Software Fault Tree Analysis

Brian Schmidt

Abstract

A meaningful approach to fault tree analysis of safety critical software is provided. The approach defines boundaries within the software around the flow of critical data from input(s) to the controlled safety critical output(s). The concepts put forth include details on failures within processor peripherals, software data, flow of control, with a direct focus on limiting the scope of analysis and prevention or detection mechanisms to provide the most useful input within typically tight time constraints. Maintaining this focus provides the greatest value add for the time spent.

Background

Working in industries where products and the software on those products either play a role in maintaining a safe environment or may have detrimental impacts to a safe environment comes with agencies, standards, and an ethical obligation for due diligence in making that software safe. Standards set requirements and offer some guidance. The details of analyzing software for failures and safety and the effort put into this is still left pretty much to the developers of the product.

From the FDA (Food and Drug Administration), Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices. This guidance sets a requirement for performing a device hazard analysis.

We recommend that you submit a Device Hazard Analysis for all Software Devices. The Device Hazard Analysis should take into account all device hazards associated with the device’s intended use, including both hardware and software hazards. We recommend that you present the information in tabular form with a line item for each identified hazard. This document can be in the form of an extract of the software-related items from a comprehensive risk management document, such as the Risk Management Summary described in ISO 14971. In this format, each line item should include:

- identification of the hazardous event
- severity of the hazard
- cause(s) of the hazard
- method of control (e.g., alarm, hardware design)
- corrective measures taken, including an explanation of the aspects of the device design/requirements, that eliminate, reduce, or warn of a hazardous event; and
verifying that the method of control was implemented correctly.

When performing a hazard analysis, we recommend that you address all foreseeable hazards, including those resulting from intentional or inadvertent misuse of the device.

The guidance also recommends including internal software tests and checks; error and interrupt handling; fault detection, tolerance, and recovery characteristics; and safety requirements in the software requirements.

In addition, with respect to probability that a failure will occur, the following guidance is provided.

The risk associated with Software Devices varies over a continuum from negligible to very severe. In general, FDA considers risk as the product of the severity of injury and the probability of its occurrence. However, software failures are systemic in nature and therefore the probability of occurrence cannot be determined using traditional statistical methods. Therefore, we recommend that you base your estimation of risk for your Software Device on the severity of the hazard resulting from failure, assuming that the failure will occur. We also recommend that you use risk identification and control techniques described in consensus standards such as ISO 14971.

From AAMI (Association for the Advancement of Medical Instrumentation), ANSI/AAMI SW68 Medical device software – Software life cycle processes.

Compliance with ANSI/AAMI/ISO 14971, Medical devices—Risk management—Application of risk management to medical devices is required for use of this standard.

At least one individual directly involved in the software development shall participate in the device risk management activity to ensure that risks associated with software are adequately addressed.

Software functionality whose failure could result in the hazards identified in the medical device risk analysis activity of ANSI/AAMI/ISO 14971 (see 4.2.1) shall be identified. This identification shall include hazards that could be the direct result of software failure or for which software implements a risk control measure.

Potential causes to be considered shall include, as appropriate:

a) Software defects in the identified functionality;

b) Failure or unexpected results from OTS software; and
c) Hardware failures or other software defects that could result in unpredictable software operation.

From ISO (International Organization for Standardization), ISO 14971 Medical devices – Application of risk management to medical devices. This standard sets requirements for performing a risk analysis. The opening paragraph captures that risk analysis is not black and white, but should be done with a systematic approach. This paper helps define that approach with a transition plan between the system risk analysis down into a software fault tree analysis for software affected items captured at the system level.

The requirements contained in this International Standard provide manufacturers with a framework within which experience, insight and judgment are applied systematically to manage the risks associated with the use of medical devices.

From AAMI (Association for the Advancement of Medical Instrumentation), AAMI TIR32 Medical device software risk management. This report, while not a standard setting requirements, provides good technical information for identifying and managing risks in software.

From IEC (International Electrotechnical Commission), IEC 61025 Fault Tree Analysis. This standard provides the notation for performing a fault tree analysis. Fault tree analysis is a tool that may be applied at many levels throughout the design. At its core, it is a top down approach, starting with a known hazard and identifying failures that can cause the hazard and breaking those failures down into component parts.

