



Assessing the Safety of Integrity Level Partitioning in Software

John A McDermid and David J Pumfrey
Department of Computer Science, University of York
York, UK

Abstract

In order to exploit the capability and performance of modern processors in safety critical applications, it is desirable to be able to run software of differing integrity levels on the same processor. To do this safely, however, requires the ability to enforce partitioning between these different integrity levels. For certification, there is a need to demonstrate the effectiveness of these partitioning mechanisms. In practice, this means analysing the hardware, e.g. memory management units, and software, e.g. operating system functions, which implement the protection mechanisms – and analysing the hardware-software interactions. This paper describes a method, known as LISA, developed to analyse low-level hardware-software interactions in multiprocessor computer systems, and draws some conclusions from experience of applying the method on a complex avionics system.

1 Introduction

Traditionally, safety critical computer systems have tended to be bespoke developments, with custom-written software, very often hand-crafted in assembly code, running on a unique hardware platform. Although it has not been the primary reason for building systems this way, safety engineering has benefited from the bespoke approach. Every part of the system is specified, designed and implemented for the application, and this allows a very high degree of visibility of, and control over, both process and product.

It is becoming generally accepted that completely bespoke development is no longer viable except for the most critical of applications. The primary reason for this change is economic; as hardware costs have fallen, development (particularly software development) costs have become dominant, even in products with high sales volumes. There is also a desire to find ways to allow safety critical systems to follow the general computing “power curve”. Because of the need to use components with a good track record of reliable operation, safety critical systems projects have typically been restricted to using hardware that is no longer state of the art even when design begins. By the end of a lengthy development, the product may be a couple of generations behind leading technology, leading to a perception of poor performance when compared to contemporary non safety critical systems.

Possible approaches to reducing development costs and addressing performance issues include:

- moving to program generators
- increased use of standard components and hardware
- increased reuse of software components
- use of commercial-off-the-shelf (COTS) software.

All of these potentially offer significant benefits, but all also pose new challenges for achieving and assessing safety.

Essential to many of these changes will be the use of system architectures more similar to those found in general computing systems, implying the use of operating systems which can provide flexible scheduling, guarantee independence of separate processes on the same processor, and provide cross-platform commonality to allow applications to be migrated to new or upgraded hardware. Already, a number of operating system vendors offer products which they claim are suitable for use in safety critical applications, although these claims are not universally accepted (see for example, the debates on the selection of operating systems in the archives of the `hise_safety_critical` mailing list [`hise_safety_critical` 1999]).

The design and analysis of operating systems presents many new challenges for safety engineers. Significant progress has been made in some areas, notably in the guaranteeing of timing properties of different scheduling schemes on a range of architectures – see, for example [Audsley 1995; Joseph 1996; Kopetz 1997] – and in static analysis techniques for analysing the use and protection of critical data. However, most of these techniques assume perfect execution of the operating system software. There are few techniques that can provide evidence about independence of processes, protection of critical data or timing behaviour once potential unintended behaviour of the underlying hardware is taken into account.

In current industrial practice, the most common method of including computer hardware failures in safety analyses is simply to assume that any failure within the computer system will be sufficient to cause any failure mode of concern at the computer's outputs. Thus, in system level fault trees, it is common to see a basic (or undeveloped) event called simply "computer hardware failure" (see, for example, the fault trees for inadvertent aircraft braking developed in ARP 4761 Appendix L [SAE 1996]). Whilst this is clearly a conservative assumption sufficient to ensure safe treatment at the system level, it is of no use to software designers, who need to know what the symptoms of various hardware failures will be in order to develop detection and protection strategies.

Safety engineering always starts by identifying hazards at the platform level, and propagating safety requirements down to system and component level. It is clearly not possible to construct a complete safety argument for any operating system supplied as a "stand-alone" component (i.e. without knowledge of its eventual application context). However, it should be possible to develop "sub-arguments" – effectively, a description of a set of properties that the operating system is claimed to guarantee, together with the evidence to support the claims. Even this will

require a combination of several different techniques, covering different aspects of correct behaviour and appropriate management of a range of failure modes.

Although superficially similar to designing any other engineered artefact (such as, say, a pump or valve), the main problem with operating systems is one of complexity. The designer of a pump may not know what systems his component will eventually be used in, but its capabilities and failure modes can be expressed relatively simply in familiar engineering terms, and the system designer can select a pump that meets his requirements.

It is perhaps better to compare an operating system with, say, the electrical power supply to a complete plant, rather than with a single component. The selection of electrical power supply characteristics will determine some of the requirements for components – suitable operating voltage, for example. A complete electrical system failure could disable all of the controlled components in a plant, or perhaps cause common damage to all of them through excessive voltage or current. In a similar way, selection of an operating system will set requirements for the development of other software components, in terms of interfaces, scheduling model etc., and operating system failures have the potential to disable or interfere with the behaviour of every other software function in a system.

