Volume 28 (1), pp. 19–35
http://www.orssa.org.za

ORiON
ISSN 0529-191-X

Modelling soft error probability in firmware: A case study

DG Kourie*

Received: 3 August 2011; Revised: 20 March 2012; Accepted: 18 April 2012

Abstract

This case study involves an analysis of firmware that controls explosions in mining operations. The purpose is to estimate the probability that external disruptive events (such as electromagnetic interference) could drive the firmware into a state which results in an unintended explosion. Two probabilistic models are built, based on two possible types of disruptive events: a single spike of interference, and a burst of multiple spikes of interference. The models suggest that the system conforms to the IEC 61508 Safety Integrity Levels, even under very conservative assumptions of operation. The case study serves as a platform for future researchers to build on when modelling probabilistic soft errors in other contexts.

Key words: Mining industry, probability, soft errors, safety integrity level.

1 Introduction and Background

This case study reports on an investigation into the SafeBlast^{TM} Central Blasting System (CBS) — a timed detonation system marketed by Sasol Nitro, a subsidiary of Sasol (Pty) Ltd., which is in turn is a multinational petro-chemical company, headquartered in South Africa and jointly listed on both the Johannesburg and New York stock exchanges.

The timed detonation system utilises detonators that have specialised built-in electronics such that the moment of firing of a detonator can be controlled by sending it an appropriate sequence of firing signals. The CBS comprises of control software in a computer that communicates with a set of these detonators via various cables, control boxes and switches. A rough and notional schematic of the components involved are supplied in Figure 1. By attaching the detonators to strategically positioned explosives, the system allows mining engineers to configure very accurately the timing and sequence of explosions with a view of achieving optimal blasting results during mining operations.

In considering the purchase of the CBS system, Anglo-Platinum embarked on a number of experiments to test the system's robustness. (Anglo-Platinum is the world's leading primary producer of platinum group metals. The company is listed on the Johannesburg

*Espresso Research Group, Department of Computer Science, University of Pretoria, Pretoria, South Africa, email: dkourie@cs.up.ac.za

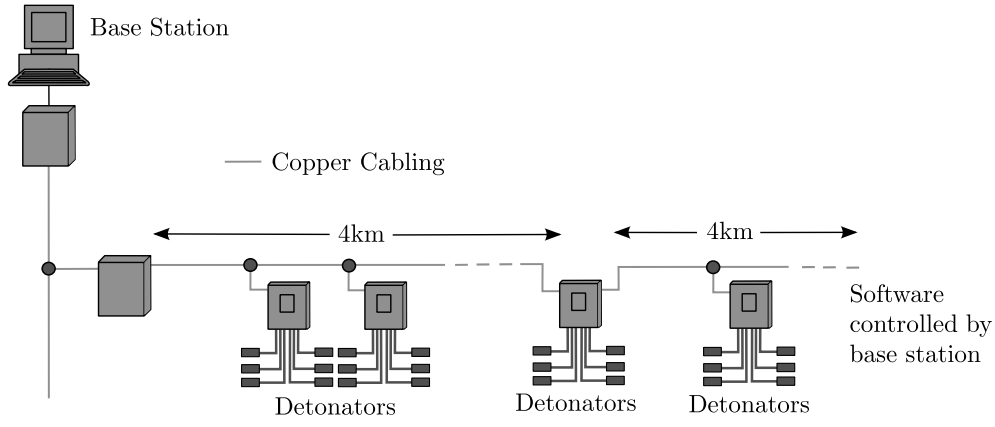


Figure 1: *Schematic layout of the CBS.*

and London stock exchanges.) In an early experiment on a now defunct version of the CBS (specifically on the Mk1 NSS component) electro-magnetic signals over a range of frequencies were artificially induced onto the connecting cables. It was found that at a particular frequency, the firmware on the CBS could be diverted into a state that resulted in an inadvertent firing of an detonator.

The possibility of such inadvertent firings became a source of general concern, even though the experiment had been carried out in a highly artifical context that violated standard hardware operating procedures relating to cable type and length.

To remedy the matter, a so-called magic number protection algorithm was designed by the system's designer, and was incorporated into subsequent versions of the firmware. The algorithm will be outlined below. The algorithm appeared to be effective, since all subsequent experiments with firmware into which it had been incorporated (whether the Mk1 or Mk2 NSS versions of the firmware) have had positive outcomes.

Nevertheless, safety concerns remained. "Was it not possible," it was asked, "that some random electro-magnetic source (a lightening strike, a cellular phone, a generator in the mine, cosmic radiation) in the vicinity of the system could alter the state of operation of the firmware so that it would inadvertently activate its firing routine to the detonators, resulting in an unintended explosion?" Anglo-Platinum commissioned a study to address this concern and this article largely coincides with the resulting report.

The chip industry is well aware of the phenomenon of randomly occurring software and hardware errors caused by external sources such as ionizing radiation. Firm errors refer to irreparable damage to the hardware elements, and attempts to mitigate the problem generally rely on the provision of redundant elements that take over if and when a firm error occurs. Soft errors arise when there are unintended transitions in a circuit's logic states. Generally, a soft error will cause a system crash. Error checking and correction (ECC) techniques at the hardware level in the form of extra bits to check each data item can mitigate the effects of soft errors. In safety-critical real-time situations these risks of soft errors can be further mitigated by fault-tolerant computing strategies, such as using one or more shadowing systems that run on separate platforms. Unlike a firm

error (*i.e.* when the hardware is permanently damaged), a soft error disappears when the system is rebooted. It is also well-known that these kinds of errors tend to occur more frequently when chip densities increase, when voltage is lower, when clock rates are higher, when systems are located higher up in the atmosphere, *etc.* These failures are generally reported as FIT per Mbit, where a FIT represents one "failure in time" during 10^9 hours of use. A 2004 source estimated the then-extant soft error rate to be in the range of 1000 to 5000 FIT per Mbit [4]. It points out that: "Even using a relatively conservative error rate (500 FIT per Mbit), a system with 1 GByte of RAM can expect an error every two weeks; a hypothetical Terabyte system would experience a soft error every few minutes. Existing ECC technologies can greatly reduce the error rate, but they may have unacceptable tradeoffs in power, speed, price, or size."

