**An Introduction to
Safety Critical Systems**

Executive Summary

This paper provides an introduction to the development of software for safety critical systems. It is aimed to serve as a tutorial for developers who are new to the development of software for safety critical systems, discussing the issues involved, introducing some of the techniques available to developers, and providing an overview of how AdaTEST and Cantata can be used to assist with the development of software for safety critical systems.

IPL is an independent software house founded in 1979 and based in Bath. The company provides a comprehensive range of software services and also supplies the AdaTEST and Cantata software testing and verification packages. IPL was accredited to ISO9001 in 1988, and gained TickIT accreditation in 1991. AdaTEST and Cantata have been produced to these standards.

1. What is a Safety Critical System?

A safety critical system is a system where human safety is dependent upon the correct operation of the system. The emphasis of this paper is on the software element of safety critical systems, which for convenience is often referred to as safety critical software. However, safety must always be considered with respect to the whole system, including software, computer hardware, other electronic and electrical hardware, mechanical hardware, and operators or users, not just the software element.

Safety critical software has been traditionally associated with embedded control systems. As awareness of how systems can impact safety has developed, the scope of safety critical software has expanded into many other types of systems.

An obvious example of a safety critical system is an aircraft fly by wire control system, where the pilot inputs commands to the control computer using a joystick, and the computer manipulates the actual aircraft controls. The lives of hundreds of passengers is totally dependent upon the continued correct operation of such a system.

Moving down to earth, railway signalling systems must enable controllers to direct trains, while preventing trains from colliding. Like an aircraft fly by wire, lives are dependent upon the correct operation of the system. However, there is always the option of stopping all trains if the integrity of the system becomes suspect. You can't just stop an aircraft while the fly by wire system is fixed!

Software in medical systems may be directly responsible for human life, such as metering safe amounts of X-rays. Software may also be involved in providing humans with information, such as information which a doctor uses to decide on medication. Both types of system can impact the safety of the patient.

Big civil engineering structures are designed on computers and tested using mathematical models. An error in the software could conceivably result in a bridge collapsing. Aircraft, trains, ships and cars are also designed and modelled using computers.

Even something a simple as traffic lights can be viewed as safety critical. An error giving green lights to both directions at a cross road could result in a car accident. Within cars, software involved in functions such as engine management, anti-lock brakes, traction control, and a host of other functions, could potentially fail in a way which increases the likelihood of a road accident.

We can see from these examples that many everyday systems can have an impact on safety. However, the fact is that software safety has not been a special consideration in the development of many everyday systems. As responsible engineers, we should be asking three questions about every system we develop:

- Can it present hazards to safety?

- What can we do to reduce hazards to an acceptable level?

- How can we verify that the developed system is safe?

Collectively, the documented process of answering these questions and applying our answers to the system development is referred to as the **safety case**. A well conceived and executed safety case is a key element in bringing a safety critical system into use. In areas which have been traditionally concerned with safety critical systems, such as the aviation industry and the nuclear industry, a certification body will have to be convinced

that a system is safe before a system can enter use. In some other areas, users have their own safety monitoring groups. Nevertheless, the vast majority of software safety is entirely in the hands and conscience of the software developers and suppliers.

2. Integrity Levels and Standards for Safety Critical Systems

The first step in developing a system is perform a preliminary hazard analysis, to determine whether the system could present a hazard to safety. If the answer is no, then hazard analysis need go no further, other than to periodically review the validity of this decision. If the answer is yes, we must conduct a more detailed hazard analysis:

- How likely is it that an error in the system will result in a particular hazard?

- How likely is the hazard to actually cause an accident?

- What is the likely magnitude of the accident in terms of injuries or deaths?

The collective results of these questions must be weighed against an ethical judgement; what is an acceptable level of safety? A reasonable starting position is that a software based system should be at least as safe as any system it replaces.

While it is good practice to make all software as reliable as possible, not all errors will lead to safety hazards. To illustrate this point, consider a medical system providing diagnostic information. If the software failed by giving a misleading output, a patient could be given the wrong medication. However, if it failed by crashing the computer, users would avoid making such a mistake. One error presents a direct hazard to safety, whilst the other error affects system availability, but not safety.