This analogy is over-simplistic, but helps to identify further characteristics of operating systems that make them particularly problematic:

- *Unknown behaviour in case of hardware failure*
The sheer complexity of computer hardware makes it impossible to predict all of its potential failure behaviours, or to guarantee continued correct behaviour of software if the underlying hardware fails. However, operating systems compound the problem due to the level of indirection between application code and hardware.
- *Lack of independent action in components* (all tasks rely on the operating system)
The designer of the plant in the electrical example above could attempt to ensure that individual components were designed to behave in the safest way possible in the event of anticipated problems with the power supply – for example, fitting springs to make valves self-closing when un-powered. This sort of independent protection within components is not possible with software, as it is not possible to guarantee any specific behaviour of software in the event of hardware or operating system failure.
- *Un-needed functionality*
The more general a computer, the fewer of its capabilities will be required by any particular application. This means that complete analysis of an operating system in any given context must include demonstration that the functions that are not wanted cannot in any way interfere with the required behaviour.

Thus, the major obstacle to a detailed combined safety analysis of hardware and software is how to manage complexity. Since the state space of even simple programs is very large, there can be no tractable approach (at least for manual

analysis) based on evaluating the effect of individual hardware failures in every possible system state. This rules out analysis techniques based on conventional event or state models. The challenge for safety analysis is to find meaningful ways of identifying the critical parts of the system and modelling the potential impact of hardware failures at a more abstract level. We have developed an approach to this problem that we call Low-level Interaction Safety Analysis.

2 Low-level Interaction Safety Analysis - Principles

In a presentation to the International System Safety Conference in 1998 [McDermid 1998] we described the principles of an analysis method called Low-level Interaction Safety Analysis (LISA). This paper briefly reviews those principles, defines a complete method for the technique, and presents further results from a large-scale industrial trial.

LISA was developed specifically to study the way in which an operating system manages system *resources*, both in normal operation and in the presence of hardware failures. **Instead of analysing the system functionality, the LISA method focuses on the interactions between the software and the hardware on which it runs. A set of *physical resources* and *timing events* is identified, and a set of *projected failure modes* of these resources is considered.** The aim of the analysis is to use a combination of inductive and deductive steps to produce *arguments of acceptability* demonstrating either that no plausible cause can be found for a projected failure, or that its consequences would always lead to an acceptable system state.

The method starts from an identification of critical code to identify and classify resources. Critical code consists of safety critical application functions, plus any parts of the operating system that are essential to the correct operation of these application functions. (Note that if there is no partitioning, then all of the application code must be considered to be critical.) The criticality of resources is then determined by whether, and how, they are used by critical code.

2.1 Identifying and Classifying Resources

Physical resources consist of the processor registers, memory locations, I/O and other special registers. This is the programmer's model of the computer. Hardware features such as buses, arbitration logic etc. are considered in terms of the registers that control them. Physical resources are partitioned into five classes based upon the *criticality* of the resource usage:

- *intrinsically critical*
resources that are used by critical application processes, or which contain data essential to the continued correct functioning of the operating system itself. For a general purpose operating system, it may be necessary to regard all memory locations that are available to application processes as intrinsically critical.

- *primary control*
these are resources which directly control the use or function of an intrinsically critical resource; examples include memory management unit (MMU) registers, I/O control registers etc.
- *secondary control*
resources which either provide a backup to primary controls (e.g. a secondary MMU giving redundancy in memory protection), or control access to primary resources (for example, “key” registers which must be set to particular values before protected locations can be altered).
- *non-critical*
resources which are never used by critical software, and do not affect the correct functioning of the operating system itself.
- *unused*
locations in the memory map which do not correspond to a physical device. The importance of these locations is that there should be no attempts by any part of the software to access them; such an attempt indicates a failure, and must be trapped and handled safely.

The model of time used in the analysis is based on the identification of discrete timing *events* that have associated hardware actions. Examples of these include interrupts, the use of system timers and counters, and synchronisation actions. Timing events can be identified as either critical or non-critical, depending upon whether they affect the execution of critical code. Note that there will be a set of primary (and possibly some secondary) control resources associated with each timing event. For example, the primary resources associated with a timer-generated interrupt will include the control registers for the timer, and the CPU registers that determine the response to the arrival of the interrupt.