The standard includes a couple of techniques for risk mitigation, the concept of failures being AND’ed together in order to cause a hazard and combining probabilities of failures. Multiple failures at the same time are more unlikely than a single point of failure. This can be leveraged to allow for more time between a single failure and detection of the failure, however, don’t let this lead to simply not trying to detect a failure.

Probability of a hazard increases if any one of many failures lead to the hazard. Probability of a hazard decreases if multiple failures are required to lead to the hazard. However, predicting probability of a software failure is difficult to justify. Be careful when using probabilities in a software fault tree analysis, take guidance from the FDA guidance document indicated above assuming a software failure will happen. Some use of probability is probably justified in setting the length of time a failure can exist before it must be detected when multiple failures are required to cause a hazard.
A Process of Risk Analysis from System through Software

Getting to the level of defining safety related software failures is guided by a process. A software failure alone does nothing; it is internal to a processor and has no effect on the outside world or safety on its own. The problem is the resulting incorrect data or control, seen by the outside world. While this seems obvious, by recognizing this fact we can help bound our efforts of software fault analysis. In addition, recognize the difference between mitigations for hardware failures that place requirements back on software, versus real software failures.

Risk Analysis
The starting point for a good process is a System Risk Analysis. ISO 14971 is a standard for this effort. During this effort, the software should be viewed as a black box, with its controlled outputs and data flow examined for failures which may generate a hazard.

Let’s take a look at an example… *a simplified example*…

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>The system is a medical cutting laser. The inputs include a power setting (ADC) and a trigger (GPIO). The output is the laser beam (DAC and GPIO).</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1 shows a sample System Risk Analysis matrix, it certainly isn’t complete.

The terms used in the table are:

- **Harm**: Physical injury or damage to the health of people, or damage to property or the environment.
- **Hazard**: Potential source of harm
- **Cause**: Potential cause of a Hazard
- **Severity**: Measure of the possible consequences of a hazard
- **Likelihood**: Probability of a cause occurring
- **Risk**: Combination of the severity and likelihood
- **Safety**: Freedom from *unacceptable* risk of harm

Common definitions of severity are:

- **Negligible**: Little or no potential for injury
- **Marginal**: Could result in minor, non-permanent injury to the patient or user; little or no damage to the environment
- **Major**: Could result in serious injury or illness to the patient or user; may cause serious damage to the environment
- **Critical**: Could result in death to the patient or user without intervention; may cause significant damage to the environment
- **Catastrophic**: Could result in multiple deaths or serious injuries; may cause severe damage to the environment
<table>
<thead>
<tr>
<th>Harm</th>
<th>Severity</th>
<th>Hazard</th>
<th>Potential Causes</th>
<th>Likelihood (Pre)</th>
<th>Risk (Pre)</th>
<th>Recommended Mitigation</th>
<th>Likelihood (Post)</th>
<th>Risk (Post)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User burn from laser</td>
<td>Marginal</td>
<td>Unexpected Laser Activation</td>
<td>Control output shorted</td>
<td>Remote(^1)</td>
<td>ALARP</td>
<td>Use redundant control output distinctly separated from each other.</td>
<td>Unlikely</td>
<td>Acceptable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Software error(^3)</td>
<td>Frequent(^2)</td>
<td>Intolerable</td>
<td>Perform software FTA.(^3)</td>
<td>Remote(^6)</td>
<td>ALARP(^7)</td>
</tr>
<tr>
<td>User eyesight damaged from laser</td>
<td>Major(^8)</td>
<td>Unexpected Laser Activation</td>
<td>Software error</td>
<td>Remote(^4)</td>
<td>ALARP</td>
<td>User manual and labeling will require use of eye protection. Perform software FTA.(^9)</td>
<td>Incredible</td>
<td>Acceptable</td>
</tr>
</tbody>
</table>

*Figure 1- Representative system risk analysis*

*There are certainly other hazards and causes even in this simple example, the point here is to demonstrate selections of severity, likelihood, and risk, and the incorporation of the use of a software FTA as a mitigation at the system level for software errors.*

\(^1\) Listed as Marginal because a minor burn is expected which can easily heal. In many cases consultation with an expert in the domain should be sought for properly setting severity levels.

\(^2\) Performing MTTF analysis or some other statistical approach could validate this. With a severity of Marginal anything from Remote to Probable still falls in the ALARP region.

\(^3\) Attempting to define this in greater detail is not appropriate. A look into the software is really required.