While a comprehensive survey into soft- and firm error research is beyond the scope of this study — see Tezzaron Semiconductor [4] and Saha [2] for examples of such suveys — the foregoing indicates that Anglo-Platinum's concern was not entirely unjustified, even though the experiments mentioned above did not simulate the type of soft errors conventionally addressed in the literature. The response to the company's concern is outlined below, following quite closely the report that was submitted to the company. Section 2 commences by outlining the hardware and firmware under study as well as the operating assumptions that were made. In §3 and 4 two probabilistic models are then developed, the first assuming a single spike of interference, and the second, a burst of spiked interference. Implications of the these models with respect to compliance with the international basic functional safety standard known as IEC 61508 is considered in §5. This paper is concluded in §6 concludes this paper with a few reflections about this case study.

2 Problem specification

This section briefly ennumerates the principle features of the hardware, firmware and operational context that are relevant to the analysis at hand. The specific questions addressed by this study is then formulated.

2.1 The hardware context

The CBS runs on an 8-bit microcontroller (ATmega16) whose basic memory components are shown schematically in Figure 2. The chip includes, *inter alia*, 16K bytes of in-system programmable flash memory. This memory space is addressed by a 13-bit Program Counter (PC) register, therefore implying that the 16K bytes may be viewed as an $2^{13} =$ 8K address space, each address referencing a 2-byte word.

It also has 32 8-bit general purpose registers — *i.e.* specialised high-speed memory designed to integrate closely with the execution of each instruction, and to serve as temporary storage while a program is running. In the ATmega16, six of the 32 registers can be used in pairs, effectively behaving as 16-bit registers. These are labelled as Reg27 to Reg32 in Figure 2 and may also be referred to as the X-, Y- and Z-registers respectively.

It is assumed that one of these serves as the so-called stack pointer (SP) to store a return

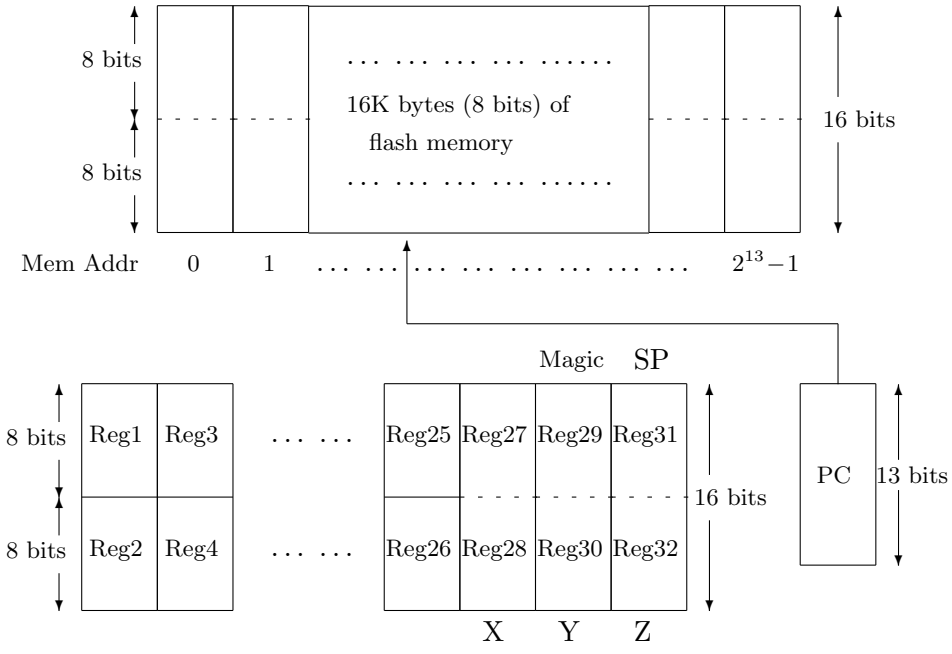


Figure 2: *An architectural overview of ATmega16.*

address when control in a compiled C-program is to return from a routine that has been called. It is therefore assumed that SP has a width of 16 bits.

It is also assumed that another of these 16-bit registers stores the "magic number" that plays a critical role in the safety measures that have been built into the firmware.

Conventional sequential programs, including the one embedded in the CBS system execute as a sequence of instruction cycles which is under control of the system's clock. During each instruction cycle all activity implicit in that instruction is completed before the next instruction is fetched from memory. Most of the 131 instructions of the ATmega16 microcontroller execute in a single clock cycle, and since the clock operates at 16Mhz, the maximum throughput is specified as 16 MIPS (million instructions per second).

For the benefit of readers not fully apprised with the notion of an instruction cycle, we provide a brief overview of the concept.

- At the start of each instruction cycle, the hardware is designed to reference the contents of the PC, copying the instruction stored at the address pointed to by the PC into a so-called instruction register. (The instruction register is not shown in Figure 2 because it is not relevant to this study.)
- Next, the instruction in the instruction register is executed, typically involving copying data into or out of registers, and/or carrying primitive operations on the register contents (*e.g.* adding two registers, or shifting bits to the left in a register).
- Still within that instruction cycle, the PC is updated to point to the next instruction

to execute. The default is to move one instruction forward, but if the currently executing instruction specifies a jump (*e.g.* because of a function call or conditional test), then the destination address is stored in the PC.