From the descriptions of the resource classes, it is clear that there are dependencies between resources; that is, the state of one resource affects the behaviour of another. Indeed, this is explicit in the definition of primary and secondary control resources. However, there are other, less direct dependencies. The most significant of these is that, in most systems, there must be an initialisation phase, in which the software configures the hardware to the state required for the execution of the main body of the application.

A complete safety argument for the system must therefore demonstrate that the system powers up in a safe state, and respects minimum safety requirements throughout every stage of initialisation. To guarantee the correct execution of the application, it must also be shown either that successful completion of the initialisation guarantees that the hardware is correctly configured, or that it is impossible (or at least extremely improbable) that the main body of the software could fail to detect, and safely respond to, any incorrectness in its execution environment.

2.2 Failure mode identification in LISA

It is infeasible to consider every type and cause of failure of each device in a computer system individually, so it is necessary to make an abstraction. Experience applying HAZOP and related techniques to computer systems [McDermid 1995] has shown that a small set of fairly general guide words can be used to prompt consideration of many aspects of computer system failure provided that it is interpreted carefully in each new system context. The set of general guide words used as a basis for LISA is:

- *Omission* – a necessary action does not occur
- *Commission* – an unwanted action is performed (i.e. a perfectly functioning system would have done nothing)
- *Early* – an action is performed before the time (either real time, or relative to some other action) at which it is required
- *Late* – an action is performed after the time at which it is required
- *Value* – the timing of the action is correct, but the data it is performed with or upon is incorrect.

2.3 Arguments of acceptability

An argument of acceptability states why the normal use of a resource, and the way in which potential failures are detected and handled, result in acceptable safety properties at a system level. Suitable argument structures will vary from project to project. If the system to be analysed is being developed specifically for one application, and hazard and safety analyses are available, the LISA acceptance arguments will be able to relate low-level behaviour to system level effects. For systems that are being developed as components without detailed knowledge of the eventual application, the acceptance criteria will depend upon the target integrity level of the system, and might typically include requirements to demonstrate that:

- the intended usage of a resource does not allow low-integrity processes to access / alter high integrity data;
- no plausible cause can be found for a suggested failure;
- a suggested failure has been shown to be mitigated by a completely independent mechanism (i.e. different hardware and independently coded software);
- a suggested failure has been shown to produce no effect on the correct operation of the critical functions of the system;
- a suggested failure may have an adverse effect on the correct operation of critical functions, but the failure can be reliably flagged to application code, which can implement acceptable handling or mitigation.

2.4 Selection of analysts

As with all hazard and safety analyses, many factors will influence the selection of appropriate people to conduct a LISA analysis. This is a very low-level technique,

requiring extremely detailed knowledge of both the software and hardware design; case study experience has shown that this is difficult and time-consuming for an outsider to obtain. However, LISA is intended to be a demonstration of achieved safety, and it is reasonable to expect that LISA analysts will be required to be as independent as possible.

Industrial experience has shown that easy access to members of the design team is vital, and our recommended way of applying LISA is to have a member of the design team working in an explanatory role, together with an independent analyst whose responsibility is to produce the final report and conclusions.

3 LISA Method

This section outlines the conduct of a LISA analysis.

Step 1: Agree principles for acceptability

Section 2.3 above explains the purpose of, and identifies some possible structures for, arguments of acceptability.

Step 2: Assemble source material

The minimum required source materials for LISA are:

- an overall description (specification or design) of the intended operation of the system, including strategies for managing expected failure modes;
- a complete system memory map;
- definitions of the purpose and usage of all special device registers, I/O etc. This may take the form of programmers' manuals for each hardware device in the system;
- a specification or design document which describes all the timing events in the system;
- specification or design documents which define the system start-up, initialisation, exception handling and normal and emergency shutdown procedures.

Additional sources that may be necessary or useful for some analyses include subsystem specification or design documentation, hardware failure data, and program source code.

Step 3: Analyse timing events

From the system documentation, identify all the timing events which involve a hardware / software interaction. These will include all uses of interrupts, system timers, counters or clocks, inter-processor synchronisations and time-dependent interactions with external devices or other systems. Care must be taken to ensure that events used in every mode of system operation, including initialisation, shutdown etc., are included.

For each event identified, describe its intended operation, including the preconditions necessary for the event to be generated (e.g. programming of timers)

and the correct response(s) to the event. Ensure that the intended behaviour defined does not in itself create potential safety problems (e.g. check that context switches initiated by interrupts cannot leave critical data in an inconsistent state, that interrupt mask levels in the new context are appropriate etc.)

Step 3.1: Suggest deviations from intended behaviour of events

Use the guide words *omission*, *commission*, *early* and *late* to prompt consideration of possible deviations from the expected operation of each event. Each guide word may suggest more than one deviation.

