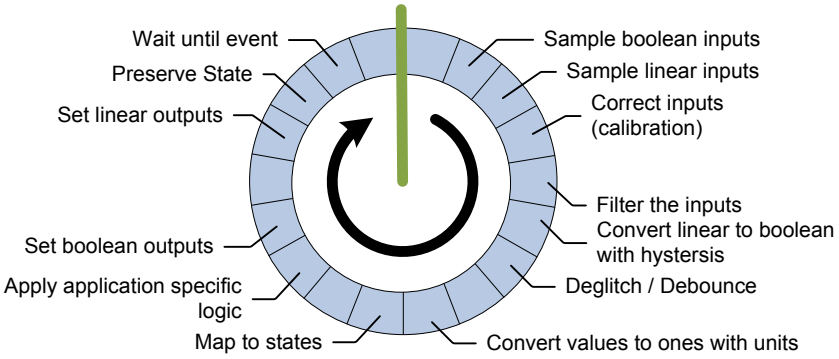

Figure 18: *Typical instrumentation loop*

The time thru the loop is more or less constant / bounded. Takes the same steps each time. The order is must be a topological ordering of the directed graph.

25.2.1 Division into modules

Consider the AUTOSAR organization of its HAL to name the microcontroller relate peripheral modules and other instrumentation modules. (For example, Chibios HAL did this).

The main types of modules include:

- Digital input (DIn). This can be one or more modules. One module may gather boolean inputs from the microcontrollers GPIO inputs. Other modules may gather boolean inputs from memory mapped input; I^2C, SPI and other remote peripherals (e.g. port expanders), and so on.
- Digital output (DOut). This can be one or more modules. One module may drive boolean outputs from the microcontrollers GPIO outputs. Other modules may gather boolean outputs from memory mapped output; I^2C, SPI and other remote peripherals (e.g. port expanders), and so on.

Each of these modules would get its own documentation.

25.2.2 DMA for gathering sampled linear inputs and linear outputs

The linear inputs and outputs – often called analog inputs and outputs on a microcontroller – might be performed by a DMA.

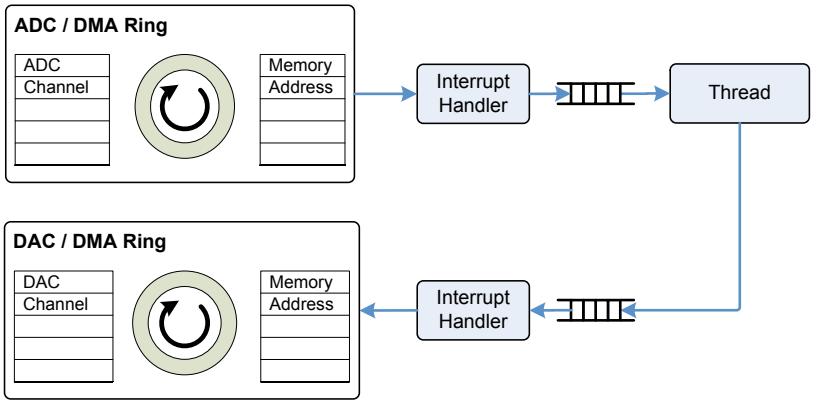

Figure 19: *DMA driven linear input and output*

For the linear inputs, the ADC samples a sequence of channels and the DMA places them into a circular memory buffer. For the linear outputs, the DMA takes bytes from a circular memory buffer and applies them to a sequence of DAC output channels.

25.3. COMMUNICATION STACK

This section gives a template for communication subsystems. Such a design might look like:

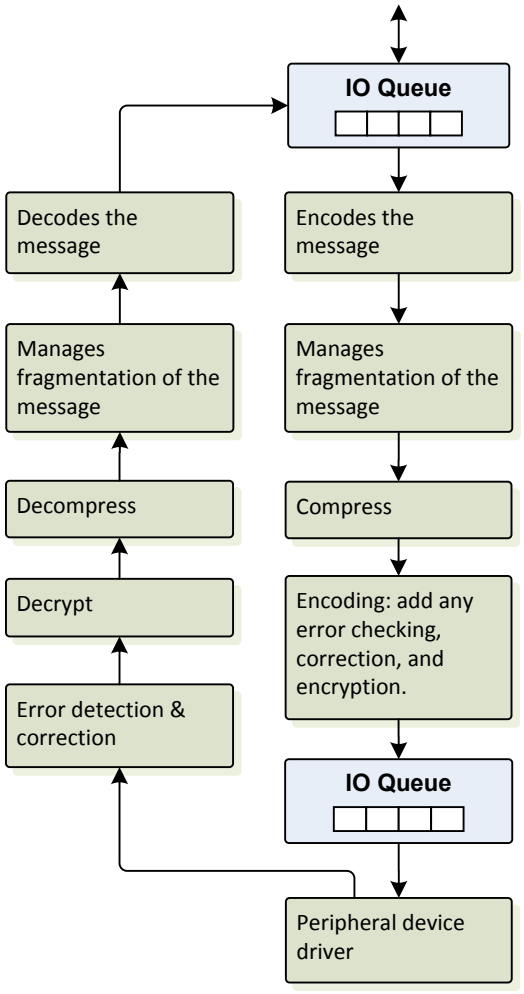

Figure 20: *Typical communication stack*

25.3.1 DMA for communication

The communication input & output might be performed by a DMA. (The output is well suited as the software is control of how much output is to be sent, and can delegate this to the DMA easily)

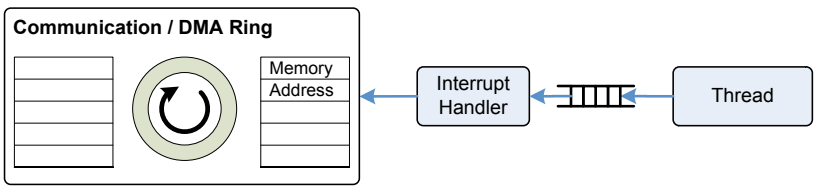

Figure 21: *DMA driven communication*

25.4. STORAGE

This section gives a template for storage subsystems. Such a design might look like:

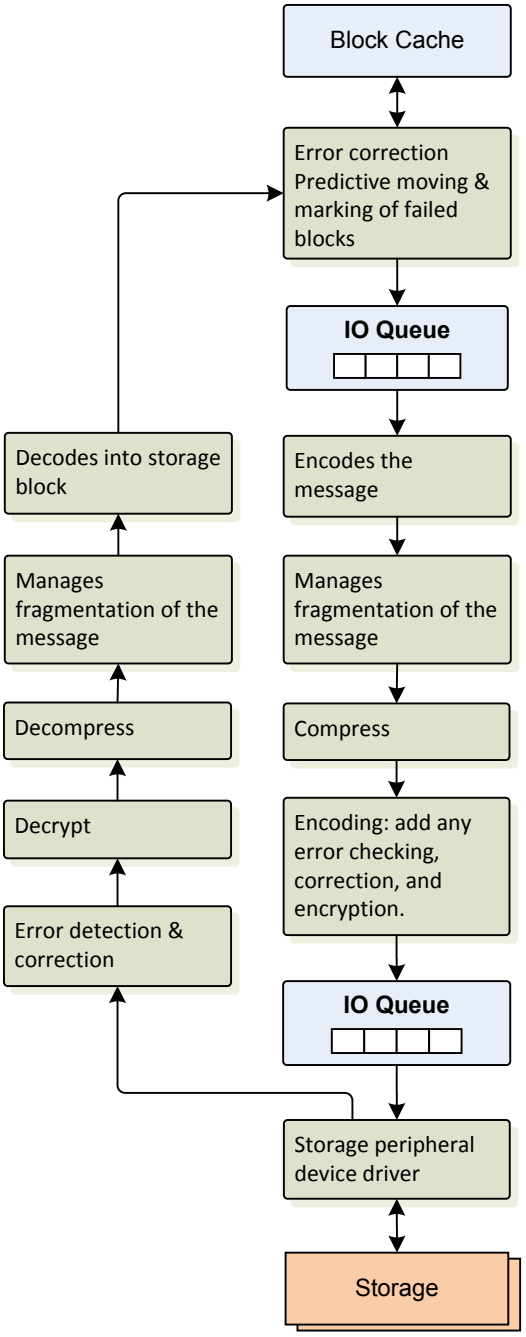

Figure 22: *Typical storage stack*

25.4.1 DMA for storage communication

The storage input & output might be performed by a DMA. (The output is well suited as the software is control of how much output is to be sent, and can delegate this to the DMA easily)

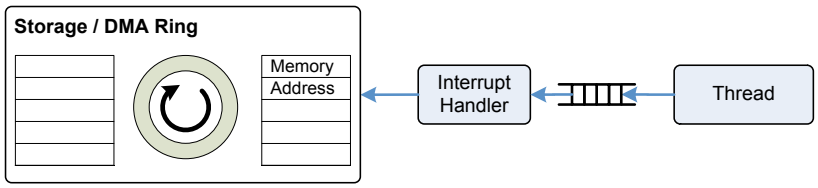

Figure 23: *DMA driven storage*

25.5. MOTOR CONTROL

This section gives a template for motor driver subsystems. Such a design typically looks like:

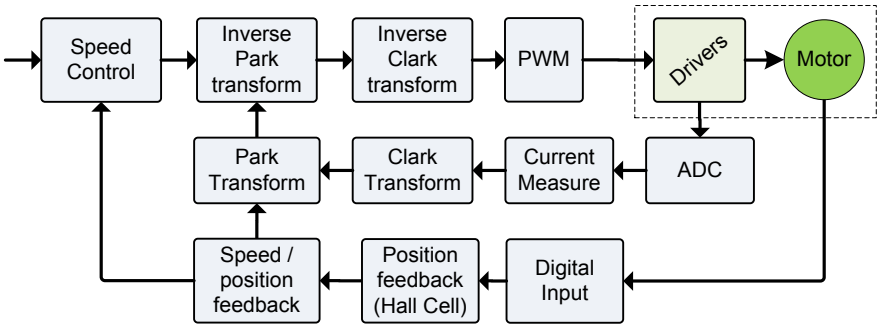

Figure 24: *Typical field oriented control of motor speed*

Most are transformations to different representations and acting on that or that some representations can drive hardware

26. FIRMWARE UNIT AND SUBSYSTEM TEST

This section discusses design for unit tests of the software modules and networks of modules. This approach allows:

- Testing of individual software elements
- Testing of test conditions that are too tedious, hard, or speculative to replicate
- Test boundary conditions, roots, pools, and other fiducial values with precision
- Test sequences of interactions that are too tedious, hard, or speculative to replicate
- Testing that error returns and error conditions are appropriately handled
- Regressions tests before software release
- Greater examination and validation of the internal software state
- Debugging the above scenarios

26.1. DESIGN TO BE DEBUGGED

Some design tips to improve debuggability:

- Think creatively about what could go wrong and what would be needed to figure out that it has gone wrong. (Too often, designs assume that nothing will go wrong.)
- Provide readable and resettable event counters to track the occurrences of key events.
- Provide read access to view the current state of key input and output signals, such as digital input pins and analog signals.

- Provide read access to variables that shows the current state of each state machine.
- Save a copy of the timer / counter value to a separate variable each time an event occurs. These values could be placed into a trace buffer, or read access to them could be provided.
- Function units – often procedures and modules – should check their parameters and internal state. (Do not use assert.) Instead clean up, increment an error count, add trace-point or break-point, and return a sensible / appropriate error.
- Measure performance counters for key functions and units. This can help identify inconsistent behaviour.

Read access should be interpreted to include support for a debugger watching the values, but can also include providing the values in a trace system, shared memory with a monitor, and via communication stack. The communication stack can be very useful for implementing the software tests.

26.2. SUBSYSTEM TESTS

The starting point to test the design is the software design description. The software design description should include structural diagrams of the modules, {a stacked (or a layered) of the modules}, a detailed list of the software modules, the signals that flow thru the software, and how to refer them.

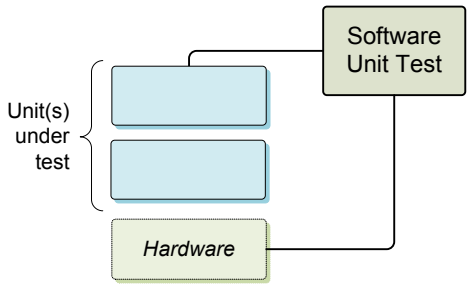

Figure 25: *Unit and subsystem test configuration*

The test concept is to

1. Identify the path or slice of these modules to test,
2. Develop a test to initialize and control just those software elements. This can be by injecting a stimulus into it using a DAQ. Note the projects electronic design should allow injecting into well-defined points, such as the board test points, or connectors.
3. Inject a stimulus into the slice. This can be sending a signal into the microcontrollers input, or invoking the function.
4. Check that the results produced are correct. This can included checking that the results produced at each stage of the along the path are correct, or that the rest of the system operates as expected. The output may be directing the results to a GPIO output, such as a DAC or digital output.

When planning the tests, one would start with tests for a single unit under test. Later tests would expand to more modules & layers, using (where possible) only units that have already been tested. The electrical hardware tests discussed earlier test the lowest layer hardware interfaces.

26.3. SOFTWARE SUPPORT FOR WHITE-BOX TESTING

The design documentation for software modules (and structural group), should include a section that describes how to test the module (or stack of modules). This section should include a description of:

1. How to observe when the module is performing work, when, and for how long
2. How to confirm that the module performs its intended function
3. How to find and test the limits of the unit performing its intended function

Many of the modules will have an accessible test interface. This many let one query its state. However, some procedures within the module will not be easily accessible in this way. The next section describes support for testing them.

26.4. TESTING INTERRUPT HANDLERS, TIMERS, AND PROCEDURES

This section provides an approach to answering the following test questions:

- How does one test that an interrupt (or timer) was raised?
- How long was it from the point that the timer was start to when it was raised?
- How does one test that a procedure was called?
- How frequently?
- How long is it the procedure execution duration?

The approach recommended here is to employ a GPIO signal to indicate entry and/or exit from a procedure. That is, an interrupt or timer handling procedure would be configured to raise a digital output signal when the procedure is entered, and lower the signal when the procedure exits. This signal can then be observed with an oscilloscope or DAQ on the test bench.

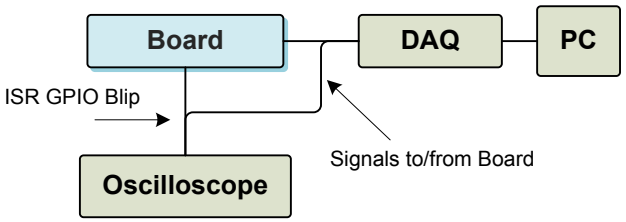

Figure 26: *White box test station configuration*

This can be extended to many other, non-interrupt related, procedures and key points in the software implementation.

26.5. TESTING THAT THE SOFTWARE IS NOT OVERSUBSCRIBED

To test that the software should not crash or become overburdened, no matter the input, the above techniques can be applied. Sending a high rate of switch signals – or other signals – should show negligible impact on the activity blips of the brake actuator. Blips of interest

- CPU "system tick" timer
- Run loop execution
- Interrupt handler

- Key timed events

The test procedure would configure the item to blip the GPIO signal. The blip should be regular, and bounded. A long gap between is a concern, and is evidence that the CPU is over loaded, interrupts are disabled too long, or the instruction engine is halted.

It is a concern if the line stays high too long as well: the item under test takes too long to execute, a higher priority interrupt or thread has occurred and impaired its execution time.

26.6. TESTING WATCHDOG TIMER(S)

Ability to disable "reseting" watchdog timer... Check that the timer expires, and that the progression to safety occurs as expected

27. REFERENCES AND RESOURCES

Blahut, Richard E; *Digital Transmission of Information*, Addison-Wesley, 1990

DI-IPSC- 81432A, *Data Item Description: System/Subsystem Design Description* (SSDD), 1999 Aug 10
http://everyspec.com/DATA-ITEM-DESC-DIDs/DI-IPSC/ DI-IPSC-81432A_3766/

IEC 61508-7: *Overview of techniques and measures* 2010

Part 7 outlines a good number of resources on how to approach the design process

Labrosse, Jean *MicroC/OS-II: The Real-Time Kernel,* 2nd Ed, CMP Books, 2002

An excellent example of the design a RTOS, with great detail given to detailed design, including algorithms.

27.1. CLEANROOM SOFTWARE ENGINEERING AND BOX STRUCTURED DESIGN

Hevner, A; Harlan Mills; *Box-structured methods for systems development with objects,* IBM Systems Journal, V32 No2, 1993

Harlan Mills write extensively on a process he called *Cleanroom Software Engineering*. His approach to structure decomposition, which he called *Box-structured design*, is a clear description on the process.

CMU/SEI-96-TR-022, Richard Linger, Carmen J. Trammell, *Cleanroom Software Engineering Reference,* Software Engineering Institute, Carnegie Mellon University, 1996 Nov

27.2. INSTRUMENTATION & SIGNAL PROCESSING

Garrett, Patrick H. *Advanced Instrumentation and Computer I/O Design: Real-Time System Computer Interface Engineering,* IEEE Press, 1994

Redmon, Nigel *Biquad Formulas* 2011-1-2
http://www.earlevel.com/main/2011/01/02/biquad-formulas/

Smith, Steven W *"The Scientists and Engineer's Guide to Digital Signal Processing,"* Newnes, 1997, http://www.dspguide

SPRU352G, *MS320 DSP Algorithm Standard, Rules and Guidelines,* Texas Instruments, 2007

An excellent example of detailed design documentation, and a good reference on microcontroller facing design of a signal processing framework.

27.3. MOTOR CONTROL

ST Microelectronics, BRSTM32MC "*Motor control with STM32 32-bit ARM-based MCU for 3-phase brushless motor vector drives*" (brochure)

ST Microelectronics, DM00195530 "*STSW-STM32100 STM32 PMSM FOC Software Development Kit Data brief*" #025811 Rev 2, 2014 Mar

Texas Instruments, BPRA073, *Field Orientated Control of 3-Phase AC-Motors*, 1998 Feb

Texas Instruments, SPRA588, Simon, Erwan; *Implementation of a Speed Field Oriented Control of 3-phase PMSM Motor using TMS320F240*, 1999 Sept

CHAPTER 10

Communication Protocol Template

This chapter is my template for communication protocol documentation.

- The kinds of activities that can be done thru communication channels
- Interaction sequences
- Overview of the communication protocol stack
- The link message formats (data structures)

Note: this chapter is placed before the detailed software design as it often drives some module design.

28. COMMUNICATION PROTOCOL OUTLINE

AN OVERVIEW, which includes:

- Name, designator, or unique identifier for the protocol
- A synopsis of the functions that it is responsible for
- The roles of the communicating parties
- A description of the transport methods, organized with OSI-like layers or TCP/IP-like layers.

INTERACTIONS. This section describes the typical interactions that would take place between the communicating parties.

THE PHYSICAL LAYER(S). This section describes the configurations employed with the different types of interconnection.

THE LINK / DATA LINK LAYER(S). This section describes the detailed framing and other differences employed with the different types of interconnection.

THE FRAME FORMAT for each type of link/transport media. (This corresponds to the network layer).

THE MESSAGE FORMAT covers the information in a command and response, and how it is encoded. (This often corresponds to the application layer).

29. INTERACTIONS

This section describes the typical interactions that would take place between the communicating parties. This is often flow diagrams.

29.1. READING A BIG BLOB OF DATA

The XYZ data is a binary "file" stored on the slave. The intended algorithm to retrieve the XYZ data is:

1. Read the size of the XYZ data, in number of bytes. For convenience, this will be called "*size*."
2. Set the current offset (which will be called "*offset"* here) to zero.
3. Send a read command with the read offset to the new offset value. The slave will send the data corresponding to that area of the XYZ data. This is synopsized in the diagram below

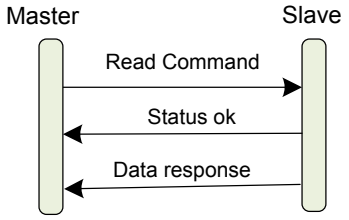

Figure 27: *Sequence for reading portion of the XYZ data*

4. The packet received will be an offset – this should match the one set – and a number of bytes of XYZ data. Place these bytes onto the end of the local copy of the XYZ data.
5. Increment *offset* by the number of bytes of data received.
6. If the *offset* is less than *size*, continue with step 3.

The figure below captures this process:

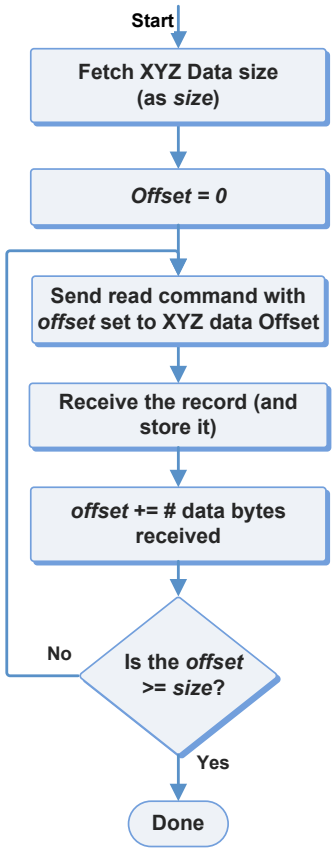

Figure 28: *The XYZ data retrieval algorithm*

30. THE DIFFERENT TRANSPORT MECHANISMS

The protocol is often possible to be conveyed over several different underlying interconnect methods. This section describes the detailed framing and other differences employed with the different types of interconnection.

For example:

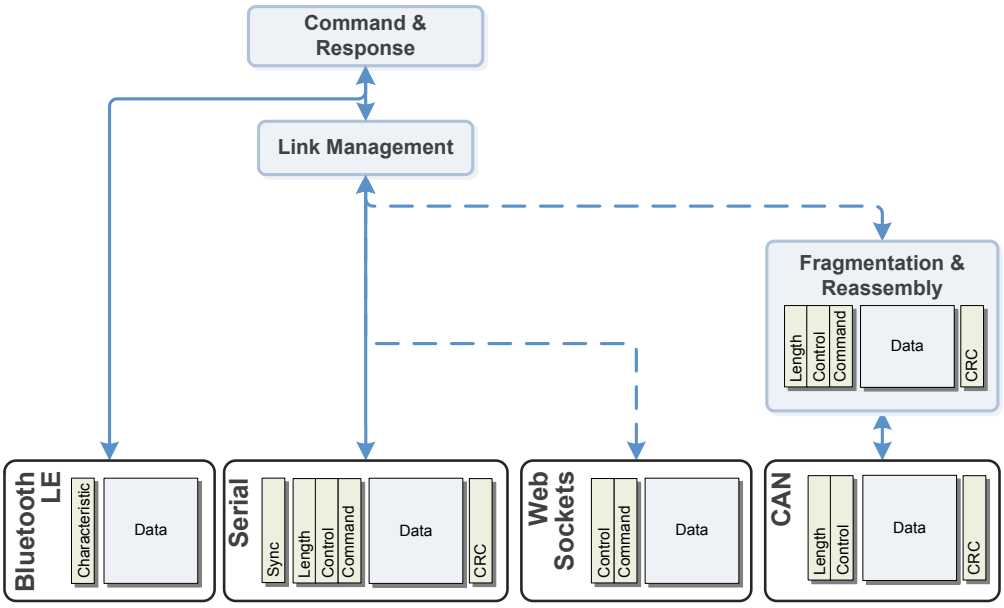

Figure 29: *Logical overview of the Communication stack*

- Bluetooth LE
- Serial communication, such as RS232, RS485, VCOM (over USB), or RFCOMM (over Bluetooth). This is a common protocol linking different microcontrollers together.
- WebSockets is a message-oriented protocol used on networks, such as available with Wifi, Ethernet, or Cellular data.
- CAN is a message-oriented protocol. This is a common protocol linking different microcontrollers together.

The communication links vary in the features they offer:

- Bluetooth LE handles the delivery, error detection, encryption, authentication, and much of the timeout of exchanging message frames. Bluetooth LE handles errors signaling, and the reference to the object being queried or acted on.
- Serial handles the delivery of the message. The software must provide mechanisms to detect errors, and lost messages. Serial has no encryption, authentication, or other security measures.
- WebSockets handles delivery, error detection, encryption (if a TLS module is employed), and much of the timeout of exchanging message frames. The software must provide its own error signaling, and means to reference the object being queried or acted on.
- CAN handles the delivery, and error detection of exchanging message frames. The software must detect damaged or missing messages, as wells as frames received in a non-sequential order.

30.1. THE COMMON LINK FIELDS

The link structures share many of the following fields:

- The *length* field is the number of bytes (octets) that follow the length byte, including the CRC field.
- The *control* field distinguishes between fetching a value, storing such a value, confirming the receipt of one, and so on. {A detailed explanation of these should be included below}
- The *status* field indicates success, any error, an indication, or notification. {A detailed explanation of these should be included below}
- The *command* field is which element to modify, and corresponds directly to a Bluetooth LE attribute / characteristic.
- The *data* fields are variable length, and optional.
- The CRC is the check value of message to help detect errors. {Of course, this is the place to describe in detail the parameters of the CRC, what is fed into it, the format of the value and so on.}

30.2. SERIAL FORMAT

RS232 serial interconnections lack the CRC checks, the start of packet, packet length and other information. This information is often added in by protocols using a serial interconnect. This section should describe that.

The commands/queries and responses have the following format

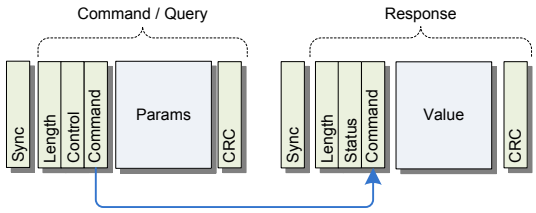

Figure 30: *The format of the command/query and response messages*

Other things to describe:

- What to do if the message doesn't pass CRC check? Ignore it?
- What about header field doesn't make sense?

This section should include the relevant (custom) configuration of the physical layers. Bit rate, number of bits, parity, etc.

31. TIMING CONFIGURATION AND CONNECTION PARAMETERS

This is the section to discuss the varied connection parameters and timing of recurring events on a per interconnect protocol basis.

32. MESSAGE FORMATS

This section should describe format and interpretation of the messages that go between the parties. What the fields are, how they are encoded, their units/dimension, scale, range, etc.

The example here is for a command-response mode, but can be adapter to other modes of communication.

It is easiest to specify the types as format, size, and sign types that match the C Coding Style (see Chapter 16), with a little endian encoding (or big endian if you prefer). The command and responses then provide

32.1. READ DATA

This command is used to retrieve a segment of data.

Command Code	{the hex value for the command goes here}
Characteristic UUID	{the hex for a Bluetooth LE characteristic UUID goes here}
Modes	Read, Notify, Indicate {this is more useful for Bluetooth LE}
Response Code	{the hex value of the response message}
Signature	offset× nBytes → MemStore → offset × $bytes^{nBytes}$
Command Size	4
Response Size	4-252
Equivalent Procedure	Foo_data_encode()

Table 23: *Summary of the Read Data command*

32.1.1 Command

The parameters of the command body are:

Offset	Size	Type	Parameter	Description
0	2	uint16_t	*offset*	The offset to retrieve the data from
2	2	uint16_t	*size*	The number of bytes to retrieve

Table 24: *Parameters for Read Command*

32.1.2 Response result

The parameters for the Read response message:

Offset	Size	Type	Parameter	Description
0	4	uint16_t	*offset*	The offset of data
4	*varies*	uint8_t[]	*data*	The retrieved data

Table 25: *Parameters for Read Response*

The intended use is to read a segment of the data buffer. The typical read sequence is below:

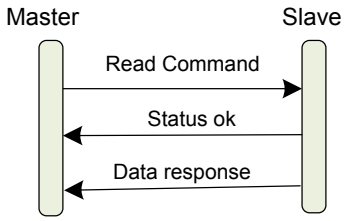

Figure 31: *Read command sequence on success*

The sequence for an invalid read command is show below:

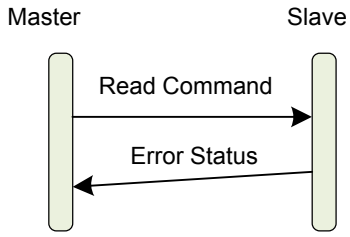

Figure 32: *Read command with error response*

33. REFERENCES AND RESOURCES

DI-IPSC-81436A, *Data Item Description: Interface Design Description* (SDD), 1999 Dec 15
http://everyspec.com/DATA-ITEM-DESC-DIDs/DI-IPSC/DI-IPSC-81436A_3748/

CHAPTER 11

Software Module Documentation Template

This chapter is my template for a detailed software design description of a module.

- Detailed design outline
- The overview of the module
- The software interface documentation
- The detailed design (internals)
- The configuration interface

34. DETAILED DESIGN OUTLINE

I use the following template for the documentation of each software module:

AN OVERVIEW, which includes:

- Name of module
- A synopsis of the functions that it is responsible for
- Diagram and description of the module's main organization. This isn't intended to be the design diagram; it is intended to show where it fits into the bigger design.

SOFTWARE INTERFACE DOCUMENTATION. This section describes how software would communicate with the module system using procedure calls.

DETAILED DESIGN. This section describes the detailed internal structures and procedures used within the module.

THE TESTING section describes how to test the module.

35. THE OVERVIEW SECTION

The overview section introduces the module, and its role. I include a diagram that shows the where the module fits in the bigger design, and how the other modules interface to it:

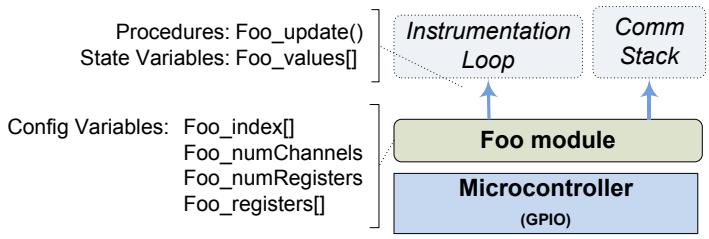

Figure 33: *Overview of the Foo module*

The diagram is usually vertically organized. The upper layer communicates with the rest of the system; the lower layer works with the hardware or more specific work tasks. Between the nodes for the different modules (and hardware elements) are callouts synopsizing the procedures, variables, and IPC structures that act as the links between the nodes. The typical major interfaces include:

- Interface that the system can use to configure the module.
- Interfaces that the rest of the stack or software system may interact with the module
- Interfaces from the module to the underlying layers, or the lower-layers of the stack that it interacts with.

36. THE SOFTWARE INTERFACE DOCUMENTATION

The software interface section describes how software would communicate with the module system using procedure calls. This includes a description of the procedures, structures, the respective parameters of these, calling sequences, responses, timing, and error handling.

A good software interface is…

- Easy to learn / memorize
- Leads to readable code
- Hard to misuse
- Easy to extend
- Sufficient or complete for the tasks at hand

The overview should describe:

- INITIALIZATION, which is passed information about how the microcontroller is connected to the board, and which of the internal resources that should be used.
- DATA ACCESS. The procedures that get data from the module or provide data to the module
- CONFIGURATION. How the module is configured to use lower-level resources, and the parameters (such as data rate) in how it should use the resources. (Included where appropriate.)

After the overview there should be:

- Description of operations
- Diagram of interaction, algorithm
- The #defines and enumerations used in the software interface
- The data structures employed by the software interface

- The variables provided by the software interface
- The procedures (and their parameters) provided the software interface

36.1. CALLING SEQUENCES FOR THE INTERFACE

All interfaces should provide a BNF-style description of the acceptable calling sequences, or phrases, for the API. For example:

```
::= open [optional_calls] (read | write | lseek)* close;
```

or

```
::= open mmap  (MemoryOp | mincore | lseek | read | write) munmap close
  |  open shmat (MemoryOp | mincore | lseek | read | write) shmdt  close ;
```

The conventions for such BNF-like statements include:

- Parameters aren't specified in the rules
- Only specify calls related, usually in a context. That is, specify only the API related to an 'instance' (object, file channel, etc.) from its creation and manipulation through its destruction.
- Items in italic refer to other rules
- Items in parenthesis form a regex-like set of alternatives
- Items in braces are optional, the equivalent of a null option in an alternative grouping
- A sequence of calls is only valid if it is accepted by the rules outlined. Under the rules of software validation, the software is erroneous if it is possible that the software executes a calling sequence not recognized by the BNF.
- Keep the number of rules small, but reflect the real constraints on the calling sequence

36.2. DEFINES

This sections describes the #defines used in the software interface.

```
#define CMD_READ  (0xA000u)
```

The read command value.

```
#define CMD_WRITE  (0x2000u)
```

The write command value.

36.3. ENUMERATION TYPE DOCUMENTATION

This section describes the enumerations used in the software interface.

enum ABC

This enumeration is such and such, used for so and so.

36.4. DATA STRUCTURE DOCUMENTATION

This section describes the data structures used in the software interface. The table below synopsizes the data structures:

Structure	Description
Foo_t	This structure is used to track info

Table 26: *Foo Structures*

Foo_t struct Reference

This structure tracks the hours of operation.

Field	Type	Description
secondsElapsed	uint32_t	The number seconds since the start of operation
prevSeconds	uint32_t	The number of seconds of operation that were logged
startTime	uint32_t	The time that the operation was started.

Table 27: *Foo _t structure*

36.5. CLASSES

The table below describes the classes employed in the module:

Class	Parent	Description
Foo		An abstract base class to do some interesting things.

Table 28: *Module classes*

Foo class Reference

This is an abstract class intended to do some interesting things.

Field	Type	Description
someField	ItsType	Describe the field
someOtherField		

Table 29: *Foo class structure*

Method	Description
isOutOfDate ()	Checks to see if the foo bar is out of date.
unload()	Unloads the foobar from memory.