\(^4\) While this certainly isn’t expected, we take guidance from the FDA guidance document and assume it will happen. This is the easiest to justify.

\(^5\) This indicates the more detailed analysis looking at the software design and code will be done. Trying to set generic mitigations at this level adds little value.

\(^6\) There is no rule for setting likelihood following the FTA. Good practice would be to see how well bounded the error conditions can be based on the FTA, also be sure to test FTA driven mitigations by injecting software errors. Finally, base this more on the severity than the Likelihood. If the severity is very high, the software self checking may not be sufficient.

\(^7\) Resulting risk can remain ALARP, it does not need to be Acceptable, it can not be Intolerable.

\(^8\) Listed as Major because eyesight may be permanently damaged.

\(^9\) This is listed as Remote because not only does the software have to fail, but the laser has to be pointed directly at the eye.

\(^10\) Use of eye protection may be sufficient, but since the hazard and cause is the same as above for which we are already performing an FTA, we will document that this is being done, it simply strengthens the case being made.
Common definitions for likelihood are:

- **Incredible**: $<10^6$ (1:1,000,000) events per year per device
- **Unlikely**: $10^4$ to $10^6$ (1:10,000 to 1:1,000,000) events per year per device
- **Remote**: $10^2$ to $10^4$ (1:100 to 1:10,000) events per year per device
- **Occasional**: $10^1$ to $10^2$ (1:10 to 1:100) events per year per device
- **Probable**: $1 - 10^1$ (1:1 to 1:10) events per year per device
- **Frequent**: $>1$ events per year per device

A common approach for combining severity and likelihood into risk is based on the following table:

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>Negligible</th>
<th>Marginal</th>
<th>Major</th>
<th>Critical</th>
<th>Catastrophic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequent</td>
<td>ALARP</td>
<td>Intolerable</td>
<td>Intolerable</td>
<td>Intolerable</td>
<td>Intolerable</td>
</tr>
<tr>
<td>Probable</td>
<td>ALARP</td>
<td>ALARP</td>
<td>Intolerable</td>
<td>Intolerable</td>
<td>Intolerable</td>
</tr>
<tr>
<td>Occasional</td>
<td>Acceptable</td>
<td>ALARP</td>
<td>Intolerable</td>
<td>Intolerable</td>
<td>Intolerable</td>
</tr>
<tr>
<td>Remote</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>ALARP</td>
<td>ALARP</td>
<td>ALARP</td>
</tr>
<tr>
<td>Unlikely</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>ALARP</td>
<td>ALARP</td>
</tr>
<tr>
<td>Incredible</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>Acceptable</td>
<td>ALARP</td>
<td>ALARP</td>
</tr>
</tbody>
</table>

- **Intolerable**: Risk is not acceptable
- **ALARP**: Risk is to be reduced to “As Low As Reasonably Possible”
- **Acceptable**: Risk level acceptable

**Discussion on severity, likelihood, and risk**

Selecting a likelihood of software failure may be difficult to justify. For hardware failures a statistical mean-time-to-failure (MTTF) analysis or reliability testing can be used to add science to the numbers. For software the FDA guidance suggests assuming the failure will happen. Does this mean all software failures should be listed as frequent?

Severity is a combination of domain knowledge, length of exposure times, in cases such as the laser example – levels of energy, etc… It is possible to reduce the severity level through mitigation if length of exposure time is reduced. Before performing mitigations one should assume that the device does not limit exposure to the hazard, the user may themselves be able to limit exposure.

For risk, there are three broad categories: Acceptable, ALARP, Intolerable. The only hard and fast rule is that Intolerable is unacceptable. Something must be done to mitigate this situation. Once risk is down to ALARP, good practice is to review and agree on a determination of ‘reasonably mitigated’ with the stakeholders of the project. Further mitigation becomes unreasonable in areas such as: making the product too costly or otherwise
incapable of being built or sold, or severely limiting the necessary functionality of the device. Risks deemed Acceptable should not simply be ignored. The risks should still be reviewed and if there are simple mitigations that can be put in place, they should be.

Hardware faults placing requirements on software
For failures occurring outside of software, mitigations may be put in place which place requirements on software to perform additional monitoring, redundant tasks, etc… These should be handled as any other software requirement with traceability performed to testing. The mitigation does not need to be further analyzed for failure.