- One of the registers that may be changed or referenced during an instruction cycle is the SP. By convention, compilers are designed to use this register to store the address to which control must be returned when a currently executing function completes. Thus, for example, the C compiler breaks down a call to a function into a sequence of instruction cycles, one of which will update SP to store the return address when the function completes. Another instruction cycle will load the address of the first of the function's instructions into the PC. When encountering a return instruction at the end of the function, the contents of the SP is copied into the PC. In the next instruction cycle, therefore, the PC will carry this address, so that control returns to the point in the program just after the invocation of the function. In the present context, only the first 13 bits from the SP will be copied into the PC, since the address space is limited to 2^{13}.

2.2 The firmware context

Firmware is a generic term to refer to software that is *embedded* on a device. The CBS firmware consists of compiled C code. It is embedded in the flash memory schematically shown in Figure 2. It occupies about 12K bytes of the available 16K bytes.

At any given moment, the system may be viewed as being in a particular *state*. Such a state is defined by the contents of the various registers. In every state, the PC points to some instruction in flash memory which is executing or about to be executed. Part of the execution of the instruction involves the updating of the PC to point to the next instruction in memory to be executed. Generally, this will be the instruction adjacent to the previous one in flash memory, unless the previous instruction was a jump instruction, in which case the PC points to the the corresponding destination instruction in flash memory.

Normal calling sequence

The normal calling sequence of CBS code which results in an intended firing is in Figure 3 in terms of the structure of the high level code. It is assumed that this structure reflects reasonably the structure of the corresponding machine-level instructions.

The CBS code contains two critical routines: the `set_magic` routine and the `fire` routine. These two routines are called from one position only within the program. The outline of the code context from which this call takes place is given in skeletal form in Algorithm 1. Line numbers are provided for reference to code. Only relevant instructions are given, the remaining sections of code being indicated by elipses (`...`).

Line 4 shows the routine `ExecLong` that has to be called if the `fire` is to be executed. It is positioned within an infinite loop. The call only takes place if two conditions hold, namely: `buff[0] == 0` (line 2) and `byte_cnt <= 0` (line 3).

```
1  while(1) { ...
2    if(buf[0] == 0) { ...
3       if(byte_cnt-- <= 0 {
4          ExecLong(); ...
5       }
6    }
7   }
8   ...
9   void ExecLong(void){ ...
10       switch (buf[3]){ ...
11              ...
12           case 0xdd : if(buf[1] == 0xff) && FlagIsSet(ARMED){
13                  P_reg = ~BadPanelMask;
14                  latch_P();
15                  ClearBit(Q_reg,VBL_en);
16                  latch_Q;
17                  Clear(LED_W);
18                  set_magic();   //Sets magic = 0xaaaaaaaa
19                  ClearFlag(ARMED);
20                  Set(FIRE_en);
21                  fire(1);
22                  ...
23                  }
24          }
25  }
```

Figure 3: *CBS Code Extract*

Once control is in `ExecLong` (line 9), the calls to `set_magic` (line 18) the `fire` (line 21) take place within the body of one of many `cases` of a `switch` statement (line 10). That particular `case` is only selected if a given 8-bit variable, `buf[3]`, has a specific value (line 12). Moreover, once selected, two more conditions have to be fullfilled, namely `(buf[1] == 0xff)` and `FlagIsSet(ARMED)` (line 12) and four subroutines have to be called (lines 14 to 17) before the calls to `set_magic` and `fire` take place in that specific order.

Within `set_magic`, the clock is read and a loop with an empty body executes T times, where T is a number that had previously been determined to cause a delay of 0.5 seconds. Upon completion of the loop, clock is again read, and the time taken to execute the loop is computed. If this time is precisely 0.5 seconds, then a 16-bit variable (called `magic`) is set to a specific "magic" value. It is assumed that if external interference had caused a soft error within that intended sampling period of 0.5 seconds, then this would be exposed by an incorrect metered time reading. This could come about for a variety of reasons. For example, the timing loop might have been skipped or might have executed an incorrect number of times; or the contents of the registers containing the metered clock ticks might have been altered. At any event, the post-condition of `set_magic` is taken to be that the `magic` variable has a specific value that it did not have before, if and only if there has been a 0.5 sec period of uninterrupted processing. This `magic` variable's value is not changed anywhere in the correct subsequent code execution path.

After returning from `set_magic`, two further routines are called, after which `fire` is called with parameter 1. In determining whether or not to cause a firing of a detonator, this `fire` routine therefore takes the post-condition of the `set_magic` routine into account — a firing will only take place if the `magic` variable is correctly set.

For the present purposes, it is unnecessary to give full details of the `fire` routine. In

broad outline, it has been designed to generate approximately 0.5 seconds of pulses to the detonator, and has the following structure: `fire` executes three loops, each of which contains essentially the same logic. The first loop executes 12500 times, the second 37500 times and the third 50000 times. In the body of each loop, a boolean 8-bit variable, (`MODE`) is checked. If its value is `false`, then a sequence of instructions is executed. Irrespective of the value of `MODE` another sequence of instructions is executed, after which a test is carried out on `magic`. If the value of `magic` is different from that set by the `set_magic` routine, then bail out instructions are carried out; otherwise the routine continues normally, resulting in a firing. Note, therefore, that before a successful firing, the `fire` routine checks the value of the `magic` variable $12\,500 + 37\,500 + 50\,000 = 100\,000$ times.

2.3 The assumed operational context

Under normal operating conditions, the state of the system at any given moment is entirely determined by its previous state, and the next firmware instruction to be executed.