Table 30: *Foo methods*

36.6. VARIABLES

This section describes the variables in the software interface. The table below describes the variables provided by the module:

Variable	Description
Foo_errorCount	The number of errors encountered
Foo_successCount	The number of successes encountered

Table 31: *Foo variables*

36.7. PROCEDURES: SYNOPSIS

This section introduces the procedures used in the interface. The table below describes the modules procedure interface:

Procedure	Description
Foo_update()	Called to update the state of the module each time step.
Foo_write()	Write a data block to the device

Table 32: *Foo interface procedures*

36.8. PROCEDURE DOCUMENTATION

This section describes procedures that the module exports.

void Foo_update(void)

This updates the internal state of the module with each time step, and prepare output results.

Parameters:

none

Returns:

none

This should describe the behaviour of the procedure, its algorithm, or other steps that it may take.

Err_t Foo::write (void* *address*, uint8_t* *buffer*, uint16_t *length*)

Write a data block to the device

Parameters:

address	The address within the device to store at
buffer	The buffer holding the data to write; this must hold *length* bytes
length	The number of bytes to write

Returns:

Err_NoError	The data was successfully written
Err_Address	The address is not a valid memory page
Err_Timeout	The operation did not complete timed out
other	Other access error

This should describe the behaviour of the procedure, its algorithm, or other steps that it may take.

37. THE DETAILED DESIGN SECTION

The detailed design section describes the detailed internal structures and procedures used within the module. This includes a description of the procedures, structures, the respective parameters of these, calling sequences, responses, timing, and error handling:

- Diagram(s) breaking down the module
- Description of operation, such as the main functions of the module, any threads and/or interrupt service routines
- Diagram of interaction, algorithm
- Detailed design info
- The #defines and enumerations used within the module
- The data structures employed by the module
- The variables internally employed in the module
- The procedures (and their parameters) within by the module
- The files employed in the module

Most of these sections follow the same format as used in the software interface.

The diagram below synopsizes the organization of the Foo module:

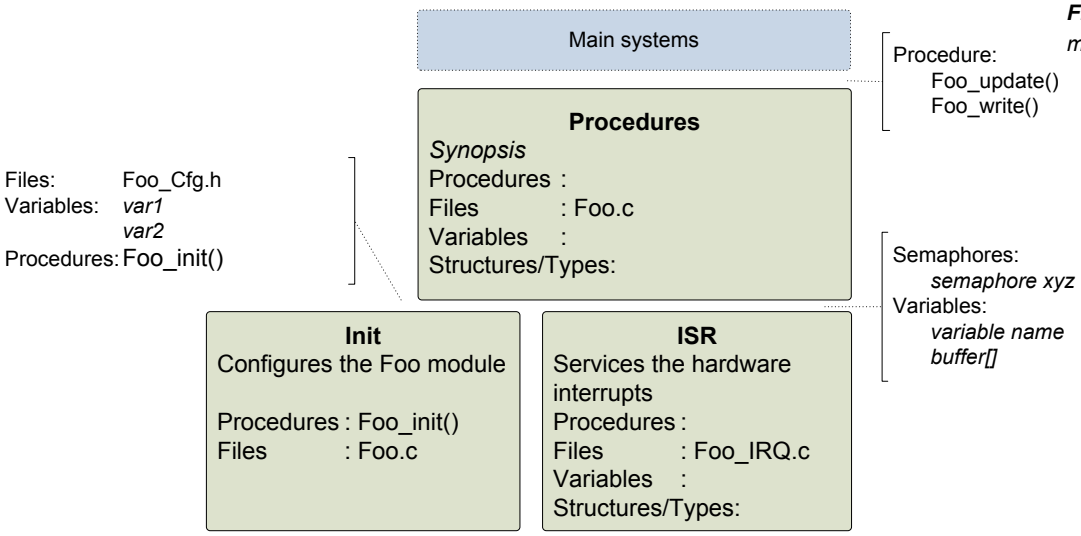

***Figure 34:** Detailed module organization*

37.1. INTERRUPT SERVICE ROUTINES

This section should introduce and describe the interrupt service routines. This should define why they are called, what action they take, and how they interact with the rest of the system.

37.2. DEFINES

This sections describes the #defines used internally.

same format as in the interface section

37.3. ENUMERATION TYPE DOCUMENTATION

This section describes the enumerations used internally.

same format as in the interface section

37.4. DATA STRUCTURE DOCUMENTATION

This section describes the data structures used internally.

same format as in the interface section

37.5. CLASSES

This section describes the classes used internally.

same format as in the interface section

37.6. VARIABLES

This section describes the variables used internally.

same format as in the interface section

37.7. PROCEDURES: SYNOPSIS

This section introduces the procedures used internally.

same format as in the interface section

37.8. PROCEDURE DOCUMENTATION

This section describes the procedures used internally.

same format as in the interface section

37.9. FILES EMPLOYED IN THE MODULE

The table below describes the files employed in the module:

Table 33: *Module files*

File	Description
Foo.c	The foo modules API procedures
Foo.c	Public interface to the Foo module
Foo_cfg.h	Public interface to the configuration of the Foo module
Foo_IRQ.h	Header file describing the local interface to the Foo Interrupt service routine
Foo_IRQ.c	The interrupt service routines.

38. CONFIGURATION INTERFACE

This section describes the configuration of the module.

The configuration is usually defined statically, at build time. The main application defines const variables with the values to configure the module. This allows a module to be reused in many applications, without specifying the exact size of resources used or coupling to the hardware

38.1. VARIABLES

The table below describes the BSP configuration variables provided to the module to configure it:

Variable	Description
Foo_numChannels	The number of channels used by the module.
Foo_numRegisters	The number of peripheral registers defined.
Foo_register[]	The set of peripheral registers

Table 34: *Configuration of the Foo module*

39. THE TEST SECTION

The test section describes how to test the module. It should include a description of

1. How to observe when the module is performing work, when, and for how long
2. How to confirm that the module performs its intended function
3. How to find and test the limits of the unit performing its intended function

Planning the test:

- Start with tests for a single unit under test, and expand to more layers.
- Different mechanisms of tests

The rest of the test section should focus on three different mechanisms for performing the tests:

- The software-based tests are intended to catch coding and calculation bugs. These checks typically cannot catch hardware interaction bugs, but they can do *regression checks* on software and (some) hardware configuration bugs.
- Desk checks look at the actual system execution, probed by hand
- Bench checks are more automated checks, with software and hardware probes

Note: the test documentation is often placed in other documents. I find it beneficial to include an outline of tests. It helps ensure that the design is focused on the testability of the module (and module stack).

40. REFERENCES AND RESOURCES

DI-IPSC-81435A, *Data Item Description: Software Design Description* (SDD), 1999 Dec 15
http://everyspec.com/DATA-ITEM-DESC-DIDs/DI-IPSC/DI-IPSC-81435A_3747/

SPRU360E, *TMS320 DSP Algorithm Standard, API Reference,* Texas Instruments, 2007 Feb

An excellent example of describing how the modules, algorithms, and interfaces work and are intended to be used.

CHAPTER 12

Software Design Guidance

This chapter describes my guidance for detailed software design:

- Specific guidance
- Design to be debuggable
- System microcontroller specific guidance
- Tests

41. CODE REUSE

Some observations on code reuse:

- Library design, creation, and support are a development effort in and of itself, with many factors that impair success;
- Small pieces of code are more readily reused;
- The file is the easiest basis of reuse. If a function is potentially reusable, pull it out into its own file. Files should be small, single-purposed, and well focused.
- Good design & coding practices facilitate easier reuse.

Some approaches and techniques to help promote reuse of code and designs:

- Use small size procedures and files in the design
- Divide the procedures and sub-procedures, where possible, into those that can be used across a wide range, and those which are specific to the project (i.e., the hardware or application). Keep feature-specific code segregated into different files to for later leverage and reuse.
- Provide a segment of time (e.g. at the end of the project) for reviewing and identifying reusable code.
- Identify, during design reviews, areas where prior design *should*[10] have been reused (but was not).

Existing code is (mostly) worked out, so can accelerate a project schedule – *if* the code is appropriate to the project. Such code must be stored in manner so that it is accessible, easily

[10] Note the emphasis is on *should*, not *could.* This step is also fraught with politics in some offices, which will undermine quality.

found, and readily reusable. Each successive project may produce more of the "library" of reusable code.

Decide what code will be reused. There levels of code reuse

4. Verbatim: The file is picked up and used with zero changes.
5. The code is copied and modified it to fit some custom nature of the project
6. A code template is reused where the structure is used unchanged, but the contents are customized for each project. This avoids re-inventing mechanisms, and supports (to some extent) a fill-in-the-blank approach to reuse.

Limits and conditions for reuse. Some code modules have license restrictions, such as to the hardware supplied by the vendor. Some are certified; so the reuse must match the certification. Use in a more stringent domain, should plan on TBD;

The application must cover (1) the areas that the certification doesn't cover (2) areas that the coverage isn't sufficient (2)

CAVAET. It's socially unpopular to suggest reuse is overrated. Yet it is overrated. The intent of reuse is noble; the effort, time, skill, and development organization required to accomplish these is too often provides insufficient return on that expenditure.

42. DESIGN TO BE DEBUGGABLE

If you don't design it to be debuggable, you won't implement it to be debuggable, and you won't have a debuggable system. You're stuck hacking till the problem that you are experiencing goes away.

- Make a clean design
- Use good implementation techniques
- Fault detection support (this is covered in detail in the next section)
- Performance support is covered in the performance document.
- Gathering instruction traces
- Gathering other kinds of traces

42.1. SUPPORT STATIC ANALYSIS

Static analysis tools – tools like lint that are fed all of the source code for the project – can often find bugs. They tend to find bugs of misuse, out of range, pointer errors, potential buffer overflows, and the like.

The design can support this analysis making it easier to analysis. Here are some tips (not all are practical):

Use indices rather than pointers. Why? Pointer analysis is still a difficult subfield of static analysis – that is, a pointer have any address, but only a discrete set is valid, and it is hard to know what those are. In contrast, an index can be check for a small range of valid values. Lint and other tools may do a better job in those cases for analyze correct usage, data flow, and spotting bugs / issues.

42.2. CLEAN DATA FLOW

Intent easy to reason how a variable (or other element) took on the value it has, where the data came from, and was calculated. Avoid coding style that generates confusion[11]

- Use meaningful names the variables.
- Don't reuse variables in a procedure. At least with Keil based debugging use –o0 until the function is correct. The compiler will optimize away the variables (in a properly constructed procedure).

42.3. CLEAN CONTROL FLOW

Intent easy to reason how the execution reached a particular point in the code, and why the software took the actions that did.

Code complexity measures[12] tell you when it's too complex to maintain, understand, reason about.

- MC/DC branch metrics tell you as well (branching and/or indeterminate values) where you will have increasing inference pain.

42.4. DON'T USE LARGE PROCEDURE: USE MORE SMALLER PROCEDURES

A large procedure is not good. It is hard to reason about what variables are live, dead, and what the control flow was to the failure point. The use of procedures partition the problem down – in the sense of what is relevant, what isn't.

Give them good names too.

42.5. EVENT COUNTERS

Include a series of event counters that can be watched in a debuggers "live watch" window. The can include counters of:

- Send events
- Receive events
- Send errors
- Receive errors
- Allocation events
- Free events

43. FAULT DETECTION: DETECTING AN ERROR CONDITION

Something like "smart breakpoints" can be used. These are breakpoints that trigger the debugger (or signal the test system in other ways) when certain conditions are met (or, conversely, if certain rules are broken). They can be implemented in software (as assertions or other techniques), in some debugger (which may break on certain memory accesses or modifications, or software break instructions), or in simulation environment monitors.

---

[11] single point of return forces confusion and questions where the return variable was bound
[12] The formulary doesn't matter

The key is the condition or rule: How do we know when something is wrong?

- Check parameters – range and conditions
- Check returned values
- Check intermediate values / conditions, error flags
- Canary methods to find buffer overruns
- Canary methods to find stack overruns
- Library exceptions
- Hardware exceptions and faults

43.1. CHECKING PARAMETER VALUES

AT THE COMPONENT LEVEL, messages are checked for correctness. These checks mimic those above for a procedure call.

PROCEDURES check their parameters at the start, and reserve any resources they will need to complete the task. If the checks do not pass, or resources cannot be reserved, the procedure exits early with an error code. Check error returns from calls. Their use of other procedures use appropriate timeout values.

The parameter value range and constraints is typically specified at the interface level.

43.2. CHECKING THE RETURN VALUES AND ERROR CONDITION

The procedure or message processing can signal a fault when the return values are out of specified bounds, or there is an error return.

43.3. CHECKING INTERMEDIATE VALUES / CONDITIONS, ERROR FLAGS

Check the values of intermediate calculations.

Check the error return or flags of procedures it calls.

"This application has requested the Runtime to terminate it in an unusual way."
– An actual Microsoft error message

43.4. MEMORY SEGMENTATION

Working memory should be segmented by the criticality classification of its use or owning module. *Canaries* should be placed between segments to detect over run/under run between segments.

An example partitioning of memory into segments is shown below

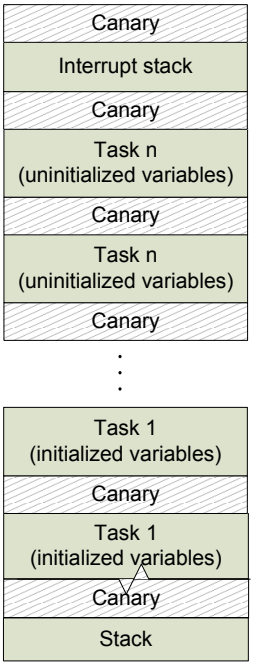

Figure 35: *Segmentation of memory with canaries*

The canary values should be checked frequently, such as during a timer tick, or every run thru a main loop. It is recommended to use different canary values (`0xdeadbeef`, `0xc0fecafe`, etc) to help id what's going on in a memory dump situation.

43.5. BUFFER OVERFLOW CHECK

BUFFER ADDRESSING CHECKS. Buffers should have *canary* values before and after the buffer area to aid in identifying stack overflow and underflow events. Buffer over and under runs are very common form of software bug, this will help detect such out-of-bounds modification:

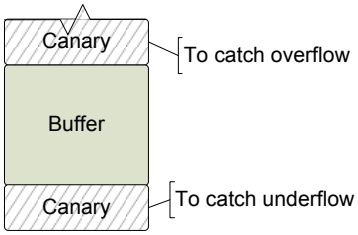

Figure 36: *Overview of buffers with canaries*

The canary values should be checked frequently, such as during a timer tick, or every run thru a main loop. It is recommended to use different canary values (`0xdeadbeef`, `0xc0fecafe`, etc) to help id what's going on in a memory dump situation.

43.6. STACK OVERFLOW CHECK: CANARY METHOD (AKA RED ZONES)

Stacks should be monitored for overflow conditions by checking that the memory surrounding the stack has not been modified.

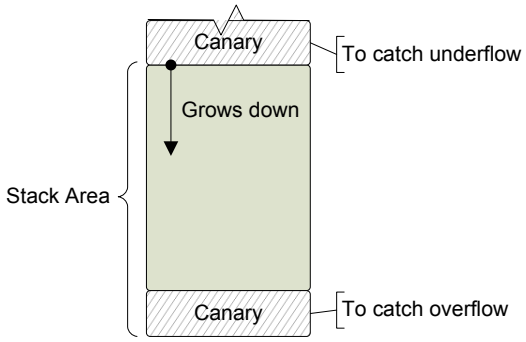

Figure 37: *Overview of the stack structure with canaries*

Software should place *canary* values on either end of the stack – or stacks, when an RTOS is used. Some compilers or linkers automate this.

FINDING UNDERFLOWS AND OVERFLOWS are a matter of checking each of the aforementioned buffers to verify that that the canary's still have valid values at the start and end.

The canary values should be checked frequently, such as during a timer tick, or every run thru a main loop. It is recommended to use different canary values (0xdeadbeef, 0xc0fecafe, etc) to help id what's going on in a memory dump situation.

THE ADVANTAGE of this approach is that it is easy to implement, easy to understand, and has low overhead costs on the executing firmware.

THE DRAWBACK is that it still possible to miss overflows and underflows: the stack pointer can be incremented by large amounts, completely skipping over the canary area.

43.7. DETECTING LOSS OF INTEGRITY OF A RAM VARIABLE OR DATA STRUCTURE

This section will look at how to prepare variables and access variables in a way that will detect a loss of integrity.

A typical module has, for globals, one or more of the following:

- An array of values (e.g. myValues[])
- Individual variables
- An array of structs (e.g myStructs[])

STEP 1: First, gather the individual variables into a struct. For purpose of an example, this struct will be typedef'd with the name bar_t.

STEP 2: Create primary and secondary instances of these. For example:

```
static bar_t barPrime, barMirror;
```

Arrays of values and structs would be renamed, and a mirror added. For example, myValues[] becomes:

```
static type myValuesPrime[size], myValuesMirror[size];
```

And myStructs[] becomes:

```
static type myStructsPrime[size], myStructsMirror[size];
```

The mirror and primary variables should be stored in separate memory segments (see section 43.4)

STEP 3: Define wrapper macros around these. Respectively the examples become:

```
#define barFetch(field)                  structFetch(barPrime, barMirror, field)
#define barStore(field, value)           structStore(barPrime, barMirror, field, value)
#define myValuesFetch(field)             aryFetch(myValuesPrime, myValuesMirror, field)
#define myValuesStore(field, value)      aryStore(myValuesPrime, myValuesMirror, field, value)

#define myStructsFetch(idx,field)        structFetch(myStructsPrime[idx], myStructsMirror[idx], field)
#define myStructsStore(idx,field, value) structStore(myStructsPrime[idx], myStructsMirror[idx], field, value)
```

STEP 4: Modify references in the code to use these wrapper macros. The table below summarizes how to convert from conventional accesses.

***Table 35:** Rewriting*

Conventional	Becomes
value = myValues[idx]; myValues[idx] = value;	value = myValuesFetch(idx); myValuesStore(idx, value);
value = bar.field; bar.field = value;	value = barFetch(field); barStore(field, value);
value = myStructs[idx].field; myStructs[idx].field = value;	value = myStructsFetch(idx, field); myStructsStore(idx, field, value);

43.7.1 The Theory of operation

Each variable (or data field) has a mirror, which holding complementary value. The mirror values are ones-complemented. A one's complement is used rather than a direct copy to better reject common-mode faults such as a brown-out. These would flip bits toward the same bias (usually ground).

The process of storing is simple:

1. Store the value in the primary variable or storage field
2. Store the ones complement (~value) in the mirror variable or field.

Fetching the field is a bit

1. Get the value from primary field, and perform a 1's complement on it. We intentionally don't complement the same field as above.
2. Get the value from the mirror field
3. Xor the mirror and complemented primary value.
4. If the result is non-zero, at least one was corrupt. This triggers the fail-safe handler, where it will remain until the software or hardware resets the microcontroller. The reset will reinitialize the values of the programs variables.
5. If the result is zero, the value is good and can be used.

43.7.2 The detailed implementation

An earlier section discussed use of structFetch(), structStore(), aryFetch() and aryStore() to gain the benefits of error detection. Their implementation is tricky, but important, so it is explained here. We will look at the macros for structures. The ones for arrays are nearly identical, so we will not look at them.

THE STORE MACRO is:

```
#define structStore(prime,mirror,fieldName, value) \
                ((mirror).fieldName = ~(value),         \
                  (prime).fieldName = (value))
```

This matches the steps in the previous section. Some notes about this macro:

- A macro lets us use a field name (rather than identify a field by index and employing an enumeration.) So this means a small number of fetch macros, and store macros total, no matter how many fields or structures
- The whole macro is wrapped in a parenthesis. This lets the substituted text act as an expression, especially a statement. It also prevents the internal steps from interacting with other expressions, and control structures in surprising ways.
- The "*mirror*" and "*prime*" macro variables are wrapped in a parenthesis. This allows an expression to be passed into the macro, and not have unexpected evaluation results when it is expanded. That is, a pointer expression could be used with this macro for prime or mirror.
- The "*value*" macro variable is wrapped in a parenthesis, for similar reasons as above. Wrapping the variable before the complement (~) operator is especially important. If an expression "0+x" was passed in, only the "0" would be complemented, not the result of the addition.
- The comma operator is used to allow to separate expression statements to be employed in the macro. The comma operator should be employed rarely, but this is an instance it is needed. Without it, we would have to employ a more complicated structure to allow access to the fields without ballooning the code, and protecting the macro contents from unexpected interactions.

THE FETCH MACRO also follows the corresponding steps in the previous section. It also has similarities to the store macros construction. The macro is:

```
#define structFetch((rime,mirror,fieldName)                     \
                (~((prime).fieldName ^ (mirror).fieldName) ?    \
                FailSafe(Err_ramErrorDetected)  :               \
                (prime).fieldName)
```

- The macro is wrapped in parenthesis, as described above. So are the macros parameters.
- The second line of the macro performs the complement of the primary; it's XOR against the mirror, and comparison against zero.

- The third line handles the case of a mismatch. It calls FailSafe(), which is intended to places the system into a safe state and then reset the microprocessor. It is not intended to return, but for C syntax it is defined as returning a value.
- The fourth line returns the primary value, if there was a match.

43.7.3 Commentary on design alternatives

There were several other design alternatives considered.

- Using direct procedures would increase the amount of software to be written, making it harder to understand and maintain over all. There would be two procedures per field per structure. The procedures written without a macro risk a local defect that is not spotted by tool or inspection. A helper macro to write them would provide consistency, yet would still leave a lot of code.
- It is possible to reduce the number of procedures is by referencing the fields with an index (and an enumeration, to name the index). This would be a small number of procedures per structure – two for every type used in the structure.
- Returning an error code from a procedure is the usual, mandated design approach in a long running, reliable system. In this design, it is not necessary as there is no clean up to perform, and external peripherals to manage. The RAM has been compromised, so the best, least burdensome recovery is to restart.

These macros make heavy use of the comma operator, and are the only known place to do so in the code.

43.8. MEMORY PROTECTION

Many microcontrollers (e.g. Cortex-M) include some facilities to protect regions of memory. This is recommended to be used where possible. It is not as fine grained as the techniques above, so it is no substitute.

43.9. INCLUDE A MEMORY MANAGER THAT CAN DETECT LEAKS

In systems that dynamically allocate and release memory, it is helpful to have debug-builds that can detect when memory has been leaked.

43.10. TASK SWITCH

The RTOS task switcher can be modified (or hook, such as with App_TaskSwHook() in µC/OS-II) to implemented a check of the stack canaries when task switch occurs. If the canary variables do not have their proper value, then the software can signal a fault condition.

43.11. LIBRARY ASSERTS, EXCEPTIONS

Many of the firmware libraries perform checking, and signal errors by called a procedure, similar to "assert".[13] By supplying this error procedure, the software can signal an error condition. The error procedure should trigger a software breakpoint (to trigger the debugger) and handle the error, perhaps by putting the machine into a safe state and halting.

[13] I personally think that this is a sign of a very poor design and implementation.

44. SYSTEM AND MICROCONTROLLER SPECIFIC DETAILED DESIGN ELEMENTS

44.1. CORTEX-M PROCESSORS

This section covers design features specific to Cortex-M based microcontrollers.

44.1.1 Atomicity

On the Cortex-M processors, loads and stores are atomic *only* if:

- It is an 8-bit transaction, or
- It is a 16-bit transaction to an address aligned 16-bits, or
- It is a 32-bit transaction to an address aligned 32-bits

Normally the compiler takes care of this of this alignment. The exceptions – which will void the atomicitiy – are if

- a compiler option has been used to change padding or alignment
- The variable was specified with an address
- The C "pack" qualifier was used

This means that volatiles are not read or written atomically on the Cortex-M unless all of the conditions mentioned are followed. Compare and swap techniques or disabling interrupts must be used when modifying memory shared with an interrupt routine

44.1.2 A note on ARM Cortex-M0 processors

The ARM Cortex-M0 instruction core cannot do:

- Compare and swap (LDREX/STREX)
- Atomic writes or increments
- Bit-banding
- Detecting that debugger is attached

The techniques below are still (largely) applicable, but will have Cortex-M0 specific adaptations.

44.1.3 Software breakpoints

44.1.4 Hardware Exceptions

Exceptions, and faults, are a type of error detected by the processor at run-time. By supplying the appropriate handling procedure, the software can signal an error condition. The handlers can preserve the call stack, key register values, and key global variables. This may be helpful for identifying what was going on.

44.1.5 Memory barriers

Memory barrier are a necessary mechanism to force the commit of memory access before next step. Specifically it is ensure that data has been moved from any cache / buffer to the destination, and blocks execution until that has been done.

- Some instruction cores have write buffers – the Cortex-M0 does not.

- The microcontroller may have a cache at system level (outside of instruction engine)
- There may be queue or buffer between the memory mapped peripheral (esp. external item on memory bus). *Note: the memory region often should also be marked as non-cacheable.*
- There may be a queue or buffer between the processor and the event bus.

Memory barriers should be employed

- Before wait for event/interrupt (sleep)
- In the construction of IPC mechanisms, e.g. mutexes and semaphores

The barrier CMSIS wrappers are:

```
_DAB()
_DSB()
_ISB()
```

44.1.6 Digital inputs and outputs

The majority of microcontrollers have "Input Data Registers" and "Output Data registers" per port. Save the data register, and the masks (for the relevant ones to access), and possibly any index substitution index from internal reference to the data register and pin.

No microcontroller I've seen has more than 32 pins per port; most keep to 16 or fewer.

44.1.7 Bitband

Cortex-M3 and above processors have bit-banding. This can be leveraged for simplifying IO. It can create a pointer to a single pin. For instance, for the chip select on I^2C or SPI communication. (Assuming that the hardware peripheral doesn't already handle the chip select).

44.1.8 Procedure blip

One useful technique is to have procedures raise a digital output line when they enter and lower it when they exit. This can used to

- Validate that key procedures execute when stimulated
- Measure the duration of interrupt or other procedure
- Check that procedures execution timing holds, even under load or high events
- To demonstrate the regularity of procedure execution
- To demonstrate regularity of events, such as CPU timers, and interrupt servicing.

The design is simple. Create a variable for each potential procedure of interest, defined like so:

```
uint32_t volatile* XYZ_blip= &XYZ_null;
```

In side of each procedure – called `XYZ_procedure()` here – have the following template:

```
void XYZ_procedure()
{
   XYZ_blip[0] = 1;
      … do work ….
   XYZ_blip[0] = 0;
}
```

When the procedure it will set the value at the destination of the pointer to 1, and when it exit it will set the value at the destination of the pointer to 0. The probe effect is minimal: the procedure executes the same code no matter what XYZ_blip points to. Both steps take only an instruction or two; there are no conditions, branches or other variations.

To cause the procedure to blip a digital output pin:

1. Ensure that the GPIO is configured to be an output
2. Set XYZ_blip to point to the bitband address for the pins bit in the digital output register of the target port.

Note: multiple procedures can drive the same output pin.

The disable the procedure blip:

1. Set XYZ_blip to point to XYZ_null. This way the procedure only stores to a dummy variable.

The execution time of the procedure is the same whether or not the probe is enabled, and the over head is negligible.

Note: as stated above, Cortex-M0 based (and non-Cortex) processors do not have banding. The above technique can be adapted in a straight forward manner to those processors.

44.1.9 Find-first set bit

Finding the first set bit in O(1) time is an important utility procedure. It is used to find, for example, the highest queued item in a bit list. Cortex-M3 and above include a "count left zeros" instruction which will tell one the highest bit set in a 32-bit word:

```
highest bit set = 32- clz(x)
```

However, the usual convention is that bit 0 is the highest priority and bit 32 is the lowest. This conventions allows working with longer bit queues by using a hierarchy. To find the right most bit set, one could (but should not do):

```
FFS(x) = 32- clz(x&(-x))
```

This takes several instructions. Finding the highest priority is often performed in a time critical procedure, such as PendSV exception handler to switch tasks. The next option is to employ the ARM "reverse bits" instruction:

```
arm_clz(arm_rbit(xx))
```

gcc:

```
__builtin_clz(__builtin_bswap32(x))
```

This is better, but still 1 instruction slower than need be.

The fix is to reverse the bits when they are set in the mask:

```
mask |= 0x80000000uL >>idx;
```

On the ARM that takes the same number of instructions as:

```
mask |= 1uL << idx;
```

44.1.10 Interrupt prioritization

The ARM Cortex-M microcontrollers have a prioritizable interrupt controller. Many processors can have as few as four levels of prioritization. Others can have a great range of

prioritization. The diagram gives some idea of how higher interrupts & exceptions can interrupt lower ones.

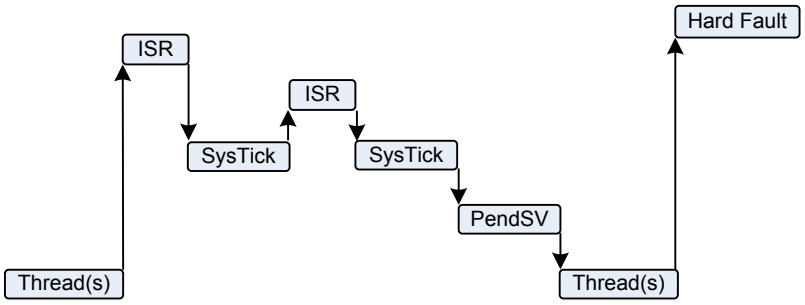

Figure 38: *Prioritized interrupts and exceptions*

The *Hard fault* exception (and other similar faults, such as NMI, etc.) is at the highest priority, and is fixed in the hardware. Interrupts cannot occur within these handlers. If this is invoked, the software (and/or hardware) has failed. The software design should place the hardware in safe state, but take no complex actions.

PendSV is an exception at the lowest priority, in that it is invoked infrequently – only when a thread, timer (in systick), and other IPC object in the interrupt changes the CPU's ready-to run list.

The *System Tick* is an exception that occurs regularly. It is (in this design guide) at a priority lower than all of the interrupts. This is done to service the interrupts with lower latency, preserving the quality of their function. It is the same priority (or higher than) PendSV's priority. If it were at a priority lower than PendSV, the regular switching of tasks would be a much higher cost.

The low priority of the system tick handler serves an integrity role: this is how interrupt overload is detected. A watchdog timer – such as the *windowed watchdog timer* on the STM32 product family – will be serviced (or partly serviced) in the system tick routine. If the system tick routine can't execute regularly either because of interrupt overload, or someone having disabled interrupts for too long, the watchdog servicing will be inhibited, triggering the microcontroller to reset, and proceed to the fail safe state.

Note: by partially serviced can mean that the watchdog timer in question can only by reset if the system tick has hand-shaked with some other thread. By requiring all the parties to handshake (or other demonstration of liveliness) the watchdog timer can detect failure to service those parties in a timely fashion.

45. TESTS

This section offers basic tests of the software units, starting with the most fundamental units. The test can check that the software module function as expected:

- Basic input or outputs
- Time based behaviour
- Basic function of the module
- Signal processing qualities

and can be employed as a hardware test:

- Signals stuck. e.g. stuck high or stuck low

- Signals shorted together
- Signals that are open

The digital input tests:

- Test 1: Test CPU input with a line high
- Test 2: Test CPU input with a line low

The digital output tests include:

- Test 3: Test CPU output with a line high
- Test 4: Test CPU output with a line low

The analog input tests include:

- Test 5: Test CPU input with a line high
- Test 6: Test CPU input with a line midrange
- Test 7: Test CPU input with a line low

This analog output tests include:

- Test 8: Test CPU output with a line high
- Test 9: Test CPU output with a line midrange
- Test 10: Test CPU output with a line low

This polynomial correction tests include:

- Test 11: Test CPU input with a line high
- Test 12: Test CPU input with a line midrange
- Test 13: Test CPU input with a line low

This IIR signal processing tests include:

- Test 14: Inject a stable signal
- Test 15: Inject a signal with a fast rising pulse
- Test 16: Inject a signal with a fast rising step
- Test 17: Inject a signal with a fast descending pulse
- Test 18: Inject a signal with a fast descending step

This debounce module tests include:

- Test 19: Check that a rising edge is passed thru
- Test 20: Check that a falling edge is passed thru
- Test 21: Check that rising-edge bounces are rejected
- Test 22: Check that falling-edge bounces are rejected

45.1. TEST 1: TEST CPU INPUT WITH A LINE HIGH

The basic test is:

1. Set the input high to the pin, using an external tool

2. Read the digital input (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.2. TEST 2: TEST CPU INPUT WITH A LINE LOW

The basic test is:

1. Set the input low to the pin, using an external tool
2. Read the digital input (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

It integrates several elements together that are, in other approaches, *separate* documentation efforts. The testing is often separate, later pass. This is included here for several reasons. Control flow errors: how did it get to the wrong spot? Bug in control flow implementation? Individual values right, but altogether not right. Wrong implementation of control flow.

45.3. TEST 3: TEST CPU OUTPUT WITH A LINE HIGH

The basic test is:

1. With the diagnostic tool, have the software set the output high
2. Using an external tool, read the digital pin. Confirm that it is high.

Stretch: This should be done with all output lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a software problem, or a hardware short.

45.4. TEST 4: TEST CPU OUTPUT WITH A LINE LOW

The basic test is:

1. With the diagnostic tool, have the software set the output low
2. Using an external tool, read the digital pin. Confirm that it is low.

Stretch: This should be done with all output lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a software problem, or a hardware short.