The concept of "safety critical" is not absolute; failure of some systems will not impact safety, failure of other systems could occasionally result in minor injuries, and failure of some systems could lead to disasters. The level of safety integrity required varies from none through to a very high level of integrity.

Standards for safety critical software (table 1) have now standardised on a scale of five discrete levels of safety integrity, with an integrity level of 4 being "very high", down to a level of 0 for a system which is not safety related. The term "safety related" is used to collectively refer to integrity levels 1 to 4. Further analysis will assign an integrity level to each component of a system, including the software.

Standard	Description
Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing. ISO9001/EN29001/BS5750 part 1	This is the recommended minimum standard of quality system for software with a safety integrity level of 0, and an essential prerequisite for higher integrity levels.
Functional Safety : Safety Related Systems IEC1508	A general standard, which sets the scene for most other safety related software standards.
Railway Applications: Software for Railway Control & Protection Systems. EN50128	A standard for the railway industry.
Software for Computers in the Safety Systems of Nuclear Powers Stations.	A standard for the nuclear industry.

3

Standard	Description
IEC880	
Software Considerations in Airborne Systems and Equipment Certification. RTCA/DO178B	A standard for avionics and airborne systems.
MISRA Development Guidelines for Behicle Based Software	Issued by the Motor Industry Software Reliability Association for automotive software.
Safety Management Considerations for Defence Systems Containing Programmable Electronics. Defence Standard 00-56	A standard for the defence industry.
The Procurement of Safety Critical Software in Defence Equipment. Defence Standard 00-55	Detailed software standard for safety critical defence equipment.

Table 1 - Relevant Standards

Differing constraints are placed on the methods and techniques used through each stage of the development lifecycle, depending on the required level of safety integrity. For example, formal mathematical methods are "highly recommended" by most standards at integrity level 4, but are not required at integrity level 0 or 1. The required integrity level can consequently have a major impact on development costs, making it important not to assign an unnecessarily high integrity level to a system or any component of a system. This is not just limited to deliverable software. The integrity of software development tools, test software and other support software may also have an impact on safety.

3. Specification and Design of Safety Critical Systems

The general principle when designing any safety critical system is to keep it as simple as possible, taking no unnecessary risks. There is less that can go wrong with a simple system. From a software point of view, this usually involves minimising the use of interrupts and minimising concurrency within the software. Use of concurrency also necessitates the use of an executive or run time environment to manage the concurrency, which is unlikely to have been produced to safety critical standards. Ideally, a safety critical system requiring a high integrity level would have no interrupts and only one task. However, this is not always achievable in practice.

There are two distinct philosophies for the specification and design of safety critical systems.

- To specify and design a "perfect" system, which cannot go wrong because there are no faults in it, and to prove that there are no faults in it.

- To aim for the first philosophy, but to accept that mistakes may have been made, and to include error detection and recovery capabilities to prevent errors from actually causing a hazard to safety.

The first of these approaches can work well for small systems, which are sufficiently compact for formal mathematical methods to be used in the specification and design, and for formal mathematical proof of design correctness to be established. However, formal mathematical specification of larger and more complex systems is difficult, with human error in the specification or proof becoming a significant problem. Tools to assist with

formal mathematical methods are now becoming available, which will enable the effective application of formal mathematical methods to larger systems in the future.

The second philosophy, of accepting that no matter how careful we are in developing a system, that it could still contain errors, is the approach more generally adopted. This philosophy can be applied at a number of levels:

- Within a routine, to check that inputs are valid, to trap errors within the routine, and to ensure that outputs are safe.

- Within the software, to check that system inputs are valid, to trap errors within the software, and to ensure that system outputs are safe.

- Within the system, as independent verification that the rest of the system is behaving correctly, and to prevent it from causing the system to become unsafe. The safety enforcing part is usually referred to as an interlock or protection subsystem.

Software at higher integrity levels is more expensive to develop. A cost conscious designer may consequently try to isolate functionality requiring a high integrity level from other less important functions. Such an approach is essential if is intended to use off-the-shelf software, such as a database manager, because the vast majority of off-the-shelf software products can only be attributed an integrity level of 0.