Omission – the failure of an event to occur. Possible cases of omission to consider include:

- Source (writer) omission
All events will have a single source, which in this case fails to produce the expected event. Depending on the type of source, it may be necessary to consider whether it is plausible for the source to proceed to its next expected action, or whether it experiences a fail-stop condition. A special case of this is *precondition* omission, in which the event is generated when a set of preconditions is satisfied, and the preconditions are never true.
- Transmission omission
The medium through which the event is transmitted “loses” the event.
- Destination (reader) omission
The destination process(es) or object(s) that was (were) expected to recognise and respond to the event do(es) not do so, e.g. through being in the wrong state or having incorrect interrupt masks. In a multiprocessor system, where events affecting more than one processor are possible (e.g. broadcast interrupts or synchronisation events), it is necessary to consider symmetric (where no recipient responds to the event) and asymmetric (one or some recipients respond) omission.

Commission – the spurious occurrence of an event. Source, transmission and destination may apply as for omission, and it may also be necessary to consider:

- Number of commission errors
A single unintended event may have a different effect to multiple repetitions
- Repetition vs. insertion
An expected event that is repeated may have a different effect from a completely unexpected event. This particularly applies when events are expected in a predetermined sequence, which is violated.

Again, there may be symmetric and asymmetric cases to consider in a multiprocessor system.

Early and *late* may be interpreted either with respect to real time, or as prompts to consider incorrect ordering (relative timing). They may also prompt consideration of jitter (i.e. where supposedly regular periodic events actually

occur at irregular intervals). It should also be noted that it is important to define the boundary between an event that is late, and one that is considered to have failed entirely (omission), and similarly for early / commission.

Step 3.2: Investigate possible causes of event deviations

For each of the deviations suggested in step 3.1, identify possible causes, making certain that both direct hardware failures and indirect causes in software (e.g. incorrect programming of control registers) are considered. If no plausible cause can be found for a deviation, this should be noted.

Step 3.3: Investigate effects of event deviations

Investigate the effects of each deviation for which plausible causes were found in step 3.2. Consider how the deviation will be detected and handled by the system, and ensure that this will result in a safe system state. If there are no detection and handling mechanisms for the deviation, consider what its ultimate effect on system safety will be.

Step 3.4: Produce arguments of acceptability

Decide which of the principles of acceptability agreed in step 1 is appropriate for each suggested deviation, and produce arguments of acceptability showing how the system design meets the principles. These arguments do not need to be extensive, but should be sufficient for a reviewer reading the analysis to identify all of the components and mechanisms involved. If a suitable argument of acceptability cannot be found for a suggested deviation, this is an indication of a possible flaw in the system design.

Record the investigation of the deviation, and repeat for each suggested deviation of every identified event.

Step 4: Analyse physical resources

Identify physical resources in the system. The primary information source for this is the system memory map, although it may be necessary to consult processor, device and subsystem documentation to identify the function of all of the registers within the blocks allocated to each device. As for events, describe the intended usage of each resource, ensuring that start-up, initialisation, shutdown and other modes are considered, and check that the intended use does not in itself create safety problems.

Step 4.1: Group resources by common factors

There is no need (and, indeed, it would be impractical) to analyse every aspect of every resource separately. Resources should therefore be grouped by common characteristics so far as possible. For example, access permissions are typically assigned to blocks of memory locations; the effects of granting inappropriate access (or denying access where it is required) can be examined for the whole block of locations. Similarly, access timing properties will usually apply to all of the locations within a single hardware device; again, these can be examined once for all locations within the device. If the initial suggested grouping is not correct,

this will become apparent as the analysis proceeds; either identical results will be obtained for many resources, suggesting they should be grouped, or the analysis of a block will show that the effects of deviations vary for different locations within the block, showing that it must be split.

Step 4.2: Classify resources

Resources are classified as *intrinsically critical*, *primary control*, *secondary control*, *not critical* or *unused* (the definition and implications of each of these classes is discussed in section 2.1 above).

Step 4.3: Suggest deviations from the intended use and operation of each resource

Use the guide words *omission*, *commission*, *early*, *late* and *value* to prompt consideration of possible deviations from the expected use and function of each resource. Each guide word may suggest more than one deviation.

- *Omission* and *commission* are interpreted as access permission violations. An omission failure occurs if a process that should be able to access a resource is denied permission. Commission failure occurs where a process is granted access to a resource that it should not have.
- *Early* has two interpretations in the case of a physical device such as memory, both leading to (unpredictably) corrupt data:
 - the processor reads from a location in the device, and attempts to latch the data from the bus before it is stable, or
 - the processor writes to a location in the device, and de-asserts data before the device has latched it correctly.