45.5. TEST 5: TEST CPU INPUT WITH A LINE HIGH

The basic test is:

1. Set the input high to the pin, using an external tool
2. Read the analog input (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.6. TEST 6: TEST CPU INPUT WITH A LINE MIDGRANGE

The basic test is:

1. Set the input to the mid range value, using an external tool
2. Read the analog input (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become low unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.7. TEST 7: TEST CPU INPUT WITH A LINE LOW

The basic test is:

1. Set the input low to the pin, using an external tool
2. Read the analog input (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.8. TEST 8: TEST CPU OUTPUT WITH A LINE HIGH

The basic test is:

1. Set the output high to the pin, using an external tool
2. Read the analog output (using external tool).

Stretch: This should be done with all output lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a software problem, or a hardware short.

45.9. TEST 9: TEST CPU OUTPUT WITH A LINE MIDRANGE

The basic test is:

1. Set the output midrange to the pin, using an external tool
2. Read the analog output (using an external tool).

Stretch: This should be done with all output lines. All lines should be read to check that their states are at the commanded level, ± a range. If a line is in the wrong state, this may indicate a software problem, or a hardware short.

45.10. TEST 10: TEST CPU OUTPUT WITH A LINE LOW

The basic test is:

1. Set the output low to the pin, using an external tool
2. Read the analog output (using an external tool).

Stretch: This should be done with all output lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a software problem, or a hardware short.

45.11. TEST 11: TEST CPU INPUT WITH A LINE HIGH

The basic test is:

1. Set the input high to the pin, using an external tool
2. Read the conversion results (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.12. TEST 12: TEST CPU INPUT WITH A LINE MIDGRANGE

The basic test is:

1. Set the input low to the mid range value, using an external tool
2. Read the conversion results (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become low unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.13. TEST 13: TEST CPU INPUT WITH A LINE LOW

The basic test is:

1. Set the input low to the pin, using an external tool
2. Read the conversion results (using diagnostic tool).

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.14. TEST 14: INJECT A STABLE SIGNAL

The basic test is:

1. Route the filter output to an analog output test point.
2. Using an external tool, set the input to the pin to a known, stable voltage
3. Read the filter results (using diagnostic tool). The reported voltage should match the injected voltage, after the input divider.
4. Check that the output signal is stable (i.e. no self-induced noise)

Stretch: This should be done with all input lines. All lines should be read to check that their states do not become high unexpectedly. If a line is in the wrong state, this may indicate a hardware short.

45.15. TEST 15: INJECT A SIGNAL WITH A FAST RISING PULSE

The basic test is:

1. Route the filter output to an analog output test point.

2. Using an external tool, set the input to the pin to a known, stable voltage
3. Read the filter results (using diagnostic tool). The reported voltage should match the injected voltage, after the input divider.
4. Check that the output signal is stable (i.e. no self-induced noise)
5. Inject a fast rising pulse, returning to the prior level
6. Read the filter results (using diagnostic tool). The reported voltage should match the prior injected voltage, after the input divider.
7. Check that the output signal is stable (i.e. no self-induced noise as a response), and that blip negligible pass-thru.

Stretch: This should be done with all input lines. All lines should be read to check that their states did not have an unexpected pulse. If a line did have a pulse, this may indicate a coupling or error with the filter implementation.

45.16. TEST 16: INJECT A SIGNAL WITH A FAST RISING STEP

The basic test is:

1. Route the filter output to an analog output test point.
2. Using an external tool, set the input to the pin to a known, stable voltage
3. Read the filter results (using diagnostic tool). The reported voltage should match the injected voltage, after the input divider.
4. Check that the output signal is stable (i.e. no self-induced noise)
5. Inject a fast rising step to a higher voltage level
6. Read the filter results (using diagnostic tool). Within TBD msecs, the reported voltage should match the new injected voltage, after the input divider.
7. Check that the output signal is stable (i.e. no self-induced noise as a response), and that blip negligible pass-thru.

Stretch: This should be done with all input lines. All lines should be read to check that their states did not have an unexpected pulse. If a line did have a pulse, this may indicate a coupling or error with the filter implementation.

45.17. TEST 17: INJECT A SIGNAL WITH A FAST DESCENDING PULSE

The basic test is:

1. Route the filter output to an analog output test point.
2. Using an external tool, set the input to the pin to a known, stable voltage
3. Read the filter results (using diagnostic tool). The reported voltage should match the injected voltage, after the input divider.
4. Check that the output signal is stable (i.e. no self-induced noise)
5. Inject a fast descending pulse, returning to the prior level
6. Read the filter results (using diagnostic tool). The reported voltage should match the prior injected voltage, after the input divider.

7. Check that the output signal is stable (i.e. no self-induced noise as a response), and that blip negligible pass-thru.

Stretch: This should be done with all input lines. All lines should be read to check that their states did not have an unexpected pulse. If a line did have a pulse, this may indicate a coupling or error with the filter implementation.

45.18. TEST 18: INJECT A SIGNAL WITH A FAST DESCENDING STEP

The basic test is:

1. Route the filter output to an analog output test point.
2. Using an external tool, set the input to the pin to a known, stable voltage
3. Read the filter results (using diagnostic tool). The reported voltage should match the injected voltage, after the input divider.
4. Check that the output signal is stable (i.e. no self-induced noise)
5. Inject a fast descending step to a lower voltage level
6. Read the filter results (using diagnostic tool). Within TBD msecs, the reported voltage should match the new injected voltage, after the input divider.
7. Check that the output signal is stable (i.e. no self-induced noise as a response), and that blip negligible pass-thru.

Stretch: This should be done with all input lines. All lines should be read to check that their states did not have an unexpected pulse. If a line did have a pulse, this may indicate a coupling or error with the filter implementation.

45.19. TEST 19: CHECK THAT RISING EDGE IS PASSED

The basic test is:

1. Set the input signal low
2. Check that the output signal is low
3. Set the input signal high
4. Check that the output signal is high within TBD ms.

45.20. TEST 20: CHECK THAT FALLINGING EDGE IS PASSED

The basic test is:

1. Set the input signal high
2. Wait TBD ms
3. Check that the output signal is high
4. Set the input signal low
5. Check that the output signal is low within TBD ms.

45.21. TEST 21: CHECK THAT RISING-EDGE BOUNCES ARE REJECTED

The basic test is:

1. Set the input signal low
2. Check that the output signal is low
3. Set the input signal high
4. Check that the output signal is high within TBD ms.
5. Set the input signal low
6. Check that the output signal is high
7. Raise signal within TBD ms
8. Check that the output signal is high

45.22. TEST 22: CHECK THAT FALLING-EDGE BOUNCES ARE REJECTED

The basic test is:

1. Set the input signal high
2. Check that the output signal is high
3. Set the input signal low
4. Check that the output signal is low within TBD ms.
5. Set the input signal high
6. Check that the output signal is low
7. Set the input signal low within TBD ms
8. Check that the output signal is low

45.23. NON-VOLATILE STORAGE TESTS

See Mikitjuk et al for a description of marching memory tests and what they diagnose

The Foo module includes a non-volatile storage (e.g. Flash) to retain the program, and non-volatile information. The non-volatile data storage can be confirmed to be functional with the conventional "marching" tests. To describe just one test, I will sketch the "walking ones" test below. The steps are:

1. In the storage, area set all bits, save one, to zero. The single bit should be set high.
2. Check the storage area contents match the expected value.
3. Repeat the above for each bit.
4. Repeat the above, but with the majority of bits set high, and the single bit set low.

This test checks that each bit in the storage area can hold clear and set values; that a bit does not clear or set other bits in the storage area. Note this is a test that the storage area works as intended, not that the access is done on a bit level.

The storage area is non-volatile – it retains the intended values after power has been removed from the system. To check this non-volatile property:

1. Setting the values in non-volatile area to known, but non-default values.

2. Remove power. The duration should be for a time longer than it takes internal power caps to deplete.
3. Applying power
4. Checking that the storage area holds the expected values.

45.24. FIRMWARE STORAGE

The program memory is integrated into the microcontroller. The software shall include features to test it. This may include ability to read program storage, and/or perform a CRC check on it.

45.25. INFORMATION STORAGE

The firmware shall include a means of setting, clearing, and reading the information storage. This includes at a low-level (accessing the Flash by address), and a service level (clearing and setting the values by communication).

46. REFERENCES AND RESOURCES

Warren, Henry S., *Hacker's Delight*, 2nd Edition, Addison-Wesley Professional; 2012 October

46.1. CORTEX MICROPROCESSOR RELATED RESOURCES

ARM, DDI0403, "*ARM v7-M Architecture Reference Manual*," Rev E.b, 2014-Dec

ARM, DDI0406, "*ARM Architecture Reference Manual*, *ARM v7-A and ARM v7-R edition*" Rev C.c, 2014-May

ARM, DDI0419, "*ARMv6-M Architecture Reference Manual*," Rev D 2017-May

ARM, DDI 0432C, *"Cortex-M0: Technical Reference Manual"* r0p0 Rev C 2009 Nov 30

ARM, DDI 0439B, *"Cortex-M4: Technical Reference Manual,"* Rev r0p0 2009-2010

ARM, DUI 0497A *"Cortex-M0 Devices: Generic User Guide"* Rev A 2009 Oct 8

ARM, QRC0011, *"ARMv6-M Instruction Set Quick Reference Guide,"* Rev L 2007 March

Keil, *"Using Cortex-M3 and Cortex-M4 Fault Exceptions,"* Application Note 209. 2010

CMSIS *"Cortex Microcontroller Software Interface Standard,"* Version: 1.10 - 24. 2009 Feb

Doulos, *"Getting started with CMSIS"* 2009

46.2. MEMORY PROTECTION

Atmel, *"AT02346: Using the MPU on Atmel Cortex-M3 / Cortex-M4 based Microcontrollers,"* 2013

ST Micro, DocID029037, "*AN4838 Managing memory protection unit (MPU) in STM32 MCUs*", Rev 1, 2016 Mar

CHAPTER 13

Design Review Checklists

This chapter provides checklists for use in reviewing the software designs (before implementation proceeds too far):

- Design review checklist
- Detail design review checklist

See also

- Appendix F for the *Code Complete* Design Review check lists
- Appendix G for a rubric to apply in the reviews

47. DESIGN REVIEW

A software design review is intended to answer a basic set of questions:

1. What the inputs and how does the design turn them into outputs?
2. What are the major elements that make up the system?
3. How do these elements work together to achieve the goals set out by the requirements?

A good design is:

- Simple
- Able to be constructed in a timely fashion
- Adaptable to other applications
- Dependable: no bugs, or unexplainable behaviour and can achieve long-lasting operation
- Efficient: applies its key resources to useful work (skillfully)

47.1. STARTING

- ☐ Are the requirements sufficiently defined to create the high-level design?
- ☐ Is the high-level design understandable?
- ☐ Are there terms or concepts introduced / defined before they are used?
- ☐ Are the requirements realizable?

47.2. MODULES AND FLOWS

- ☐ Are the main areas of functionality explained?
- ☐ Are the main inputs and outputs described?
- ☐ Are the main modules or components and their roles described?
- ☐ Is a structural diagram showing the flows given?
- ☐ Are the main signal chains and logic flows show and described?
- ☐ Are the roles of the signals and logic explained?
- ☐ Is the application logic discussed and outlined?
- ☐ Does it describe the approach to testing and diagnostics?
- ☐ Is the approach to power management outlined?
- ☐ Is data management outlined? Is the roughly what will be stored, whether it will be non-volatile discussed?
- ☐ Is the communication outlined?
- ☐ Is the safety model discussed? Timeouts? Watchdog timers?
- ☐ Is the approach to software configuration (of features, parameters, etc) discussed?
- ☐ Is the approach to other IO described?
- ☐ Are the module prefixes provided and described?
- ☐ Are the main file groupings provided and described?

47.3. NAMES

- ☐ Are the module names well chosen?
- ☐ Are the signals, and other object names well chosen? Are the names clear? Do the names convey their intent? Are they relevant to their functionality?
- ☐ Is the name format consistent?
- ☐ Names only employ alphanumeric and underscore characters?
- ☐ Are there typos in the names?

48. DETAILED DESIGN REVIEW CHECKLISTS

48.1. BASIC FUNCTIONALITY

- ☐ Does the detailed design match the overall design and requirements?
- ☐ Are the requirements sufficiently defined to create the detailed design?
- ☐ Is the high-level design sufficient and agreed upon to support the detailed design?
- ☐ Is all the detailed design easily understood? Is the design simple, obvious, and easy to review?
- ☐ Is the detailed design sufficiently detailed to create/update a work breakdown structure?
- ☐ …to create a schedule, down to half-day increments?
- ☐ Is the design sufficiently detailed to delegate work?

48.2. DOCUMENTATION

- ☐ Are all modules and interconnecting mechanisms documented?

- ☐ Do they properly describe the intent of the module?
- ☐ Is the modules interface (procedures, data structure, sequences) documented?
- ☐ Are all parameters of the procedures documented?
- ☐ Is the use and function of third-party libraries documented?
- ☐ Are data structures and units of measurement explained?

48.3. DIAGRAMS

- ☐ Are block diagrams employed?
- ☐ Are the boxes labeled with their designator?
- ☐ Are the boxes connected?
- ☐ Do the diagrams show the flow of signals and external control?
- ☐ Code complexity measure is low (below set threshold)?
- ☐ Is there sufficient annotation on the connection to understand how they communicate? Is this covered in the expository text?
- ☐ Are there sequence diagrams?
- ☐ Are there flow charts? Does the text in the diagrams match the terms used in the expository?

48.4. MAINTAINABILITY AND UNDERSTANDABILITY

- ☐ Is the design unnecessarily ornate or complex?
- ☐ Is the design appropriately modular? Would it be better with more modules? Fewer?
- ☐ Can any of the modules be replaced with library or built-in functions?
- ☐ Does the design have too many dependencies?
- ☐ Any changes that would improve readability, simplify structure, and utilize cleaner models?

48.5. NAMES & STYLE

- ☐ Are the module names well chosen? Are they relevant to their functionality?
- ☐ Are the signals, variables, and other object names well chosen? Are the names clear? Do the names convey their intent? Are they relevant to their functionality?
- ☐ Do the names of these objects use a good group / naming convention? e.g. related items should be grouped by name
- ☐ Is the name format consistent?
- ☐ Do the names only employ alphanumeric and underscore characters?
- ☐ Are there typos in the names?

48.6. PRIORITIZATION REVIEW CHECKLIST

- ☐ Are all threads identified? These should be in a table summarizing them.
- ☐ Are the resource protecting mutexes identified? These should be in a table summarizing them.
- ☐ Are all of the interrupts and their sources identified?

- ☐ Has a Rate Monotonic Analysis (RMA) and dead-line analysis been performed?
- ☐ Have the task/threads and mutexes been assigned priorities, based on this analysis?
- ☐ Have the interrupts been prioritized based on a similar analysis?
- ☐ Have the DMA channels been prioritized based on a similar analysis?
- ☐ Have the CAN message been prioritized based on a similar analysis?
- ☐ Does the ADC use prioritization? Have the ADC priorities been prioritized based on a similar analysis?
- ☐ Have the Bluetooth LE notification/indication priorities been prioritized based on a similar analysis?

48.7. CONCURRENCY REVIEW CHECKLIST

- ☐ Are the protect resources, and how they are protected listed?
- ☐ Are there resource missing protecting mutexes?
- ☐ Is the acquisition order of locks/mutexes defined?
- ☐ Are the appropriate IPC mechanisms specified?
- ☐ Is the order of multiple accesses defined?
- ☐ How do interrupts signal threads? Which threads do they signal?
- ☐ Are there ways to reduce the blocking time?

48.8. CRITICAL FUNCTION / SUPERVISOR REVIEW CHECKLIST

Check that critical functions (e.g. Class B and C of IEC 60730) are suitably crafted:

- ☐ Does the detailed design identify the critical functions?
- ☐ Are the critical functions limited to a small number of software modules?
- ☐ Is the relation between the input and output parameters simple? Or at least, simple as possible?
- ☐ Is a power supervisor / brown-out detect employed? Should one be?
- ☐ Are self-tests and/or function tests performed before any action that depends on the critical functions?
- ☐ Are periodic self-tests or functional tests performed? How do they work? Is a vendor supplied module performing the test? Which one(s)?
- ☐ Is there a defined acceptable state for when self-check (or other functions) fail?
- ☐ Are the clock(s) functionality and rates checked?
- ☐ Is a watchdog timer is employed? Correctly? Does the design only reset the watchdog after all protected software elements are shown to be live? An example of a bad design would be to reset the watchdog.
- ☐ Does the design describe where the watchdog timer may be disabled? Is this acceptable?
- ☐ Is an external watchdog employed? Is the external watchdog handshake done only after all of the software has checked liveliness? A bad approach is to use a PWM for the handshake, as a PWM can continue while software has locked up or is held in reset.
- ☐ Is there a fail-safe and fail-operational procedure defined to bring the product to the defined acceptable state? There should be a very small number of such procedures: only one or two.
- ☐ Is there acceptable handling of interrupt overload conditions?

- ☐ Is the critical program memory is protected from modification? How? Hardware? Software?
- ☐ Is the program memory checked for validity? How? CRC check? Hardware based? Software?
- ☐ Is the stack checked for overflow? How?
- ☐ Is the critical data separated, checked, and protected? How?
- ☐ Are independent checks / reciprocal comparisons to verify that data was exchanged correctly? How does it work? For example, how does it know that the correct device and correct address within the device was modified or read?
- ☐ Are there possible partition violations from data handling errors, control errors, timing errors, or other misuse of resources?

48.9. MEMORY HANDLING REVIEW CHECKLIST

Has the memory been partitioned in a manner suitable for Class B? i.e., does the software isolate and check the regions?

- ☐ Does the detailed design outline good practices to prevent buffer overflows – bound checking, avoid unsafe string operations?
- ☐ Are memory regions write protected?
- ☐ Is the memory protection unit enabled?
- ☐ What is the access control configuration?
- ☐ Is it appropriate?

Non-volatile storage:

- ☐ Does the design have a plan to not overwrite or erase the non-volatile data that is in use? Or does the design use a "replacement" strategy of writing the most recent/highest good-copy of the data?
- ☐ Does the design account for loss of power, reset, timeout, etc during read/write operation? This should include checking supply voltage before erasing/writing non-volatile memory, performs read back after write, and CRC data integrity checks
- ☐ Are data recovery methods used? Will the design work?
- ☐ Does the design ensure that the correct version of stored data will be employed (such as on restart)?

48.10. POWER MANAGEMENT REVIEW CHECKLIST

Power configuration for low power modes:

- ☐ Are power management goals defined?
- ☐ Are the target power performance characteristics/requirements defined?
- ☐ How will it enter the states?
- ☐ How will it exit the states?
- ☐ Are the states of clocks, IOs, and external peripherals defined for the low power states?
- ☐ Is there a race condition in going into low-power state and not being able to sleep or wake?

48.11. NUMERICAL PROCESSING REVIEW CHECKLIST

Check for correct specification of numerical operations, such as might be used in signal processing, kinematics, control loops, etc.:

- ☐ Is there a description of the numerical processing that will occur?
- ☐ Is the theory of operation (e.g. that forms the system of equations) sound?
- ☐ Is it numerically sound?
- ☐ Are the equations ill-conditioned?
- ☐ Is the method of calculation slow? Is the algorithm slow? Is floating point emulated on the target platform?
- ☐ Would use of fixed point be more appropriate?
- ☐ Is simple summation or Euler integration specified? This is most certainly lower quality than employing Simpsons rule, or Runge-Kutta.
- ☐ Floats and doubles are not used in interrupt handlers, fault handlers, or the kernel.
- ☐ The RTOS is configured to preserve the state of the floating point unit(s) on task switch.

48.12. SIGNAL PROCESSING REVIEW CHECKLIST

- ☐ Is the signal chain described?
- ☐ Is the relation between the input and output of the signal chain simple? Or at least, simple as possible?
- ☐ Is the sampling approach to linear signals (aka analog inputs) described?
- ☐ Is the description of sample acquisition time defined? Does it match with the hardware design description and target signal? (e.g. input impedance, signal characteristics)
- ☐ Is the method for acquiring samples appropriate? If the processing requires low jitter, the design should support this. For instance, a design that uses a DMA ring-buffer has low variation, while run-loop or interrupt trigger can have a great deal of time variation.
- ☐ Is oversampling applied? Is the design done in a proper way?
- ☐ Is simple summation or Euler integration specified? This is most certainly lower quality than employing Simpsons rule, or Runge-Kutta.
- ☐ Are appropriate forms of filter specified? Is an unstable form used? (Would the form have ringing, feedback, self-induced oscillation or other noise?)
- ☐ Is the signal processing unnecessarily complex?

48.13. TIMING REVIEW CHECKLIST

- ☐ Is the sequence of interactions documented?
- ☐ Is the timing of interactions documented? Are the timeouts defined and documented?
- ☐ From the time the trigger is made to the action, what worst case round-trip? Include interrupts, task switching, interrupts being disabled, etc. Is this timing acceptable?

48.14. TESTABILITY

- ☐ Is the design testable?

48.15. OTHER

- ☐ Are there regular checks of operating conditions? Should there be?

CHAPTER 14

Software Detailed Design Risk Analysis

This chapter provides an initial template for software detailed design risk analysis.

49. SOFTWARE DETAILED DESIGN RISK ANALYSIS

The outputs of a software detailed design risk analysis include:

- A table mapping the software requirements to the detailed design element (e.g. procedure) that addresses it. *This table may have been produced by another activity and is only referenced in the output.*
- List of software risks, acceptability level, and their disposition
- A criticality level for each hazard that can be affected by software
- Recommended changes to the software design, software architecture, software requirements specification, programmable system architecture, etc. For example, actions required of the software to prevent or mitigate the identified risks.
- Recommended test Verification & Validation activities, especially tests

The steps of a software detailed design risk analysis include:

1. Identify the design elements that address each requirement. *This may have been produced by another activity and is only referenced in the output.*
2. Examine the risks of errors with values
3. Examine the risks of message capacity
4. Examine the risks of timing issues
5. Examine the risks of software function
6. Recommendations for rework

49.1. STEP 1: IDENTIFY THE DESIGN ELEMENTS THAT ADDRESS EACH REQUIREMENT

Go thru each of the software requirements and list the design elements that address it.

49.2. STEP 2: EXAMINE ALL VALUES FOR ACCURACY

Identify all the elements of the design – sensor 1, sensor 2, actuator, motor, calculations, operator inputs, operator outputs, each parameter received, etc. For each of these elements, create a copy of **Table 4** ("*Value accuracy risks*") and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition.

49.3. STEP 3: EXAMINE THE MESSAGES CAPACITY

Identify all the messaging elements of the system – I^2C sensor, task 1, user input, etc. For each of these elements, create a copy of **Table 7** ("*Message capacity risks"*) and populate it with respect to the architecture. In reviewing each condition, identify the least acceptable risk for each applicable condition.

49.4. STEP 4: EXAMINE THE CONSEQUENCES OF TIMING ISSUES

Identify all the input elements of the system – button #1, frequency input, I2C sensor, task 1, user input, etc. This list should include elements those receive messages, and send messages. For each of these elements, create a copy of **Table 17** ("*Timing capacity risks*") and populate it with respect to the design. In reviewing each condition, identify the least acceptable risk for each applicable condition.

49.5. STEP 5: EXAMINE SOFTWARE FUNCTION

This step examines the ability of the software to carry out its functions.

Identify all the functions of the system; that is, the operations which must be carried out by the software. For each of these elements, create a copy of **Table 36** (below) and populate it with respect to the architecture. (Strike inapplicable conditions)

***Table 36:** Software function risks*

Condition	Hazard, likelihood & severity
Hardware or software failure is not reported to operator	
Data is passed to incorrect process	
Non-deterministic	
Non-terminating state	
Software fails to detect inappropriate operation action	

In reviewing each condition, identify the least acceptable risk for each applicable condition.

The risk analysis shall illustrate how events, or logical combinations of events, are capable of leading to an identified hazard

An analysis shall be conducted to identify states or transitions that are capable of resulting in a risk.

49.6. STEP 6: RECOMMENDATIONS FOR REWORK

Summarize each of the identified conditions with a risk level of "medium" or "high." These items mandate rework, further analysis, and/or Verification & Validation activities.

[This page is intentionally left blank for purposes of double-sided printing]

PART III

Source Code Craftsmanship

This part provides guides for source code workmanship

- OVERVIEW OF SOURCE CODE WORKMANSHIP.
- C CODING STYLE. This chapter outlines the style used for C source code.
- JAVA CODING STYLE. This chapter outlines the style used for Java source code.
- CODE INSPECTION & REVIEWS. Describes code reviews.
- CODE INSPECTION & REVIEWS CHECKLISTS. Provides checklists for reviewing source code.

[This page is intentionally left blank for purposes of double-sided printing]

CHAPTER 15

Overview of Source Code Workmanship

This part promotes good source code construction:

50. SOURCE CODE WORKMANSHIP

This part seeks to reduce bugs from language mistakes and mis-implementing the detailed design. It presents coding guides and review tools.

Source code should follow good practices. Some of these practices are covered in industry guides, such as MISRA C, and Lint. Chapters 16 and 17 give specific coding guidance that the industry guides do not cover. These guides provide direction to producing clear code, with a low barrier to understanding and analysis.

Chapter 18 discusses review the resulting source code against the guides and the detailed design to help ensure that the result has a good construction.

The source code should be reviewed (and otherwise inspected) against those guides, designs, and against workmanship evaluation guides. The purpose of reviewing the work is to examine quality of construction – it is not an evaluation of the engineers, and it is looking for more than finding defects. It is to get a second opinion on the implementation.

The review checklists & rubrics are a dual to the coding style; everything in one should be in the other.

50.1. WHAT DOES GOOD CODE LOOK LIKE?

Good code is

- *Well-structured*. It is consistent & neat, using accepted (or mandated) practices
- *Structured simply*. It uses simple operations, with one action per line. It makes minimal use of macros. It modularizes effectively. It limits a function to fit one screen of code.
- *Clean interfaces*. It passes minimal data, reducing memory requirements and increasing speed. It exposes only variables that are necessary: minimal use of global variables (IO port and configuration registers as well as variable use to communicate with ISRs are notable exceptions.) It minimizes dependencies and confines processor dependent code to specific functions.
- *Functional*. The code is simple, and compact. It has been tested frequently, completely, and thoroughly. It uses a layered approach to add the needed complexity.

50.2. THE ROLE OF REVIEWS AND INSPECTIONS

The purpose of reviewing the work is to examine quality of construction (the workmanship). (Note: it is not an evaluation of the engineers.) Code review is looking for more than finding defects. Reviews check that:

- The construction is consistent, and coherent
- That the style is easy to understand, and clear
- That the work is maintainable over time, by many people
- That it avoids known and potential defects
- Consistent execution
- Evaluate quality of construction
- Planning goals for schedule and quality
- Improve meeting quality goals

The reviews can also be used as an education for new team members.

Tools can be used to automate some of the checks, relieving some of the reviewer labor:

- LINT
- MISRA C checks
- Compiler tool warnings
- Klockwork Coverity
- TI Optimization assistant

CHAPTER 16

C Coding Style

This chapter describes the subset of C that we will use, and how to format the source code.

- Coding style overview
- The language used
- The source code file format
- Preferred types

51. CODING STYLE OVERVIEW

The goals of a coding style guide are to promote understandable source code that is

- Easier to maintain (than would be the case without such a guide),
- Reduces misinterpretations and
- Discourages the use of unreliable techniques.

To do so, the guidelines often emphasize:

- Readability of layout and comments so as to be clear about the source codes intent
- Decomposing the code so as to make it clear what is going on
- Being consistent, so that it is predictable about intent based on other analogs
- Value checking

51.1. SOFTWARE LANGUAGE

The software for applications created with this guide are written in the ANSI C99 programming language.

Compiler specifics should be used frugally. This includes Keil, IAR, MDK and GNU C extensions. Where possible, use the more portable forms. For instance, use the ARM C Specification forms for portability across

Tools should be used to check for possible misinterpretation of intent. This can include:

- LINT
- MISRA C checks
- Compiler tool warnings
- Klockwork Coverity

51.2. THE REMAINDER

The rest of this chapter will narrow the subset of C that we will use:

- Source code files – spaces (no tabs), file layout, and code must be well commented
- Code layout – braces, spacing,
- Preferred Types, esp. number types
- Control flow: conditionals, goto/label/return/break/continue
- Pointers

52. SOURCE CODE FILES

52.1. FILE NAMES

The file names should be prefixed with the modules prefix.

52.2. FILE GROUPINGS FOR A MODULES IMPLEMENTATION

A module has an .h that declares the procedures, variables, and macros that other modules may use. This file should not have 'internal' only information; that is information that other modules *should not use*. It may have many .c files that implement the module – it is better to break down a module into groups of relatively short files rather than one large file a thousand lines or longer. The module may have other .h files (suffixed as –int.h) that are for use only within the module.

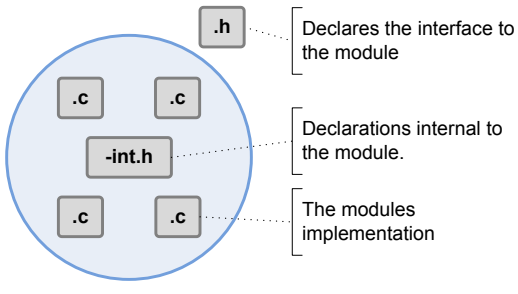

Figure 39: *How .h and .c files related to a module*

The documentation distinguished between external interface (procedure and variables other modules may use) and an internal one.

52.3. A BRIEF NOTE ON CHARACTER SETS

Tab characters shall not be used in software source code. Text editors have different tabbing configuration and this guide is independent of any particular text editor. Indents shall use spaces, not tab characters.

52.4. HEADER FILES

Header files describe the interface to the modules and the system.

Header files should not define variables or procedures; it should only declare them.

52.4.1 Guards

Header files (file ending with the extension .h) should have guard defines in the file, so the file's declarations/definitions are not made twice.

GOOD:

```
#ifndef MYSTUFF_H
#define MYSTUFF_H

…

#endif
```

RATIONALE. Some poorly planned header files create include cycles, meaning that they are forever including each other. Although it is better to resolve the include cycle and improve division of declarations to have acyclic dependencies, this [BANDAID] is used.

52.4.2 Extern declaration / procedure prototypes

Non-static extern declarations and procedure prototypes are to be done in header files. (Not in C files). There is to be only one declaration of a variable, macro, procedure or any other symbol.

52.4.3 Documented code

Header files describe the interface to the modules and the system. Thus, the header files must be documented completely – they should contain all the information a developer needs to understand and use the interface. Each declaration must have descriptive comments.

Procedure or function headers must contain:

- The procedure declaration,
- A description of the procedure including requirements for input parameters
- The specification of the return value, results, and output parameters.
- Changes to any global data.

52.5. C SOURCE FILES

Local data declarations must have comments.

A procedure should look like

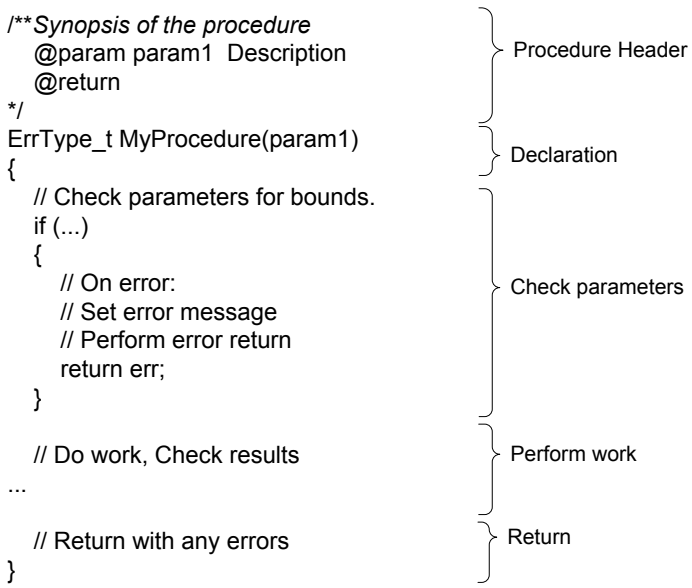

Figure 40: *Typical procedure template*

The elements include

1. Description (in comments)
2. Declaration, with parameter list. Must be declared in header file or top of current file. The header file declaration was discussed in a previous section
3. The procedure itself checks it's parameters
4. Performs work, and checks the error return of all calls
5. Returns with error code

Like the header file, each the procedure (or function) header must contain:

- The procedure declaration,
- A description of the procedure including requirements for input parameters
- The specification of the return value, results, and output parameters.
- Changes to any global data.
- Details of the implementation

Comments are required when the code is non-trivial. It is better to explain every line than to argue that the code is the documentation.

52.6. BRACE PLACEMENT

The diagram earlier showed the placement of braces and indentation style. Open braces and their closing brace are to be in the same column.

52.7. LONG LINES

When you split an expression into multiple lines, split it before an operator, not after one:

```
if (foo_this_condition && bar > win(x, y, z)
   && remaining_condition)
```

Try to avoid having two operators of different precedence at the same level of indentation. For example, don't write this:

```
mode = (inmode[j] == VOIDmode
      || GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])
      ? outmode[j] : inmode[j]);
```

Instead, use extra parentheses so that the indentation shows the nesting:

```
mode = ((inmode[j] == VOIDmode
      || (GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])))
      ? outmode[j] : inmode[j]);
```

53. PREFERRED TYPES

Use proper, standardized C99 types, rather made up or ad hoc type names.

53.1. BOOLEAN TYPES LARGER THAN 1 BIT ARE BAD

Do not use *BOOL*. It does not work the way you think. Use of BOOL may be mandated if an unchangeable 2nd or 3rd party code employs. If you do have to use a BOOL – never ever cast it. A Boolean value may *only* be checked against false.