It is conjectured that human (or natural) activity in the CBS's operational environment could conceivably produce electromagnetic radiation that might interfere with the normal operating conditions of the CBS. This interference could either be harmless in that the CBS unexpectedly aborts and needs to be reset to continue functioning; or harmful, in that the CBS software unexpectedly enters its firing routines, resulting in an inadvertent firing. To date, this claim of potential electromagnetic interference is entirely a matter of conjecture, since no interference of this nature has been noted in an operational context at all.

It was assumed that random disruptive events could potentially alter the contents of the system's registers — *i.e.* registers were assumed to be subject to soft errors, as discussed in §1. However, on the advise of the electronic engineer who was acting on behalf of this project's client, it was assumed that the contents of the firmware in flash memory should not be regarded as susceptible to soft errors. Although firm errors could conceivably occur, it was considered beyond the scope of this investigation.

Before a disruptive event, the system's state (*i.e.* as reflected by the contents of its various registers) is dictated by the code executed to date. In particular, the PC points to some instruction in flash memory which is executing or about to be executed. When a disruptive event occurs, the state of the system can potentially change. Immediately after the disruptive event, it is assumed that the system continues to execute the code, based on the existing state of the system, despite the fact that the state might have unexpectedly changed. This means, *inter alia*, that the contents of the PC is taken as the next instruction to execute even though it might have been changed; the contents of SP is used as the return address should a return instruction be encountered in a subroutine; and the contents of various registers continue to be used to represent the value of various program variables.

It is worth briefly reflecting on the instruction register discussed earlier. Recall that its contents is changed every instruction cycle. It was decided not to model directly the possibility that a disruptive event may distort its contents. Instead we model the secondary effect that such a change might cause by bringing about unintended changes to the values

of SP and/or PC.

2.4 The assumed interference patterns

In the absence of hard statistical evidence about (a) the frequency of occurrence of the conjectured electromagnetic sources in the operational context of the CBS; (b) the nature of such electromagnetic sources; and (c) the likelihood that such sources will have an impact on the operation of the CBS, the best that can be done is to build probabilistic models of one or more scenarios that might 'plausibly' occur. Two kinds of electromagnetic disruptive events could be envisaged:

1. A single spike of interference occurs, which may change the contents of one or more CBS registers according to some assumed probability.
2. A burst of N spikes of interference occurs, where no restriction is placed on N, and each spike may change the contents of one or more registers according to some assumed probability.

In the latter case, it will be assumed that spikes are produced at a frequency of 0.5Hz or higher. This will guarantee that the 'Magic Number' mechanism embedded in the firmware will be activated. If this assumption does not hold (*i.e.* if the intervals between spikes in a burst are larger than 0.5 seconds) then the burst could be analysed as a succession of single spike occurrences, in terms of the first model.

It is conjectured that if disruptive events occur at all, then they are more likely to be of a bursty type than of the single spike type. Nevertheless, a probabilistic model of the single spike case will illuminate the probabilistic model needed for the bursty case. For this reason a single spike probabilistic model is first considered.

3 Single spike probabilistic model

In the face of these single spike disruptive events, an estimate, $P(N)$, is sought for the probability that the system will inadvertently fire at least once, if it is used N times during its lifetime. Let

- p_d be the probability that a disruptive event capable of altering register contents occurs when the system is running,
- p_f be the probability that the system inadvertently fires, *given* that a disruptive event occurs while it is running,
- $p_d p_f$ be the probability that while the system is running, a disruptive event does indeed occur, *and* the system inadvertently fires,
- $(1 - p_d p_f)$ be the probability that while the system is running, either a disruptive event occurs but the system does *not* inadvertently fire, or a disruptive event *does not* occur at all,
- $(1 - p_d p_f)^N$ be the probability the system runs N times without ever inadvertently firing,

- $P(N) = 1 - (1 - p_d p_f)^N$ be the probability that the system runs N times and inadvertently fires one or more times.

It is apparent that the smaller p_d is, the smaller $P(N)$ becomes, and indeed, if $p_d = 0$ then $P(N) = 0$. On the other hand, if $p_d = 1$ then $P(N) = 1 - (1 - p_f)^N$. This expression corresponds to a probability estimate of an inadvertent firing as a result of a 'burst' of N spikes, separated by intervals of more than 0.5 seconds.

3.1 Components of p_f

It is assumed that when a disruptive event occurs, each register changes independently with probability p_r. This means that the probability of a given register not changing when a disruptive event occurs is $(1 - p_r)$.

It is also assumed that the change is not biased to a particular set of values in a register. Thus, in an n-bit register, a change to each of the $2^n - 1$ bit configurations other than the original one is equally probable. This means that p_n, the probability that an n-bit register contains a *given* configuration after a spike, is

$$p_n = \frac{p_r}{(2^n - 1)} \approx \frac{p_r}{2^n}.$$

Thus, the probability that the PC, SP or 8-bit register change from their respective existing values to some *specific* new value is $p_r/2^{13}$, $p_r/2^{16}$ and $p_r/2^8$, respectively.

The following scenarios could lead to an unintended firing.

Only the PC changes (p_{f_1}): At the level of the source code analysis, the program appears to be most vulnerable if the PC incorrectly points to line 13, 14, 15, 16, 17 or 18. In each case, no further tests are carried about before calling `set_magic` and subsequently calling `fire`. On the assumption that each of these lines of C code are separately compiled to a few bytes of machine code, the above model suggest that the probability of the PC being changed to point to one of the six C statements could be estimated at $p_{f_1} = 6p_r/2^{13} = 7.32p_r \times 10^{-4} \approx p_r \times 10^{-3}$.