These may seem unlikely failures, and only possible with poor hardware design. However, there are systems in which parameters such as the number of wait states inserted on accessing a particular device are programmable; the system can dynamically alter its own timing characteristics. In such systems, this type of timing failure is plausible and extremely important.
- *Late* refers to delay in accessing the resource, arising either from effects such as contention for a shared bus, or from the same type of configuration fault that could lead to *early* failures. Unlike *early* failures, lateness cannot cause data corruption. Excess wait states in a device access will merely extend the cycle and result in reduced performance; the data will be held stable until the end of the cycle. Late latching of data being read by the processor (i.e. after it has been de-asserted by the device being accessed) is highly unlikely, since the processor itself generates the timing control signals that control the device. In general, therefore, *lateness* is only of interest in our analysis if the delay is great enough to be treated as an *omission*, e.g. by triggering a bus timeout.
- The *value* of a resource is its data content. For control resources, the correct value can often be determined in advance, and the effects of changes predicted. In the case of memory (RAM or ROM), the effect of unwanted changes can usually only be determined with knowledge of the application software.

Step 4.4: Investigate possible causes deviations in resource use or function

For each of the deviations suggested in step 4.3, identify possible causes, making certain that both direct hardware failures and indirect causes in software (e.g. incorrect programming of control registers) are considered. If no plausible cause can be found for a deviation, this should be noted.

Step 4.5: Investigate effects of deviations in resource use or function

Investigate the effects of each deviation for which plausible causes were found in step 4.4. Consider how the deviation will be detected and handled by the system, and ensure that this will result in a safe system state. If there are no detection and handling mechanisms for the deviation, consider what its ultimate effect on system safety will be.

Step 4.6: Produce arguments of acceptability

As with the event analysis, decide which of the principles of acceptability agreed in step 1 is appropriate for each suggested deviation, and produce arguments of acceptability showing how the system design meets the principles.

Record the investigation of the deviation, and repeat for each suggested deviation of every resource. It is helpful to produce two tables to record the analysis for each block of resources, one containing the investigation of properties such as access permissions and timing that apply to all resources in the block, and one that records deviations, effects and arguments which are specific to a single resource. As generic arguments become apparent (i.e. those that are used repeatedly), a separate table of these should be compiled to reduce the volume of the analysis.

4 Industrial Application of LISA

LISA has now been successfully applied to the development of a large, multiprocessor avionics system. The relative proportion of critical software to that of lower integrity levels, together with the relatively high levels of data sharing between the classes of software, meant that the conventional approach of partitioning out the critical software to separate processors would have resulted in very inefficient use of resources in a heavily loaded system. The design team therefore took the decision to implement a partitioned software system, with a core of critical software operating on all processors to implement initialisation, communication and scheduling / synchronisation tasks.

Although this was a completely bespoke development, the critical software effectively provided some of the functions of a rudimentary operating system. These included:

- memory and I/O protection
- synchronisation of activity across all processors
- scheduling

The critical software did not provide the uniform interface to hardware that would be expected of a full conventional operating system (i.e. application functions were able to access hardware directly).

Evidence was required that the partitioning system genuinely provided an acceptable level of independence between the critical functions and the rest of the system. No existing analysis technique could be identified which could provide this evidence so, with the agreement of the certifying authority, it was decided to develop a new approach. It was decided that the critical software that implemented the partitioned software environment should be analysed independently of the rest of the system. In effect, it was to be treated as an operating system to be analysed in isolation, without detailed application knowledge. The primary reason for this decision was that it was hoped to make the analysis of the critical code completely independent from the rest of the software, so that changes could be made to non-critical parts of the system without the need to repeat the complete safety analysis.

4.1 Case Study Safety Principles

In discussion with system developer, the primary requirements identified for acceptability of the partitioning scheme were:

1. Data flow corruption must be prevented
 - Modification of critical data by software of lower integrity levels shall not occur
2. Control flow corruption must be prevented
 - No action of the low integrity software shall prevent the critical software from executing when it should
 - Modification of critical code by software of lower integrity levels shall be prevented
3. Corruption of the execution environment shall not occur
i.e. corruption of processor registers, device registers and memory access privileges shall not occur.

From these a further, secondary, requirement was identified:

4. If any of the primary requirements 1 to 3 is violated, this shall be detected, and the system caused to shut down promptly, setting a failure flag so that a stand-by unit can assume control.

Note that requirement 2 must be interpreted carefully, in that requirement 4 means that execution errors in the non-critical software may cause a shutdown, thus preventing further execution of critical code. This is acceptable, as it is the defined safe state of the system. However, the self-shutdown procedures, which are executed if any hardware failures or internal inconsistencies are detected, are a part of the critical software, and this gives a further derived requirement:

5. The ability of the self-shutdown procedures to run to completion (i.e. safe shutdown) must be guaranteed regardless of the state of the non-critical parts of the software.