RATIONALE: Only "false" has a defined value in C (0). Only expressions are true or false. When comparing a number as a Boolean, C casts the number to an expression and compares it against zero to determine if it is false (if zero) or true (not zero). The reverse produces a non-zero number for true expressions, and 0 for false ones. However, casting to a smaller type does not preserve the important "non zero" property. It only keeps the lower bits sufficient for the smaller type. For instance, $FF00_{16}$ when cast to 8 bits is 00_8. C would consider the former "true" and the later "false."

53.2. CHARACTERS AND STRINGS

`char` should only be used to represent characters, and nothing should be assumed about its sign. Characters might use `char` type, or a wider type, as appropriate.

Similarly, strings might use `char const*`, but strings might use a more specific type.

Text strings should be zero-terminated UTF-8 strings without embedded nulls. Unicode has many flaws.

53.3. NUMERICAL TYPES

Variables of a numerical type are to specify

- If they may be signed, or only unsigned;
- Their representation format; and
- Their size.

53.3.1 Integer numbers

Preferred number types. C99 style, with all lower cases, and '_t' suffix:

Size	Signed	Unsigned
8 bits	int8_t	uint8_t
16 bits	int16_t	uint16_t
32 bits	int32_t	uint32_t

Table 37: *The preferred C integer type for a given size*

Number literals are to use a suffix to match type.

Suffixes are uppercase.

Wrong:

```
int I;
for (I = 1; I < 32; I++)
{
…
}
```

Correct:

```
uint8_t I;
for (I = 1U; I < 32U; I++)
{
…
}
```

53.3.2 Floating point numbers

Floating-point representation is expected to be used in this project. The microcontroller has a standard implementation of IEEE floats. If you need to represent a wide dynamic range (and floating-point representation is chosen), use IEEE 754 floats.

Comparison of floating-point values should be used sparing, be treated with care, and regularly reviewed for correctness. Floating-point are not to be used where discrete values are needed.

- A float (or double) must not be used as a loop counter
- Exact comparisons – equality (==) and inequality (!=) – must not be used with floats and doubles.
- Comparisons are not transitive for floating-point values. "a <= b" and "b < a" can both be false.

Math operations (especially division) of non-zero numbers can create "NANs." The software design should have a structured approach to checking for NANS, Infinites, and Out of range values.

Floating point should be employed (directly or indirectly) during interrupts or exceptions. Regular review the compiler generated code to ensure that it has not employed floating points for intermediate calculation. Not all processors preserve the floating point (sub)processors state during these; or the processor may have been configured to not save the state. (Saving the state can increase interrupt latency). Clear rules should be stated and enforced on the configuration of the processor, the acceptable use of floating point and *when*.

See the appendix H for the limits of float precision.

53.4. POINTERS

Pointer should be to the specific type, if known. A generic type should be `void*`.

Pointers are to be pointers to a `const` target, except where they explicitly will change the referent.

53.5. QUALIFIERS

Internal functions and variables should be declared static.

Scope Qualifiers (e.g. `static`, `extern`), should always go before the thing they modify, not after.

53.5.1 The const qualifier

PRINCIPLE OF USE: The *const* qualifier should be used on all reader interfaces to data structures and variables. Only a single writer should have the non-const access.

WHY: This helps ensure that there is a single modifier of the shared memory. It helps catch programmer flubs, which (unintentionally) modify the memory.

PRINCIPLE OF USE: The *volatile* qualifier must be used to access anything modified in an ISR, or exception handler and another ISR, exception handler or main task. Some examples of where to use:

- When accessing CPU registers or peripheral registers
- Non-locals accessed in an ISR or SysTick handler.
- Variables/structures modified in a SysTick handler, accessed in the main execution

WHY: Without volatile, compiler has the option to delay, or reorder committing changes to the memory (or register). The programmer's mental model is that modification occurs right away. The compiler also has the option to reuse previously accessed values, rather than fetching an updated value from the underlying storage.

Note, for accessing things larger than a single atomic unit (e.g. in the ARM Cortex non-32bits aligned), further protection is needed.

The `const` qualifiers should always go after the thing they modify, not before.

53.5.2 The volatile qualifier

PRINCIPLE OF USE. Use it with anything that may be accessed in one context, and modified in another context. Contexts could be interrupt handlers, OS tasks, OS timer handlers, peripheral register access, multi CPU (or multi core) shared memory, etc.

Examples of where to use:

- When accessing CPU registers or peripheral registers
- Non-locals accessed in an ISR
- Variables/structures modified in one OS task (or thread), accessed in another
- Variables/structures modified in a OS timer handler, accessed in an OS task (or thread)

MOTIVATION. Without volatile the compiler has the option to delay, or reorder committing changes to the memory (or register). The programmer's mental model is that modification occurs right away. The compiler also has the option to reuse previously accessed values, rather than fetching an updated value from the underlying storage

EXAMPLES OF EFFECTS. The device works one without optimizations (or with a particular optimization setting), but with optimizations, it doesn't anymore. The following pseudo code as an example:

```
set GPIO pin high
wait 1uSec
set GPIO pin low
```

This code might not work to create a blip. The compiler might toss out all of the GPIO pin modifications, except the last one.

SEE ALSO: const, mutexes, and disabling interrupts

53.6. DATA BUFFERS AND CROSS CHECKS

53.6.1 Canary method (aka Red Zones)

Buffer over and under runs are very common form of software bug. To help detect these bugs, the software is to place a canary around each buffer or array:

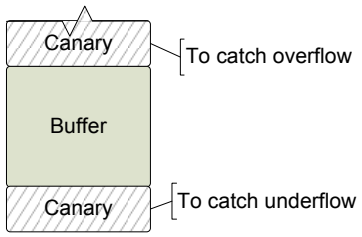

Figure 41: *Overview of buffers with canaries*

53.7. MULTIDIMENSIONALS ARRAY

Do not use C's multidimensional arrays.

RATIONALE. Dereferencing multidimensional is frequently (nearly universally) misunderstood… and incorrect. For instance,

```
int array[9][20];
```

produces 9 arrays of 20 integer arrays. Too often, it may be misunderstood to produce 20 arrays, each holding 9 integers.

54. MACROS

There are three parts:

- Macros that act as expressions,
- Macros that act as statements (or control flow).
- Macros parameters

54.1. MACROS THAT ACT AS EXPRESSIONS

PRINCIPLE OF USE: #define macros that are (or act as) an expression must be wrapped in parenthesis.

WHY: The macro expansion can have unintended effects

54.1.1 Examples of effects

The following provides an example of a bad case:

```
#define MyMacro(x)  1L + x

MyValue = 3L * MyMacro(v);
```

The above will expand to:

```
3L * 1L + v
```

Rather than the intended expansion of:

```
3L * ( 1L + v)
```

54.2. MACROS THAT ACT AS STATEMENTS (OR CONTROL FLOW)

PRINCIPLE OF USE: #define macros that use complex expressions – those with statements, if-then, whiles, etc – must be wrapped in do{}while(0)

WHY: The macro expansion can have unintended interactions with other control structures

54.2.1 Examples of effects

The following provides the first example a bad case:

```
#define BlipOn()    if (blipPtr)  *blipPtr = 1;
#define BlipOff()   if (blipPtr)  *blipPtr = 0;

if (myVar == 3)
  BlipOn();
else
  BlipOff();
```

Expands to the equivalent of

```
if (myVar == 3)
{
    if (blipPtr)
    {
       *blipPtr = 1;
    }
    else if (blipPtr)
    {
       *blipPtr = 0;
    }
}
```

Rather than the intended:

```
if (myVar == 3)
{
    if (blipPtr)
    {
      *blipPtr = 1;
    }
}
```

```
else
{
    if (blipPtr)
    {
      *blipPtr = 0;
    }
}
```

The following provides an example, where while's can interact inappropriately with the surrounding code:

```
#define WaitForSignalToGoLow()    while (*input1)
```

Used within code:

```
// Wait for signal #1 to go low and then set led on
WaitForSignalToGoLow();
*ledPtr = 1;
```

This becomes

```
while(*input1)
{
  *ledPtr =1;
}
```

Rather than the intended:

```
while(*input1)
{
}
*ledPtr =1;
```

54.2.2 How to fix these problems

The "better" is that the body of these kind of #define macros can be wrapped in do{...}while(0) statements. In the first example:

```
#define BlipOn()    do{if (blipPtr)  *blipPtr = 1;}while(0)
#define BlipOff()   do{if (blipPtr)  *blipPtr = 0;}while(0)
```

The example expands to the equivalent of

```
if (myVar == 3)
{
    do
    {
        if (blipPtr)  *blipPtr = 1;
    }while(0);
}
else
{
    do
    {
       if (blipPtr)  *blipPtr = 0;
    }while(0);
}
```

In the second example:

```
#define WaitForSignalToGoLow()    do{while (*input1) ;}while(0)
```

The example expands to:

```
do
{
```

```
    while(*input1);
} while(0);
*ledPtr =1;
```

54.2.3 Other comments

There are two other mitigations for the problems in the example code:

1. The body in if, else, while, for, do, etc. should be wrapped in {}.
2. Avoid using macros with statements, conditionals, loops, etc.

54.3. MACRO PARAMETERS

PRINCIPLE OF USE: The parameters to #define macros must be wrapped in parenthesis [within the macro body]

WHY: The macro expansion can have unintended effects

EXAMPLES OF WHERE TO USE. The following example shows how the parameters are wrapped in a parenthesis:

```
#define multipleAccumulation(m,b)          ((m)*3L + (b))
```

EXAMPLES OF EFFECTS. The following example shows how the parameters, when not wrapped in a parenthesis, can interact in unintended ways:

```
#define MyMacro(v)                  (v * 3)

local3 = MyMacro(local1 + local2);
```

This will expand to

```
local3 = local1 + local2 * 3;
```

Rather than the intended

```
local3 = (local1 + local2) * 3;
```

55. OPERATORS & MATH

55.1. THE PRECEDENCE OF C'S SHIFT OPERATORS

The C shift operators have a non-intuitive precedence. They should be used carefully:

1. Shift operations must be inside of a parenthesis – at least, if there are any operations to the left or right of it.
2. The left hand and right hand operands must be in parenthesis, if they are an expression. (That is to say, it must be "(4+2)" not "4+2".)
3. If a compiler has a flag to force precedence checking on >> as an error, it should be used;
4. If a compiler has a flag to report possible errors on >>, it should be used.

COMMENT: Lint and many compilers, like Microsoft C's compiler, do give a warning. The bad news is that the error messages are pretty hard to understand:

```
warning C4554: '>>' : check operator precedence for possible error; use parentheses
to clarify precedence
```

PRINCIPLE OF USE: The results of a computation should be as expected

EXAMPLES OF WHERE TO USE

Wrong:

```
unsigned A = 4 + 2 >> 1;
unsigned B = 2 + 1 << 1;
```

Correct:

```
unsigned A = 4 + (2 >> 1);
unsigned B = 2 + (1 << 1);
```

Correct:

```
unsigned A = (4 + 2) >> 1;
unsigned B = (2 + 1) << 1;
```

Note that this has a different result than the previous example of correct.

WHY & EXAMPLES OF EFFECTS. What are the computed values, for the C/C++ language, of A and B below?

```
unsigned A = 4 + 2 >> 1;
unsigned B = 2 + 1 << 1;
```

The answers are 3 and 6, respectively. Many programmers would expect 5 and 4. In other words, it is common to expect the shift operators to have more precedence than addition and subtraction, but less than multiplication and division.

55.2. COUNTABLE AND FLOATING POINT NUMBERS ARE NOT ASSOCIATIVE NOR DISTRIBUTIVE

One of the subtlest way to create bugs in embedded systems is with the math in C. Or, at least to assume that it is good enough, without considering how the compiler and hardware do math.

Arithmetic operators in C are *not* distributive. You need to know – and validate with – more information, such as the actual possible range of values in the variables. To wit, the countable numbers (int, short, unsigned and signed, etc) preserve the *least significant digits* under arithmetic operation. Worse – and something few people understand – is that floating point values (floats and doubles) only preserve the *most significant digits*.

55.3. THE OOPS OF INTS

In the integer family of types in C (and C-like) language values can silently overflow, leaving you with a surprisingly small number (even a very negative one when you expect otherwise). It helps if I give an example.

```
int X = (A-B) + (D - C);
```

is not always the same as:

```
int Y = (A + D)  - (B + C);
```

The sum of A and D could be large enough to overflow the integer (or whatever) type. The same for the sum of B and C. But – and the likely reason that they were written as two

subtractions before the addition – B might shrink A enough, and C might shrink D enough to not overflow. I've seen it a lot in small microcontrollers that are doing control loops.

Actually, there is subtle, but frequent bug. What happens is that both the `A+D` and the `B+C` overflow almost always at the same time, making for the difference to be pretty close. But there are a few cases where they don't. I haven't found a test engineer that can design cases to test this. Code review occasionally catches this. And nature usually triggers it only after the code deploys (ìshipsî), and the person likely to work on the bug has no idea which formula is correct for stability.

There isn't a simple solution. Often I just write down a derivative of the equations so that other people can check them. (I do make mistakes after all). And we try to use a wider type – more bits – than we think we'll need. And we double check. It helps.

55.4. THE OOPS OF FLOATING POINTS

Floating point preserves the most significant digits, dropping the least. That is its major appeal – it prevents the problems you see with the integer family above. (Well, float point can overflow too). Let's just assume that you have a DSP, microcontroller or processor where floating point is practical. There still is a class of bugs waiting to happen. (Fortunately is rare if you're just replace the equations you were using ints for earlier)

```
double X = (A+B) + (D + C);
```

is not always the same as:

```
double Y = (A + D) + (B + C);
```

When, say, A and B are small numbers, and C and D are big ones here is what happens. A plus D is D, because the digits of A are insignificant and dropped. And, similarly, the digits of B are insignificant and are dropped. But, A plus B does some up the digits, enough so that they do add with D and C, giving a different result.

The answer, for such simple cases, is to arrange the arithmetic operations from the smallest number to the biggest.

It all seems pretty trivial. Until you get into linear algebra, which is very heavily used in signal processing and control systems. In those systems, the pretty matrix operations we learn as sophomores is very unstable. Matrices get ridiculous numbers doing, say, an eigenvector. (By ridiculous, not only the computed results not work very well, they can have not-a-number results ñ singularities and infinites and such). One way to prevent this is to permute the matrix before performing the operation, like that sort from smallest to largest, and the rearrange back to the proper order when done.

And that is where I have to cop out. Numerical stability for these networks of multiplies and divides, sums and differences, is a specialty. How to permute and all the other things you need to do is something for good textbooks, and why you should use really, really good libraries.

55.4.1 Use of rational number forms

For a moment, back to microcontrollers where floating point is not a practical option. Even if you aren't going to be doing linear algebra. What then? Use rational numbers. Basically it is multiplying by a 100, to track the pennies in sales figures, and knowing that you are dealing

with currency in terms of pennies not dollars. (And you need to use a wide enough integer type). It is faster than floating point on some machines.

56. CONTROL FLOW, AVOIDING COMPLEXITY

SUMMARY: Prefer techniques that simplify control flow structure. Complex control structures tend to be harder to maintain, hard to evaluate for correctness, and more likely to have bugs.

56.1. BLOCK BODY

The flow control primitives if, else, while, for and do should be followed by a block, even if it is an empty block. For example:

Wrong:

```
while( /* do something */ )
  ;
```

Correct:

```
while( /* do something */ )
  {
  }
```

The block following a flow control primitive should always be bounded by brackets even it the block contains only one statement. For example:

Wrong:

```
if( isOpened() )
  foobar();
```

Correct:

```
if( isOpened() )
  {
  foobar();
  }
```

56.2. COMMA OPERATORS

Do not use the comma operator. (Exceptions may be made for very restricted use cases, and must be reviewed.)

56.3. CONDITIONS

Do not nest if-then statements more than 2-levels.

Do not nest “switch” blocks.

56.4. LOOPS

Things to avoid with loops (as they create complete control flow)

- Do not nest “for” loops more than 2-levels.
- Too many ‘continues’ or ‘break’ statements in for loop

56.4.1 Loop conditions

Where possible, the loop conditions other than index variable should be *const* variables.

Wrong:

```
for (Idx = 0; Idx < length-2; Idx++)
{
…
}
```

Correct:

```
int const End = length-2;
for (Idx = 0; Idx < End; Idx++)
{
…
}
```

RATIONALE. This creates smaller, faster code, which uses fewer memory accesses and reduces power consumption (in lower power designs).

The compiler may reload (and recalculate) the variables used in the comparison, even though they have not changed. The compiler has to be conservative and assume that the block (somehow) may affect the value, and so it must reload the variables with each comparison. The exception is if it can prove (via aggressive analysis) that the block will not modify it.

56.5. EARLY RETURNS

Better to have a clear procedure than to muddle it with nesting, convoluted control flow and return values that pass thru temporary variables.

Return errors and set the fault in the ISR or near the top of the main loop.

56.6. NO RECURSION / CALL LOOPS

Recursion – direct or indirect – is not allowed.

56.7. GOTOS

The `goto` statement is not allowed. However there may be instances where the use of a `goto` statement may actually make the source code more understandable and robust. The software engineer must document the use of the `goto` in the source code and must be prepared to defend his/her actions rigorously in software source code reviews.

57. PROCEDURE STRUCTURE

57.1. PARAMETER LISTS

Procedures with no parameters shall be declared with parameter type void.

RATIONALE: A procedure declared without a parameter list in C, does *not* mean no parameters are to be passed. It means that nothing was said about what the parameters may be. This is ambiguous.

57.2. DO NOT USE VARARGS

Procedures shall not use variable numbers of arguments, such as varargs.

RATIONALE: A variable number of arguments frequently introduce several kinds of bugs. A procedure may erroneously access more parameters than were passed. Or a procedure may erroneously use a different type of access than was used to pass it. There is no type checking on passing values.

57.3. DO NOT USE A STRUCT AS A PARAMETER VALUE TYPE

Do not use a struct as a value type for parameter. Use a const pointer instead.

RATIONALE: This copies the entire struct onto the stack to pass it.

57.4. DO NOT USE A STRUCT AS A RETURN VALUE TYPE

Do not use a struct as a value type for return.

RATIONALE: This copies the entire struct onto the stack to return it.

57.5. PARAMETER CHECKING

The input parameters should be checked for acceptable value ranges. This should prior to performing any other work.

57.6. RETURN VALUE CHECKING

Return values are to be checked. If (at this is unlikely) they are to be ignored, comment must explain why, and use a construction like

```
(void) funcCall(param1, etc); // Error doesn't matter in this case
```

RATIONALE: Return codes often include error indications or resource handles. Not checking the return values is a common source of software flaws, and incorrect error handling.

57.7. SIZE

Procedure should be small. Procedures should be small enough to fit comfortably on a screen.

RATIONALE: Big procedures are poor modularization, and undermine maintainability.

Longer procedures tend to have redundant code, something that rarely is a benefit.

57.8. INTERRUPT SERVICE ROUTINES

The interrupt service handler

- Prefix the interrupt handler with the _IRQ_ pseudo-qualifier
- The name ends with _IRQHandler (to match the CMSIS guidelines)

For convention, I am prefixing interrupt handlers in code (their declaration and definition) with the _IRQ_ pseudo-qualifier. It is defined as nothing in the Keil environment, and as an interrupt attribute in the GNU C environment.

An example

```
 IRQ  void fun IRQHandler()
{
```

Example 1: *IRQ handler*

```
        .. do stuff ...

    }
```

There are procedures that are effectively interrupt service routines. Use the following guidelines for these and interrupt service routines:

- Do very little in the interrupt service routine, only what is necessary. Push the rest of the work to the main application.
- Do not use unbounded loops in an interrupt service routine
- The volatile qualifier must be used to access anything modified in the ISR and another ISR, fault handler or main task.
- The interrupt service routine must not use mutexes or pend on IPC mechanisms.
- The interrupt service routine must not disable global interrupts.
- The interrupt service routine must not use floating point.

The ISR documentation should include:

- The function of the ISR. Common ones include:
    - GPIO rising/falling edge input
    - Compare / capture
    - ADC interrupts
- The work of the interrupt service routine, including its flow.
- The bounds of the ISR execution time.
- The work of the access procedure – the main procedure that receives the results of the interrupt. How does it check the values?
- When should the other contexts disable the routing?

57.9. EXCEPTION HANDLING ROUTINES

The exception handler – or microcontroller fault handler –

- The name ends with _Handler
- The volatile qualifier must be used to access anything modified in the exception handler and another ISR or main task.
- The handler must not use floating point.

Excepting PendSV, and SysTick the handler should

- Trigger a software breakpoint, to allow debugging
- Put the outputs into a safe state
- Reset the system

58. NAMING CONVENTIONS

- Each module is named. Stick to standard to acronyms and abbreviations for the modules identifier. See the table at the start of this document for recommended ones.

- Procedures are prefixed with their module identifier
- Variables
  - Parameter names begin with a lower case variable.
  - local variables – no special designation
  - module private variables – no special designation
  - global & module exported variables are prefixed with their module identifier
- type names end with '_t'
- tag names – no special designation
  - union – no special designation
  - structure – no special designation
  - enumeration – no special designation
- macro – no special designation

59. MATH, STRINGS, AND ASSEMBLY

59.1. FLOATING POINT ARITHMETIC PROHIBITIONS

Floating is not to be used in interrupt handlers, exception/fault handlers, or in the kernel.

RATIONALE: Many processors do not preserve the state of the floating point unit on interrupt or exception. Kernels– which are preferred to execute quickly – do not preserve the state of the floating point unit on entry to kernel space. (They do preserve it on context switch.)

59.2. HOW AND WHEN TO USE ASSEMBLY

In C, most of the math operations, such as `fabs()`, are procedures. This is done because a standardization document says it should be this way, and to (presumably) make it possible to refer the math procedure with a pointer. Many compilers include a technique to automatically inline a procedure when possible, but defer to an external procedure if such a pointer is necessary. Although math procedures are not commonly provided in this manner – at least in the standard libraries – replacement functions can be made to do this. Macros can be employed as well.

Using assembly is inherently processor specific, so it should only be created in important blocks. The assembly must be rigorously tested against a set of known values at critical points. The blocks that use these optimizations must be similarly tested.

59.3. STRING PROCEDURES NOT TO USE

Do not use `scanf()`,`sprintf()`, `strcat()`, `strncat()`, `strcpy()` or `strncpy()`.

59.4. ASCIIZ STRING COPIES

If the length of the string is known, do not use `strlcat()`, `strlcpy()` or similar procedures. Use the memory copy procedures instead.

60. MICROCONTROLLER SPECIFIC GUIDELINES

60.1. CORTEX-M FAMILY OF MICROCONTROLLER GUIDELINES

60.1.1 Do not use floats on Cortex-M0 and Cortex-M3

The ARM Cortex-M0 thru Cortex-M3 do not support floating point (floats and doubles0. If used, they have to be emulated in software, which is slow.

60.2. MICROCHIP PIC MICROCONTROLLER GUIDELINES

60.2.1 Using the "sleep" instruction

The "`sleep`" construction should almost always be:

1. The "`sleep`" instruction
2. A "`nop`" instruction (this is executed before any interrupt)
3. A conditional check – with a branch back to step 1.
4. A call for the CPU initialization

60.2.2 Use of multiplication and division

Microchip PIC microcontrollers do not include a multiplication or division unit. The compilers are quite good, especially if only one of the terms in the multiplication is a variable. Under some circumstances, the compiler is also able to analyze the code and translate a formula of two or more variables into a small set of formulas of a single independent variable. It is best to for the programmer to do this manually.

When the above technique cannot be applied, and the variables can be large in value, it may be better to convert the value to a logarithmic form, do the operation as arithmetic, and exponentiate the value back.

60.2.3 Interrupt Time and normal time

A procedure must not be called both at interrupt time and during "normal" execution of MCU. Procedures store their calling parameters and local variables at a fixed location.[14] If a procedure was to be interrupted during its execution and then called by the interrupt service routine the second call may scramble its parameters and local variables. Although there are some ways to make a procedure "reentrant", it is best to avoid mixing calling context for a subroutine.

Occasionally merely avoiding the re-entrancy is not sufficient. Depending on the circumstances, this may involve:

- Making *two* versions of the same subroutine, one suitable for being called at interrupt time, and one for normal time.
- Eliminating `switch()`'s or a procedure, opting for an array lookup.

[14] The linker works hard to identify which procedures are used (or might be used) at the same time. Only if two procedures are not used at the same time can their parameters and local variables reuse the same memory.

60.2.4 Use of Arrays instead of Switches or Pure Functions

With the Microchip MCU's it is sometimes better to transform `switch()` statements and many pure functions into a pre-computed table. Arrays have two advantages: first they are near constant time (the upper bounds is often very near the lower bounds), and reduce issues related to re-entrancy (see previous section) resulting from temporary variable allocation.

Try to avoid switches and functions of constructions like:

```
switch(x)
{
  case  0: return  3;
  case  1: return  7;
  default: return  9;
  case  3: return 13:
   …
}
```

Instead employ something like:

```
const int _Ary[]={3,7,9,13};
#define MySwitch(x) (x<0?9 : x>3?9 : _Ary[X])
```

61. REFERENCES AND RESOURCES

Barr, Michael, *How to use the volatile keyword*
http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword

Barr, Michael, *Coding standard rule for use of volatile*
http://embeddedgurus.com/barr-code/2009/03/coding-standard-rule-4-use-volatile-whenever-possible/

Boswell, Dustin; Trevor Foucher, "*The Art of Readable Code,"* O'Reilly Media, Inc. 2012

Ellemtel Telecommunication Systems Laboratories, *"Programming in C++ Rules and Recommendations"*, Document: M 90 0118 Uen, Rev. C, 1992-April 27
http://www.doc.ic.ac.uk/lab/cplus/c++.rules/

Exida Consulting, "*C/C++ Coding Standard Recommendations for IEC 61508"* V1 R2 2011 Feb 23, http://exida.com/images/uploads/exida_C_C++_Coding_Standard_-_IEC61508.pdf

Gimpel Software *"Reference Manual for PC-lint/FlexeLint, A Diagnostic Facility for C and C++"*, Rev. 9.00, 2009

IAR *"C-STAT Static Analysis Guide"*, 2015

IAR *"MISRA C:1998 Reference Guide"*, 2011 January

IAR *"MISRA C:2004 Reference Guide"*, 2011 January

Labrosse, Jean *MicroC/OS-II: The Real-Time Kernel,* 2nd Ed, CMP Books, 2002

This includes a chapter on the coding style guide employed in the RTOS.

Lockheed Martin Corporation, "*Joint Strike Fighter, Air Vehicle, C++ Coding Standards*", Document: 2RDU000001 Rev. C, 2005 December

Microsoft, *Secure Coding Guidelines*
https://docs.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines

MISRA Limited, *"MISRA-C: 2004, Guidelines for the use of the C language in critical systems"* 2004

NASA; Steven Hughes, Linda Jun, Wendy Shoan, *"C++ Coding Standards and Style Guide",* 2005
https://ntrs.nasa.gov/search.jsp?R=20080039927

Seebach, Peter *"Everything you ever wanted to know about C types"* 2006
Part 1: http://www.ibm.com/developerworks/library/pa-ctypes1/
Part 2: http://www.ibm.com/developerworks/power/library/pa-ctypes2/index.html
Part 3: http://www.ibm.com/developerworks/power/library/pa-ctypes3/index.html
Part 4: http://www.ibm.com/developerworks/power/library/pa-ctypes4/index.html

SEI CERT, *C Coding Standard*
https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

Turner, Jason; *C++ best practices*
https://github.com/lefticus/cppbestpractices/blob/master/00-Table_of_Contents.md

---

"The compiler doesn't know whether you know what's good for you."
– Raymond Chen

CHAPTER 17

Java Coding Style Guide

This chapter is a Java specific guide on coding style. This guide describes:

- Suggestions on iterating over `collections'
- Improvements on handling synchronization
- How to improve GUI response time
- How to manage constants

62. BASICS

62.1. SUGGESTION ON LOOP ITERATION

The following two loops will exhibit different performance, especially as the arrays get longer

***Example 1**: Two different loops*

```
for (int i = 0; i < MyArray.size(); i++);

int L = MyArray.size();
for (int i = 0; i < L; i++);
```

The second form of the loop will perform better since the costs of the size() method invocation will not be done L times; this is also known as elimination of loop invariants. (Note: size() has different overhead for different collections; some collections do synchronization or scanning during invocation).

63. LOCKS AND SYNCHRONIZATION

63.1. AVOID SYNCHRONIZED THREAD RUN() METHODS.

The following code will not work as expected; that is another thread cannot tell the thread below to stop:

***Example 2**: Thread stopping problems*

```
private boolean iv_runnable=true;
public void synchronized setRunnable (boolean runnable)
{
  iv_runnable = runnable;
}

public void synchronize run()
{
  while (iv_runnable)
  {
```

```
    ....
  }
}
```

Discussion: The thread (I'll call it A) acquires its own mutex; if any second wants to tell A to stop, it must acquire the mutex via setRunnable(). Since A is already locked, the second thread can never acquire the mutex, never modify the instance variable.

63.2. CONSIDER THE VOLATILE KEYWORD FOR FLAGS READ IN ONE THREAD AND SET IN ANOTHER.

The following code may not always work as expected; that is another thread setting iv_runnable to false may not cause the thread to stop:

***Example 3**: volatile keyword*

```
public boolean iv_runnable=true;
...
public void run()
{
  while (iv_runnable);
}
```

This one is hard to track down in debugging. What can happen is that the while() loop gets the value once for `iv_runnable`, and any updates don't check field. This is not a violation of the language in any way. If the while loop is sufficiently complex, the compiler tends not to cache the value for iv_runnable, and fetches the value. One solution is to apply the 'volatile' keyword to iv_runnable; the second is to use lots of synchronize calls. The former in some cases is much faster. The second option can be slower if there is any lock contention. Sun's own recommendation for the volatile keyword is:

> "The volatile modifier requests the Java VM to always access the shared copy of the variable so the its most current value is always read. If two or more threads access a member variable, AND one or more threads might change that variable's value, AND ALL of the threads do not use synchronization (methods or blocks) to read and/or write the value, then that member variable must be declared volatile to ensure all threads see the changed value"

63.3. RECOMMENDATIONS ON AVOIDING CERTAIN SYNCHRONIZE CONSTRUCTS:

Example: The following probably does not provide the safety expected.

```
public ArrayList getMyArray()
{
   synchronized (iv_myarray)
   {
     return iv_myarray;
   }
}
```

I'll ignore the obvious potential that iv_myarray could be null. There are two probable intentions of the above code. The first intention is to return a valid array reference for the iv_myarray field – even another thread assigns a different array to iv_myarray during the call. However, the object is not being locked to prevent this inconsistency. The better way to achieve this objective is:

```
public synchronized ArrayList getMyArray()
{
  return iv_myarray;
}
```

The second intention may be to provide the caller with the array, but not have it change while the caller uses it. In this case, the goal may not be totally achievable. At best, the caller should synchronize on the returned array, accepting that the array may have been altered just before the synchronization succeeded.

There is a similar problem with the following idioms:

```
synchronized (iv_myarray)
{
  iv_myarray = new ArrayList();
}
```

This should probably be:

```
synchronized (this)
{
  iv_myarray = new ArrayList();
}
```

In the worst case, synchronizing on the instance variables instead of the object can cause dead lock. For example, the following fragment would eventually deadlock:

Thread1:

```
public Object getObj1()
{
  synchronized (iv_obj1)
  {
    return iv_obj1;
  }
}

public void run()
{
  ...
  synchronized (iv_obj1)
  {
    iv_obj1 = Thread2.getObj2();
  }
  ...
}
```

Thread2:

```
public Object getObj2()
{
  synchronized (iv_obj2)
  {
    return iv_obj2;
  }
}

public void run()
{
  ...
  synchronized (iv_obj2)
  {
    synchronized (iv_obj2)
    {
      iv_obj2 = Thread2.getObj1();
    }
    ...
}
```

One way the deadlock would happen is that

1. Thread 1 locks iv_obj1 (in the run())
2. Thread 2 locks iv_obj2 (in the run())
3. Thread 1 calls getObj2(), which blocks waiting to get the mutex for iv_obj2
4. Thread 2 calls getObj1(), which blocks waiting to get the mutex for iv_obj1.

64. TYPE CONVERSION

The JDK is very inefficient at type conversion, creating a large number of temporary objects. In the `normal' case, that is not a problem. The problem is where there is a large number of conversions in a time sensitive matter. I'll explain my skepticism about the JDK's behaviour below, and why I created custom int/long/double/float to string conversion routines.

64.1. WORST CASE AREAS

The worst-case behaviour will always be in big tables. The Log table is pretty good example. A log table may easily have 25000 rows * 4 columns, or about 10^5 cells. Scrolling up once, the getValueAt() will be called 10^5*Number of Rows on screen times. This is means about 10^7 times. (Note: right now the log table is limited to a few thousand rows because it is slow for several reasons, some of which have been addressed, but not committed). The getValueAt() will typically execute the following code path:

```
JLabel tmp = new JLabel();
tmp.setText("");
tmp.setText(""+someIntValue+"/"+someIntValue);
```

This is very, *very* slow. First, the JLabel is created (and everything else it creates). And the string concatenation creates many temporary objects. And the intValue.toString() method creates half a dozen temporary objects. (floatValue.toString() and doubleValue.toString() are much worse, by about an order of magnitude). In the end getValueAt() typically creates 10 to 20 temporary objects. I'll skip for now the JLabel – someone else can bring the issue up in more detail about what has been looked at regarding the JLabel.

