Safety-critical embedded system architectures

Jef Mangelschots

EECS X497.36 Embedded Systems Architecture

Summer 2006

Lecture Assignment 1

Introduction to safety critical embedded systems

Safety-critical systems are defined as any human-implemented system, that can cause harm to human life, through intended action as well as inaction. These systems have existed for as long as humans acquired the ability to construct stuff.

The advent of computer-driven system has not simplified matters, on the contrary. The emerging flexibility is a blessing and a curse at the same time. Computer-driven systems allow far more flexibility in controlling equipment, it has asymptotically increased the complexity and the derived hazardous conditions.

Hardware being physical in nature can show any number of defects at any given time. Software however can not go ‘bad’ over time. Most errors attributed to software are caused by problems with the requirements. Either someone didn’t specify them correctly or completely or someone didn’t understand them as intended.

The problem with software is that it creates a combinatorial explosion of possible states. This results for even simple systems in huge amount of test cases, providing you would like to test every possible case.

Theorists have postulated that it is possible to model the requirements and use mathematical techniques to validate the design. However, there are problems with this approach:

- any realistic system exhibits a huge number of possible states. These require discrete rather than continuous math.
- Logic based proofs can be attempted but usually resort to the same size or even larger than the actual software, they are more difficult (and error-prone) to construct than software, and are harder to understand than software. On top of that, there is no guarantee that the outcome of a logic proof is correct.

Black box testing, which derives all test vectors solely from the specification, is faced with a huge task of finding a manageable subset of relevant test cases. But no more. It is not sufficient to come up with a set of nominal test cases (which can be huge itself), it also needs to come up with a substantial set of test cases for off-nominal conditions. This test set is even larger than the nominal cases.

Whitebox testing, which investigates the source code is fraught with its own problems. Testing all possible branches is impossible. Even if it tests all relevant branches, it is no guarantee that the implementation is correct.

Many safety efforts in designing safety-critical applications end up with problems because they fail to identify the importance of considering safety from the earliest stages. E.g. early refrigerators had a door handle. It allowed kids to lock themselves in the fridge. One approach of slapping on safety-mitigation on safety-mitigation until theoretically no unsafe conditions remains can lead you down the wrong path. In this example overeager engineers could have built in door handles on the inside, interior lights going one while living creatures inside, emergency air supply, communication links to outside, temperature control mechanisms, explosive bolts, escape hatches, …. Instead they replaced the door handle with a magnetic strip, strong enough to keep the door closed, but not strong enough to withstand a panicked person trapped inside. Brilliant.

Safety issues are best dealt with during the earliest requirements phase. They should be the result of a systematic hazard analysis effort. In general, one tries to carefully control the conditions how a system can move between hazardous and safe states. A mitigation ensures that is more difficult to go from a safe state to an unsafe state than the reverse.

This however impact reliability. There is a misconception that safe systems are reliable. This is not necessarily true. Reliable systems work under all conditions. Safe systems do whatever is necessary to prevent hazardous situations. That could mean shutting down the system. In some cases, safe systems mean reliable systems: e.g. avionics systems. A reliable avionics systems is one that works under any conceivable situation. Such a system has the highest chance of keeping the plan in the air. Passengers on that plane tend to appreciate that capability.

If a hazard can not be prevented, the system needs to control the hazard as it occurs, e.g. a leaky valve can increase the pressure in a closed vessel. If not able to prevent a gas from being added, an independent system can reduce the pressure by venting it.

Lastly, if all else fails, safety mitigations should look at minimizing damage when it occurs.

Description of typical safety mitigations in embedded software systems

In the following section we give a list of possible mechanisms one could want to include in an embedded system design to mitigate against safety hazards:

- **dual-channel source code:** it is possible that corruption of the memory in which the program code resides, results in erratic execution, possibly resulting in hazardous conditions. A mitigation of this can be to duplicate vital code sections (e.g. calculation of control value, …) and executing both consecutively. A mechanism needs to be built in that compares both results before using them.

Consider this a poor-mans N-version programming. Beware, this technique does not mitigate against design flaws, since code is merely duplicated.