Note that this probability may be considerably reduced, merely by changing line 18 to `if(buf[1] == 0xff) set_magic()`. In this case, the probability of an unintended firing reduces to $p_{f_1} = p_r/2^8 \times p_r/2^{13} = 4.77p_r^2 \times 10^{-7} \approx p_r^2 \times 10^{-6}$.

This risk can be even further mitigated by inserting additional tests into the `if` condition. For example, `if(buf[1] == 0xff && buf[3] == 0xdd) set_magic()` reduces the risk to $p_{f_1} = p_r/2^8 \times p_r/2^8 \times p_r/2^{13} = 1.86p_r^3 \times 10^{-9} \approx p_r^3 \times 10^{-9}$.

The PC and the SP changes (p_{f_2}): A single spike could cause an unintended firing if it changed the PC to point to the start of the `set_magic` routine *and* also changed SP to ensure that the return from this routine would direct control to line 19. This would mean that the 16-bit variable, `magic`, would be set to the correct value and control would lead the execution of the `fire` routine. The probability of this happening can be estimated at $p_{f_2} = p_r/2^{13} \times p_r/2^{16} = p_r^2/2^{29} = 1.86p_r^2 \times 10^{-9} \approx p_r^2 \times 10^{-9}$.

The PC and a specific 16-bit variable changes (p_{f_3}): A single spike could cause an unintended firing if it changed the PC to point to the start of the `fire` routine *and* also changed the 16-bit `magic` variable to its unique 'magic' value. This would mean that the firing sequence would commence, with no bailout as execution progressed. The probability is similar to the previous one, namely $p_{f_3} = p_r/2^{13} \times p_r/2^{16} = p_r^2/2^{29} = 1.86p_r^2 \times 10^{-9} \approx p_r^2 \times 10^{-9}$.

Under the assumption that there are no other routes to a firing sequence, p_f may be estimated as $p_f = p_{f_1} + p_{f_2} + p_{f_3}$.

This line of argument has not accounted for the fact that each C instruction is compiled to several machine instructions. Most of these instructions are two-bytes long (and thus start at an address to which the PC could point). In some cases, an additional two bytes of data could follow on the instruction, and if the PC erroneously pointed to such bytes, the data would be construed by the hardware as a machine instruction. Even though the precise consequences of a PC jump to one of these bytes would require a detailed analysis, the eventual outcome is clear: processing would continue along an execution path that either aborts the program, places it in some non-critical state, or leads to an unintended firing.

A single instruction in a high-level language such as C typically compiles down to between 1 and 10 machine instructions. (In some instances, there could be considerably more machine instructions. For example, in the case of an arithmetic expression with several operators.) There is no *a priori* reason for believing that the proportion of paths leading to an unintended firing will increase above p_f if the PC erroneously points to data or machine instructions that do not correspond to the start of a C instruction. Nevertheless, to err on the side of conservative estimates, it was decided to increase the estimate of p_r by a factor of 10. Under such an assumption, without the recommended change in code mentioned above $p_f \approx k_1.p_{f1} \approx p_r \times 10^{-2}$. If the recommended code changes are made, then this estimate for p_f becomes considerably smaller, namely $p_f \approx k_2.p_{f2} \approx p_r^2 \times 10^{-8}$.

To construct a highly accurate statistical model soft errors at this byte code level would clearly be extremely complicated. It would require, *inter alia* a detailed analysis of the actual byte-level contents of the compiled code. Such a detailed analysis is beyond the scope of this paper.

Under these assumptions, therefore, the probability of an unintended firing due to a single spike that occurs while the system is running, was estimated as

$$P(1) \approx p_d.p_r^2 \times 10^{-8}. \quad (1)$$

The interpretation of this equation depends on the interpretation given to p_d. Below, it will be assumed that p_d is the probability of a disruptive event occurring during an arbitrary hour of the system's use. In that case, the equation gives the probability of an unintended firing during an arbitrary hour of use.

4 Burst of spikes probabilistic model

The second scenario that is considered as a possible cause of disruptive events leading to an unintended firing, could have the nature of a sequence of N spikes occurring at a frequency of f spikes per second, each spike behaving probabilistically as the single spike scenario just described. It is also assumed that $f \geq 2$, *i.e.* that these spikes occur within 0.5 seconds or less from one another. This assumption has two implications:

- If the `fire` routine executes (inadvertently or not) during a burst, then at least one spike will occur. This is because it is assumed that at least 0.5 seconds worth of pulses generated during this sequence are needed to effect a firing. The spike(s) that occur may or may not deflect the program's execution from completing the firing sequence, and/or alter the `magic` variable's value.
- If the `set_magic` routine is executing (inadvertently or not), at least one spike will occur, and as a result, the `magic` variable will not be set. As explained above, this routine was specifically designed to 'detect' such a spike, by counting the determined number of ticks emitted by a hardware timer, in a loop that terminates in 0.5 seconds under normal operating conditions. The `magic` variable is only set if the measured number of ticks at the end of this period corresponds identically to the hardcoded number of ticks that occurs under normal operating conditions. Should a spike occur during the 0.5 second period, it is assumed the timer will be momentarily disrupted. This assumption is supported by the empirical evidence of the previously-mentioned simulation experiments. Thus, it is assumed that an unintended firing cannot occur as a result of inadvertently jumping into the `set_magic` routine (provided, of course, that the jump was not caused by the last spike of the N spikes in the burst). If later required, the timer synchronisation behaviour can be probabilistically modelled in greater detail. However, it is not expected that such modelling will alter the broad order-of-magnitude conclusions.

As before, it is assumed that a register's contents changes with probability p_r in the presence of a spike. Thus a register remains unaltered in the presence of a spike with probability $q_r = (1 - p_r)$. Moreover, we assume that register probabilistic behaviour is mutually independent, not only over registers but over spikes, such that the above probabilities characterise each register's behaviour at each spike that occurs.