In a case where the implicit string constructor was used, it was used for only ONE column, so scrolling up the table created 10^6 temporary objects, each object being an average of 55 bytes in length. The performance penalties from the new() call alone are bad. Worse, the garbage collector had a huge burden, because it had to garbage collect 1 million objects, a total of several megabytes. Even worse, the garbage collector is triggered multiple times during the rendering, causing rendering slow triggered multiple times during the rendering, causing rendering slowdowns.

Other common scalar convections is roughly of the form

```
String.valueOf(intValue)
""+intValue;
```

64.2. MEASURING THE COST OF CHURN

It is easy to measure the cost of object churn. You will need to get the jar file, and

1. use a command line like:

```
java  -cp MyJar.jar -Xms1m -Xmx4m -verbose:gc com.MyClass Arg1 ... Argn
```

or

```
java  -cp MyJar.jar -Xms1m -Xmx4m -verbose:gc com.MyClass Arg1 … Argn
```

2. Then go to a series of screens under test, and run the scroll bar up and down. Resize the window several times (use the grabber on the bottom)
3. Note the CPU time, in seconds, that the application used, and quit the application.
4. The console window will have a long list of Garbage Collector times. (The right most column). Sum these times up.
5. Calculate the percentage of time wasted in the garbage collector = 100*(GCTime/CPUTime). It should be less than 10%. If it is more than 15%, it is a serious problem.

Of course, you need to spend a lot of time doing #2. This methodology does have some measurement limits, too. If someone wants to *cheat* to get a particular target number in step 5, here's how:

To make the application & garbage collector look evil, skip step 2.

The only garbage collection times you'll see are the costs associated with load java, AWT, SWING, etc. These costs are huge and one time, not the recurring costs.

To make the application look golden, change -Xmx4m to -Xmx800m or some such huge number. Skipping step 2 is optional. The -Xmx option manually controls how often the garbage collection runs (usually). By setting it so large, the application is allowed to create a huge amount of temporary objects, but never report it to you, since the garbage collector is disabled.

Don't get tempted to set -Xmx to too small a number. The application will stop running since there is a certain (large) amount of memory the application just needs to run. This number can only really be reduced thru larger design changes, which are another topic.

64.3. REPLACEMENT METHODS

int2String (and their ilk) are intended to be much more efficient. It creates one object per call, the resulting string. Its main drawback is that it employs a single shared preallocated character buffer, that has to be synchronized. In this case, nearly all of the calls happen in a single thread, so the synchronization time bounds to an uncontested mutex acquisition (a few multiples of an integer op on the 1.4 JVM). If this character changes, then the synchronization should be struck, and the buffers should be dynamically created.

One other peculiarity about their design – the JDK seems to not know ahead of time how long the string will be before it converts the scalar. This causes it to use many concatenations (and hence the temporary objects). int2String uses the log method of determining the length (along with a sign check).

For an alternate implementation using StringBuffers, and various other performance comparisons, anyone interested might like to see pages 135 to 150 of [Shirazi 2003].

*Shirazi, Jack "*Java Performance Tuning,*" 2nd Ed, O'Reilly 2003*

65. GUI RELATED CODE

Below are some suggestions to make a GUI seem more responsive; they aren't necessarily applicable to anything else.

The AWT/Swing thread runs a loop pretty much like

1. Check the Event Queue, and process those events
2. Check the repaint list, and call `paint()` for those items
3. Check the timer queue, and `SwingWorker` queues; put those items into the Event Queue

In order for a GUI to be fast, any Event Listener, Timer, or Repaint helper should be very very fast, in all cases. (Conversely, if they are not, the GUI may appear to be slow, unresponsive, or stutter).

The AWT/Swing Run-loop may call any of the following methods, so they should be as fast as possible:

```
actionPerformed
changedUpdate
getTableCellRendererComponent
getValueAt
getColumnName
getColumnClass
itemStateChanged
keyPressed
keyReleased
keyTyped
mouseClicked
mouseEntered
mouseExited
mousePressed
mouseReleased
paintComponent
removeUpdate
treeExpanded
treeCollapsed
valueChanged
windowActivated
windowClosed
windowClosing
windowDeactivated
windowDeiconified
windowIconified
windowOpened
insertUpdate
```

65.1. IMPLEMENTATION OF THESE METHODS

By fast, each implementation of these methods:

1. *Must not* call anything that blocks (or call anything that calls anything that blocks, etc.). That is, they must not call:
   - Any synchronized methods
   - Any `System.out.` methods
   - Throw an exception
   - Any IO writes or reads
   - Any `while(true)` loops
   - Any RMI function
   - Any JMS function.

If a method blocks – perhaps because it is waiting on RMI, or waiting on synchronization blockage (which happen even with a `new') – the whole display loops grinds to a halt.

2. *Must not* call anything that is not O(1) time, include binary search, non-small loops, or get() on JDK's hash maps or hash tables. Instead use arrays, switch statements, and direct field access.

3. Avoid creating temporary objects in these routines. It may be more responsive for the critical methods to access arrays or fields. The objects populating these arrays or fields can be created at a more 'idle' time when temporary objects don't impact the display loop. Earlier we discussed methods for reducing temporary objects, and how to measure their impact.

4. Avoid calling repaint() for a GUI component unless its value has actually changed. For example, the temperature gauge can be modified so that the only call path to repaint() looks like:

```
public void setCurrentTemp(int X)
{
  if (iv_enabled && X==currentTemp) return;
  currentTemp=X;
  repaint();
}
```

5. Avoid calling fireTableDataChanged() if the fireTableCellUpdated() can be called instead.

6. Like 4, avoid calling fireTableCellUpdate() unless the value of the cell has changed:

```
if (IOs[row] != newValeForRow)
 {
   Ios      [row] = newValueForRow;
   IOsString[row] = int2String(newValueForRow);
   fireTableCellUpdated(row, col);
 }
```

7. Consider a hysteresis timer. If it is likely that a large number of fireTableDataChanged() calls will be issued, consider starting a timer, and have it call the fireTabledDataChanged() after (say) 100 ms. In the code where fireTableDataChanged() would normally be placed, insert a check for the timer, and start one if it is not already started. This reduces the load on the repaint() loop, and reduces some of the flickering the user would otherwise see. For example, when pulling down several 10,000's logs in an event driven GUI, each received log could trigger a fireTableDataChanged(), but that would make the GUI slow. Instead, if a single-shot timer doesn't exist (or expired), one is created to call fireTableDataChanged() several hundred ms from now.

8. Consider reusing objects. To use a more common example, the getValueAt() will typically execute the following code path:

```
JLabel tmp = new JLabel();
tmp.setText(someValue);
```

This is slow (very slow if there is lots of string concatenation and temporary objects to create someValue). The JLabel is created (and everything else it creates). It is then return to the caller, which uses it just once to paint the value on the screen, and then dereferences it. Of

course, this is not true of all AWT/Swing callers, but it is true for many tables. It can be better to create one `JLabel()` for the Table Model instance, and reset its contents each time `getValueAt()` is called. (The contents have to be reset so that some table cells don't inherit bogus text, icons, colors, etc. from other cells). This reduces the number of temporary objects by the number of cells displayed in the table, since a `JLabel` would be otherwise created once every time `getValueAt()` is called, and `getValueAt()` can be called for every displayable cell (and is when the user is scrolling).

CHAPTER 18

Code Inspections and Reviews

This chapter discusses checking the source code for good workmanship, thru inspections & reviews:

- When a review should occur
- Who should review
- How to inspect and review
- What to report, outcomes

Note: There is no universally accepted and adopted approach to peer review. Each work enviornment has its own norms for peer review. These are checklists and templates that I have constructed over years. (I've found little available elsewhere.)

66. WHEN TO REVIEW

A review might occur when

- There are proposed changes to a stable codebase,
- When closing out a bug
- When a project reaches a control gate

67. WHO SHOULD REVIEW

What kind of person should participate in a review?

- The reviewers should have experience with the class of hardware being used. In typical embedded development today, they should be experienced with 32-bit embedded software, and Cortex-M microcontrollers.
- In some cases, the reviews will require someone with experience in the particular microcontroller family.
- Reviewers should have a lot experience with the way software, microcontrollers, and hardware can go wrong.
- Some of the reviewers should be independent; they should not be working on this artifact.
- The owner of the subsystem or other area of code

Others may participate in a review, of course, but they are optional. This includes:

- The author need not attend, as the code should stand on its own
- People interested
- People being brought into the team
- People with little experience in this area of engineering.

The later are not expected to contribute specific technical comments, but they may learn the system, the performance of reviews, and provide feedback on the understandability & maintainability of this foreign code.

68. HOW TO INSPECT AND REVIEW CODE

How can reviews be performed? One may apply any of the well-documented review and inspection techniques that can be found in the references. Common review methods are:

- The reviewers can meet and perform a formal inspection: e.g. with presentation, roles, and sign-offs.
- Some reviews can be reviewed at each person's desk. Often a tool such as CodeCollaborator (https://smartbear.com/product/collaborator/overview/) is helpful

This applies to general reviews, as well as specialized inspections.

- General reviews emphasize the workmanship of the code – maintainability (is it clear enough for others to work on in the future), basic quality-of-construction, and appropriateness.
- Specialized inspections are used to focus attention on specific areas that may be esoteric or require particular technical skill to judge.

The reviews take, as inputs:

- Style and other workmanship guides,
- Evaluation guides and rubrics
- The top level and detailed designs
- Supporting data sheets, application notes, vendor documentation

The reviewers should be provided a summary of areas to look at. The reviewers would examine these areas (and inputs), looking for such things as defects that can create bugs, or constructions that can be difficult to maintain.

68.1. SPECIALIZED INSPECTIONS

Specialized inspections are used to focus attention and effort. These delve into key areas and slices of code to answer narrow questions. Typical questions may be:

- Is the processor set up properly – are the clocks / oscillators turned on properly, etc?
- Are the watchdog timers (or similar protective timers) set up properly and detect enough unresponsiveness in the code?
- Is the source code that is very intimately coupled with microcontroller / hardware specifics done correctly?

- Do the critical and supervisory sections of software only perform their intended functions and do not result in a risk?
- Consistency in the data and control flows across interfaces.
- Correctness and completeness with respect to the safety requirements
- Coverage of each branching condition and function evaluation that addresses and remediates risks associated with abnormal operations, or involves a risk associated with its normal operation

See also

- Chapter 19 checklists
- Appendix I for the *Code Complete* Code Review check lists,
- Appendix J for a rubric to apply in the reviews
- Appendix D for Bug classification

69. THE OUTCOMES OF A CODE REVIEW

Reviewers comment on the aspect of the code quality:

- Detailed design
- Functionality
- Complexity
- Testing
- Naming
- Comment Quality
- Coding style
- Maintainability
- Understanding/comprehension.

The results of a review ideally should:

- Be actionable and easy to fix
- Produce few false positives
- Emphasize / focus on where there can be improvements with significant impact on code quality.

The results of a review might be realized one or more of the following ways:

- Gathering the results in a document (or spreadsheet) in a tabular fashion
- Annotate the source code, e.g. using a tool such as Code Collaborator
- Fill out bug reports
- Provide written feedback

69.1. A TIP ON FEEDBACK

When you are providing feedback, consider:

- Should it be said? Is the comment necessary, kind, true and helpful?

- Does it have the right emphasis? The emphasis of the feedback (especially critiques) should be proportionate. Scale using a rubric; some are included in Appendix F and Appendix I.
- How should the comment be said? Specific, actionable, measureable or distinct (that it has an effect when performed; can tell that it was done), timely (can be done immediately, or has time bounds)
- What is the person try to accomplish? {with the thing they are getting feedback on?}

69.2. REWORK CODE AFTER A REVIEW

The rework, in most cases, can be done by a second person or the primary developer.

70. REFERENCES AND RESOURCES

IEEE Std 1028-2008 - *IEEE Standard for Software Reviews and Audits*
The standard provides minimum acceptable requirements for systematic reviews:

Wiegers, Karl *Peer Reviews in Software: A Practical Guide* 2001, Addison-Wesley Professional

CHAPTER 19

Code Inspection & Reviews Checklists

This chapter summarizes the code review checklists

- The types of reviews to perform checklist
- Basic review checklist
- Software revision control setup checklist
- Software Release checklist

71. REVIEWS

These are the kinds of reviews to perform

- ☐ Basic reviews
- ☐ Microcontroller / Hardware Initialization review
- ☐ Error returns review
- ☐ Fault handling review
- ☐ Memory/Storage handling review
- ☐ Prioritization review
- ☐ Concurrency review
- ☐ Critical function / Supervisor review
- ☐ Low power mode review
- ☐ Numerical processing review
- ☐ Signal processing review
- ☐ Timing review

72. BASIC REVIEW CHECKLIST

Before a review proceeds:

- ☐ Code has clean-result when checked with analysis tools – MISRA C rules, lint, compiling with extensive warning checks enabled.

See also

- Appendix I for the *Code Complete* Code Review check lists
- Appendix J for a rubric to apply in the reviews

72.1. BASIC STYLE

Layout checks:

- ☐ Commenting: there are comments at the top of the file, the start of each function, and with all the code that needs an explanation
- ☐ Does the source code conform to the coding style guidelines & other conventions? These cover location of braces, variable and function names, line length, indentations, formatting, and comments.
- ☐ Code naming, indentation, and other style elements are applied consistently (esp in areas beyond the style guidelines)

Names:

- ☐ Are the file names well chosen?
- ☐ Are the files in the correct location in the file tree? In the repository?
- ☐ Are the names – for variables, files, procedures, and other objects – clear and well chosen? Do the names convey their intent? Are they relevant to their functionality?
- ☐ Do they use a good group / naming convention (e.g. related items should be grouped by name)
- ☐ Is the name format consistent?
- ☐ Names only employ alphanumeric and underscore characters?
- ☐ Are there typos in the names?

Values and operators:

- ☐ Parentheses used to avoid operator precedence confusion
- ☐ Are `const` and `inline` instead of `#define`?
- ☐ Is conditional compilation avoided? Can it be reduced?
- ☐ Avoid use of magic numbers (constant values embedded in code)
- ☐ Use strong typing (includes: sized types, structs for coupled data, const)

Control flow checks:

- ☐ Are all inputs checked for the correct type, length, format, range?
- ☐ Are invalid parameter values handled?
- ☐ Are variables initialized at definition?
- ☐ Are output values checked and defined?
- ☐ Are `NULL` pointers, empty strings, other boundary conditions (for results) handled?

72.2. BASIC FUNCTIONALITY

- ☐ Does the code match the detailed design (correct functionality)?
- ☐ Does the code work? Does it perform its intended function? Is the logic is correct? etc.
- ☐ Is the update/check of state correct? Any incorrect updates or checks?
- ☐ Is the wrong algorithm/assumption/implementation used?
- ☐ Is the work performed in the correct order?
- ☐ Check that proper types are employed

72.3. SCOPING

- ☐ Proper modularity, module size, use of .h files and #includes
- ☐ Is the code as modular as possible?
- ☐ Minimum scope for all functions and variables; e.g. few globals?
- ☐ Can any global variables be replaced?
- ☐ Are there unused or redundant variables? Macros?
- ☐ Do the variables have an appropriate storage class (and scope) – static, extern, stack?
- ☐ The register storage class is not used?

72.4. CONTROL FLOW

- ☐ There is forward progression: loops are bounded, delays are bounded, etc.
- ☐ Do loops have a set length and correct termination conditions?
- ☐ Loop entry and exit conditions correct; minimum continue/break complexity
- ☐ Conditionals should be minimally nested (generally only one or two deep)
- ☐ Conditional expressions evaluate to a boolean value
- ☐ Conditional expressions do not assignments, or side-effects
- ☐ All switch statements have a default clause, with error return
- ☐ Do the work events/messages get submitted backwards in the IO queue network? Is there a potential infinite work loop?

72.5. DOCUMENTATION

- ☐ Are all procedures/functions/variables/etc commented?
- ☐ Do they properly describe the intent of the code?
- ☐ Is any unusual behavior or edge-case handling described?
- ☐ Are all parameters of the procedure are documented?
- ☐ Is the use and function of third-party libraries documented?
- ☐ Are data structures and units of measurement explained?
- ☐ Is there any incomplete code? If so, should it be removed or flagged with a suitable marker like 'TODO'?

72.6. MAINTAINABILITY AND UNDERSTANDABILITY

- ☐ Is all the code easily understood? Is the code simple, obvious, and easy to review?
- ☐ Is the code unnecessarily, ornate or complex? Are there more intermediate variables than necessary? Is the control flow overly complex? (Look for variables that hold the return value far from the return)
- ☐ Code complexity measure is low (below set threshold)?
- ☐ Is there any redundant or duplicate code?
- ☐ Is there any dead or commented out code?
- ☐ Can any of the code be replaced with library or built-in functions?

- ☐ Any changes that would improve readability, simplify structure, and utilize cleaner models?
- ☐ Does the code have too many dependencies?

72.7. TESTABILITY

- ☐ Is the code testable?

72.8. PERFORMANCE

- ☐ Are there obvious optimizations that will improve performance?
- ☐ Can any of the code be replaced with library functions built for performance?

Performance changes to improve the implementations:

- ☐ Can the data access be improved? E.g. caching and work avoidance.
- ☐ Can the I/O scheduling be improved? E.g. batching of writes, opportunistic read ahead and avoiding unnecessary synchronous I/O.
- ☐ Are there better / faster data structures for in-memory and secondary storage?
- ☐ Are there other performance improve techniques that can be applied?

Synchronization-based performance improvements:

- ☐ Are the synchronization methods inefficient?
- ☐ Can a pair of unnecessary locks be removed?
- ☐ Can finer-grained locking be employed?
- ☐ Can write locks be replaced with read/write locks?

72.9. OTHER

- ☐ Can any logging or debugging code be removed?
- ☐ Are there regular checks of operating conditions?
- ☐ Data structure ordering is efficient for access pattern? Alignment and padding will not be an issue?
- ☐ Do the variables have the appropriate qualifiers? `volatile`? `const`?

73. SPECIALIZED REVIEW CHECKLISTS

This section provides checklists for specialized, focused reviews:

- Microcontroller / Hardware Initialization review
- Error returns review
- Fault handling review
- Memory/Storage handling review
- Prioritization review
- Concurrency review
- Critical function / Supervisor review
- Low power mode review

- Numerical processing review
- Signal processing review
- Timing review

Note: these can be used in conjunction with the detailed design review checklists. If the detailed design review covered these, the review is much faster; often there is no detailed design review.

See also

- Chapter 13 Design review check lists
- Appendix I for the *Code Complete* Code Review check lists
- Appendix J for a rubric to apply in the reviews

73.1. MICROCONTROLLER / HARDWARE INITIALIZATION REVIEW CHECKLIST

Looks for bugs in the initialization and configuration of the hardware:

- ☐ Check the initialization order
- ☐ Are the clocks set correctly? i.e., no over-clocking at the voltage and/or temperature
- ☐ Does the code handle oscillator (or clock) startup failures?
- ☐ Does the code check the initial clock rate? Properly?
- ☐ Check that the source clock, prescalar, divisor, and PLL configuration are setup correctly.
- ☐ Check the peripherals are configured and enabled properly
- ☐ Is the software using the right bus for the peripheral?
- ☐ Check that the proper clock source is enabled for the peripheral.
- ☐ Check that the peripheral is not over-clocked for the power source and temperature range. (Some peripherals have tighter constraints)
- ☐ Check that the correct power source / enable is used in setting up the peripheral
- ☐ DMA channel assignments match hardware function constraints
- ☐ GPIO mode, direction (in/out), biasing (pull-ups, pull-downs) are configured correctly.
- ☐ Power supervisor / brown-out detect is configured properly.
- ☐ Lock bits are set on peripherals – GPIO, timer, etc.
- ☐ The microcontroller's errata has read and applied?

73.2. ERROR RESULTS REVIEW CHECKLIST

A lack of checking results, or incorrectly handling the results, is a frequent source of critical failures. Look for bugs in the handling (or lack thereof) of return values and error results:

- ☐ Check that NULL pointers, empty strings, other result boundary conditions are handled
- ☐ Error handling for function returns is appropriate
- ☐ Does it check the correct (or wrong) set of error codes?
- ☐ Is there missing or incorrect error code handling?
- ☐ Where third-party utilities are used, are returning errors being caught?

73.3. FAULT HANDLING (WITHIN PROCEDURES) REVIEW CHECKLIST

Defects in fault handling is a frequent source of critical failures. Look for bugs on failure paths.

- ☐ Check that the semantics for the failure are handled correctly. Is metadata updated properly? Are the resources freed?
- ☐ Check that release allocated resources
- ☐ Check that the locks/semaphores/mutexes are released correctly
- ☐ Look for null-pointer dereferences, and code that incorrectly assume the pointers are still valid after failure
- ☐ Check that it returns correct error code – i.e. not the wrong error code

73.4. MEMORY HANDLING REVIEW CHECKLIST

Has the memory been partitioned in a manner suitable for Class B? i.e., does the software isolate and check the regions?

- ☐ Are the potential buffer overflows?
- ☐ Are there good practices to prevent buffer overflows – bound checking, avoid unsafe string operations?
- ☐ Dereferences of free'd memory
- ☐ Dereferences of NULL pointer
- ☐ Dereferences of undefined pointer value
- ☐ incorrect handling of memory objects
- ☐ didn't release memory / resource
- ☐ Free'd memory resource twice
- ☐ Parity checking enabled
- ☐ Redundant memory is segregated and stored in a different format
- ☐ Check that the data access will be performant; that an slow approach is not employed unnecessarily
- ☐ Memory pages write protected
- ☐ Memory protection unit is enabled? Access control is configured properly?

Non-volatile storage:

- ☐ Doesn't overwrite or erase the non-volatile data in use
- ☐ Doesn't use a "replacement" strategy of writing the most recent/highest good-copy of the data.
- ☐ Accounts for loss of power, reset, timeout, etc during read/write operation
- ☐ Checks supply voltage before erasing/writing non-volatile memory
- ☐ Performs read back after write
- ☐ Checks that software detects bit-flip and other loss of data integrity (e.g. employs CRC)
- ☐ Check that data recovery methods will work, if employed
- ☐ Check that the correct version of stored data will be employed (such as on restart)
- ☐ Interrupts and exceptions are disabled during program memory is modified.

- ☐ Cache/instruction pipeline is flushed (as appropriate) after program memory modification.
- ☐ Check that the data access will be performant; that an slow approach is not employed unnecessarily
- ☐ Check that the data access will not interfere with the other timing.

73.5. PRIORITIZATION REVIEW CHECKLIST

- ☐ Rate Monotonic Analysis (RMA) and dead-line analysis performed
- ☐ Task/thread prioritization based on the analysis
- ☐ Mutex prioritization based on the analysis
- ☐ Events, Messages and IO queue prioritization based on the analysis
- ☐ Interrupt prioritization are based on the analysis
- ☐ DMA channel prioritization are based on the analysis
- ☐ CAN message priorities are based on the analysis
- ☐ ADC priorities are based on the analysis
- ☐ Bluetooth LE notification/indication priorities are based on the analysis

73.6. CONCURRENCY REVIEW CHECKLIST

- ☐ Are there any missing locks, and IPC mechanisms?
- ☐ Check acquisition order of locks/semaphores/mutexes – is the order wrong or potential for dead locks?
- ☐ Check for violations of access atomicity: missing volatile keyword, assuming that read/write is atomic when it is not, missing write barriers, etc.
- ☐ Check order of multiple accesses
- ☐ Check for missing release of lock/semaphore/mutex
- ☐ Check for unlocking lock / posting semaphore/mutex multiple times
- ☐ Look for forgotten unlock locks/semaphores/mutexes
- ☐ Are there ways to reduce the blocking time?

73.7. CRITICAL FUNCTION / SUPERVISOR REVIEW CHECKLIST

Check that critical functions (e.g. Class B and C of 60730) are suitably crafted:

- ☐ Is the code for the critical functions in a limited (and small) number of software modules?
- ☐ Is the code for the critical functions small?
- ☐ Is the code complexity low? Are there no branches – or only simple branches?
- ☐ Are the possible paths thru the critical function code small, and simple?
- ☐ Is the relation between the input and output parameters simple? Or at least, simple as possible?
- ☐ Are complex calculations used? They should not be. Especially as the basis of control flow, such as branches and loops.
- ☐ Power supervisor / brown-out detect is configured properly.
- ☐ Checks the clock functionality and rates

- ☐ Watchdog timer is employed (and correctly)
- ☐ Is the watchdog reset only after all protected software elements are shown to be live? An example of a bad design would be to reset the watchdog in the idle lop
- ☐ Check that the watchdog timer is not disabled anywhere in the code
- ☐ Is the external watchdog handshake done only after all of the software has checked liveliness? A bad approach is to use a PWM for the handshake, as a PWM can continue while software has locked up or is held in reset.
- ☐ Handles interrupt overload conditions
- ☐ Critical program memory is protected from writes. How: Hardware level? Software?
- ☐ Program memory CRC check.
- ☐ Stack overflowing checking
- ☐ Critical data is separated, checked, protected.
- ☐ Cross checks values
- ☐ Performs read backs of sent values
- ☐ Independent checks / reciprocal comparisons to verify that data was exchanged correctly.
- ☐ Periodic self-tests or functional tests
- ☐ Are there possible partition violations from data handling errors, control errors, timing errors, or other misuse of resources?
- ☐ That the software can meet the scheduling requirements and the timing constraints specified.
- ☐ Do the fail-safe and fail-operational procedures bring the product to the defined acceptable state?

73.8. LOW POWER MODE REVIEW CHECKLIST

Power configuration for low power modes:

- ☐ Does it switch to low clock source(s) and disable the others?
- ☐ Are the IOs set to a low direction, mode (e.g. analog in?) and bias (e.g. pull-down, pull-up)?
- ☐ Are peripherals disabled where they can be?
- ☐ Are peripheral clocks disabled where they can be?
- ☐ Are the proper flushes, barriers, etc. executed before going into a sleep state?
- ☐ Is the proper low-power instruction used?
- ☐ Is there a race condition in going into low-power state and not being able to sleep or wake?
- ☐ Check coming out of low power mode restores the operating state

73.9. NUMERICAL PROCESSING REVIEW CHECKLIST

Check for correct arithmetic, and other numerical operations:

- ☐ Check that division by zero, other boundary conditions are handled
- ☐ Is lazy context save of floating point state (LPSEN) disabled on ARM Cortex-M4s? (See ARM Cortex-M4 errata, id 776924)
- ☐ Floating point is not used in interrupts, exception handlers, or the kernel

- ☐ Check that floating point equality is used properly – i.e., something other than ==. Does it handle denormals, non-zeros, NaNs, INFS and so on?
- ☐ Are the equations ill-conditioned?
- ☐ Is the method of calculation slow?
- ☐ Check that denormals, NaNs, INFs, truncation, round off that may result from calculations are properly handled.
- ☐ Are the use of rounding and truncation proper?
- ☐ Would use of fixed point be more appropriate?
- ☐ Is simple summation or Euler integration employed? This is most certainly lower quality than employing Simpsons rule, or Runge-Kutta.

73.10. SIGNAL PROCESSING REVIEW CHECKLIST

- ☐ Are the ADCs over-clocked for the signal chain? Check that the sample time and input impedance are aligned.
- ☐ Is the sample time sufficient to measure the signal?
- ☐ Is there a potential time variation (e.g. jitter) in the sampling? The code should be implemented for low jitter. For instance, a design that uses a DMA ring-buffer has low variation, while run-loop or interrupt trigger can have a great deal of time variation.
- ☐ Is oversampling applied? Is the oversampling done in a proper way?
- ☐ Is simple summation or Euler integration employed? This is most certainly lower quality than employing Simpsons rule, or Runge-Kutta.
- ☐ Is the proper form of the filter used? Is an unstable form used?
- ☐ Does it have ringing, feedback, self-induced oscillation or other noise?
- ☐ Does handle potential saturation, overflows?
- ☐ Efficient, fast implementation?
- ☐ Is there good instruction locality on the kernel(s)?
- ☐ Is there good data locality on the kernel(s)?
- ☐ Is the signal processing unnecessarily complex?
- ☐ Check the step response of the signal processing

73.11. TIMING REVIEW CHECKLIST

- ☐ Does the timing meet the documented design and requirements?
- ☐ Are there possible timing violations?
- ☐ Are there race conditions?
- ☐ Is enough time given to let a signal/action/etc propagate before the next step is taken?
- ☐ Is there a potential for hidden delays (e.g. interrupt, task switch) that would violate the timing?
- ☐ From the time the trigger is made to the action, what worst case round-trip? Include interrupts, task switching, interrupts being disabled, etc. Is this timing acceptable?
- ☐ The length of operations, in the worst case, do not cause servicing the watchdog timer to be missed.

Appendices

- ABBREVIATIONS, ACRONYMS, & GLOSSARY. This appendix provides a gloss of terms, abbreviations, and acronyms.
- PRODUCT STANDARDS. This appendix provides supplemental information on standards and how product standards are organized
- BUG REPORTING TEMPLATE. This appendix provides a template (and guidelines) for reporting bugs
- TYPES OF DEFECTS. This appendix provides a classification of different kinds of software defects that are typically encountered.
- CODE COMPLETE REQUIREMENTS REVIEW CHECKLISTS. This appendix reproduces checklists from *Code Complete, 2nd Ed* that are relevant to requirements reviews.
- CODE COMPLETE DESIGN REVIEW CHECKLISTS. This appendix reproduces checklists from *Code Complete, 2nd Ed* that are relevant to design reviews.
- DESIGN REVIEW RUBRIC. This appendix provides rubrics relevant in assessing the design and its documentation.
- FLOATING POINT PRECISION. This appendix recaps the limits of floating point precision.
- CODE COMPLETE CODE REVIEW CHECKLISTS. This appendix reproduces checklists from *Code Complete, 2nd Ed* that are relevant to code reviews.
- SOFTWARE REVIEW RUBRIC. This appendix provides rubrics relevant in assessing software workmanship.

[This page is intentionally left blank for purposes of double-sided printing]

APPENDIX A

Abbreviations, Acronyms, Glossary

Table 38: Common acronyms and abbreviations

Abbreviation / Acronym	Phrase
ADC	analog to digital converter
ANSI	American National Standards Institute
ARM	Advanced RISC Machines
BNF	Backus-Naur Form
BSP	board support package
API	application programming interface.
CAN	controller-area network
CRC	cyclic redundancy check
DAC	digital to analog converter
DMA	direct memory access
EN	European Norms
GPIO	general purpose IO
Hz	Hertz; 1 cycle/second
I^2C	inter-IC communication; a type of serial interface
IEC	International Electrotechnical Commission
IPC	interprocess communication
IRQ	Interrupt request
ISO	International Organization for Standardization
ISR	Interrupt service routine
JTAG	Joint Test Action Group
MCU	microcontroller (unit)
MPU	memory protection unit
NMI	non-maskable interrupt
NVIC	nested vector interrupt controller
NVRAM	non-volatile RAM
PWM	pulse width modulator

QMS	quality management system
RAM	random access memory; aka data memory
RISC	reduced instruction set computer
RTOS	real time operating system
SDK	software development kit
SDLC	software development lifecycle
SPI	serial peripheral interface
SRAM	static RAM
SWD	single wire debug
TBD	to be determined
TMR	timer
UART	universal asynchronous receiver/transmitter
WDT	watchdog timer

***Table 39:** Glossary of common terms and phrases*

Phrase	Description
abnormal operating condition	A condition when an operating variable has a value outside of its normal operating limits.[15] See also *fault, normal operating condition.*
allowed operating condition	A condition when each of the operating variables (flow, pressure, temperature, voltage, etc.) has a value within of its respective normal operating limits, and so the "system will satisfy a set of operational requirements" [IEC 62845 3.10]. See also *abnormal operating condition, fault.*
analog to digital converter	An analog to digital converter measures a voltage signal, producing a digital value.
application logic	Application logic is a set of rules (implemented in software, or hardware) that are specific to the product.
Backus-Naur form	A notation used to describe the admissible calling sequences for an interface. Traditionally this form is used to define the syntax of a language.
bitband	An ARM Cortex-M mechanism that allows a pointer to a bit.
black-box testing	Testing technique focusing on testing functional requirements (and other specifications) with no examination of the internal structure or workings of the item.
board support package	The specification to an RTOS and/or Compiler of what peripherals the MCU has internally, and is directly connected to.
certification	A "procedure by which a third party gives written assurance that a product, process or service conforms to specified requirements, also known as conformity assessment" [IEC 61400-22 3.4] longer description at [IEC 61836 3.7.6]
coding style guide	"specif[ies] good programming practice, proscribe unsafe language features (for example, undefined language features, unstructured designs, etc.), promote code understandability, facilitate verification and testing, and specify procedures for source code documentation." [IEC 61508-3 7.4.4.13] aka *coding standard*
coefficient	A measure of a property for a process or body. This number is constant under specified, fixed conditions.