You can only have different integrity levels for different parts of a system or its software if it can be proven that lower integrity parts cannot violate the integrity of higher integrity parts or the overall system. Consequently, split levels of integrity are usually only viable when used between processors in a system, and not within the software executing on a single processor.

Figure 1 shows a system in which control and protection has been implemented on separate processors, enabling control software to be implemented to a lower integrity level.
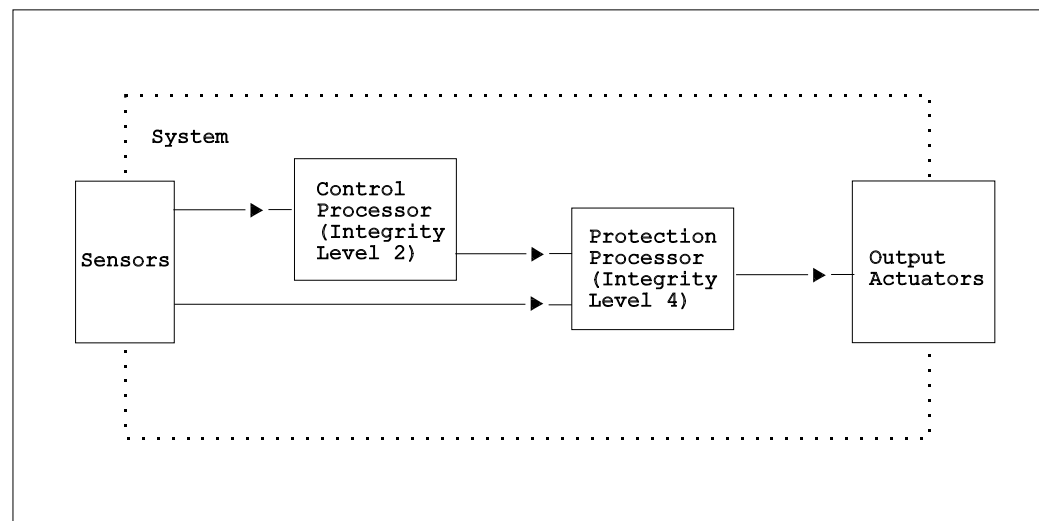
Figure 1 - Split Integrity Levels Between a Control and Protection System

Whatever philosophy for the specification and design of a safety related software you adopt, basic quality management principles apply, as required by the ISO9001 standard. These include defined procedures for each stage of the lifecycle, records and documentation, configuration management and quality assurance.

4. Programming Languages for Safety Critical Systems

The general principle of taking no unnecessary risks can also be applied to the selection of programming languages for safety critical software development. Some programming language features are more prone to problems than others. This could be for one or more of a number of reasons:

- Programmers may be prone to making errors when using the feature

- Compilers may be prone to poor or incorrect implementation of the feature

- Programs written using the feature may be difficult to analyze, test or prove

- The feature may introduce implementation dependencies, reducing portability

There are quite a few programming language features which are prone to causing problems:

- It is easy for programmers to become confused as to just what a **pointer** is pointing at. Manipulation of pointers often requires an understanding of and dependence on the underlying processor architecture. Programs which use pointers can be difficult to understand or analyze.

- **Dynamic memory allocation and deallocation** is usually closely connected with the use of pointers. Programmers often allocate memory and subsequently forget to deallocate it. Compilers and operating systems frequently fail to fully recover deallocated memory. The result is errors which are dependent on execution time, with a system mysteriously failing after a period of continuous (and up to that point correct) operation.

- **Unstructured programming**, including the use of **goto,** is perhaps the most widely recognised source of programming error. Most programmers shun the use of goto.

- **Multiple entry points and exit points**, to loops, blocks, procedures and functions, is really just a variation of unstructured programming. However, there are circumstances where carefully controlled use of more than one exit can simplify code and consequently reduce risk.

- **Variant data**, where the type of data in a variable changes, or the structure of a record changes, is difficult to analyze, and can easily confuse a programmer leading to programming errors.

- **Implicit declaration** and **implicit initialisation**. A simple spelling mistake or typographical error can result in software which compiles, but does not execute correctly. In the worst case individual units may appear to execute correctly, with the error only being detectable at a system level.