Let $m = f/2$ denote the number of spikes occurring in a time interval of 0.5 seconds — the time needed for the `fire` routine to complete execution. Two unintended firing scenarios are possible: (a) an unintended firing sequence could start *and complete* during the burst; or (b) the unintended firing sequence could start during the burst but complete *after* the burst. We shall designate the probability of the first eventuality $P_a(N)$ and the probability of the second eventuality $P_b(N)$.

Under scenario (a) an unintended firing will occur during the burst of interference when all of the following happens.

1. One of the N spikes, say spike j, changes the 16-bit `magic` variable into the specific value required for a firing. This occurs with probability $p_r/2^{16}$ for each given j.
2. That same spike or some subsequent spike, say spike i, changes the PC to point to

the `fire` routine. This occurs with probability $p_r/2^{13}$ for each given i.

3. None of the spikes $i+1, i+2, \ldots, i+m$ disturb the PC. This occurs with probability $(1-p_r)^m$, irrespective of the value of i.
4. None of the spikes $j, j+1, \ldots i+m$ disturb the value of `magic`. For a given value of j and i, this occurs with probability $(1-p_r)^{(i+m-j)}$.

It is important to note that a *change* in `magic` and the PC is being contemplated here, implying that prior to spike j, `magic` is assumed to have contained a value other than the value required for a firing, whether its initialised value or some other value resulting from a previous spike. Likewise, prior to i the PC is assumed to be pointing to some place other than the `fire` routine.

There are, of course, potential paths to effect an unintended firing that differ from the above. For example, during the above scenario, a succession of spikes may conceivably cause the PC to be deflected off its proper course through the `fire` routine, and then return again to the point where it left off within a relatively short period of time. For the purposes of the analysis below, such eventualities are regarded as too unlikely to affect our estimations.

Furthermore, we assume that an unintended firing cannot arise in a scenario where a spike sets the `magic` variable only *after* an earlier spike has directed the PC to the `fire` routine. This is because the `fire` routine checks the `magic` variable almost immediately after it starts (and indeed multiple times after that). It is therefore reasonable to assume that `magic` would not have been set at one of these checking points and that the bail out instructions would consequently be executed.

Let $E(j,i)$ represent the event of an unintended firing that happens because of changes at spikes j and i as described by scenario (a) above. Note that j and i are constrained to be within the ranges: $1 \leq j \leq i \leq N-m$. Letting $p(j,i)$ be the probability of event $E(j,i)$, we have

$$\begin{aligned} p(j,i) &= \frac{p_r}{2^{13}}(1-p_r)^m \times \frac{p_r}{2^{16}}(1-p_r)^{i+m-j} \\ &\approx (1.86 \times 10^{-9}) p_r^2 q_r^{2m} q_r^{(i-j)}. \end{aligned}$$

Consider the two events $E(j,i)$ and $E(j+\delta,i)$ where $\delta \leq i$. These two events may be considered mutually exclusive because it cannot happen that `magic`'s value *changes* to the firing value at j and remains that value until the unintended firing at $i+m$, and also that `magic`'s *changes* to the firing value at $j+\delta$ and retains that value until the unintended firing at $i+m$. This mutual exclusiveness for any j and $j+\delta$ allows that probabilities over the different possibilities may be summed. Thus $\Sigma_{j=1}^{i} p(j,i)$ gives the probability of an unintended firing sequence starting somewhere by virtue of a change of `magic`'s value followed by a subsequent PC change at spike i.

Now let $k(i)$ denote the probability that an unintended firing does *not* occur before spike i. For the present purposes, it is not important to derive an explicit expression for this probability, but merely to keep in mind that $k(i) \leq 1$.

Then $k(i)\Sigma_{j=1}^{i} p(j,i)$ is the probability that the *first* unintended firing sequence results from a PC change at spike i. Since only one firing can actually take place, the probability

of multiple unintended firings is not relevant in this present context.

Note that there are $N - m$ opportunities at which spikes that change the PC can result in such a *first* unintended explosion. Again, each of these eventualities mutually excludes the other from occurring, since there can only be one first unintended firing.

Let $P_a(N)$ denote the probability that a PC change occurs at any one of these $N - m$ opportunities to result in the first unintended firing. Then, taking into account that $k(j) \leq 1$, it can be shown that

$$\begin{aligned} P_a(N) &= \sum_{i=1}^{N-m} k(i) \sum_{j=1}^{i} p(j,i) \\ &\leq \sum_{i=1}^{N-m} \sum_{j=1}^{i} p(j,i) \\ &= (1.86 \times 10^{-9})[(N-m+1)q_r^{2m} p_r - q_r^{2m} + q_r^{N+m+1}], \end{aligned} \tag{2}$$

where the final line has been derived by substitution and elementary algebraic manipulation. The computation of $P_b(N)$ under scenario (b) described above follows a similar line of reasoning. The probability, $p'(j,i)$, of an unintended firing as a result of events at spikes j and i in the range $1 \leq j \leq i$ and $N - m + 1 \leq i \leq N$, is given by

$$\begin{aligned} p'(j,i) &= \frac{p_r}{2^{13}}(1-p_r)^{N-i} \times \frac{p_r}{2^{16}}(1-p_r)^{N-j} \\ &\approx (1.86 \times 10^{-9}) p_r^2 q_r^{(N-i)} q_r^{(N-j)}. \end{aligned}$$

There are now m opportunities at which a spikes can initiate the *first* unintended firing sequence, namely at spikes $N - m + 1, N - m + 2, \ldots, N$. Similarly to the case of $P_a(N)$, it can be shown that the probability $P_b(N)$, that the first unintended firing happens at any one or more of these opportunitiesm is bound from above by

$$\begin{aligned} P_b(N) &\leq \sum_{A}^{N} \sum_{j=1}^{i} p'(j,i) \\ &= \frac{q_r^2 (1.86 \times 10^{-9})(1-q_r^m)(1-q_r^{m-1})}{1+q_r}, \end{aligned} \tag{3}$$

where $A = N - m + 1$As a result, for the given frequency, $f = m/2$, given probability of a register change, p_r, and a given number of spikes in a burst, N, the probability of a first inadvertent firing in such a burst of N spikes can be computed as $p_f = P_a(N) + P_b(N)$ and an upper bound on p_f can be derived directly from (2) and (3) above.