Software faults
Software errors should be expected, analyzed and dealt with through external hardware mitigations or surrounding safety system or with software self checking and defensive coding techniques derived out of analysis. This paper is presenting the use of FTA for this with a way to scope this effort. It is insufficient to simply say the errors will be mitigated by good design practices and testing.

Need to Consider Hazard Timing and Impact on Architecture
Before continuing full steam ahead analyzing the software for what types of errors might generate the conditions defined in the system risk analysis, it is important to consider the timing of the failure in relation to when it becomes a hazard.

Virtually any mitigation that comes out of an analysis with a single core processor with no MMU support will be a detect-and-correct type mitigation rather than a prevent type mitigation. This means the failure will exist for a period of time, and it must be confirmed that this length of time is acceptable for the failure to exist without becoming a hazard or causing harm. If the failure becomes an immediate hazard or causes immediate harm it is likely the system architecture will need to include some type of redundancy or hardware based safety wrapper around the software. If not, the time limit is still necessary to understand for defining mitigations through the upcoming software analysis. This definition of time will strongly influence the design of mitigations. How often do you refresh outputs? How quickly does the watchdog have to fire, if a timing deadline is missed?

Considering a Failure in a Mitigation to a Failure
Typically systems are analyzed to a single level of failure. This means that failures within mitigations do not need to be further analyzed. One should be careful to stick to this rule, also recognizing when this rule can not be followed.

Some industrial safety standards require analysis to multiple levels of failure (usually two or three) for certain safety categories. This may likely lead to a redundant design.

Also, even when only required to analyze to a single level of failure by standards, consideration should be given to how long the mitigation must run. For example, if the mitigation is a back up system to keep the device operational, then consideration should be given to the length of time for repair once the back up system has taken over. Another consideration is how long the system runs without service (by humans or self tests) for checking that the mitigations are still in working order.

Types of Systems and Software with Respect to Hazard Mitigation
Many systems and mitigations are designed to fail safe. If not working properly the system shuts down. This is great until the shutdown of the device is a hazard itself. It is important to identify any requirement that must be met even in the presence of failures. The system must
be able to degrade in the face of failures, but in the face of a defined number of failures still be able to perform specific requirements. In general such a system can be very difficult if not impossible to design without redundancy. A level of reliability is likely required with preventative maintenance required before any expected failure. All in all, it gets much more complicated if fail safe is not an option. While analyzing the internals of the software as presented in this article may still be appropriate, it likely won’t be a solution in itself, and possible decreases in importance with more focus on the redundant design.

Some examples of redundant system architecture are: two or more processors running the same software (homogeneous); two or more processors running different software designs (heterogeneous), and it can be desirable to maintain independence of designers in the design of the different versions of software; or a primary controlling application with a safety monitoring wrapper.

For non-redundant solutions it can be helpful if the processor includes a memory management unit (MMU) for true process support, although often designs use a single core processor with no MMU support, limited to a threading model in which all software has access to all resources. Some processors without MMU support may still support privileged execution modes which may be leveraged for mitigation purposes.

**Fault Tree Analysis**

After performing a system risk analysis, it is time to consider a deeper dive on software failures causing hazards. It should be recognized that while the system risk analysis can be performed at the time the system architecture is done and the role of software is defined, a deeper dive is difficult and certainly can not be completed until much later into the design and even implementation of the software. This can imply rework if the software design is not able to be properly analyzed, due to things such as: lack of cohesion, coupling, or inability to implement identified mitigations. The software design team must be aware of the system risk analysis, and must at least design with this in mind.

A deeper dive may not be necessary if the software failure can be mitigated by a hardware wrapper. This is often an ideal solution, as it is easier to understand, justify, test, and control. Of course, it may be cost or space prohibitive.

**Defining a Safe Channel through the Software Cloud**

Without constraints, fault analysis of software is difficult if not impossible given the nature of software. All of the software is ultimately running within the same processor core, all of it having access to the same memory space and peripherals, and all of it interdependent on itself to meet timing requirements.

Looking at one very common failure, memory corruption, will quickly branch out of control without techniques to close the boundary of analysis. There is typically not time or budget to look at one hundred percent of the source code identifying reasons why memory would get corrupted, and being honest with ourselves, this would probably not be considered a valid approach anyway.