- **Procedural parameters**, or passing one procedure or function as a parameter to another procedure or function, is difficult to analyze and test thoroughly.

- **Recursion** is difficult to analyze and test thoroughly. Recursion can also lead to unpredictable real time behaviour.

- **Concurrency** and **interrupts** are supported directly by some programming languages. Use of concurrency and interrupts has already been discussed under the topic of specification and design.

The use of such programming language features in safety critical software is discouraged. Obviously, enforcement must be stronger where higher integrity levels are required, and may be relaxed where lower integrity levels are required.

In some circumstances the careful and constrained use of features such as multiple exits, pointers, recursion, concurrency and interrupts, can actually reduce complexity and enhance reliability. In a few rare circumstances, the programmer may have no option but to use these features. Such uses must be carefully designed, controlled and justified, to ensure that the integrity of the software is not adversely affected.

Most modern programming languages encourage the use of block structure and modular programming, such that programmers take good structure for granted. Well structured software is easier to analyze and test, and consequently less likely to contain errors. Other features, provided by some programming languages, which can be used to increase reliability are:

- **Strong typing**, to ensure that data is only used and assigned where it is of a compatible type.

- **Run time constraint checking**, to ensure that arrays bounds are not violated, that data does not overflow, that zero division does not occur, and similar numeric and constraint checks.

- **Parameter checking**, to ensure that parameters passed to or from procedures and functions are of the correct type, are passed in the right direction (in or out) and contain valid data.

There are no commonly available programming languages which provide all of the good language features, while not providing any of the bad language features. The solution is to use a language subset, where a language with as many good features as possible is chosen, and the bad features are simply not used. Use of a subset requires discipline on behalf of the programmers and ideally a subset checking tool to catch the occasional mistake. An advantage of a subset approach is that the bounds of the subset can be flexible, to allow the use of some features in a limited and controlled way.

Ada is the preferred language for the implementation of safety critical software because it can be used effectively within the above constraints. The most popular Ada subset for safety critical software is the SPARK Ada subset. Ada has the added advantage of being highly standardised through the compiler validation scheme. Similar languages such as Pascal and Modula-2 are also suitable and have been used extensively in the past, but have now effectively been superseded by Ada.

The C language provides none of the good language features, and depends heavily on the use of pointers, making C an unsuitable choice for safety critical software. Other languages, such as Basic and Fortran, fall down by allowing the implicit declaration of variables, have weak typing and no parameter checking. PL/M and C++, like C, depend too heavily on pointers.

The choice of language, use of a subset, and any deviations from that subset, have to be justified as part of the safety case. Use of an Ada subset simplifies this process. Other languages can be used successfully for safety critical software development, but with an increased overhead associated with the use of undesirable language features.

5. Verification of Safety Critical Systems

Verification is the most important and most expensive group of activities in the development of safety critical systems, with verification activities being associated with each stage of the development lifecycle. The actual activities are basically the same as a conscientious developer would apply to any other software development, but with a more formal and rigorous approach.

An added complication is that **independent verification** is usually required. The means by which this is achieved depends upon the integrity level and the user or certification body. Independent verification can vary from independent witnessing of tests, participation at reviews and audit of the developer's verification, to fully independent execution of all verification activities. The degree of independence can vary from a separate team within the overall project structure, to a verification team supplied by a completely independent company, who may never even meet the development team.

Irrespective of how much independent verification there is, developers should not forget their own verification work. Independent verification is an addition to verification conducted by developers, not a substitute for it. A developer's objective should be to come as close as possible to passing independent verification first time, with only minor corrections to make.

Tools used in the development of safety critical software can influence the safety integrity of the software under development. This can be a problem, because most tools are only developed to integrity level 0. Care must therefore be taken during verification to ensure that tools such as compilers and CASE tools have not introduced errors into the software under development.

Care must also be taken to ensure that faults in verification tools do not give an incorrect "pass" result to any verification activity, when the results should have been a "fail". Some verification tools have been developed with this requirement in mind, using safety critical standards during the development of the tool.