It was further agreed that requirement 4 applied only to cases of single failure, i.e. the analysis did not need to consider cases of multiple simultaneous or near-simultaneous failures. This limitation was justified by two arguments:

1. Self-shutdown procedures were very quick and simple. It was considered that, provided that the first failure did not prevent the shutdown procedures from being initiated, the probability of a second, independent failure occurring that could prevent their completion was negligible.
2. In the case of multiple, coincident failures of sufficient severity to prevent orderly detection and shutdown, it was considered that the behaviour of the system would be sufficiently incoherent to be readily detectable by other systems or operators.

In order to meet the requirement that the analysis produced should be as independent of the application software as possible, it was decided that analysis should proceed by assuming that all non-critical application software would always behave in the worst conceivable way. This was interpreted as meaning that any failure of protection, synchronisation or scheduling would always result in the non-critical software causing the maximum possible interference to the operation of the critical functions. If satisfactory protection of the critical functions could be demonstrated under this assumption, no future change to the non-critical software could invalidate the safety arguments.

The LISA analysis of the system was carried out by one of the authors with considerable assistance and support from the project team, and also from Dr. Neil Audsley, who carried out a timing analysis of the system which was included in the final report. The case study extended over an elapsed period of approximately 30 months, during which time it received about 10 man months of direct effort. Of this, approximately 4 months was dedicated to developing an initial detailed understanding of the design and intended operation of the system. The initial analysis took approximately 2 months of effort, and there were then a number of updates of the design that necessitated repeating or updating parts of the analysis. The final report took a further two weeks to write.

Arguments of compliance with the safety principles were considered complete when either:

- the failure had been shown to be mitigated by two or more independent mechanisms
- the failure had been shown to produce no effect on the correct operation of the partitioning system or critical application software

or

- the failure had been shown to produce no effect on the correct operation of the partitioning system, but there were potential effects on the critical application

software. In this case, the need for subsequent application software analysis was noted.

The complete final report contained 98 pages of safety analysis, including 7 tables of event analysis, which occupied 22 pages, and 18 tables of physical resource analysis, which occupied a further 20 pages. The report also contained 22 pages of Functional Failure Analysis, which was produced for comparison with the LISA results. The comparison showed that the LISA analysis contained every failure mode considered in the FFA, plus a significant number of additions, and much more detail (FFA, of course, does not consider the causes of the failure modes it examines).

In conducting this analysis, it was found that the method was extremely good at highlighting areas where the design intent was not clearly expressed in the available documentation, and several recommendations were made for improvement in the specification and supporting documentation. No safety related problems were found in the actual design at any stage, although completing the arguments of acceptability for some items required quite extensive investigation, including requests for additional information from subsystem suppliers. In a number of cases, the analysis also suggested additional tests that could be incorporated into the built in test (BIT) routines to improve the detection of certain classes of deviation. Most of these suggestions made use of self-test capabilities already present in the hardware, and all of the suggested additional tests were incorporated into the system. The LISA analysis was also used to investigate the iteration rate required for some of the continuous built-in test (CBIT) routines to meet the worst-case detection and shutdown response times agreed in the safety principles.

In a number of cases where particularly complex mechanisms were involved in the detection and handling of faults, the analysis recommendations also included bench tests involving fault injection to prove that the system response would be as predicted. It was also found that, when design changes were required for other reasons (e.g. to improve system performance), the results of completed parts of the analysis were valuable in developing modifications and in selecting between alternatives.

The final conclusion of the report was that the system was suitable and sufficiently robust for its intended use. However, this was strongly qualified by a number of limitations and exclusions. These included:

- The analysis was conducted on the design *as presented*; there was no attempt to verify that the implementation matched the design, and that this was a task that was essential to ensuring the accuracy and applicability of the report's conclusions.
- Throughout the analysis, it was assumed that previous hazard and risk assessments were correct, i.e. that the hazards were as described, and that the identified safe states were acceptable.

- The safety principles outlined at the start of this section were assumed to be sufficient to guarantee safe operation of the system. No attempt was made to ascertain, either quantitatively or in broad qualitative terms, the availability or reliability of the system. It was noted that extremely low levels of reliability or availability could invalidate the conclusions about the acceptability of the “fail stop” safe state.

The report was presented to customer representatives at the Critical Design Review, at which the analysis was described as thorough and credible. The customer representatives have formally requested that the LISA analysis is maintained as the system progresses through the remaining stages of development and commissioning to in-service status.