Defining a channel through the software starting with the controlled outputs of interest back to the inputs from which the control is derived confines the analysis and mitigations. The channel captures code, data, timing, and hardware peripherals that participate in the transformation of input to controlled output. A good software design is critical to success in narrowing this scope.
Specific Types of Software Faults and Mitigation Techniques

With the scope defined through the system risk analysis and defined channels through the software, it is time to take a look at types of software faults that may corrupt the channel and affect the controlled output.

The goal is to clearly define types of software failures and ways to mitigate them that allow for a reasonable fault tree analysis effort that is justifiable with high value add.

Memory Corruption
Memory corruption can be one of the most difficult items to encompass in a fault tree analysis. Corruption is random and the affect of corruption produces a great many unpredictable results. One could say too many unpredictable outputs to honestly capture in any type of realistic schedule and budget. So a better approach is to capture where specific memory corruption causes a hazard and defend against this, usually by detection of corruption and correction.

Prevention is limited, at a minimum it would require employing an MMU, but even then it would be somewhat difficult to justify that a process couldn’t corrupt its own memory. An approach is to use small processes that are very well analyzed and tested.

Data (Stack or Heap)
Data memory, whether the stack or the heap, may be corrupted, causing invalid values resulting in invalid decisions. Pointers may be corrupted, causing a cascading effect to further corruption. Stack corruption can affect the program counter, leading to out of order execution.

A processor with MMU support is a great first layer of protection. It can isolate areas of concern so that the risk is of corrupted memory causing a safety hazard is lower, and allowing
for more focused review and testing. Many processors available don't have an MMU and one still has to worry about corruption coming within the process of interest.

The first thing to realize is that prevention cannot be guaranteed, and memory usage cannot be continuously checked. So the goal is to detect the corruption and react before a failure is generated or before the failure becomes a hazard while recognizing the code may be off executing incorrectly. Not an easy task.

To catch an error before it creates an external failure that is a hazard consider these techniques: safety variables (see Figure 3), refreshing input data (reading from source rather than saving and using temporary values), and CRC checks done right before the data is used.

For data outputs that could be corrupted for awhile before presenting a hazard, the output data can be periodically refreshed. This may be done by keeping a shadow copy of the output data that is CRC checked, and refreshing the output data out of a timer that is checked by the watchdog.

Think about how close the check can get to the actual point at which the error would be a fault. Think about how long the error can persist before it is a hazard. Think about the error being caused by bad software (which may have overwritten an output), or bad hardware (which may never allow an output to be set properly even if refreshed). For mitigating bad hardware a feedback loop may be needed for self test and checking of the output.

```cpp
template <class T>
class SafetyVariable {
  private:
    T value;
    T inverseValue;

  public:
    SafetyVariable() {
      value = 0; inverseValue = ~value;
    }
    ~SafetyVariable() {
      value = 0; inverseValue = 0;
    }
    void Set( T newValue ) {
      value = newValue;
      inverseValue = ~newValue;
    }
    T Get( void ) {
      if ( ~value != inverseValue ) throw "Corruption";
      return( value );
    }

    SafetyVariable<T>& operator= ( T newValue ) {
      Set(newValue); return *this;
    }

  } // class SafetyVariable

Figure 3- Sample safety variable code
```
Code
Many times code runs out of FLASH or other non-volatile, difficult to write media. This is a benefit. Alternatively, system requirements sometimes mandate that code runs out of RAM. This opens up challenges similar to the memory corruption of data. Corrupted code can lead to out of order or incorrect execution, or exceptions from invalid opcodes, etc...

A very common mitigation is to perform a CRC of the code when powering up or periodically during runtime. This is a good answer for code running out of read-only memory (and FLASH can probably be included in this bucket as well). One thing to ask is how often the CRC is checked? If just out of power up, consider how often the device will go through a reset, many products are not meant to reset. While the CRC checking will likely not be done quicker than a code failure could present a hazard, the justification is that the read-only memory is unlikely to be corrupted.

Other mitigations that are helpful are watchdog checks and sequence checks that check for out of order execution covered in the next section.

Out of Order Execution
Beyond running incorrectly, when conditions need to be met and checked before changing the state of an output, or changes to multiple outputs must be performed in a specific order, out of order execution can be hazardous.

Out of order code execution can be caused by at least a few things that could actually be quite real and plausible. While corruption of the program counter in the CPU is unlikely, the program counter is set by things that could be corrupted, such as being reloaded off the stack, or loaded with an interrupt vector from an interrupt vector table stored in RAM, and sometimes stored at address zero (NULL). It is possible to imagine any number of things affecting what the program counter is loaded from.