Taking an ISO9001 accredited quality management system as a baseline, let's look at how the individual verification activities may vary. **Reviews** become more formal, including techniques such as detailed walkthroughs of even the lowest level of design. The scope of reviews is extended to include safety criteria. If formal mathematical methods have been used during the specification and design, formal mathematical proof is a verification activity.

Static analysis is the analysis of program source code before it is executed. If static analysis is conducted at all for non safety critical software, it is usually limited to checking coding standards and gathering metrics. For safety critical software, more complex static analysis techniques such as control flow analysis, data flow analysis, and checking the compliancy of source code with a formal mathematical specification can be applied. This kind of sophisticated static analysis is provided by tools such a SPARK Examiner and Malpas.

Dynamic testing is the mainstay of verification, extending from the testing of individual units of code in isolation from the rest of the software, through various levels of integration, to system testing. For safety critical software, dynamic test results are only valid in the target environment. You can however develop dynamic tests in the convenience of a host environment, then repeat fully developed tests in the target environment.

8

- Safety critical software standards require techniques such as equivalence partitioning, boundary value analysis, and structural testing to specified levels of dynamic coverage. **Dynamic coverage** levels required when testing to safety critical standards are much more rigorous than simple statement coverage or decision coverage. For example RTCA/DO178B requires modified decision coverage, where a developer must show that every element of a Boolean expression can independently affect the outcome of the expression.

Most developers provide traceability between system requirements and system test cases, to ensure that all requirements have been implemented and tested at the system level. For safety critical software, the level of detailed traceability required is much higher. Traceability of requirements extends through all levels of design, into individual units of code, and into all levels of testing, through to individual test cases.

6. Using AdaTEST and Cantata for Safety Critical Systems Development

Tool support for dynamic testing and coverage analysis, to the levels of functionality required by standards for safety critical software development, is provided by AdaTEST (for Ada) and Cantata (for C and C++). It is believed that AdaTEST and Cantata are the only tools to offer this functionality.

AdaTEST and Cantata have been produced to IPL's ISO9001 and TickIT accredited Quality Management System. The core part of AdaTEST was in fact developed to safety critical standards, to ensure it's suitability for use in the verification of safety critical software.

AdaTEST and Cantata provide repeatability of tests and portability of test scripts between host and target environments. AdaTEST and Cantata can be used in both analysis mode and in non-analysis mode in both the host environment and in the target environment. A suggested strategy is:

a) Use the AdaTEST or Cantata instrumentor (in the host environment) to perform static analysis and to prepare an instrumented version of the software under test for dynamic (coverage) analysis.

b) Execute tests in analysis mode in the host environment, using instrumented software. Ensure that coverage objectives are achieved. Correct any errors in the software under test and in the test script.

c) Verify that coverage is achieved in the target environment, by executing tests in analysis mode in the target environment.

d) Verify that the instrumentation and environment has not influenced the outcome of tests, by repeating tests in non-analysis mode in the target environment, using uninstrumented software.

The portability of AdaTEST and Cantata test scripts between the host and target environments has been specifically designed to enable tests to be fully developed in the host environment, and then simply repeated in the target environment.

There are a number of papers available from IPL which provide more details on the use of AdaTEST and Cantata in association with particular standards required for safety critical software development. The majority of standards identified in table 1 are covered

9

by such papers and IPL's coverage of such standards is constantly growing. Please contact IPL for more details.

7.   Conclusion

Safety critical software is a complex subject. This paper has necessarily had to be brief and superficial. The main objective of the techniques discussed in this paper is to minimise the risks of implementation errors. The premise is that errors could result in system behaviour that causes a hazard to safety.

Although safety critical systems have been in use for many years, the development of safety critical software is still a relatively new and immature subject. New techniques and methodologies for safety critical software are a popular research topic with universities, and are now becoming available to industry. Tools supporting the development of safety critical software are now available, making the implementation of safety critical standards a practical prospect.

Suitably trained and experienced engineers are key to the success of any software development. In the development of safety critical software, the risk of an inexperienced or untrained engineer making an error which could then lead to an accident must be avoided.

Even if you are not developing safety critical software, the reliability of your systems may benefit from adopting some of the techniques and methodologies discussed in this paper. Above all, the best way to minimise risk, both to safety, reliability and to the timescale of a software development project, is to keep it simple.