4.2 Sample analysis output

The tables at the end of this paper contain samples of the three types of tables produced by the case study. Due to the sensitive nature of the case study, details in the tables have been altered, but the style and technical nature of the entries is representative.

Table 1 contains a sample of the timing event analysis, in this case for the master interrupt that prompted Processor 1 to start a new cycle of the schedule. Table 2 contains samples of the generic arguments developed for blocks of physical resources, which were then used in the analysis tables as shown in Table 3. Note that, as so much information was contained in the generic arguments, or the whole device arguments in the top part of the table, the entries in the address-specific sections of the table were mostly extremely brief, as shown here. The only exception to this was in the analysis of special device registers, where complex arguments were frequently required.

5 Conclusions

This paper has presented an overview of some of the safety analysis problems associated with the transition to safety critical systems designed around conventional operating systems. It has presented a method for safety analysis of the hardware / software interactions in such a system, structured around study of the system *resources* (timing events and physical devices).

As a basis for analysis, this model has several advantages:

1. it is possible to ensure that the set of resources analysed is complete (i.e. includes the entire memory map, all non-mapped devices such as processor registers, and all interrupts and synchronisation events);
2. the model is familiar to the system’s designers and programmers, so it is possible to discuss safety analysis in well-understood terms;
3. although potentially large, the set of resources in any system is of a fixed and predetermined size, so the effort required for analysis can be predicted reasonably accurately in advance.

Overall, we believe that the approach represents an important step towards the analysis of more complex architectures, such as integrated modular avionics, which are being proposed for future generations of systems.

References

[Audsley 1995] Audsley, N. C., Burns, A., Richardson, M.F., Tindell, K. and Wellings, A.J., "Fixed Priority Preemptive Scheduling: An Historical Perspective", *Journal of Real-Time Systems* 8(2): 173-198, 1995.

[hise_safety_critical 1999] Archives of the hise_safety_critical mailing list, ftp://ftp.cs.york.ac.uk/hise_reports/sc.list/archive99.txt

[Joseph 1996] Joseph, M., "Real-Time Systems: Specification, Verification and Analysis", Prentice-Hall, 1996.

[Kopetz 1997] Kopetz, H., "Real-Time Systems: Design Principles for Distributed Embedded Applications", Kluwer Academic, 1997.

[McDermid 1995] McDermid, J. A., Fenelon, P., Nicholson, M. and Pumfrey, D. J., "Experience with the Application of HAZOP to Computer-Based Systems", *COMPASS '95: Proceedings of the Tenth Annual Conference on Computer Assurance*: 37-48, 1995.

[McDermid 1998] McDermid, J. A. and Pumfrey, D. J., "Safety Analysis of Hardware / Software Interactions in Complex Systems", *Proceedings of the 16th International System Safety Conference*, Seattle, Washington: 232-241, 1998.

[SAE 1996] "ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment", Society of Automotive Engineers, Inc., 1996

Table 1 – Sample of LISA event analysis

Master (start of cycle) Interrupt – periodic, 20ms, processor 1 only				
Guide Word and Deviation	Cause	Effect	Detection / Protection	Argument of Acceptability
Omission No master interrupt (more than 5ms late, as this is the limit imposed by the protection mechanism)	<ul style="list-style-type: none"> • Hardware failure • Programmable timer or interrupt control not programmed correctly • Interrupts remain masked during application code execution 	Processor 1 will remain in application mode indefinitely	Back-up interrupt programmed for 25ms on Processor 2, which is reset when Processor 2 receives a start of cycle synchronisation message from Processor 1. If this interrupt occurs, Processor 2 will run the system shutdown procedure.	<ul style="list-style-type: none"> • Failure will be detected and shutdown initiated on first occurrence. • Backup mechanism uses completely diverse hardware (i.e. different processor, no reliance on shared bus)
Commission (single) Excess master interrupts (interrupt occurs less than 20 ms after preceding interrupt)	<ul style="list-style-type: none"> • Hardware failure • Programmable timer not programmed correctly 	If Processor 1 is executing operating system code, no effect until context switch to application code, since interrupts are masked. In application mode, will cause immediate re-entry into operating system, resulting in shortened (no) time for application software. Result will be single-cycle application mode over-run.	Single spurious / early interrupts may not be detected.	<ul style="list-style-type: none"> • All resource protection mechanisms will function as intended despite mis-timed cycle.
Commission (multiple) Excess master interrupts	As above.	As above, except multiple-cycle application mode over-run is certain.	Multiple spurious / early interrupts will be detected by the application mode software completion detection mechanism. Repeated short application mode time slots be detected and will result in shutdown.	<ul style="list-style-type: none"> • Failure will be detected and result in shutdown within a maximum of 4 cycles. • Backup mechanism relies on Processor 1 hardware, but <u>not</u> on timer hardware, which is most plausible cause of failure.
Early Master interrupt early by less than the slack time in application schedule	<ul style="list-style-type: none"> • Hardware failure • Programmable timer not programmed correctly 	Small reduction in master cycle time. Possible slip before effects become as commission depends on slack in application mode schedule.	Will not be detected.	<ul style="list-style-type: none"> • All resource protection mechanisms will function as intended despite mis-timed cycle.