While easy to imagine it would be nearly impossible to prevent the possible failures with any level of confidence. It is better to detect the problem with a more general solution and focus on detecting the problem around the code path which if executed out of order could cause a hazard.

It is appropriate to fill in unused interrupt vectors to a default error interrupt handler. Also filling unused code space with an invalid instruction which can be captured in an invalid instruction interrupt handler is a good idea.

A watchdog is a common answer and may catch this, but is not a complete solution. Code could be skipped to an endpoint where the watchdog is kicked, allowing execution to erroneously continue.

Consider inline sequence checking. Either local checking that makes sure inputs are evaluated before outputs are set, or checking with a CRC code that can provide check pointing through an extended algorithm or software flow. Some watchdogs are available that require sequence coding for a proper kick rather than just a simple write.

Figure 4 shows a simple example of CRC checking a sequence. If an inadvertent jump occurs to setting the outputs before inputs are properly evaluated and an algorithm executed, a check is executed to catch and prevent this from happening. This concept can be expanded across multiple files and functions and even be combined with kicking a watchdog.
Missed Timing Deadlines (too fast or too slow)

After the code is designed to be deterministic with a timing analysis and testing performed to ensure timing deadlines are met, an FTA should focus on what can go wrong to affect timing outside of design flaws. This might include corrupted timers, or bad code execution (from things like the failures already discussed), or an unexpected flood of other events at a higher priority that weren’t properly accounted for during design.

A good way to catch missed timing deadlines is by tying them into the watchdog, with the watchdog running at a similar rate or a rate fast enough to detect a missed deadline before the error becomes a hazard. The use of the watchdog disconnects the checking from the timer subsystem of the CPU. It is not sufficient to use another timer to check timing as corruption could likely affect multiple timers at the same time.

While not always thought of when discussing timing, as the usual concern is missing deadlines, another problem may be a timer running too fast. This could be self checked by the timer routine checking against another independent timer. Be careful that the other timer is truly independent.

Corrupted Peripheral Configuration (Inputs / Outputs / PWM / …)

Processor peripheral registers may be corrupted causing various undesired results such as inputs changed to outputs, outputs changed to inputs, output values changed, A/D triggering causing out of order reading when meant to be used, PWM outputs changed, and many other things, specific to the peripherals being used.
This is similar to memory corruption, but can have an immediate effect on controlled outputs, with no ability to later code detection before effecting outputs. An MMU can again help with this. Also, some CPUs require code running in a supervisor mode, or some type of unlock/lock sequence to modify certain registers. These are all helpful. Consider the time that the outputs can be in error before causing a hazard. Consider a refresh of these registers on a basis quicker than the changed outputs cause harm. Then tie the refresh method(s) and their timing into the watchdog.

One way to do this is to maintain a shadow copy of register settings, the settings can be refreshed from this copy without rerunning algorithms. Of course, be sure to put a CRC check around the copy.

Example Fault Tree Analysis

For performing the fault tree analysis, one can follow the technique defined in IEC 61025. One thing to consider is the use of a tool which supports the symbols, or simplifying the notation so as to avoid spending a lot of time on making it look good and allowing more time for getting good content.

Let's look at the software error causing the unexpected laser activation hazard from the system risk analysis in Figure 1. First, define what will be in the safe channel from input to output. This is done by looking at the design and source code. Do this by identifying all of the code that plays a role from the trigger input to the laser on/off control output. For example, this is identified as the fairly simplified source code in Figure 5.

```
1: void init( void ) {
  2:   /* Setup GPIO for trigger as input and laser control
  3:        as output. */
  4:   GPIO_CTRL_REG = 0; /* all input */
  5:   GPIO_CTRL_REG |= LASER_ON_BIT; /* output */
  6:
  7:   /* The input needs to be polled, do that with a
  8:        timer every 10ms. Set the timer to auto reload. */
  9:   TIMER_CLKDIV_REG = TRIGGER_TMR_CLKDIV; /* count rate */
 10:  TIMER_RELOAD_REG = TRIGGER_TMR_RELOAD; /* reload */
 11:  TIMER_CTRL_REG   = TRIGGER_TMR_CTRL;    /* enable tmr */
 12: }
 13:
 14:  _interrupt void timer( void ) {
 15:    if( GPIO_REG & TRIGGER_BIT ) {
 16:      GPIO_REG |= LASER_ON_BIT;
 17:    } else {
 18:      GPIO_REG &= ~LASER_ON_BIT;
 19:    }
 20: }
```