[15] Modified from http://www.wartsila.com/encyclopedia/term/abnormal-condition

comment	Text, usually to provide context, clarify or explain the requirement(s).
control function	"functions intended to regulate the behaviour of equipment or systems" [IEC 61892-2 3.9], it typically "evaluates input information or signals and produces output information or activities" [IEC 62061 H.3.2.14] see also *safety-related control function*
control function (class B)	Those "control functions intended to prevent an unsafe state of the appliance... Failure of the control function will not lead directly to a hazardous situation" [IEC 60730-1:2013 H.2.22.2]
customer requirement	A requirement in any of the top-level documents, but especially in the customer (or user) requirements specification.
cyclic redundancy check	A form of error-detecting code. A check value is computed from a block of data.
data integrity	That the stored data – such as program memory – is intact, unchanged, in the expected order and complete; that is, that the entire program memory area matches *exactly* with the data defined for a particular revision.
data retention	The ability for a storage to hold bits
debounce	Switches and contacts tend to generated multiple rising & falling edges when coming into contact; debouncing removes the extra signals.
diagnostic	A "process by which hardware malfunctions may be detected" [IEEE 2000]
defect	An "imperfection in the state of an item (or inherent weakness) which can result in one or more failures of the item itself, or of another item under the specific service or environmental or maintenance conditions, for a stated period of time" [IEC 62271-1 3.1.16]
design document	A design document explains the design of a product, with a justification how it addresses safety and other concerns.
digital to analog converter	A digital to analog converter is used create a voltage signal from an internal value.
direct memory access	A special purpose microcontroller peripheral that moves data between the microcontrollers storage and another peripheral or storage; this is useful to reduce work done in software.
error	An error is the occurrence of an incorrect (or undesired) result.
exception	An "event that causes suspension of normal execution" [IEC 61499-1 3.36] A special condition – often an error – that changes the normal control flow. On an ARM Cortex, this can cause the processor to suspend the currently executing instruction stream and execute a specific exception handler or interrupt service routine.
$failure_1$	A failure "is a permanent interruption of a system's ability to perform a required function under specified operating conditions." (Isermann & Ballé 1997).
$failure_2$	An incident or event where the product does not perform functions (esp. critical functions) within in specified limits. e.g. the product did not meet its requirements.
$fault_1$	A fault is an abnormal condition, or other unacceptable state of some subsystem (or component) that will disallow the intended operation. The part or subsystem did not meet its requirements. See also *abnormal condition, normal operating condition.*
$fault_2$	A fault is represented an interrupt or exception on ARM processors that pass control to handler of such an abnormal condition.
fault tolerant	"The capability of software to provide continued correct execution in the presence of a defined set of microelectronic hardware and software faults." [ANSI/UL 1998]
firmware	A program permanently recorded in ROM and therefore essentially a piece of

	hardware that performs software functions.
flash	A type of persistent (non-volatile) storage media.
frequency monitoring	"a fault/error control technique in which the clock frequency is compared with an independent fixed frequency" [IEC 60730-1]
function	The "specific purpose of an *entity* or its characteristic action" [IEC 61499-1] That is, what the product is intended to do, and/or what role it is to serve.
function block	A self contained unit with specific functionality
functional hazard analysis	An "assessment of all hazards against a set of defined hazard classes" [IEC 62396-1 3.21] see also *hazard analysis*
hard fault	A type of microcontroller fault.
harm	A "physical injury or damage to health" [ISO 12100-1:2003]
hazard	A "potential source of physical injury to persons."
hazard analysis	The "process of identifying hazards and analysing their causes, and the derivation of requirements to limit the likelihood and consequences of hazards to an acceptable level" [IEC 62280 section 3.1.24] see also *functional hazard analysis, preliminary hazard analysis, risk analysis*
hazard class	Energy (electric: voltage, current, electric & magnetic fields, radiation, thermal energy, vibration/torsion/kinetic energy/force, acoustic), biological & chemical, operational (function and use error), are informational (labeling, instructions, warnings, markings) [ISO 14971]
hazard list	A list of all identified hazards that a product may present.
high-level specification	System specification, customer inputs, marketing inputs, etc.
identifier	A label that can refer to product, specific version of the product, a document, requirement, test, external document, or comment.
initialization	Places each of the software and microcontroller elements into a known state; performed at startup.
input comparison	"a fault/error control technique by which inputs that are designed to be within specified tolerances are compared." [IEC 60730-1]
integrity	"The degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data." [ANSI/UL 1998]
integrity check	Checks to see that a storage unit has retained its data contents properly and that the contents have not changed unintentionally.
internal fault condition	A programmable element resets for a reason other than a power-on reset; or a fault occurs with any programmable-element, or power supervisor; or a self-test did not pass.
interface	An interface is a defined method of accessing functionality. An object may support several interfaces.
non-maskable interrupt	A type of microcontroller fault.
non-volatile memory	A storage mechanism that will preserve information without power.
parameter	A controllable quantity for a property.
parity check	A simple form of error detection. Each byte in SRAM has an extra check bit that can catch memory errors.
peripheral lock	The microcontroller's peripheral registers can be locked, preventing modification until microcontroller reset.
power management	An "automatic control mechanism that achieves the .. input power consistent with

	a pre-determined level of functionality" [IEC 62542 5.10]
power on reset	A type of microcontroller reset that occurs when power is applied to the microcontroller; release from reset allows software to execute.
preliminary hazard analysis	"This evaluates each of the hazards contained in the [preliminary hazard list], and should describe the expected impact of the software on each hazard."
programmable component	"any microelectronic hardware that can be programmed in the design center, the factory, or in the field." [ANSI/UL 1998] This includes FPGAs, microcontrollers, microprocessors, and so on.
programmable system	"the programmable component, including interfaces to users, sensors, actuators, displays, microelectronic hardware architecture," and software [ANSI/UL 1998]
protective control	A control whose "operation … is intended to prevent a hazardous situation during abnormal operation of the equipment" [IEC 60730-1]
protective electronic circuit	An "electronic circuit that prevents a hazardous situation under abnormal operating conditions" [IEC 60335]
quality management system	A "management system with which an organization will be directed with regard to product quality" [IEC 60194 10.141]
realization	An implementation, or a mathematical model or design that has the target input-out behaviour and can be directly implemented.
redundant monitoring	"the availability of two independent means such as watchdog devices and comparators to perform the same task" [IEC 60730-1]
requirement	An "expression ... conveying objectively verifiable criteria to be fulfilled and from which no deviation is permitted " [ISO/IEC Directives, Part 2, 2016, 3.3.3]
requirements specification	A set of requirements
risk	"a measure that combines the likelihood that a system hazard will occur, the likelihood that an accident will occur and an estimate of the severity of the worst plausible accident." [UCRL-ID-1222514]
risk analysis	A "systematic use of available information to identify hazards and to estimate the risk" [ISO 14971:2007 2.17]
risk management	The "systematic application of management policies, procedures and practices to the tasks of analyzing, evaluating and controlling risk" [ISO 14971:2007 2.22]
safety-critical function	A "function(s) required … the loss of which would cause the tool to function in such a manner as to expose the user to a risk that is in excess of the risk that is permitted … under abnormal conditions" [EN 62841]
safety-related function	"Control, protection, and monitoring functions which are intended to reduce the risk of fire, electric shock, or injury to persons." [ANSI/UL 1998]
safety-related control functions	A "control function … that is intended to maintain the safe condition of the machine or prevent an immediate increase of the risk(s)" [IEC 60204-32 section 3.62]
	note: not all are safety critical functions.
signal	TBD active and deactivated state; forms can include a digital logic signal (which may be active high, or active low), an analog signal, some logical state conveyed by a communication method, etc.
single event upset	An ionizing particle flipped a bit or transistor state
single wire debug	An electrical debugging interface for the ARM Cortex microcontrollers.
software development lifecycle	"conceptual structure spanning the life of the software from definition of its requirements to its release" [ISO/IEC 12207 3.11]
software risk analysis	A risk analysis applied to the software
software safety	A safety requirement applied to the function or operation of software

requirement	
test monitoring	"the provision of independent means such as watchdog devices and comparators which are tested at start up or periodically during operation" [IEC 60730-1]
test report	A report of test outcomes describing how a product performs under test.
test requirement	A requirement that define what a test must do for a product must pass the test.
test specification	A requirements specification that describes a set of tests intended to check that the product meets it requirements. This may be in the form of test requirements – what the tests are to do – and test procedures.
to be determined	The information is not known as of the writing, but will need to be known.
traceability	Ability to follow the steps from output back to original sources. For products, this allows tracing all of the products design, and features back to the original documents approved by the company. For information, this allows tracing to measurements, methodology and standards.
trace matrix	A tool that is used to identify high level requirements that are not realized by a low-level requirement or design element; and low-level requirements or design requirements that are not driven by a high-level requirement.
validation	Check that the product meets the users specification when the item is used as an element of the product
verification	Checking that an item meets its specification
watchdog reset	A microcontroller reset triggered by the expiration of a watchdog timer.
watchdog timer	A hardware timer that automatically resets the microcontroller if the software is unable to periodically service it.
white-box testing	Testing technique focusing on testing functional requirements (and other specifications), with an examination of the internal structure or workings of the item.

APPENDIX B

Product Standards

This appendix provides further, supplemental discussion of standards.

74. STANDARDS

I did not provide a definition of "standard" earlier. Circular No A-119 provides a useful definition of *technical standard*, being that a standard that includes:

1. [The] common and repeated use of rules, conditions, guidelines or characteristics for products or related processes and production methods, and related management systems practices[; and]
2. The definition of terms;
   classification of components;
   delineation of procedures;
   specification of dimensions, materials, performance, designs, or operations;
   measurement of quality and quantity in describing materials, processes, products, systems, services, or practices;
   test methods and sampling procedures; or
   descriptions of fit and measurements of size or strength.

OMB Circular No A-119, Revised OMB (US Government) 1998 Feb 10

74.1. OTHER IMPORTANT SOFTWARE SAFETY STANDARDS

DO-178C is the aerospace industry's software quality standard. It employs five levels (instead of 3) and in descending order of concern (as opposed to the IEC 60730's & 62304 ascending order):

- Level A for Catastrophic
- Level B for Hazard/Severe
- Level C for Major
- Level D for Minor
- Level E for no effect

DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc. 2012 Jan 5*

NASA-STD-8719.13 is NASA's software assurance standard. It classifies software criticality in descending level of concern, based on its role and/or complexity. This classification is based on MIL-STD-882C (the last revision to have such a classification).

- Category IA. "Partial or total autonomous control of safety-critical functions by software[; or] Complex system with multiple subsystems, interacting parallel processors, or multiple interfaces[; or] Some or all safety-critical software functions are time critical exceeding response time of other systems or human operator[; or] Failure of the software, or a failure to prevent an event, leads directly to a hazard's occurrence."

NASA-STD-8719.12., NASA Software Safety Standard*, Rev C 2013-5-7*

- Category IIA & IIB. "Control of hazard by software but other safety systems can partially mitigate. Software detects hazards, notifies system of need for safety actions.[or] Moderately complex with few subsystems and/or a few interfaces, no parallel processing[; or] Some hazard control actions may be time critical but do not exceed time needed for adequate human operator or automated system response.[; or] Software failures will allow, or fail to prevent, the hazard's occurrence. "
- Category IIIA & IIIB. "Several non-software mitigating systems prevent hazard if software malfunction[; or] Redundant and independent sources of safety-critical information[; or] Somewhat complex system, limited number of interfaces[; or] Mitigating systems can respond within any time critical period[; or] Software issues commands over potentially hazardous hardware systems, subsystems or components requiring human action to complete the control function."
- Category IV. "No control over hazardous hardware. No safety-critical data generated for a human operator. Simple system with only 2-3 subsystems, limited number of interfaces. Not time-critical."

NASA-STD-8739.8 is NASA's software quality standard. It classifies software criticality in descending level of concern, but based on a classification of intended use rather than hazard:

NASA-STD-8739.8, "Software Assurance Standard" *NASA Technical Standard 8739.8 2004, 2004 Jul 28*

- Class A Human Rated
- Class B Non-Human Space rated
- Class C Mission support software
- Class D Analysis and Distribution software
- Class E Development support

75. PRODUCT STANDARDS

75.1. TYPES OF ISO SAFETY & PRODUCT STANDARDS

ISO 12100-1:2003 proposes organizing standards into a hierarchy of how broadly or specifically they apply.

- *Basic safety standards* (type A), give generic concepts & principles applicable to all machinery of a class. (ISO 12100 is itself a type A standard)
- *Generic safety standards* address wide range of machinery, but focus on a narrow area of safety (type-B),
    - Type B1 are those that focus on safety "aspect" – some safe operating region often defined along a physical dimensions
    - Type B2 are those that focus on safeguards or mechanisms
- Standards for groups or a particular machine (type C) are the narrowest

75.2. TYPES OF IEC SAFETY STANDARDS

IEC safety standards are similarly grouped, from broadest to narrowest:

- *Basic safety publications* give general safety provisions, generic concepts & principles applicable to many products.

- *Group safety publications* address all safety aspects of a specific group of products
- *Product publication* for "a specific product or group of related products" [IEC 2011]

An extra, informal, variant is that a country (or region) may adopt the standards, modifying them in the process. This is important as these are the ones *recognized* (accepted) for the country or region.

75.3. PRODUCT STANDARDS

The table below summarizes how several safety standards adapt software safety-related material from other standards:

***Table 40:** Safety standards and where they adapt from*

Std	Adapts	Type	Sector	Notes
EN/ISO 13849	IEC 61508	B1	machine control	*"Safety of machinery - Safety-related Parts of Control Systems" Uses PL risk*
ISO 26262	IEC 61508	Group	Automotive	*"Road Vehicles Functional Safety"* Applies ASIL to automotive electrical/electronic systems
EN 50128:2011		Group	Railway	*"Railway applications. Communication, signalling and processing systems."* (includes software)
EN 60601		Group	Medical	Medical device product requirements
UL 61010				*Safety Requirements for Electrical Equipment for Measurement, Control, and Laboratory Use - Part 1: General Requirements*, 2015 May 11
IEC 61508	DIN 12950	Basic		Adapted risk assessment from DIN 12950
IEC 61511	IEC 61508	Group	Industrial process	*"Functional safety - Safety instrumented systems for the process industry sector."*
IEC 61513:2001	IEC 61508			*"Nuclear power plants - Instrumentation and control for systems important to safety - General requirements for"*
IEC/EN 62061	IEC 61508	Group	Machinery	*"Safety of machinery: Functional safety of electrical, electronic and programmable electronic control systems,"*
IEC 62279	IEC 61508		Railway	
IEC 62841	IEC 60730	Group	Garden appliances	*"Electric motor-operated hand-held tools, transportable tools and lawn and garden machinery - Safety - Part 1: General requirements"*

76. REFERENCES AND RESOURCES

IEC, *Basic Safety Publications,* 2011

IEC, *Basic Safety Publications: Tools*

APPENDIX C

Bug Report Template[16]

This Appendix describes the best means in which to file a bug report. A useful bug report is written in simple, jargon free language, and structured using the inverted hierarchy.

"The horror of that moment," the King went on, "I shall never, never forget!" "You will, though," the Queen said, "if you don't make a memorandum of it"– Lewis Carroll, Through the Looking Glass

77. OUTLINE OF A PROPER BUG REPORT

12 words	1 : Bug Header Information
1-5 words	1.1 : Product
2 words	1.2 : Classification
1-3 words	1.3 : Reproducibility
	1.4 : Version/Build Number
2 words	1.5 : Area of bug
< 20 words	2 : Bug Title & Description
< 20 words	2.1 : Title
	2.2 : Description
	2.3 : Requirements that are of interest or are relevant
	3 : Additional Information To Provide (General)
	3.1 : Configuration Information
	3.2 : Crashing Issues
	3.3 : Application resets
	3.4 : Hanging/Performance Issues
	3.5 : Screen shots, Scope Capture,
	4 : Contact Information
	5 : Product-specific Additional Information

The remainder of the

12 words

78. BUG HEADER INFORMATION

1.1: Product: *1 to 5 words*

PC Programmer, Handheld, OurPeripheral, Implant, Telemetry Module, etc, whether it is a first run engineering board, a second run engineering board, a first run production board, a second run production board, etc

Include details such as the part number, or board assembly and serial number

1.2: Classification: *2 words*

[16] This appendix is adapted from Apple's bug reporting form, as well as many others.

Classify the bug appropriately (partly by its *manifestation)* so that we can properly prioritize the problem:

Method of manifestation is the observable effect

- Crash/Hang/Data Loss: Bugs which cause a machine to crash, resulting in an irrecoverable hang, or loss of data.
- Performance: Issues that reduce the performance or responsiveness of an application.
- Usability: A cosmetic issue, or an issue with the usability of an application.
- Serious bug: Functionality is greatly affected, and has no workaround.
- Other bug: A bug that has a workaround.
- Unexpected behaviour: a bug that not only has a work around
- Feature (new): Request for a new feature
- Enhancement: Request for an enhancement to an existing feature.

1.3: Reproducibility

1 to 3 words

Let us know how frequently you are able to reproduce this problem.

1.4: Version/Build Number:

Provide the version of firmware / software you are using. (If it is an engineering change to a release version please note that)

1.5 Area of bug:

2 words

This is how the bug manifests itself, or where it has the observable effect:

- Communication
- Therapy Behaviour
- Input to output logic behaviour
- Preferences
- Recharge
- Incorrect or inaccurate results: input/output is wrong, or provides inaccurate information
- Corruption – data is corrupted, altered, lost or destroyed
- Responsiveness, Speed or Performance degradation, efficiency defects
- Power: poor battery life, high power consumption, degradation, efficiency defects
- Increased resource usage in other areas
- Other device behaviour
- It crashes my Handheld / OurPeripheral / Telemetry Module / LabPC / Display Unit

79. BUG TITLE AND DESCRIPTION

2.1: Problem Report Title:

<20 words

The ideal problem title is clear, concise, succinct and informative. It should include the following:

- Build or version of the firmware on which the problem occurred
- Verb describing the action that occurred
- Explanation of the situation which was happening at the time that the problem occurred

- In case of a crash or hang, include the symbol name

The title should also:

- Be objective and clear (and refrain from using idiomatic speech/colloquialisms/slang)
- Include keywords or numbers from any error messages you may be receiving
- Not employ vague terms such as “failed”, “useless”, “crashed”, “observed” etc....

The following examples demonstrate the difference between a non-functional title and a functional title:

Example 1:

Non-functional title:	Handheld Crashed.
Functional title:	Handheld gave a watchdog reset while performing a lead impedance measurement

Example 2:

Non-functional title:	Failed test
Functional title:	OurPeripheral return error ErrOutOfSpace when performing recharge test.

2.2: Description:

The description includes:

- A Summary
- Steps to Reproduce
- Expected Results
- Actual Results
- Workaround, and
- Regression/Isolation
- Relevant requirements.

Summary:

Recap the problem title and be explicit in providing more descriptive summary information.

Provide what happened, what you were doing when it happened, and why you think it's a problem. If you receive an error message, provide the content of the error message (or an approximation of it).

Provide specifics and avoid vague language or colloquialisms. Instead of using descriptive words or phrases when something “looks bad,” “has issues,” “is odd,” “is wrong,” “is acting up,” or “is failing,” be concise and describe how something is looking or acting, why you believe there is a problem, and provide any error messages that will support the problem being reported.

Example 1:

Non-functional description:	When printing, nothing happens. Application doesn't work.
Functional description:	Print Menu item enabled, print dialog box appears, print button enabled, but progress dialog box doesn't appear.

Example 2:

Non-functional description:	Handheld is slow.
Functional description:	Handheld is slow when incrementing therapy amplitude (provide durations)

If there is a clear safety implication, specify it (otherwise do not).

Steps to Reproduce:

Describe the step-by-step process to reproduce the bug, including any non-default preferences/installation, and the system configuration information. Note: It is better to include too much information than not enough, as this reduces the amount of back-and-forth communications. Note: Be very specific and be sure to provide details, as opposed to high-level actions. Test cases with clear & concise steps to reproduce that will enable us to reproduce this and fix.

When does the problem occur? For example:

- Does it occur after power on?
- Does it occur after unlock?
- Does it occur after power off and lock?

Important points to note when providing steps to reproduce are:

- Include information about any preferences that have been changed from the defaults.

Expected Results:

Describe what you expected to happen when performing the steps to reproduce.

Actual Results:

Explain what actually occurred.

With error codes try to include the text name of the error code

Bad:	error 0x12
Good:	ErrParameterOutOfRange (0x12)

Workaround:

If you have found a workaround for this problem, describe it.

Regression/Isolation:

Note any other configurations in which this issue was reproducible. Include details if it is new to this build, or no regression testing was done.

If there are other steps that are similar to those above, but do not create an undesired outcome, please note those. We can use this information to help resolve the issue.

2.3: Requirements that are of interest or are relevant

80. ADDITIONAL INFORMATION REQUIREMENTS (GENERAL)

Reports from developers should include:

- The hardware configuration
- The “preferences” configuration

- The device bonding or pairing configuration
- The "manufacturing data" configuration
- The embedded device(s) configuration
- If reporting an error dialog message or UI bug, provide screen shots
- Log file

Reports from developers should include

- A complete enumeration of the Revision Ids of the source files

Reports from test stations should include:

- The software / firmware version
- Event trace (e.g. log of the connection). Please provide the *smallest* trace possible that captures the issue. As traces may contain a lot of spurious information that doesn't pertain to the issue at hand, it is vital to the bug solving effort to remove distracting volume.

The generation of this information can be done in an automated fashion.

3.2: Crashing Issues:

A crash might include a NMI, Watchdog, Stack Underflow, Stack Overflow, memory fault, bus fault, usage fault, or Hard Fault. Extra information is essential. Please give us:

- The fault register values
- Call stack trace (if possible)

In addition to all the above, provide any information regarding what you were doing around the time of the problem.

NOTE: If you're able to reproduce the crash the exact same way each time and the ___ looks identical in every instance, only one crash report is required. In instances where the crash doesn't look identical, file separate reports with one crash log submitted per bug.

3.4: Hanging/Performance Issues:

If you are experiencing a "hang" (includes freeze, slow data transfer), a sample of the application while it is in the hung state is required.

3.5: Screen shots, Scope Traces and Waveform capture:

SCREEN SHOTS. Provide a screen shot when it will help clarify the bug report. In addition to providing any screen shots to error or dialog messages, be sure to also type the text of the error/dialog message you're seeing in the description of the bug report (so that the contents of the message are searchable. If there are steps involved, a sequence of screen shots, or a movie is always appreciated. Be sure to write down the steps associated with each screen shot.

SCOPE TRACE. When working with electrical signals, please provide scope trace or screen shot of the oscilloscope. Please provide a diagram of the setup, and a description where in the diagram or schematic the signals were measured.

81. CONTACT INFORMATION

Be sure to include the contact information of who found the bug. Although this sounds implicit in an email or trouble tracking system (e.g. ClearQuest, Jira), too often the bug

reporter is different than the one who found it. By including the contact information we'll be able to correspond with them as we investigate the issue.

82. PRODUCT-SPECIFIC ADDITIONAL INFORMATION

When submitting a bug report against certain tools, be sure to provide the following additional information:

- Build number & version. Put the build number at the beginning of your title as such:

  1.5.0_06-112: Title Here

If your setup is non-standard, indicate that in the bug report.

Handheld Power Management (sleep/wake) issues:

- Be aware of what is plugged into the Handheld

When submitting a bug report involved a sealed in the can device, be sure to provide:

- Whether the battery is connected or not
- Was it in saline?
- Were leads attached?
- Which version of firmware?
- Was an OurPeripheral being used – which version?

APPENDIX D

Types of Defects

This Appendix describes a system of categorizing bugs.

83. OVERVIEW

The analysis of the bug is intended to gather information about its causes and underlying defects (there may be many), and provide a basis to disposition or prioritize repairs.

Bugs are classified along four dimensions by

1. Method of manifestation.
2. Type of Defect
3. Implication
4. Means of testing

Defect is the design or implementation mistake
Method of manifestation is the observable effect

The bug analysis should try included a number of attributes about how the bug manifests itself. And include a chain of analysis to other potential underlying defects.

84. CLASSIFYING THE TYPE OF DEFECT

The types of defects include:

- Hardware problem
- Hardware misuse
- Storage / access partition violation
- Resource allocation issues
- Arithmetic, numerical bug
- Logic errors
- Syntax errors
- Improper use of API's – violates how an API should be used, including calling sequence, parameter range, etc. Errors in interacting with others in calls, commands, macros, variable settings, control blocks, etc.
- State errors
- Concurrency
- Interaction issues
- Graphic errors
- Security issue – disclosure, alteration/destruction/insertion

Various sources were used in the preparation of this. "A comparative study of industrial static analysis tools (Extend Version)" Par Emanuelsoon, Ulf Nilsson, January 7 2008

84.1. HARDWARE PROBLEM

Defect is the design or implementation mistake

- Missing component
- Component incorrectly mounted
- Component broken
    - Unable to communicate
    - Does not pass self test
    - Does not operate correctly.

84.2. HARDWARE MISUSE

- Power is too high, too low, or off
- Power transition is too fast
- Truncated addresses
- Stack overrun

84.3. STORAGE / ACCESS PARTITION VIOLATIONS

STORAGE / ACCESS PARTITION VIOLATION may have attributes of the storage violation:

- Type of access: read, write
- Location of the segment, and access: stack, or heap
- The boundary violated: above or below the segment/partition.
- How far outside of the segment was the access?
- How much data is affected with the access?
- Stride: were the access violations in a large continuous span, or were there gaps between the accesses?

An access violations can be classified into one of:

- NULL pointer dereference
    - Is a pointer possibly NULL before its use? Is it checked before use?
    - Is it checked for NULL *after* its use?
- Wild pointer dereference
- Pointer arithmetic error
    - Pointer does not point to a meaningful location
    - Pointer points outside of the bounds of its referent.
- Improper memory allocation
- Using memory that has not been initialized
    - Array cell being dereferenced in a fetch (or fetch-n-modify) operation has not been initialized.
    - Pointer being dereferenced has not been initialized (a variation on the use of a variable that has not been initialized)
- Aliasing
    - Two pointers to the same region. Especially without proper *volatile.*

    - Pointer to variable storage. Especially without proper *volatile.*
    - Pointer to an array is assigned to point to second, smaller array
- Access (segmentation) violation – using something not allowed to
    - Buffer overflow / overrun
    - Array is indexed outside of its upper or lower bound.
    - Pointer points outside of the bounds of its referent.
    - Possible causes may include pointer arithmetic errors
- Access alignment violation – e.g. having something on a odd address that must be align on 16 byte boundary
- Reference of pointer being dereferenced in a fetch (or fetch-n-modify) operation has not been initialized.
- Function pointer does not point to a function – or points to a function with a different signature.
- Casting an integer in a pointer or pointer-union when it is smaller / larger
- Use of arrays (especially large arrays) on stack. This can happen when returning a struct, or array
- Use of large strings on stack. This can happen when returning a struct, or array
- Return of a pointer to the local stack

Possible causes of these

- Earlier access violation
- Uninitialized value, variable or field used as pointer
- *Arithmetic issues,* for potential sources of erroneous index and pointer calculations
    - Conversion created incorrect value. Check implicit and explicit values for proper widening and conversion.
- Input value wrong, out of range, or does not meet implicit constraints
- String or other data structure missing a termination, e.g. a NULL terminator
- The allocation was smaller than the amount of data to process

Possible fixes and mitigations

- For large strings and arrays passed on stack, pass a pointer to the array
- Add parameter checking and return a value
- Employ sentinel values, and canaries to detect inconsistencies and misuse earlier

84.4. RESOURCE AND REFERENCE MANAGEMENT ISSUES

RESOURCE AND REFERENCE MANAGEMENT ISSUES includes leaks and resources that are not released when they are no longer used:

- Use resource after free
- Double free
- Mismatch array new / delete
- Memory leak (use more memory over time)
    - Constructor / Destructor leaks

    - Bad deletion of arrays
- Temporary files
- Resource – esp. memory and file handle – leaks
- Database connection leaks
- Custom memory and network resource leaks

84.5. ARITHMETIC, NUMERICAL BUG & INCORRECT CALCULATIONS

Calculation bugs can include:

- Relying on operator precedence, or not understanding operator precedence.
- Overflow or underflow
- Invalid use of negative variables
- Loss of precision. These can come from using the wrong size type or casting to an inappropriate type:
    - Underflow – a number too small
    - Overflow – bigger than can be represented, dropping the most significant bits
    - Truncation – dropping the least significant bits
- Inadequate precision, accuracy, or resolution of type
- Computation is inaccurate. Accuracy issues relate from the formulae used.
- Numerically unstable algorithm
    - Using an IIR with an order higher than 2
    - PID lacks anti-windup (e.g. timers)
    - PID lacks dead-band dampening
- Equality check is incorrect
    - Check for literal zero rather than within epsilon around zero
    - Check equal to NaN, rather than using `isnan()`
- Basic inappropriate values for an operation
    - Using a Not-A-Number
    - Driving by zero
    - Performing an operation, such as logarithm and `sqrt()`, on a negative number
    - Shift left by more than the size of the target
    - Shift operand is negative
    - Shift LHS is negative

84.6. ERRORS IN LOGIC

Errors in logical can include:

- Illegal values to operations
- Not checking taint or validating values properly
- Wrong order of parameters in a call
- Variables that have not been initialized

- Dead code cause by logical errors
- Under run – not sending enough on time
- Macros
- Dynamic-link and loading bugs
- Infinite loop / loss of forward progression; a procedure or loop does not terminate.
- Typo between variable and procedure names
- Error in internal check
- *See API misuse*

Logical errors can have three sub-classes of defects:

- Syntax errors
- Unused results
- Incorrect calculation

UNUSED RESULTS. Unreachable code (dead code) may indicate a logical or syntax error. Data that is computed but not used may also indicate logical errors or misspellings. Data stored via a pointer but is not used may indicate a problem.

84.7. API OR COMPONENT INTERFACE MISUSE

Interface Misuse – violates how an API should be used, including calling sequence, parameter range, etc. Errors in interacting with others in calls, commands, macros, variable settings, control blocks, etc. A description of the interface should be concise, but provide enough information to understand the intended used and limitations

- STL usage errors
- API error handling
- Misuse of `sprintf`, other `varargs`, and `argv`

84.8. ERROR HANDLING

- Uncaught fault / exception.
- Inadequate fault / exception handling.
- Not checking return values
- Not checking error values

84.9. SYNTAX ERRORS

SYNTAX ERRORS may produce some of the logical errors above:

- Use of the comma operator
- Misplacement of ";", especially in conditional statements
- Forgotten breaks.
- The use of variables that were not initialized with values
- Return statements without defined value – either the return is implicit, no value is specified or the return accesses a variable that has not been initialized.

MISRA has recommended these checks

- Return of a pointer to the local stack
- Inconsistent return values for input

84.10. STATE ERRORS

- Results in wrong state
- Transition from state A to state B is not allowed
- Does not handle event in given state
- Handles event incorrectly in given state.

84.11. CONCURRENCY

- Deadlocks
- Double locking
- Missing lock releases
- Release order does not match acquisition order of other thread means dead lock, etc. (Aka reversed order of clocking)
    - Static / dynamic analysis should check the lock order (for several locks)
- Blocking call misuse
- Associate variable/register/object access with particular locks
- Lock contention

84.12. INTERACTION ISSUES

- Thread prioritizations
- Contention for resources (including, but not limited to lock contention)
- Data rate is incorrect / mismatch
- Differing process rates
- Sourcing events faster than they can be processed
- Long communication and processing pipelines
- Timing violation, too soon / too late
    - Timer incorrectly set
    - Timer stopped
    - Timer reset
- Sequence of operation is incorrect
    - Wrong command sent
    - Missing command
- Wrong response is sent
- Sent to wrong party
- Format is wrong
- Length is wrong
- Misinterpreted
- Ignored command or response

- Mis-estimated state of other party
- Redundant interaction

84.13. GRAPHIC ERRORS

- Position incorrect
- Size incorrect / truncated
- Shape incorrect
- Parent / child relationship is incorrect
- Incorrect sibling order / tab order
- Color is wrong
- Text is wrong
- Graphic mismatch / pixels not refreshed
- Pixels not being refreshed / dirty rectangle issue
- Item is not visible when it should be
- Item is visible when it should not be

84.14. SECURITY VULNERABILITY

- Temporary files. Not using secure temporary files, file names.
- Missing / insufficient validation of malicious data and string input (*see also taint checking*)
    - SQL injection attacks
- Cross-site scripting attacks
- Format string vulnerabilities
- Faulty permission models – not a bug with access checks, but many with wrong arrangement of access controls (it's very hard to do bottom up)
- Incorrect use of `chroot`, `access`, and `chmod`.
- Bad passwords
- Dynamic-link and loading bugs
- Spoofing
- Race conditions and other concurrency issues
- Poor encryption
- Command injection
- Not checking values or their origins
- Race conditions with system calls

APPENDIX E

Code-Complete Requirements Review Checklists

Adapted from **STEVEN C. MCCONNELL,** *CODE COMPLETE, 2ND ED.*

Source: https://github.com/janosgyerik/software-construction-notes/tree/master/checklists-all

85. CHECKLIST: REQUIREMENTS

85.1. SPECIFIC FUNCTIONAL REQUIREMENTS

- ☐ Are all the inputs to the system specified, including their source, accuracy, range of values, and frequency?
- ☐ Are all the outputs from the system specified, including their destination, accuracy, range of values, frequency, and format?
- ☐ Are all output formats specified for web pages, reports, and so on?
- ☐ Are all the external hardware and software interfaces specified?
- ☐ Are all the external communication interfaces specified, including handshaking, error-checking, and communication protocols?
- ☐ Are all the tasks the user wants to perform specified?
- ☐ Is the data used in each task and the data resulting from each task specified?

85.2. SPECIFIC NON-FUNCTIONAL (QUALITY) REQUIREMENTS

- ☐ Is the expected response time, from the user's point of view, specified for all necessary operations?
- ☐ Are other timing considerations specified, such as processing time, data-transfer rate, and system throughput?

- ☐ Is the level of security specified?
- ☐ Is the reliability specified, including the consequences of software failure, the vital information that needs to be protected from failure, and the strategy for error detection and recovery?
- ☐ Is maximum memory specified?
- ☐ Is the maximum storage specified?
- ☐ Is the maintainability of the system specified, including its ability to adapt to changes in specific functionality, changes in the operating environment, and changes in its interfaces with other software?
- ☐ Is the definition of success included? Of failure?