If p_d is the probability of a burst of N spikes occurring during during an arbitrary hour of operation, then the probability of an unintended firing during such an arbitrary hour is given by

$$P(N) = p_d p_f. \tag{4}$$

The implications of these probabilistic models are discussed in the next section.

5 IEC 61508 compliance

It is instructive to consider the implication of equations (1) and (4) in relation to the Safety Integrity Levels (SIL) as specified by IEC 61 508. (See, for example, [1].) These come in two categories for a given device. The first relates to a device characterised by 'low demand mode of operation,' and reflects the average probability of failure of the device to perform its design function on demand. The second is more stringent, being characterised by 'high demand or continuous mode of operation.' In this case, the probability of a dangerous failure per hour is required. In each case, probability intervals are specified for each of four levels, level 1 being the least stringent, and level 4 being the most stringent.

5.1 The single spike scenario

Table 1 should be interpreted as follows. The SIL levels shown in the first column of the table correspond to the probability ranges shown in the second column. These ranges are prescribed by the most stringent of the two IEC 61508 SIL categories. In terms of this IEC 61508 standard, a device is at SIL 1, for example, if its probability of failure within an hour of use is greater than 10^{-6} but less than 10^{-5}.

The last two columns of the Table 1 reveal the SIL performance of the CBS as derived from equation (1) to estimate the the CBS performance in the presence of a single spike of interference. To compute the probability of CBS failure during one hour of use (subject to the various assumptions that were made in deriving this equation) the probability of a potentially disruptive event during that hour of use is taken to be *absolutely certain*, *i.e.* $p_d = 1$.

SIL	Range	p_r	$P(1)$
1	$\geq 10^{-6}$ to $< 10^{-5}$	—	—
2	$\geq 10^{-7}$ to $< 10^{-6}$	—	—
3	$\geq 10^{-8}$ to $< 10^{-7}$	1	$\approx 10^{-8}$
4	$\geq 10^{-9}$ to $< 10^{-8}$	0.1	$\approx 10^{-9}$

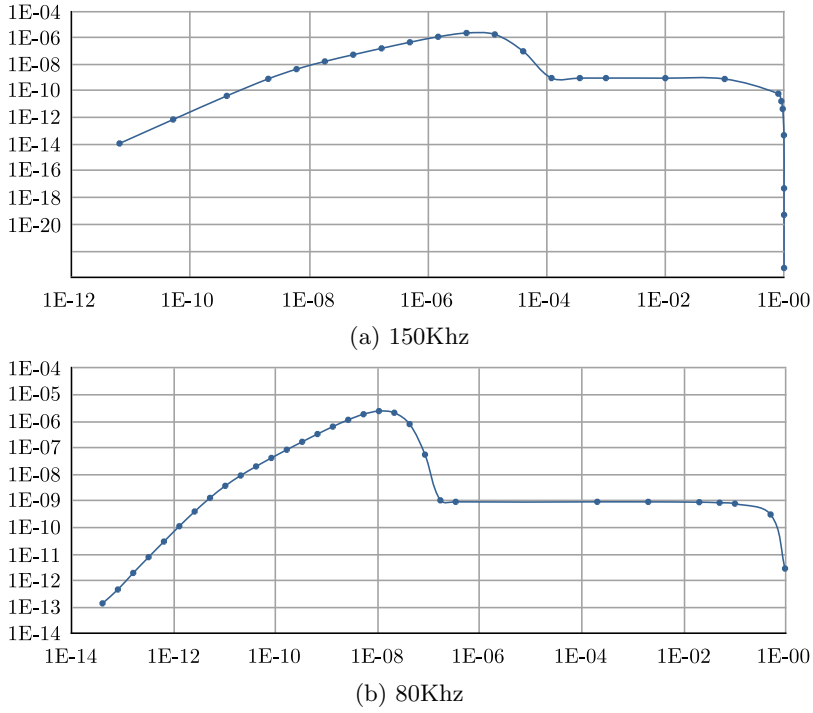
Table 1: *The IEC 61 508 Safety Integrity Levels for the Single Spike Scenario.*

Table 1 gives examples of p_r values for which the SIL level will be 1, 2, 3 or 4. (Recall that p_r is intended to reflect the probability that a register will change its value in the presence of an interference spike.)

It is seen that even in the worst case, where it is assumed that in the presence of a spike a register *definitely changes its value* (*i.e.* $p_r = 1$), the CBS attains the SIL level 3. In cases where p_r is more realistic (*e.g.* $p_r = 10\%$) SIL level 4 is attained. Of course, if the assumption of certainty of a spike every hour is relaxed, (even to assume $p_d = 10$ seems an overly pessimistic assumption in the total absence of any empirical observations to date) then a SIL level of 4 is comfortably attained under this single spike scenario.

5.2 The burst scenario

In order to characterise the CBS system in terms of the IEC 61 508 Safety Integrity Levels in relation to a burst of spikes, we use equation (4) to estimate its probabilistic behaviour.