*Figure 5 - Identified source code from input to output of laser control*

Next, perform the fault tree analysis considering the types of software faults previously mentioned, see Figure 6.
Figure 6 - Example Fault Tree Analysis
To define mitigations, the length of time the failure can exist (laser on) before it becomes a hazard must be understood. This is determined from domain experts who understand the energy level the laser can deliver and its impact. If the laser can not be on at all, then likely the architecture must be changed to use redundancy and / or hardware mitigations. For this example, the domain experts have indicated the laser can be on for a maximum of 100 milliseconds when the trigger is not pressed, or after the trigger has been released.

Before looking at software mitigations, consider hardware mitigations. For this simple example, using hardware to disable the output whenever the trigger is not pressed is an obvious and simple mitigation. This should be put in place. Now, for example, a continued look at software mitigations, see Figure 7.

**Summary**

We must recognize that software running on a single CPU without the advantage of a memory management unit, which many systems do, means that all of the software is suspect. There is no well defined safety section. It is unreasonable to assume a top down fault tree analysis or a bottom up failure modes and effects analysis can be performed well with meaningful results and few gaps across all of the software. Finding a way to confine the analysis, mitigations, and resulting test requirements not only reduces the time and budget requirements, it is likely to add greater value and be safer.

**About the Author**

Brian Schmidt is a Principal Software Design Engineer at Plexus. I have been developing embedded software for 11 years, working on projects in medical, industrial, and commercial markets. My medical work has included breast biopsy, DNA purification, and automated microbial detection devices among others. I have been a leader in tool selection, methods, and bringing the Plexus software process in compliance with ANSI/AAMI SW68:2001 while maintaining flexibility for projects across different industry sectors with varying goals. I received a BS degree in Computer Engineering from the Milwaukee School of Engineering, and a MS degree in Computer Science from the University of Illinois at Urbana-Champaign.

Brian can be reached at, brian.schmidt@plexus.com.
```c
1: void init( void ) {
    uint16_t sequenceCheck = 0x0000;
    sequenceCheck = crcUpdate( sequenceCheck, 0x0001 );
    if ( ! crcCheckCodeOk() )
        SafeStateHalt();
2:   /* Setup GPIO for trigger as input and laser control as output. */
3:   GPIO_CTRL_REG = 0; /* all input */
4:   GPIO_REG &= ~LASER_ON_BIT; /* make sure output is clr */
5:   GPIO_CTRL_REG |= LASER_ON_BIT; /* output */
6:   
7:   /* The input needs to be polled, do that with a timer every 10ms. Set the timer to auto reload. */
8:   TIMER_CLKDIV_REG = TRIGGER_TMR_CLKDIV; /* count rate */
9:   TIMER_RELOAD_REG = TRIGGER_TMR_RELOAD; /* reload */
10:  TIMER_CTRL_REG = TRIGGER_TMR_CTRL; /* enable tmr */
11:  sequenceCheck = crcUpdate( sequenceCheck, 0x0002 );
12:   if ( sequenceCheck != CORRECT_SEQUENCE_CRC )
           SafeStateHalt();
13: }
14: _interrupt void timer( void ) {
    uint16_t sequenceCheck = 0x0000;
    sequenceCheck = crcUpdate( sequenceCheck, 0x0001 );
    GPIO_CTRL_REG = 0 | LASER_ON_BIT; /* reset IO dir */
15:   if( GPIO_REG & TRIGGER_BIT ){
16:       GPIO_REG |= LASER_ON_BIT;
17:   } else{
18:       GPIO_REG &= ~LASER_ON_BIT;
19:   }
20:   sequenceCheck = crcUpdate( sequenceCheck, 0x0002 );
21:   if ( sequenceCheck != CORRECT_SEQUENCE_CRC )
           SafeStateHalt();
           KickWatchdog();
22: }
```

**Figure 7 - Source code from input to output of laser control modified with mitigations**

- Use sequence checking.
- Check for code corruption.
- Good coding practice.
- Refresh GPIO control each time.
- Use a watchdog to make sure timer stays running fast enough. Watchdog should be set based on 100ms allowed fault time.
References


5 International Electrotechnical Commission, Fault Tree Analysis, IEC 61025.