85.3. REQUIREMENTS QUALITY

- ☐ Are the requirements written in the user's language? Do the users think so?
- ☐ Does each requirement avoid conflicts with other requirements?
- ☐ Are acceptable trade-offs between competing attributes specified—for example, between robustness and correctness?
- ☐ Do the requirements avoid specifying the design?
- ☐ Are the requirements at a fairly consistent level of detail? Should any requirement be specified in more detail? Should any requirement be specified in less detail?
- ☐ Are the requirements clear enough to be turned over to an independent group for construction and still be understood?
- ☐ Is each item relevant to the problem and its solution? Can each item be traced to its origin in the problem environment?
- ☐ Is each requirement testable? Will it be possible for independent testing to determine whether each requirement has been satisfied?
- ☐ Are all possible changes to the requirements specified, including the likelihood of each change?

85.4. REQUIREMENTS COMPLETENESS

- ☐ Where information isn't available before development begins, are the areas of incompleteness specified?
- ☐ Are the requirements complete in the sense that if the product satisfies every requirement, it will be acceptable?
- ☐ Are you comfortable with all the requirements? Have you eliminated requirements that are impossible to implement and included just to appease your customer or your boss?

APPENDIX F

Code-Complete Design Review Checklists

Adapted from **STEVEN C. MCCONNELL,** *CODE COMPLETE, 2ND ED.*

Source: https://github.com/janosgyerik/software-construction-notes/tree/master/checklists-all

86. CHECKLIST: ARCHITECTURE

86.1. SPECIFIC ARCHITECTURAL TOPICS

- ☐ Is the overall organization of the program clear, including a good architectural overview and justification?
- ☐ Are major building blocks well defined, including their areas of responsibility and their interfaces to other building blocks?
- ☐ Are all the functions listed in the requirements covered sensibly, by neither too many nor too few building blocks?
- ☐ Are the most critical classes described and justified?
- ☐ Is the data design described and justified?
- ☐ Is the database organization and content specified?
- ☐ Are all key business rules identified and their impact on the system described?
- ☐ Is a strategy for the user interface design described?
- ☐ Is the user interface modularized so that changes in it won't affect the rest of the program?
- ☐ Is a strategy for handling I/O described and justified?
- ☐ Are resource-use estimates and a strategy for resource management described and justified?
- ☐ Are the architecture's security requirements described?
- ☐ Does the architecture set space and speed budgets for each class, subsystem, or functionality area?
- ☐ Does the architecture describe how scalability will be achieved?
- ☐ Does the architecture address interoperability?
- ☐ Is a strategy for internationalization/localization described?
- ☐ Is a coherent error-handling strategy provided?

- ☐ Is the approach to fault tolerance defined (if any is needed)?
- ☐ Has technical feasibility of all parts of the system been established?
- ☐ Is an approach to overengineering specified?
- ☐ Are necessary buy-vs.-build decisions included?
- ☐ Does the architecture describe how reused code will be made to conform to other architectural objectives?
- ☐ Is the architecture designed to accommodate likely changes?
- ☐ Does the architecture describe how reused code will be made to conform to other architectural objectives?

86.2. GENERAL ARCHITECTURAL QUALITY

- ☐ Does the architecture account for all the requirements?
- ☐ Is any part over- or under-architected? Are expectations in this area set out explicitly?
- ☐ Does the whole architecture hang together conceptually?
- ☐ Is the top-level design independent of the machine and language that will be used to implement it?
- ☐ Are the motivations for all major decisions provided?
- ☐ Are you, as a programmer who will implement the system, comfortable with the architecture?

86.3. CHECKLIST: UPSTREAM PREREQUISITES

- ☐ Have you identified the kind of software project you're working on and tailored your approach appropriately?
- ☐ Are the requirements sufficiently well-defined and stable enough to begin construction (see the requirements checklist for details)?
- ☐ Is the architecture sufficiently well defined to begin construction (see the architecture checklist for details)?
- ☐ Have other risks unique to your particular project been addressed, such that construction is not exposed to more risk than necessary?

87. CHECKLIST: MAJOR CONSTRUCTION PRACTICES

87.1. CODING

- ☐ Have you defined coding conventions for names, comments, and formatting?
- ☐ Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, and so on?
- ☐ Have you identified your location on the technology wave and adjusted your approach to match? If necessary, have you identified how you will program into the language rather than being limited by programming in it?

87.2. TEAMWORK

- ☐ Have you defined an integration procedure, that is, have you defined the specific steps a programmer must go through before checking code into the master sources?

- ☐ Will programmers program in pairs, or individually, or some combination of the two?

87.3. QUALITY ASSURANCE

- ☐ Will programmers write test cases for their code before writing the code itself?
- ☐ Will programmers write unit tests for their code regardless of whether they write them first or last?
- ☐ Will programmers step through their code in the debugger before they check it in?
- ☐ Will programmers integration-test their code before they check it in?
- ☐ Will programmers review or inspect each others' code?

87.4. TOOLS

- ☐ Have you selected a revision control tool?
- ☐ Have you selected a language and language version or compiler version?
- ☐ Have you decided whether to allow use of non-standard language features?
- ☐ Have you identified and acquired other tools you'll be using editor, refactoring tool, debugger, test framework, syntax checker, and so on?

88. CHECKLIST: DESIGN IN CONSTRUCTION

88.1. DESIGN PRACTICES

- ☐ Have you iterated, selecting the best of several attempts rather than the first attempt?
- ☐ Have you tried decomposing the system in several different ways to see which way will work best?
- ☐ Have you approached the design problem both from the top down and from the bottom up?
- ☐ Have you prototyped risky or unfamiliar parts of the system, creating the absolute minimum amount of throwaway code needed to answer specific questions?
- ☐ Has you design been reviewed, formally or informally, by others?
- ☐ Have you driven the design to the point that its implementation seems obvious?
- ☐ Have you captured your design work using an appropriate technique such as a Wiki, email, flipcharts, digital camera, UML, CRC cards, or comments in the code itself?

88.2. DESIGN GOALS

- ☐ Does the design adequately address issues that were identified and deferred at the architectural level?
- ☐ Is the design stratified into layers?
- ☐ Are you satisfied with the way the program has been decomposed into subsystems, packages, and classes?
- ☐ Are you satisfied with the way the classes have been decomposed into routines?
- ☐ Are classes designed for minimal interaction with each other?
- ☐ Are classes and subsystems designed so that you can use them in other systems?
- ☐ Will the program be easy to maintain?
- ☐ Is the design lean? Are all of its parts strictly necessary?

- ☐ Does the design use standard techniques and avoid exotic, hard-to-understand elements?
- ☐ Overall, does the design help minimize both accidental and essential complexity?

89. CHECKLIST: CLASS QUALITY

89.1. ABSTRACT DATA TYPES

- ☐ Have you thought of the classes in your program as Abstract Data Types and evaluated their interfaces from that point of view?

89.2. ABSTRACTION

- ☐ Does the class have a central purpose?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface present a consistent abstraction?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- ☐ Are the class's services complete enough that other classes don't have to meddle with its internal data?
- ☐ Has unrelated information been moved out of the class?
- ☐ Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
- ☐ Are you preserving the integrity of the class's interface as you modify the class?

89.3. ENCAPSULATION

- ☐ Does the class minimize accessibility to its members?
- ☐ Does the class avoid exposing member data?
- ☐ Does the class hide its implementation details from other classes as much as the programming language permits?
- ☐ Does the class avoid making assumptions about its users, including its derived classes?
- ☐ Is the class independent of other classes? Is it loosely coupled?

89.4. INHERITANCE

- ☐ Is inheritance used only to model “is a” relationships?
- ☐ Does the class documentation describe the inheritance strategy?
- ☐ Do derived classes adhere to the Liskov Substitution Principle?
- ☐ Do derived classes avoid “overriding” non-overridable routines?
- ☐ Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- ☐ Are inheritance trees fairly shallow?
- ☐ Are all data members in the base class private rather than protected?

89.5. OTHER IMPLEMENTATION ISSUES

- ☐ Does the class contain about seven data members or fewer?

- ☐ Does the class minimize direct and indirect routine calls to other classes?
- ☐ Does the class collaborate with other classes only to the extent absolutely necessary?
- ☐ Is all member data initialized in the constructor?
- ☐ Is the class designed to be used as deep copies rather than shallow copies unless there's a measured reason to create shallow copies?

89.6. LANGUAGE-SPECIFIC ISSUES

- ☐ Have you investigated the language-specific issues for classes in your specific programming language?

90. CHECKLIST: THE PSEUDOCODE PROGRAMMING PROCESS

- ☐ Have you checked that the prerequisites have been satisfied?
- ☐ Have you defined the problem that the class will solve?
- ☐ Is the high level design clear enough to give the class and each of its routines a good name?
- ☐ Have you thought about how to test the class and each of its routines?
- ☐ Have you thought about efficiency mainly in terms of stable interfaces and readable implementations, or in terms of meeting resource and speed budgets?
- ☐ Have you checked the standard libraries and other code libraries for applicable routines or components?
- ☐ Have you checked reference books for helpful algorithms?
- ☐ Have you designed each routine using detailed pseudocode?
- ☐ Have you mentally checked the pseudocode? Is it easy to understand?
- ☐ Have you paid attention to warnings that would send you back to design (use of global data, operations that seem better suited to another class or another routine, and so on)?
- ☐ Did you translate the pseudocode to code accurately?
- ☐ Did you apply the PPP recursively, breaking routines into smaller routines when needed?
- ☐ Did you document assumptions as you made them?
- ☐ Did you remove comments that turned out to be redundant?
- ☐ Have you chosen the best of several iterations, rather than merely stopping after your first iteration?
- ☐ Do you thoroughly understand your code? Is it easy to understand?

91. CHECKLIST: A QUALITY-ASSURANCE PLAN

- ☐ Have you identified specific quality characteristics that are important to your project?
- ☐ Have you made others aware of the projects quality objectives?
- ☐ Have you differentiated between external and internal quality characteristics?
- ☐ Have you thought about the ways in which some characteristics may compete with or complement others?

- ☐ Does your project call for the use of several different error-detection techniques suited to finding several different kinds of errors?
- ☐ Does your project include a plan to take steps to assure software quality during each stage of software development?
- ☐ Is the quality measured in some way so that you can tell whether its improving or degrading?
- ☐ Does management understand that quality assurance incurs additional costs up front in order to save costs later?

92. CHECKLIST: EFFECTIVE PAIR PROGRAMMING

- ☐ Do you have a coding standard to support pair programming that's focused on programming rather than on philosophical coding-style discussions?
- ☐ Are both partners participating actively?
- ☐ Are you avoiding pair programming everything, instead selecting the assignments that will really benefit from pair programming?
- ☐ Are you rotating pair assignments and work assignments regularly?
- ☐ Are the pairs well matched in terms of pace and personality?
- ☐ Is there a team leader to act as the focal point for management and other people outside the project?

93. CHECKLIST: TEST CASES

- ☐ Does each requirement that applies to the class or routine have its own test case?
- ☐ Does each element from the design that applies to the class or routine have its own test case?
- ☐ Has each line of code been tested with at least one test case? Has this been verified by computing the minimum number of tests necessary to exercise each line of code?
- ☐ Have all defined-used data-flow paths been tested with at least one test case?
- ☐ Has the code been checked for data-flow patterns that are unlikely to be correct, such as defined-defined, defined-exited, and defined-killed?
- ☐ Has a list of common errors been used to write test cases to detect errors that have occurred frequently in the past?
- ☐ Have all simple boundaries been tested – maximum, minimum, and off-by-one boundaries?
- ☐ Have compound boundaries been tested – that is, combinations of input data that might result in a computed variable that is too small or too large?
- ☐ Do test cases check for the wrong kind of data – for example, a negative number of employees in a payroll program?
- ☐ Are representative, middle-of-the-road values tested?
- ☐ Is the minimum normal configuration tested?
- ☐ Is the maximum normal configuration tested?
- ☐ Is compatibility with old data tested? And are old hardware, old versions of the operating system, and interfaces with old versions of other software tested?

- ☐ Do the test cases make hand-checks easy?

94. CHECKLIST: DEBUGGING REMINDERS

94.1. TECHNIQUES FOR FINDING DEFECTS

- ☐ Use all the data available to make your hypothesis
- ☐ Refine the test cases that produce the error
- ☐ Exercise the code in your unit test suite
- ☐ Use available tools
- ☐ Reproduce the error several different ways
- ☐ Generate more data to generate more hypotheses
- ☐ Use the results of negative tests
- ☐ Brainstorm for possible hypotheses
- ☐ Narrow the suspicious region of the code
- ☐ Be suspicious of classes and routines that have had defects before
- ☐ Check code that's changed recently
- ☐ Expand the suspicious region of the code
- ☐ Integrate incrementally
- ☐ Check for common defects
- ☐ Talk to someone else about the problem
- ☐ Take a break from the problem
- ☐ Set a maximum time for quick and dirty debugging
- ☐ Make a list of brute force techniques, and use them

94.2. TECHNIQUES FOR SYNTAX ERRORS

- ☐ Don't trust line numbers in compiler messages
- ☐ Don't trust compiler messages
- ☐ Don't trust the compilers second message
- ☐ Divide and conquer
- ☐ Find extra comments and quotation marks

94.3. TECHNIQUES FOR FIXING DEFECTS

- ☐ Understand the problem before you fix it
- ☐ Understand the program, not just the problem
- ☐ Confirm the defect diagnosis
- ☐ Relax
- ☐ Save the original source code
- ☐ Fix the problem, not the symptom
- ☐ Change the code only for good reason

- ☐ Make one change at a time
- ☐ Check your fix
- ☐ Look for similar defects

94.4. GENERAL APPROACH TO DEBUGGING

- ☐ Do you use debugging as an opportunity to learn more about your program, mistakes, code quality, and problem-solving approach?
- ☐ Do you avoid the trial-and-error, superstitious approach to debugging?
- ☐ Do you assume that errors are your fault?
- ☐ Do you use the scientific method to stabilize intermittent errors?
- ☐ Do you use the scientific method to find defects?
- ☐ Rather than using the same approach every time, do you use several different techniques to find defects?
- ☐ Do you verify that the fix is correct?
- ☐ Do you use compiler warnings?

95. CHECKLIST: CODE-TUNING STRATEGY

95.1. OVERALL PROGRAM PERFORMANCE

- ☐ Have you considered improving performance by changing the program requirements?
- ☐ Have you considered improving performance by modifying the program's design?
- ☐ Have you considered improving performance by modifying the class design?
- ☐ Have you considered improving performance by avoiding operating system interactions?
- ☐ Have you considered improving performance by avoiding I/O?
- ☐ Have you considered improving performance by using a compiled language instead of an interpreted language?
- ☐ Have you considered improving performance by using compiler optimizations?
- ☐ Have you considered improving performance by switching to different hardware?
- ☐ Have you considered code tuning only as a last resort?

95.2. CODE-TUNING APPROACH

- ☐ Is your program fully correct before you begin code tuning?
- ☐ Have you measured performance bottlenecks before beginning code tuning?
- ☐ Have you measured the effect of each code-tuning change?
- ☐ Have you backed out the code-tuning changes that didn't produce the intended improvement?
- ☐ Have you tried more than one change to improve performance of each bottleneck, i.e., iterated?

96. CHECKLIST: CONFIGURATION MANAGEMENT

96.1. GENERAL

- ☐ Is your software-configuration-management plan designed to help programmers and minimize overhead?
- ☐ Does your SCM approach avoid overcontrolling the project?
- ☐ Do you group change requests, either through informal means such as a list of pending changes or through a more systematic approach such as a change-control board?
- ☐ Do you systematically estimate the effect of each proposed change?
- ☐ Do you view major changes as a warning that requirements development isn't yet complete?

96.2. TOOLS

- ☐ Do you use version-control software to facilitate configuration management?
- ☐ Do you use version-control software to reduce coordination problems of working in teams?

96.3. BACKUP

- ☐ Do you back up all project materials periodically?
- ☐ Are project backups transferred to off-site storage periodically?
- ☐ Are all materials backed up, including source code, documents, graphics, and important notes?
- ☐ Have you tested the backup-recovery procedure?

97. CHECKLIST: INTEGRATION

97.1. INTEGRATION STRATEGY

- ☐ Does the strategy identify the optimal order in which subsystems, classes, and routines should be integrated?
- ☐ Is the integration order coordinated with the construction order so that classes will be ready for integration at the right time?
- ☐ Does the strategy lead to easy diagnosis of defects?
- ☐ Does the strategy keep scaffolding to a minimum?
- ☐ Is the strategy better than other approaches?
- ☐ Have the interfaces between components been specified well? (Specifying interfaces isn't an integration task, but verifying that they have been specified well is.)

97.2. DAILY BUILD AND SMOKE TEST

- ☐ Is the project building frequently – ideally, daily to support incremental integration?
- ☐ Is a smoke test run with each build so that you know whether the build works?
- ☐ Have you automated the build and the smoke test?

- ☐ Do developers check in their code frequently – going no more than a day or two between check-ins?
- ☐ Is a broken build a rare occurrence?
- ☐ Do you build and smoke test the software even when you're under pressure?

98. CHECKLIST: PROGRAMMING TOOLS

- ☐ Do you have an effective IDE?
- ☐ Does your IDE support outline view of your program; jumping to definitions of classes, routines, and variables; source code formatting; brace matching or begin-end matching; multiple file string search and replace; convenient compilation; and integrated debugging?
- ☐ Do you have tools that automate common refactorings?
- ☐ Are you using version control to manage source code, content, requirements, designs, project plans, and other project artifacts?
- ☐ If you're working on a very large project, are you using a data dictionary or some other central repository that contains authoritative descriptions of each class used in the system?
- ☐ Have you considered code libraries as alternatives to writing custom code, where available?
- ☐ Are you making use of an interactive debugger?
- ☐ Do you use make or other dependency-control software to build programs efficiently and reliably?
- ☐ Does your test environment include an automated test framework, automated test generators, coverage monitors, system perturbers, diff tools, and defect tracking software?
- ☐ Have you created any custom tools that would help support your specific project's needs, especially tools that automate repetitive tasks?
- ☐ Overall, does your environment benefit from adequate tool support?

APPENDIX G

Design Review Rubric

This appendix describes the rating of design.

99. DOCUMENTATION

99.1. READABILITY RUBRIC

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Process	There is a clear process to guide design requirements and choices	There rules of thumb, and senior team members are mature guides.	There is no process. Members duplicate previous projects	There is no process. People make it up as they go a long
Follows guides / process	Process & style guidelines are followed correctly.	Process & style guidelines are almost always followed correctly.	Process & style guidelines are not followed. Style guide may be inadequate.	Does not follow process or does not match style guide; style guide may not exist.
Organization	The documentation is exceptionally well organized	The documentation is logically organized.	The documentation is poorly organized	The documentation is disorganized
Readability	The documentation is very easy to follow, understandable, is clean, and has no errors	The documentation is fairly easy to read. Minor issues with consistent naming, or general organization.	The documentation is readable only by someone who knows what it is supposed to be doing. At least one major issue with names, or organization.	The documentation is poorly organized and very difficult to read. Major problems with at names and organization.
Diagrams	Diagrams are clear and help understanding	Diagrams are mostly clear and do not sacrifice understanding	Diagrams are mostly confusing, overwrought, or junk	No diagrams used
Naming	All names follow naming conventions, are meaningful or expressive, and defined. Glossary is complete.	Names are mostly consistent in style and expressive. Isolated cases may be overly terse or ambiguous. No glossary	Names are often are cryptic or overly terse, ambiguous or misleading. No glossary.	Names are cryptic; items may be referred to by multiple different names or phrases. No glossary is given.

Table 41: *Readability rubric*

99.2. ORGANIZATION AND CLARITY

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Documentation	The documentation is well written and clearly explains what the documentation is accomplishing and how, at an appropriate level of detail. All required and most optional elements are present, and follows the prescribed format.	The documentation is not compelling; consists of embedded comment and some simple header documentation that is somewhat useful in understanding the documentation. All files, procedures, and structures are given an overview statement.	The documentation is simply comments embedded in the code and pretty-printed. Does little to help the reader understand the design.	No documentation.
Overview statement	The overview is given and explains what the documentation is accomplishing.	The overview is given, but is minimal and is only somewhat useful in understanding the documentation.	The overview is not given, or is not helpful in understanding what the documentation is to accomplish.	No overview is given.
Top-Down Design	Top-down design method followed and written in appropriate detail.	Top-down method followed, but level of detail is too vague or too exact.	Top-down design method attempted, but poorly executed.	No design.
Modularization & Generalization	The description is broken into well thought out elements that are of an appropriate length, scope and independence.	Documentation elements are generally well planned and executed. Some documentation is repeated. Individual elements are often, but not always, written in a way that invites reuse.	Documentation elements are not well thought out, are used in a somewhat arbitrary fashion, or do not improve clarity. Elements are seldom written in a way that invites reuse.	
Reusability	Individual elements were developed in a manner that actively invites reuse in other projects.	Most of the documentation could be reused in other projects.	Some parts of the documentation could be reused in other projects.	The documentation is not organized for reusability.
Design & Diagrams	A design tool or diagram is correctly used	A design or diagram tool is used but does not entirely match text	A design or diagram tool is used but is incorrect.	No design or diagram tool is used.
Identification	All identifying information is shown in the documentation	Some identifying information is shown.	Only a small portion of identifying information is shown, and/or is not correct.	No identifying information is shown.

Table 42: *Documentation organization and clarity rubric*

100. DESIGN

Table 43: Implementation rubric

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Overall design	The design is elegant, complete system	The design lacks some critical design components; simpler than comparable products	The design lacks many critical design components, is not simpler than comparable products	The design is lacking most or all design components, or is excessively complex
Understanding	Shows thorough understanding of the mission, the components, underlying techniques and science	Shows moderate understanding of the mission, the components, underlying techniques and science	Shows minimal understanding of the mission, the components, underlying techniques and science	Can't describe what the design will do, shows little knowledge of why some components are employed or understanding of what they do
Design, & Structure	The design proceeds in a clear and logical manner. Structures are used correctly. The most appropriate algorithms are used.	The design is mostly clear and logical. Structures are used correctly. Reasonable algorithms are employed.	The design isn't as clear or logical as it should be. Structures are occasionally used incorrectly. Portions are clearly inefficient or unnecessarily complicated.	The design is sparse or appears to be patched together. Requires significant effort to comprehend.
Modularization & Generalization	The design is broken into well thought out components that are of an appropriate scale, scope and independence.	Components are generally well planned and executed. Individual components are often, but not always, written in a way that invites reuse.	Components are not well thought out, are used in a somewhat arbitrary fashion, or do not improve clarity. Elements are seldom written in a way that invites reuse.	
Cohesion	All of the components look like they belong together.	Most of the components look like they belong together.	Some of the components look like they belong together.	Few components look like they belong together.
Reusability	Individual components were designed in a manner that actively invites reuse in other projects.	Most of the components could be reused in other projects.	Some parts of the design could be reused in other projects.	The design is not organized for reusability.
Efficiency	The design is extremely efficient, using the best approach in every case.	The design is fairly efficient at completing most tasks	The design uses poorly-chosen approaches in at least one place. For example, the documentation is brute force	Many things in the design could have been accomplished in an easier, faster, or otherwise better fashion.

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Correctness	Prioritization properly based on Rate Monotonic Analysis. Performs error checking in all cases. Appropriately bounded time checks are used in all cases. Resources are appropriately sized.	Has potential or obvious deadlocks. Some operations do not use time limits or use limits that are inappropriate. Does not check for error/lack of resources in some case. Has prioritization, based on ad hoc experience, not on analysis. Mutexes correctly used. Semaphores may overflow, or not wake task	Has obvious deadlocks. Does not use time limits on operations. Doesn't check for error, or lack of resources. Resource sizing is not based on analysis. Has prioritization, based on ad hoc experience, not on analysis. Semaphores or mutexes misused.	Has obvious deadlocks. Does not use time limits to operations. Doesn't check for error/lack of resources. Resource sizing is not based on analysis. No prioritization, not based on analysis
Problem Prevention	Communication / resource utilization has effective (or best in class) collision avoidance algorithms	Communication / resource utilization has some collision avoidance algorithm(s), but it is not always effective (or best in class)	Communication / resource utilization has poorly thought out collision avoidance approach	Communication / resource utilization has no collision avoidance algorithm
	Has fallback on collision, reducing further errors in all cases	Has fallback on collision, reducing further errors in most cases	Has fallback on collision, but fails to significantly reduces collisions	Has no fallback on collision
Safety	Controls have been identified from analysis such as SIL or FMEA. Device handles error/exception circumstances correctly. Device engages safe conditions in all cases. Internal state is monitored. External sate is monitored. Self-checks are performed correctly. Memory and other internal protection are employed.	Internal state, such as values and Buffers are checked. Output monitoring is employed. Self-test is not performed.	Some safe bounds are used. Some value/range checking is employed. Some output monitoring is employed.	No requirements, no analysis, no action.

APPENDIX H

Floating-point precision

This appendix summarizes the limits of precision employing a floating-point representation. Floats have some corner cases, and loss of precision.

- There is a -0 with floats. IEEE 754 requires that zeros be signed
- Floats can have signed infinity (+INF, and –INF)
- Floats can be NAN; there are several different encodings for NAN. (The exponent is zero, and significand is non-zero)
- Division by zero can throw exception, and/or give a NAN as a result
- Division by non-zero numbers can also give a NAN, such as *denormals.*
- Due to subtleties of precision and other factors, two floating point values must not be compared for equality or inequality using == or !=.
- Floats are not *associative*. The order of addition matters. Adding numbers in different orders can give differing results.
- Float values can be correctly sorted by treating the format as 32 bit integers.

Table 44: *Float range*

Parameter	Value
maximum value	3.402823×10^{38}
minimum value	-3.402823×10^{38}

Table 45: *Accuracy of integer values represented as a float*

From	To	Precision
−16777216	16777216	can be exactly represented
−33554432	−16777217	rounded to a multiple of two
16777217	33554432	rounded to a multiple of two
-2^{n+1}	$-2^{n}-1$	rounded to a multiple of 2^{n-23}; n > 22
$2^{n}+1$	2^{n+1}	rounded to a multiple of 2^{n-23}; n > 22
$-\infty$	2^{128}	rounded to -INF
2^{128}	∞	rounded to +INF

APPENDIX I

Code-Complete Code Review Checklists

Adapted from **STEVEN C. MCCONNELL,** *CODE COMPLETE, 2ND ED.*

Source: https://github.com/janosgyerik/software-construction-notes/tree/master/checklists-all

101. CHECKLIST: EFFECTIVE INSPECTIONS

- ☐ Do you have checklists that focus reviewer attention on areas that have been problems in the past?
- ☐ Is the emphasis on defect detection rather than correction?
- ☐ Are inspectors given enough time to prepare before the inspection meeting, and is each one prepared?
- ☐ Does each participant have a distinct role to play?
- ☐ Does the meeting move at a productive rate?
- ☐ Is the meeting limited to two hours?
- ☐ Has the moderator received specific training in conducting inspections?
- ☐ Is data about error types collected at each inspection so that you can tailor future checklists to your organization?
- ☐ Is data about preparation and inspection rates collected so that you can optimize future preparation and inspections?
- ☐ Are the action items assigned at each inspection followed up, either personally by the moderator or with a re-inspection?
- ☐ Does management understand that it should not attend inspection meetings?

102. CHECKLIST: HIGH-QUALITY ROUTINES

102.1. BIG-PICTURE ISSUES

- ☐ Is the reason for creating the routine sufficient?
- ☐ Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?

- ☐ Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- ☐ Does the routine's name describe everything the routine does?
- ☐ Have you established naming conventions for common operations?
- ☐ Does the routine have strong, functional cohesion – doing one and only one thing and doing it well?
- ☐ Do the routines have loose coupling – are the routine's connections to other routines small, intimate, visible, and flexible?
- ☐ Is the length of the routine determined naturally by its function and logic, rather than by an artificial coding standard?

102.2. PARAMETER-PASSING ISSUES

- ☐ Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- ☐ Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- ☐ Are interface assumptions documented?
- ☐ Does the routine have seven or fewer parameters?
- ☐ Is each input parameter used?
- ☐ Is each output parameter used?
- ☐ Does the routine avoid using input parameters as working variables?
- ☐ If the routine is a function, does it return a valid value under all possible circumstances?

103. CHECKLIST: DEFENSIVE PROGRAMMING

103.1. GENERAL

- ☐ Does the routine protect itself from bad input data?
- ☐ Have you used assertions to document assumptions, including preconditions and postconditions?
- ☐ Have assertions been used only to document conditions that should never occur?
- ☐ Does the architecture or high-level design specify a specific set of error handling techniques?
- ☐ Does the architecture or high-level design specify whether error handling should favor robustness or correctness?
- ☐ Have barricades been created to contain the damaging effect of errors and reduce the amount of code that has to be concerned about error processing?
- ☐ Have debugging aids been used in the code?
- ☐ Has information hiding been used to contain the effects of changes so that they won't affect code outside the routine or class that is changed?
- ☐ Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- ☐ Is the amount of defensive programming code appropriate – neither too much nor too little?
- ☐ Have you used offensive programming techniques to make errors difficult to overlook during development?

103.2. EXCEPTIONS

- ☐ Has your project defined a standardized approach to exception handling?

- ☐ Have you considered alternatives to using an exception?
- ☐ Is the error handled locally rather than throwing a non-local exception if possible?
- ☐ Does the code avoid throwing exceptions in constructors and destructors?
- ☐ Are all exceptions at the appropriate levels of abstraction for the routines that throw them?
- ☐ Does each exception include all relevant exception background information?
- ☐ Is the code free of empty catch blocks? (Or if an empty catch block truly is appropriate, is it documented?)

103.3. SECURITY ISSUES

- ☐ Does the code that checks for bad input data check for attempted buffer overflows, SQL injection, html injection, integer overflows, and other malicious inputs?
- ☐ Are all error-return codes checked?
- ☐ Are all exceptions caught?
- ☐ Do error messages avoid providing information that would help an attacker break into the system?

104. CHECKLIST: GENERAL CONSIDERATIONS IN USING DATA

104.1. INITIALIZING VARIABLES

- ☐ Does each routine check input parameters for validity?
- ☐ Does the code declare variables close to where they're first used?
- ☐ Does the code initialize variables as they're declared, if possible?
- ☐ Does the code initialize variables close to where they're first used, if it isn't possible to declare and initialize them at the same time?
- ☐ Are counters and accumulators initialized properly and, if necessary, reinitialized each time they are used?
- ☐ Are variables reinitialized properly in code that's executed repeatedly?
- ☐ Does the code compile with no warnings from the compiler?
- ☐ If your language uses implicit declarations, have you compensated for the problems they cause?

104.2. OTHER GENERAL ISSUES IN USING DATA

- ☐ Do all variables have the smallest scope possible?
- ☐ Are references to variables as close together as possible – both from each reference to a variable to the next and in total live time?
- ☐ Do control structures correspond to the data types?
- ☐ Are all the declared variables being used?
- ☐ Are all variables bound at appropriate times, that is, striking a conscious balance between the flexibility of late binding and the increased complexity associated with late binding?
- ☐ Does each variable have one and only one purpose?
- ☐ Is each variable's meaning explicit, with no hidden meanings?

105. CHECKLIST: NAMING VARIABLES

105.1. GENERAL NAMING CONSIDERATIONS

- ☐ Does the name fully and accurately describe what the variable represents?
- ☐ Does the name refer to the real-world problem rather than to the programming-language solution?
- ☐ Is the name long enough that you don't have to puzzle it out?
- ☐ Are computed-value qualifiers, if any, at the end of the name?
- ☐ Does the name use Count or Index instead of Num? Naming Specific Kinds Of Data
- ☐ Are loop index names meaningful (something other than i, j, or k if the loop is more than one or two lines long or is nested)?
- ☐ Have all “temporary” variables been renamed to something more meaningful?
- ☐ Are boolean variables named so that their meanings when they're True are clear?
- ☐ Do enumerated-type names include a prefix or suffix that indicates the category – for example, Color for Color Red, Color Green, Color Blue, and so on?
- ☐ Are named constants named for the abstract entities they represent rather than the numbers they refer to?

105.2. NAMING CONVENTIONS

- ☐ Does the convention distinguish among local, class, and global data?
- ☐ Does the convention distinguish among type names, named constants, enumerated types, and variables?
- ☐ Does the convention identify input-only parameters to routines in languages that don't enforce them?
- ☐ Is the convention as compatible as possible with standard conventions for the language?
- ☐ Are names formatted for readability? Short Names
- ☐ Does the code use long names (unless it's necessary to use short ones)?
- ☐ Does the code avoid abbreviations that save only one character?
- ☐ Are all words abbreviated consistently?
- ☐ Are the names pronounceable?
- ☐ Are names that could be mispronounced avoided?
- ☐ Are short names documented in translation tables?

105.3. COMMON NAMING PROBLEMS: HAVE YOU AVOIDED...

- ☐ ...names that are misleading?
- ☐ ...names with similar meanings?
- ☐ ...names that are different by only one or two characters?
- ☐ ...names that sound similar?
- ☐ ...names that use numerals?
- ☐ ...names intentionally misspelled to make them shorter?
- ☐ ...names that are commonly misspelled in English?
- ☐ ...names that conflict with standard library-routine names or with predefined variable names?

- ☐ ...totally arbitrary names?
- ☐ ...hard-to-read characters?

106. CHECKLIST: FUNDAMENTAL DATA

106.1. NUMBERS IN GENERAL

- ☐ Does the code avoid magic numbers?
- ☐ Does the code anticipate divide-by-zero errors?
- ☐ Are type conversions obvious?
- ☐ If variables with two different types are used in the same expression, will the expression be evaluated as you intend it to be?
- ☐ Does the code avoid mixed-type comparisons?
- ☐ Does the program compile with no warnings?