(a) 150Khz

(b) 80Khz

Figure 4: *The value of for the function $P(N)$ when the values of f and p_r are varied.*

We make the assumption that during an hour of observed operation of the system, a burst of interference is *always* experienced that lasts for the full hour. Hence, in computing $P(N)$, we not only consider the CBS system's behaviour under the most stringent of the IEC 61508 SIL categories; we also make the unrealistically harsh assumption that $p_d = 1$.

The upper bound probability estimates shown in inequalities (2) and (3) depend on the values assigned to N, m and p_r, where N and m in turn depend on the frequency of the spikes in the burst. Given f, then m is determined by $f/2$ and N for an hour of operation is determined by $3600f$.

Figure 4 provide graphs in which the upper bound of $P(N) = p_f$ is represented on the vertical axis for the different values of p_r indicated on the horizontal axis. The upper bounds are derived from inequalities (2) and (3). Note that logarithmic scales have been used on both axes. In Figure 4(a) the value for p_f has been computed under the assumption that $f = 150$Khz, while in Figure 4(b) $f = 80$Mhz was assumed. The frequencies of 150Khz and 80Mhz were chosen by the client of this study. The reason for this choice is because frequencies in the range 150Khz to 80Mhz were recommended and used in an South African Bureau of Standards (SABS) approved susceptibility safety test relating to detonation in civil blasting operations [3, Paragraph 5.3].

The overall shape of the graphs is similar. In each case, p_f increases with increasing p_r to a maximum value, then dips to a plateau (there is actually a slight decline in the detailed plateau data) and then dips very sharply towards 0 for larger values of p_r. That p_f does not increase monotonically with increasing p_r might seem counter-intuitive. However, one should bear in mind that rising values of p_r not only increase the probability of knocking

registers into a state that will lead to an unintended firing; these rising values also increase the probability of knocking registers already in such an undesirable state back into a state where an unintended firing will *not* occur. The choice of `magic`, and other tests within the software have rendered the set of states in which an unintended firing will occur to be relatively small, and relatively easily neutralised by a subsequent disturbance in the registers. Such disturbances being more likely when p_r has a larger value.

In both examples in Figure 4, the maximum value is below 10^{-5}, the upper bound of the SIL 1 range. This maximum $P(N)$ value is reached for $p_r \approx 10^{-5}$ for 150Khz, and for $p_r \approx 10^{-8}$ for 80Mhz. In the case of 150Khz frequencies, $P(N)$ values less than the upper bound of SIL 4 level (10^{-8}) are attained when $p_r < 10^{-8}$ and when $p_r > 10^{-5.5}$. In the case of an 80Mhz frequency, $P(N)$ attains this SIL 4 upper bound level when $p_r < 10^{-11.5}$ and when $p_r > 10^{-7}$.

Overall then, under the assumptions of the study, the CBS system complies with SIL 1, 2 or 3 for $p_r \in [10^{-8}, 10^{-5.5}]$ ($p_r \in [10^{-11.5}, 10^{-7}]$) at 150Khz (80Mhz respectively) and outside those ranges it performs at the even higher safety levels of SIL 4 or better.

6 Reflections

Although the phenomenon of soft errors has been studied from a variety of perspectives, no studies similar to this one have been traced — *i.e.* studies in which models have been built that attempt to probabilistically describe software behaviour in the face of soft errors. Hopefully the broad approach taken here will highlight to future researchers the most important issues, compromises and assumptions that need to be considered.

Many assumptions had to be made in order to arrive at tractable formulae. As a matter of practicality, the analysis was constrained to view potential trajectories of program execution at the level of C code. Clearly, there could be several additional paths through the byte-code that could cause an unintended firing in the face of a burst of disruptive events. Determination of all byte-code level paths was not feasible in the time available for the study. However, it is conjectured that in probabilistic terms, each such path would add a probability term into p_f as determined in equations (2) and (3) of roughly the same order-of-magnitude as p_f itself. Such an addition is unlikely to materially affect the broad conclusions reached by this study.

It therefore seems reasonable to believe that the above models of the CBS in the presence of disruptive events reasonably reflect its order-of-magnitude performance. It should also be emphasised that these models have been built entirely in the absence of hard empirical evidence of the frequency, nature and influence of disruptive events on the CBS. The only evidence at hand was a simulated experiment that was able to induce an unintended firing on a defunct version of the firmware under conditions that violated the standard operating conditions. Even in this case, it was not possible to replicate this unintended firing, once the `magic` routine had been incorporated into the code.

This absence of evidence does not prove the non-existence of potential disruptive events. There may indeed be such disruptive events whose precise (or even approximate) nature have not been anticipated in this study, and which could lead to unintended firing, albeit

under extremely rare and unforeseen circumstances.

Acknowledgements

The support of Anglo-Platinum and Sasol Nitro in carrying out this study is gratefully acknowledged. I am also grateful to Ray Greyvenstein, the inventor of the CBS system, for his many helpful suggestions.

References

[1] Exida, 2006, IEC 61508 overview report : *A summary of the IEC 61508 standard for functional safety of electrical/electronic/programmable electronic safety-related systems*, Technical report, Sellersville (PA).

[2] Saha GC, 2006, *Software based fault tolerance — A survey*, [Online], [Cited April 25^{th} 2012], Available frm: `http://www.acm.org/ubiquity`.

[3] South African Bureau of Standards, 2005, *The design of detonator initiated systems for use in mining and civil blasting applications — Part 1*, Technical Report, Pretoria.

[4] Tezzaron Semiconductor, 2004, *Soft errors in electronic memory — A white paper*, Technical Report, Tezzaron Semiconductor, Naperville (IL).