Arg	Applies to	Justification
G1	Unused location (no hardware present)	<ul style="list-style-type: none"> • Attempted accesses to non-existent hardware indicate program corruption, and will result in system shutdown via one of the following mechanisms: • These locations are mapped out by MMU, and accesses will result in bus error on read or write; this will result in an exception in either application or operating system code, which will be handled to shutdown. • CPU bus timer provides a redundant mechanism; if an erroneous access is not refused by the memory protection hardware, an access to a non-existent device will result in the bus timer generating a transfer error signal; again, the result will be an unexpected exception handled to shutdown.
G2	Processor MMU functionality duplicated by second MMU	<ul style="list-style-type: none"> • This applies to access permissions to all locations • The two MMUs are always run in parallel, and both must give permission before a location can be accessed. This is 'fail safe', in that both devices have to grant permission for access; if either one denies it, a bus error handled to shutdown will result. • The MMU mapping tables are set up by separate sections of initialisation code.
G3	Read-only device register enforced by hardware	<ul style="list-style-type: none"> • Writes to these locations are ignored by the device (have no effect). • An attempt to write to such a location is indicative of software error, but it cannot further damage the system state.
G4	Program code area	<ul style="list-style-type: none"> • Given the degree of synchronisation within the operating system schedule, and the mechanisms which exist to detect erroneous application code behaviour, it is extremely unlikely that corrupt program code could execute for any significant period of time before causing a detectable failure. This also applies to mapping tables etc., as corruption of these will cause similar effects. The only plausible undetectable failure would be omission of a section of application code, so functions include cross-checks on successful completion (normal defensive programming techniques are sufficient).
G5	Application RAM area	<ul style="list-style-type: none"> • No corruption of application data can lead to violation of operating system integrity, as the worst case behaviour that can be produced is the malicious application code behaviour already assumed.
G6	Unused location (device register)	<ul style="list-style-type: none"> • Attempted reads from unused registers return 0, thus no detection at this level • Attempted write accesses to unused registers will cause a bus error, which will result in an unexpected exception handled to shutdown.

Table 2 – Examples of generic arguments applicable to many resources

Table 3 – Sample of LISA physical resource analysis (simplified)

EEPROM: Arguments applying to whole device:			
Guide Word	Deviation	Causes	Argument of Acceptability
Omission	Denial of read access to application or operating system software, or operating system write permission during loading	Failure of one MMU	<ul style="list-style-type: none"> • G2 applies
Commission	Granting of write permission except during loading	Requires simultaneous failure of both MMUs	<ul style="list-style-type: none"> • G2 applies
Early	Processor attempts to latch data off bus before it has stabilised	Incorrect number of waitstates inserted on access	<ul style="list-style-type: none"> • See note 1 in discussion of timing analysis.
Late	Excessive latency between device access and data read	Incorrect number of waitstates inserted on access	<ul style="list-style-type: none"> • Local device - no system bus arbitration • See note 1 in discussion of timing analysis.

EEPROM: Address - specific comments:					
Start Address	End Address	Description	Criticality	Reason	Argument of Acceptability
		Operating System Area			
010000	01FFFF	Operating System translation tables	Intrinsically critical	Tables used to initialise CPU MMU at initialisation. Define boundary between operating system and application software	<ul style="list-style-type: none"> • Region is check-summed and validity checked at initialisation • G4 applies
020000	03FFFF	Run-Time System	Intrinsically critical	Run time system code used by all code including operating system software	<ul style="list-style-type: none"> • As above
040000	0FFFFF	Operating System software	Intrinsically critical	Operating system software	<ul style="list-style-type: none"> • As above
		Application Area			
1D0000	1DFFFF	Application mode translation tables	Not critical	Non-critical application area	<ul style="list-style-type: none"> • G4 applies
1E0000	2CFFFF	Application mode software	Not critical	Non-critical application area	<ul style="list-style-type: none"> • G4 applies
2D0000	3BFFFF	Application mode subsystem software	Not critical	Non-critical application area	<ul style="list-style-type: none"> • G4 applies
3C0000	3FFFFF	NOT USED	Not used	-	<ul style="list-style-type: none"> • G1 applies