106.2. INTEGERS

- ☐ Do expressions that use integer division work the way they're meant to?
- ☐ Do integer expressions avoid integer-overflow problems?

106.3. FLOATING-POINT NUMBERS

- ☐ Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
- ☐ Does the code systematically prevent rounding errors?
- ☐ Does the code avoid comparing floating-point numbers for equality?

106.4. CHARACTERS AND STRINGS

- ☐ Does the code avoid magic characters and strings?
- ☐ Are references to strings free of off-by-one errors?
- ☐ Does C code treat string pointers and character arrays differently?
- ☐ Does C code follow the convention of declaring strings to be length constant+1?
- ☐ Does C code use arrays of characters rather than pointers, when appropriate?
- ☐ Does C code initialize strings to NULLs to avoid endless strings?
- ☐ Does C code use `strncpy()` rather than `strcpy()`? And `strncat()` and `strncmp()`?

106.5. BOOLEAN VARIABLES

- ☐ Does the program use additional boolean variables to document conditional tests?
- ☐ Does the program use additional boolean variables to simplify conditional tests?

106.6. ENUMERATED TYPES

- ☐ Does the program use enumerated types instead of named constants for their improved readability, reliability, and modifiability?
- ☐ Does the program use enumerated types instead of boolean variables when a variable's use cannot be completely captured with TRUE and FALSE?

- ☐ Do tests using enumerated types test for invalid values?
- ☐ Is the first entry in an enumerated type reserved for "invalid"?
- ☐ Named Constants
- ☐ Does the program use named constants for data declarations and loop limits rather than magic numbers?
- ☐ Have named constants been used consistently – not named constants in some places, literals in others?

106.7. ARRAYS

- ☐ Are all array indexes within the bounds of the array?
- ☐ Are array references free of off-by-one errors?
- ☐ Are all subscripts on multidimensional arrays in the correct order?
- ☐ In nested loops, is the correct variable used as the array subscript, avoiding loop-index cross talk?

106.8. CREATING TYPES

- ☐ Does the program use a different type for each kind of data that might change?
- ☐ Are type names oriented toward the real-world entities the types represent rather than toward programming language types?
- ☐ Are the type names descriptive enough to help document data declarations?
- ☐ Have you avoided redefining predefined types?
- ☐ Have you considered creating a new class rather than simply redefining a type?

107. CHECKLIST: CONSIDERATIONS IN USING UNUSUAL DATA TYPES

107.1. STRUCTURES

- ☐ Have you used structures instead of naked variables to organize and manipulate groups of related data?
- ☐ Have you considered creating a class as an alternative to using a structure?

107.2. GLOBAL DATA

- ☐ Are all variables local or class-scope unless they absolutely need to be global?
- ☐ Do variable naming conventions differentiate among local, class, and global data?
- ☐ Are all global variables documented?
- ☐ Is the code free of pseudo-global data-mammoth objects containing a mishmash of data that's passed to every routine?
- ☐ Are access routines used instead of global data?
- ☐ Are access routines and data organized into classes?
- ☐ Do access routines provide a level of abstraction beyond the underlying data-type implementations?
- ☐ Are all related access routines at the same level of abstraction?

107.3. POINTERS

- ☐ Are pointer operations isolated in routines?

- ☐ Are pointer references valid, or could the pointer be dangling?
- ☐ Does the code check pointers for validity before using them?
- ☐ Is the variable that the pointer references checked for validity before it's used?
- ☐ Are pointers set to NULL after they're freed?
- ☐ Does the code use all the pointer variables needed for the sake of readability?
- ☐ Are pointers in linked lists freed in the right order?
- ☐ Does the program allocate a reserve parachute of memory so that it can shut down gracefully if it runs out of memory?
- ☐ Are pointers used only as a last resort, when no other method is available?

108. CHECKLIST: ORGANIZING STRAIGHT LINE CODE

- ☐ Does the code make dependencies among statements obvious?
- ☐ Do the names of routines make dependencies obvious?
- ☐ Do parameters to routines make dependencies obvious?
- ☐ Do comments describe any dependencies that would otherwise be unclear?
- ☐ Have housekeeping variables been used to check for sequential dependencies in critical sections of code?
- ☐ Does the code read from top to bottom?
- ☐ Are related statements grouped together?
- ☐ Have relatively independent groups of statements been moved into their own routines?

109. CHECKLIST: CONDITIONALS

109.1. IF-THEN STATEMENTS

- ☐ Is the nominal path through the code clear?
- ☐ Do if-then tests branch correctly on equality?
- ☐ Is the else clause present and documented?
- ☐ Is the else clause correct?
- ☐ Are the if and else clauses used correctly – not reversed?
- ☐ Does the normal case follow the if rather than the else?
- ☐ if-then-else-if Chains
- ☐ Are complicated tests encapsulated in boolean function calls?
- ☐ Are the most common cases tested first?
- ☐ Are all cases covered?
- ☐ Is the if-then-else-if chain the best implementation – better than a case statement?
- ☐ case Statements
- ☐ Are cases ordered meaningfully?
- ☐ Are the actions for each case simple-calling other routines if necessary?
- ☐ Does the case statement test a real variable, not a phony one that's made up solely to use and abuse the case statement?
- ☐ Is the use of the default clause legitimate?

- ☐ Is the default clause used to detect and report unexpected cases?
- ☐ In C, C++, or Java, does the end of each case have a break?

110. CHECKLIST: LOOPS

110.1. LOOP SELECTION AND CREATION

- ☐ Is a while loop used instead of a for loop, if appropriate?
- ☐ Was the loop created from the inside out?

110.2. ENTERING THE LOOP

- ☐ Is the loop entered from the top?
- ☐ Is initialization code directly before the loop?
- ☐ If the loop is an infinite loop or an event loop, is it constructed cleanly rather than using a kludge such as for i = 1 to 9999?
- ☐ If the loop is a C++, C, or Java for loop, is the loop header reserved for loop-control code?

110.3. INSIDE THE LOOP

- ☐ Does the loop use { and } or their equivalent to prevent problems arising from improper modifications?
- ☐ Does the loop body have something in it? Is it nonempty?
- ☐ Are housekeeping chores grouped, at either the beginning or the end of the loop?
- ☐ Does the loop perform one and only one function – as a well-defined routine does?
- ☐ Is the loop short enough to view all at once?
- ☐ Is the loop nested to three levels or less?
- ☐ Have long loop contents been moved into their own routine?
- ☐ If the loop is long, is it especially clear?

110.4. LOOP INDEXES

- ☐ If the loop is a for loop, does the code inside it avoid monkeying with the loop index?
- ☐ Is a variable used to save important loop-index values rather than using the loop index outside the loop?
- ☐ Is the loop index an ordinal type or an enumerated type – not floating point?
- ☐ Does the loop index have a meaningful name?
- ☐ Does the loop avoid index cross talk?

110.5. EXITING THE LOOP

- ☐ Does the loop end under all possible conditions?
- ☐ Does the loop use safety counters – if you've instituted a safety-counter standard?
- ☐ Is the loop's termination condition obvious?
- ☐ If break or continue are used, are they correct?

111. CHECKLIST: UNUSUAL CONTROL STRUCTURES

111.1. RETURN

- ☐ Does each routine use return only when necessary?
- ☐ Do returns enhance readability?

111.2. RECURSION

- ☐ Does the recursive routine include code to stop the recursion?
- ☐ Does the routine use a safety counter to guarantee that the routine stops?
- ☐ Is recursion limited to one routine?
- ☐ Is the routine's depth of recursion within the limits imposed by the size of the program's stack?
- ☐ Is recursion the best way to implement the routine? Is it better than simple iteration?

111.3. GOTO

- ☐ Are gotos used only as a last resort, and then only to make code more readable and maintainable?
- ☐ If a goto is used for the sake of efficiency, has the gain in efficiency been measured and documented?
- ☐ Are gotos limited to one label per routine?
- ☐ Do all gotos go forward, not backward?
- ☐ Are all goto labels used?

112. CHECKLIST: TABLE DRIVEN METHODS

- ☐ Have you considered table-driven methods as an alternative to complicated logic?
- ☐ Have you considered table-driven methods as an alternative to complicated inheritance structures?
- ☐ Have you considered storing the table's data externally and reading it at run time so that the data can be modified without changing code?
- ☐ If the table cannot be accessed directly via a straightforward array index (as in the Age example), have your put the access-key calculation into a routine rather than duplicating the index calculation in the code?

113. CHECKLIST: CONTROL STRUCTURE ISSUES

- ☐ Do expressions use True and False rather than 1 and 0?
- ☐ Are boolean values compared to True and False implicitly?
- ☐ Are numeric values compared to their test values explicitly?
- ☐ Have expressions been simplified by the addition of new boolean variables and the use of boolean functions and decision tables?
- ☐ Are boolean expressions stated positively?
- ☐ Do pairs of braces balance?
- ☐ Are braces used everywhere they're needed for clarity?
- ☐ Are logical expressions fully parenthesized?
- ☐ Have tests been written in number-line order?

- ☐ Do Java tests uses a.equals(b) style instead of a == b when appropriate?
- ☐ Are null statements obvious?
- ☐ Have nested statements been simplified by retesting part of the conditional, converting to if-then-else or case statements, moving nested code into its own routine, converting to a more object-oriented design, or improved in some other way?
- ☐ If a routine has a decision count of more than 10, is there a good reason for not redesigning it?

114. REFACTORING

114.1. REASONS TO REFACTOR

- ☐ Code is duplicated
- ☐ A routine is too long
- ☐ A loop is too long or too deeply nested
- ☐ A class has poor cohesion
- ☐ A class interface does not provide a consistent level of abstraction
- ☐ A parameter list has too many parameters
- ☐ Changes within a class tend to be compartmentalized
- ☐ Changes require parallel modifications to multiple classes
- ☐ Inheritance hierarchies have to be modified in parallel
- ☐ Related data items that are used together are not organized into classes
- ☐ A routine uses more features of another class than of its own class
- ☐ A primitive data type is overloaded
- ☐ A class doesn't do very much
- ☐ A chain of routines passes tramp data
- ☐ A middle man object isn't doing anything
- ☐ One class is overly intimate with another
- ☐ A routine has a poor name
- ☐ Data members are public
- ☐ A subclass uses only a small percentage of its parents' routines
- ☐ Comments are used to explain difficult code
- ☐ Global variables are used
- ☐ A routine uses setup code before a routine call or takedown code after a routine call
- ☐ A program contains code that seems like it might be needed someday

114.2. DATA LEVEL REFACTORINGS

- ☐ Replace a magic number with a named constant
- ☐ Rename a variable with a clearer or more informative name
- ☐ Move an expression inline
- ☐ Replace an expression with a routine

- ☐ Introduce an intermediate variable
- ☐ Convert a multi-use variable to a multiple single-use variables
- ☐ Use a local variable for local purposes rather than a parameter
- ☐ Convert a data primitive to a class
- ☐ Convert a set of type codes to a class
- ☐ Convert a set of type codes to a class with subclasses
- ☐ Change an array to an object
- ☐ Encapsulate a collection
- ☐ Replace a traditional record with a data class

114.3. STATEMENT LEVEL REFACTORINGS

- ☐ Decompose a boolean expression
- ☐ Move a complex boolean expression into a well-named boolean function
- ☐ Consolidate fragments that are duplicated within different parts of a conditional
- ☐ Use break or return instead of a loop control variable
- ☐ Return as soon as you know the answer instead of assigning a return value within nested if-then-else statements
- ☐ Replace conditionals with polymorphism (especially repeated case statements)
- ☐ Create and use null objects instead of testing for null values
- ☐ Routine Level Refactorings
- ☐ Extract a routine
- ☐ Move a routine's code inline
- ☐ Convert a long routine to a class
- ☐ Substitute a simple algorithm for a complex algorithm
- ☐ Add a parameter
- ☐ Remove a parameter
- ☐ Separate query operations from modification operations
- ☐ Combine similar routines by parameterizing them
- ☐ Separate routines whose behavior depends on parameters passed in
- ☐ Pass a whole object rather than specific fields
- ☐ Pass specific fields rather than a whole object
- ☐ Encapsulate downcasting

114.4. CLASS IMPLEMENTATION REFACTORINGS

- ☐ Change value objects to reference objects
- ☐ Change reference objects to value objects
- ☐ Replace virtual routines with data initialization
- ☐ Change member routine or data placement
- ☐ Extract specialized code into a subclass

- ☐ Combine similar code into a superclass

114.5. CLASS INTERFACE REFACTORINGS

- ☐ Move a routine to another class
- ☐ Convert one class to two
- ☐ Eliminate a class
- ☐ Hide a delegate
- ☐ Replace inheritance with delegation
- ☐ Replace delegation with inheritance
- ☐ Remove a middle man
- ☐ Introduce a foreign routine
- ☐ Introduce a class extension
- ☐ Encapsulate an exposed member variable
- ☐ Remove Set() routines for fields that cannot be changed
- ☐ Hide routines that are not intended to be used outside the class
- ☐ Encapsulate unused routines
- ☐ Collapse a superclass and subclass if their implementations are very similar

114.6. SYSTEM LEVEL REFACTORINGS

- ☐ Duplicate data you can't control
- ☐ Change unidirectional class association to bidirectional class association
- ☐ Change bidirectional class association to unidirectional class association
- ☐ Provide a factory routine rather than a simple constructor
- ☐ Replace error codes with exceptions or vice versa

114.7. CHECKLIST: REFACTORING SAFELY

- ☐ Is each change part of a systematic change strategy?
- ☐ Did you save the code you started with before beginning refactoring?
- ☐ Are you keeping each refactoring small?
- ☐ Are you doing refactorings one at a time?
- ☐ Have you made a list of steps you intend to take during your refactoring?
- ☐ Do you have a parking lot so that you can remember ideas that occur to you mid-refactoring?
- ☐ Have you retested after each refactoring?
- ☐ Have changes been reviewed if they are complicated or if they affect mission-critical code?
- ☐ Have you considered the riskiness of the specific refactoring, and adjusted your approach accordingly?
- ☐ Does the change enhance the program's internal quality rather than degrading it?
- ☐ Have you avoided using refactoring as a cover for code and fix or as an excuse for not rewriting bad code?

115. CHECKLIST: CODE-TUNING TECHNIQUES

115.1. IMPROVE BOTH SPEED AND SIZE

- ☐ Substitute table lookups for complicated logic
- ☐ Jam loops
- ☐ Use integer instead of floating-point variables
- ☐ Initialize data at compile time
- ☐ Use constants of the correct type
- ☐ Precompute results
- ☐ Eliminate common subexpressions
- ☐ Translate key routines to assembler

115.2. IMPROVE SPEED ONLY

- ☐ Stop testing when you know the answer
- ☐ Order tests in case statements and if-then-else chains by frequency
- ☐ Compare performance of similar logic structures
- ☐ Use lazy evaluation
- ☐ Unswitch loops that contain if tests
- ☐ Unroll loops
- ☐ Minimize work performed inside loops
- ☐ Use sentinels in search loops
- ☐ Put the busiest loop on the inside of nested loops
- ☐ Reduce the strength of operations performed inside loops
- ☐ Change multiple-dimension arrays to a single dimension
- ☐ Minimize array references
- ☐ Augment data types with indexes
- ☐ Cache frequently used values
- ☐ Exploit algebraic identities
- ☐ Reduce strength in logical and mathematical expressions
- ☐ Be wary of system routines
- ☐ Rewrite routines in line

116. CHECKLIST: LAYOUT

116.1. GENERAL

- ☐ Is formatting done primarily to illuminate the logical structure of the code?
- ☐ Can the formatting scheme be used consistently?
- ☐ Does the formatting scheme result in code that's easy to maintain?
- ☐ Does the formatting scheme improve code readability?

116.2. CONTROL STRUCTURES

- ☐ Does the code avoid doubly indented begin-end or {} pairs?
- ☐ Are sequential blocks separated from each other with blank lines?
- ☐ Are complicated expressions formatted for readability?
- ☐ Are single-statement blocks formatted consistently?
- ☐ Are case statements formatted in a way that's consistent with the formatting of other control structures?
- ☐ Have gotos been formatted in a way that makes their use obvious?

116.3. INDIVIDUAL STATEMENTS

- ☐ Is white space used to make logical expressions, array references, and routine arguments readable?
- ☐ Do incomplete statements end the line in a way that's obviously incorrect?
- ☐ Are continuation lines indented the standard indentation amount?
- ☐ Does each line contain at most one statement?
- ☐ Is each statement written without side effects?
- ☐ Is there at most one data declaration per line?

116.4. COMMENTS

- ☐ Are the comments indented the same number of spaces as the code they comment?
- ☐ Is the commenting style easy to maintain?

116.5. ROUTINES

- ☐ Are the arguments to each routine formatted so that each argument is easy to read, modify, and comment?
- ☐ Are blank lines used to separate parts of a routine?

116.6. CLASSES, FILES AND PROGRAMS

- ☐ Is there a one-to-one relationship between classes and files for most classes and files?
- ☐ If a file does contain multiple classes, are all the routines in each class grouped together and is the class clearly identified?
- ☐ Are routines within a file clearly separated with blank lines?
- ☐ In lieu of a stronger organizing principle, are all routines in alphabetical sequence?

117. CHECKLIST: GOOD COMMENTING TECHNIQUE

117.1. GENERAL

- ☐ Can someone pick up the code and immediately start to understand it?
- ☐ Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
- ☐ Is the Pseudocode Programming Process used to reduce commenting time?
- ☐ Has tricky code been rewritten rather than commented?

- ☐ Are comments up to date?
- ☐ Are comments clear and correct?
- ☐ Does the commenting style allow comments to be easily modified?

117.2. STATEMENTS AND PARAGRAPHS

- ☐ Does the code avoid endline comments?
- ☐ Do comments focus on why rather than how?
- ☐ Do comments prepare the reader for the code to follow?
- ☐ Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- ☐ Are surprises documented?
- ☐ Have abbreviations been avoided?
- ☐ Is the distinction between major and minor comments clear?
- ☐ Is code that works around an error or undocumented feature commented?

117.3. DATA DECLARATIONS

- ☐ Are units on data declarations commented?
- ☐ Are the ranges of values on numeric data commented?
- ☐ Are coded meanings commented?
- ☐ Are limitations on input data commented?
- ☐ Are flags documented to the bit level?
- ☐ Has each global variable been commented where it is declared?
- ☐ Has each global variable been identified as such at each usage, by a naming convention, a comment, or both?
- ☐ Are magic numbers replaced with named constants or variables rather than just documented?

117.4. CONTROL STRUCTURES

- ☐ Is each control statement commented?
- ☐ Are the ends of long or complex control structures commented or, when possible, simplified so that they don't need comments?

117.5. ROUTINES

- ☐ Is the purpose of each routine commented?
- ☐ Are other facts about each routine given in comments, when relevant, including input and output data, interface assumptions, limitations, error corrections, global effects, and sources of algorithms?

117.6. FILES, CLASSES, AND PROGRAMS

- ☐ Does the program have a short document such as that described in the Book Paradigm that gives an overall view of how the program is organized?
- ☐ Is the purpose of each file described?

- ☐ Are the author's name, email address, and phone number in the listing?

118. CHECKLIST: SELF-DOCUMENTING CODE

118.1. CLASSES

- ☐ Does the class's interface present a consistent abstraction?
- ☐ Is the class well named, and does its name describe its central purpose?
- ☐ Does the class's interface make obvious how you should use the class?
- ☐ Is the class's interface abstract enough that you don't have to think about how its services are implemented?
- ☐ Can you treat the class as a black box?

118.2. ROUTINES

- ☐ Does each routine's name describe exactly what the routine does?
- ☐ Does each routine perform one well-defined task?
- ☐ Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
- ☐ Is each routine's interface obvious and clear?

118.3. DATA NAMES

- ☐ Are type names descriptive enough to help document data declarations?
- ☐ Are variables named well?
- ☐ Are variables used only for the purpose for which they're named?
- ☐ Are loop counters given more informative names than i, j, and k?
- ☐ Are well-named enumerated types used instead of makeshift flags or boolean variables?
- ☐ Are named constants used instead of magic numbers or magic strings?
- ☐ Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

118.4. DATA ORGANIZATION

- ☐ Are extra variables used for clarity when needed?
- ☐ Are references to variables close together?
- ☐ Are data types simple so that they minimize complexity?
- ☐ Is complicated data accessed through abstract access routines (abstract data types)?

118.5. CONTROL

- ☐ Is the nominal path through the code clear?
- ☐ Are related statements grouped together?
- ☐ Have relatively independent groups of statements been packaged into their own routines?
- ☐ Does the normal case follow the if rather than the else?

- ☐ Are control structures simple so that they minimize complexity?
- ☐ Does each loop perform one and only one function, as a well-defined routine would?
- ☐ Is nesting minimized?
- ☐ Have boolean expressions been simplified by using additional boolean variables, boolean functions, and decision tables?

118.6. LAYOUT

- ☐ Does the program's layout show its logical structure?

118.7. DESIGN

- ☐ Is the code straightforward, and does it avoid cleverness?

- ☐ Are implementation details hidden as much as possible?
- ☐ Is the program written in terms of the problem domain as much as possible rather than in terms of computer-science or programming-language structures?

APPENDIX J

Code Review Rubric

This appendix describes the rating of source code workmanship.

Note: This was inspired by numerous sources including the First Lego League Coaches Handlbook, and school grading rubrics.

119. SOFTWARE READABILITY RUBRIC

Table 46: Readability rubric

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Coding Style	Coding style is checked at code reviews, automated tools are used to ensure consistent formatting; audits		Program manager has ensured that the development team has followed a set of coding standards	No check that software team has and is following coding standards
Consistency	Coding style guidelines are followed correctly.	Coding style guidelines are almost always followed correctly.	Coding style guidelines are not followed. Style guide may be inadequate.	Does not match style guide; style guide may not exist.
Organization	The code is exceptionally well organized	The code is logically organized.	The code is poorly organized	The code is disorganized
Readability	The code is very easy to follow, understandable, is clean, is easy to maintain, and has no errors	The code is fairly easy to read. Minor issues with consistent indentation, use of whitespace, variable naming, or general organization.	The code is readable only by someone who knows what it is supposed to be doing. At least one major issue with indentation, whitespace, variable names, or organization.	The code is poorly organized and very difficult to read. Major problems with at three or four of the readability subcategories.
Indentation / white spaces	Indentation and whitespace follows coding style, and is not distracting.	Minor issues with consistent indentation, use of whitespace.	At least one major issue with indentation, whitespace.	The code is poorly organized and very difficult to read.
Naming	All names follow naming conventions, are meaningful or expressive without being verbose, and documented. Data dictionary is complete.	Names are mostly consistent in style and expressive. Isolated cases may be verbose, overly terse or ambiguous. No data dictionary	Names are occasionally verbose, but often are cryptic or overly terse, ambiguous or misleading. No data dictionary.	Variable names are cryptic and no data dictionary is shown.

120. SOFTWARE COMMENTS & DOCUMENTATION

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Comments	Code is well-commented.	One or two places that could benefit from comments are missing them or the code is overly commented.	File header missing, lack of comments or meaningful comments.	No file header or comments present.
Initial Comments	Initial comments are complete. Internal documentation is complete and well suited to the program	Initial comments are complete but internal documentation is in some small fashion inadequate.	Initial comments are incomplete or internal documentation is inadequate.	No internal documentation
Coding Comments	Every line is commented. Comments clarify meaning.	Many comments are present, in correct format. Comments usually clarity meaning. Unhelpful comments may exist.	Some comments exist, but are frequently unhelpful or occasionally misleading; may use an incorrect format. Complicated lines or sections of code uncommented or lacking meaningful comments. Comments do not help the reader understand the code.	No comments
Documentation	The documentation is well written and clearly explains what the code is accomplishing and how, at an appropriate level of detail. All required and most optional elements are present, and follows the prescribed format.	The documentation is not compelling; consists of embedded comment and some simple header documentation that is somewhat useful in understanding the code. All files, procedures, and structures are given an overview statement.	The documentation is simply comments embedded in the code with some header comments separating routines. Does little to help the reader understand the code.	No documentation. There might be comments embedded in the code with some simple header comments separating routines. Does not help the reader understand the code.
Overview statement	The overview is given and explains what the code is accomplishing.	The overview is given, but is minimal and is only somewhat useful in understanding the code.	The overview is not given, or is not helpful in understanding what the code is to accomplish.	No overview is given.
Top-Down Design	Top-down design method followed and written in appropriate detail.	Top-down method followed, but level of detail is too vague or too exact.	Top-down design method attempted, but poorly executed.	No design.
Design & Diagrams	A design tool or diagram is correctly used	A design or diagram tool is used but does not entirely match	A design or diagram tool is used but is incorrect.	No design or diagram tool is used.

Table 47: *Comments and documentation rubric*

		code		
Identification	All identifying information is shown in the documentation	Some identifying information is shown.	Only a small portion of identifying information is shown, and/or is not correct.	No identifying information is shown.

121. IMPLEMENTATION

Table 48: Implementation rubric

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Syntax/runtime /logic errors	The program contains no errors.	The program has no major errors.	Program executes but has errors.	Program does not execute.
Modularization & Generalization	Program is broken into well thought out elements that are of an appropriate length, scope and independence.	Code elements are generally well planned and executed. Some code is repeated that should be encapsulated. Individual elements are often, but not always, written in a way that invites code reuse.	Code elements are not well thought out, are used in a somewhat arbitrary fashion, or do not improve program clarity. Elements are seldom written in a way that invites code reuse.	
Reusability	Individual elements were developed in a manner that actively invites reuse in other projects.	Most of the routines could be reused in other programs.	Some parts of the code could be reused in other programs.	The code is not organized for reusability.
Design, & Structure	Program is designed in a clear and logical manner. Control structures are used correctly. The most appropriate algorithms are used, in a manner that does not sacrifice readability or understanding	Program is mostly clear and logical. Control structures are used correctly. Reasonable algorithms are implemented, in a manner that does not sacrifice readability or understanding	Program isn't as clear or logical as it should be. Control structures are occasionally used incorrectly. Steps that are clearly inefficient or unnecessarily long are used.	The code is huge and appears to be patched together. Requires significant effort to comprehend.
Emulation	has a whole system emulation	can emulate significant parts, individually	in concept could emulate	no emulation
Efficiency	The code is extremely efficient, using the best approach in every case.	The code is fairly efficient at completing most tasks	Code uses poorly-chosen approaches in at least one place. For example, the code is brute force	Many things in the code could have been accomplished in an easier, faster, or otherwise better fashion.
Consistency	Program behaves in a consistent, predictable fashion, even for complex tasks	Mostly predictable	Somewhat unpredictable	unpredictable

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Operating bounds	The code and design has been reviewed by independent experts for arithmetic issues. Appropriate analysis tools have been used. A sizable body of test cases and tests have been applied against the code.	The code and design has been reviewed by independent experts for resource arithmetic issues.	The Designer has ensured that the implementation is not vulnerable to arithmetic issues.	No one has checked for arithmetic issues
	The code and design has been reviewed by independent experts for buffer overflow issues. Appropriate analysis tools have been used. A sizable body of test cases and tests have been applied against the code.	The code and design has been reviewed by independent experts for buffer overflow issues.	The Designer has ensured that the implementation is not vulnerable to buffer overflow issues.	No one has checked for overflow issues
	The code and design has been reviewed by independent experts for resource exhaustion. Appropriate analysis tools have been used. A sizable body of test cases and tests have been applied against the code.	The code and design has been reviewed by independent experts for resource exhaustion issues.	The Designer has ensured that the implementation is not vulnerable to resource exhaustion issues.	No one has checked for overflow issues
	The code and design has been reviewed by independent experts for race conditions. Appropriate analysis tools have been used. A sizable body of test cases and tests have been applied against the code.	The code and design has been reviewed by independent experts for race conditions	The Designer has ensured that the implementation is not vulnerable to race conditions.	No one has checked for race conditions

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Correctness	Prioritization properly based on Rate Monotonic Analysis. Performs error checking in all cases. Appropriately bounded time checks are used in all cases. Resources are appropriately sized.	Has potential or obvious deadlocks. Some operations do not usetime limits or use limits that are too long. Does not check for error/lack of resources in some case. Has prioritization, based on ad hoc experience, not on analysis. Mutexes correctly used. Semaphores may overflow, or not wake task	Has obvious deadlocks. Does not use time limits on operations. Doesn't check for error, or lack of resources. Resource sizing is not based on analysis. Has prioritization, based on ad hoc experience, not on analysis. Semaphores or mutexes misused.	Has obvious deadlocks. Does not use time limits to operations. Doesn't check for error/lack of resources. Resource sizing is not based on analysis. No prioritization, not based on analysis
Problem Prevention	Communication / resource utilization has effective (or best in class) collision avoidance algorithms	Communication / resource utilization has some collision avoidance algorithm(s), but it is not always effective (or best in class)	Communication / resource utilization has poorly thought out collision avoidance approach	Communication / resource utilization has no collision avoidance algorithm
	Has fallback on collision, reducing further errors in all cases	Has fallback on collision, reducing further errors in most cases	Has fallback on collision, but fails to significantly reduces collisions	Has no fallback on collision
Safety	Controls have been identified from analysis such as SIL or FMEA. Device handles error/exception circumstances correctly. Device engages safe conditions in all cases. Internal state is monitored. External sate is monitored. Self-checks are performed correctly. Memory and other internal protection are employed.	Internal state, such as values and Buffers are checked. Output monitoring is employed. Self-test is not performed.	Some safe bounds are used. Some value/range checking is employed. Some output monitoring is employed.	No requirements, no analysis, no action.

122. ERROR HANDLING

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Robustness	Program handles erroneous or unexpected input gracefully; action is taken without surprises	All obvious error conditions are checked for and appropriate action is taken.	Some, but not sufficient portion of, obvious error conditions are checked for with an appropriate action is taken.	Many obvious error conditions are not checked. Or, if checked, appropriate action is not taken.
PID Control	Is stable and free of oscillation (low and high frequency) for all manner of conditions and disturbances	Is stable and free of oscillation for most conditions and disturbances; may have some high-pitch whine or oscillation for boundary conditions	Is occasionally approximately correct, frequently has oscillation or is easily disturbed	Has high oscillation, high degree of error.
Testing	Testing is complete without being redundant. All boundary cases are considered and tested.	All key items are tested, but testing may be redundant. Nearly all boundary cases are considered and tested.	Testing was done, but is not sufficiently complete. Most boundary cases are considered and tested.	Testing has not been done

Table 49: *Error handling rubric*

123. BEHAVIOUR

Trait	Exceptional	Acceptable	Unsatisfactory	Needs improvement
Analysis of comm and IPC network	full structural analysis of all software systems as a network	Structural analysis of only IPC, or communication section.	structural analysis of one unit software	no structural analysis
Communication overload	Handles overload in a graceful fashion, with predicable/defined behaviour, including honoring time bounds, priority order of responses to messages, and dropping messages & disabling services.	Thrashes on overload. Inefficient slow responses	Communication fails; does not hold safe state; is not responsive; crashes, or sends erroneous behaviour. Runs out of resources. Critical behaviours are missed.	crashes on overload
Interrupt / Event overload	Handles overload in a graceful fashion, with predicable/defined behaviour, including honoring time bounds, priority order of responses and dropping messages & disabling services.	Thrashes on overload. Inefficient slow responses	Communication fails; does not hold safe state; is not responsive; crashes, or sends erroneous behaviour. Runs out of resources. Critical behaviours are missed.	crashes on overload

Table 50: *Behaviour rubric*

References & Resources

Note: most references appear in the margins, significant references will appear at the end of their respective chapter.

124. REFERENCE DOCUMENTATION AND RESOURCES

124.1. OVERALL SOFTWARE CRAFTSMANSHIP

McConnell, Steve *"Code Complete"* 2ed 2004

IEEE Computer Society, *SWEBOK Guide to the Software Engineering Body of Knowledge*, version 3, 2014

IEEE Std 1044-2009 *IEEE Standard Classification for Software Anomalies*, IEEE-SA Standards Board, 2009 Nov 9

124.2. SOFTWARE SAFETY

Joint Software Systems Safety Committee, "*Software System Safety Handbook,"* 2000-Dec

Joint Software Systems Safety Engineering Workgroup, "*Joint Software Systems Safety Engineering Handbook,"* Rev 1 2010-Aug-27

> While both cover much the same material – although the second has more material. I prefer the style of the earlier edition.

MOD Defence Standard 0058 *Requirements for Safety Related Software in Defence Equipment*. 1996 UK Ministry of Defence

MOD Interim Defence Standard 08-58 *Issues 1: HAZOP Studies on Systems Containing Programmable Electronics* 1996 UK Ministry of Defence

SAE ARP 4761 *Guidelines and methods for conducting the safety assessment process on Civil Airborne Systems and Equipment*. 1996 Society of Automotive Engineers.

UCRL-ID-122514, Lawrence, J Dennis *"Software Safety Hazard Analysis"* Rev 2, U.S. Nuclear Regulatory Commission, 1995-October

124.3. OTHER

ISO/IEC/IEEE 60559:2011 "*Information technology – Microprocessor Systems – Floating-Point arithmetic*"

Miktijuk et al, V.G. Mikitjuk, V.N. Yarmolik, A.J. van de Goor, *RAM Testing Algorithm for Detection Linked Coupling Faults,* IEEE 1996

V

"Optimally, what is needed is something that can be added to airplanes and other systems which weighs nothing, yet is very costly, and violates none of the physical laws of the universe, such as the law of gravitation or the laws of thermodynamic.

This might appear to be an insurmountable challenge; however, as a result of the traditional ingenuity characteristic of system designers, it can be reported with confidence that such an ingredient has already been found.

It is called software."
– Norman Ralph Augustine, (1970s)