- **Stack checking:** stack usage is difficult to predict. Few techniques, if any, exist to mathematically calculate the worst-case stack usage. Stack size is usually determined based on measuring the worst stack usage during testing, and adding a certain safety factor, depending on available memory. Regardless of any best effort to predict worst-case demands, it is still possible that during execution, stack overflows. A couple of techniques are available to mitigate against this:
  - **Guard band technique:** a number of magic values (e.g. 8) are stored at the end of the stack. The system should periodically check these memory locations whether they have changed or not.
  - **Stack pointer checking:** the system can periodically do a sanity check on the stack pointer value: (1) it should be within the range indicated by top- and bottom of stack, (2) it should be within its own region, e.g. interrupt stack pointer should be within the memory block assigned to the interrupt stack. When servicing an interrupt, a stack pointer, pointing inside the main stack indicates a faulty context switch.
- **Watchdog mechanism:** it is possible that program execution stalls. A software-based watchdog mechanism consist of a separate task (e.g. an interrupt) that periodically decrements a counter. When that counter reaches zero, the system reboots. At regular intervals, the software should reset that counter to its maximum value, i.e. kicking the watchdog. It is also possible to have a hardware watchdog. Safety-critical systems typically have both.
- **Progress checking:** it is possible that faulty program code (compile errors, single-event upset, corrupted memory, …) skip erratically over blocks of code. It is not impossible that this goes unnoticed, other than that the program behaves incorrectly. This incorrect behavior can cause hazardous situations. A mitigation against this can be a progress checking mechanism. Spread over the execution of the main loop, calls can be made to a routine that updates a code word. Some techniques use pseudo-random generators, that end up with a given value after a well-specified number of calls. When used in conjunction with dual-channel source code, calls to these update functions can be inserted between invocation of primary and secondary channels of dual-channel code. At the end of the main loop, a check-routine calls the same update routine with the same seed value (typically the clock at beginning of a cycle, so that subsequent cycles will yield different code words) and same number of calls. This updates a second code word, which will be compared with the running code word. When executed correctly, both code words must be the same. If this is not the case, an error condition is declared. On top of that, a periodic task (e.g. as part of an interrupt handler, or a separate task in a multi-threaded system) can compare the code word value of any cycle with that of the previous cycle. Since they are seeded with a different value (the clock) every cycle, subsequent code words must be different. If not the case, then it can be assumed that the main loop did not proceed, or took too long to complete. The period for this test should be longer than the maximally

allowed main cycle time. A similar test can also be scheduled at a period which is less than the minimal allowed cycle time. In this case, subsequent code words must be the same.
Progress checking is complementary to watchdog mechanisms. The latter only checks whether any code is running. It does not catch endless loops in which the watchdog is kicked repeatedly.

- **Time reference checking:** real-time systems are extremely vulnerable to clock errors. No real-time clock is 100% accurate. A mitigation can be to include a secondary clock signal in the hardware, to which the primary clock is compared periodically. Short term tests can be executed every 1 to 2 seconds. Long term tests can detect marginal drift over 1 hour and 24 hour periods.
- **Tracking timing statistics:** real-time systems require timely execution of tasks and routines. Tracking timing statistics, e.g. minimum, maximum and average execution times can be range checked periodically.
- **Compatibility index checking:** when dealing with different versions of hardware and software components being deployed, incompatible combinations can cause hazardous conditions. Each software components has a compatibility matrix hardcoded embedded. Hardware components have a compatibility index encoded, accessible to software components. Each software component should check the compatibility index of each component it interacts with, against its own matrix. When an incompatibility is determined, the component should go in an unconfigured state.
- **CRC checking:** cyclic redundancy checking is a technique where checksum value is calculated over a chunk of data. This value is supposed to be unique for a given sequence of data values. CRC's can be used to guard any suspect chunk of data aginst corruption. Typical applications of CRC are protecting data packets received over a communications link, critical parameters introduced to an interface to the outside world (e.g. a GUI). Each software component (e.g. a executable, a data table, boot code, …) can be accompanied with a CRC value, and should be verifiable by the user. A system can calculate the CRC of each component and present it to the user, who can compare it with a given CRC. Also possible is to have the system compare the CRC with a hardcoded CRC.
- **UCN:** universal checksum numbers are CRC's calculated over a set of CRC's. Similar to compatibility checking, when it is crucial that a well specified combination of configurable items are to be combined (e.g. a specific version of the software must be loaded on a specific revision of the hardware, …), a CRC can be calculated over a set of all CRC's of each loaded component. Comparing this calculated UCN with a specified UCN can guarantee the user the correct combination. An even stricter application can withhold the calculated UCN from the user. The user then has to enter the required UCN. If it doesn't match with the calculate value, the system refuses to go into configured mode.
- **Copy checking:** If different copies of data reside in the memory (e.g. sometimes program code is copied from ROM to RAM and executed from there), a periodic

check of each image can guard against spurious mutations. CRC’s are a useful technique to quickly establish the integrity of large data chunks.

- **Peripheral integrity checking:** a system with peripheral devices can periodically do health check to establish the integrity of the device.
- **Initialization of unhandled exceptions:** embedded systems typically have a fixed number of possible exceptions, each of which can be associated with a separate exception servicing routine (similar to interrupt handlers). All exceptions that do not have a particular handler, should be associated with a default handler. This default handler could log an error condition and reset, or another action that restores the system in a safe state.
- **Interrupt/Exception vector checking:** the interrupt and exception vector table integrity can be periodically checked by calculating a CRC over the table and comparing it with a known expected value.
- **RAM checking:** various techniques exist to validate the integrity of the RAM: EDAC, writing an reading various bit-patterns in various orders, …
- **System integrity checking:** hardware platforms can make various information available to the software running on it (e.g. power supply voltage, current drop, temperature, …). Software can sample this information on regular intervals and check it against expected value ranges.
- **Dormant failure checking:** a variation on system integrity checking is periodically stimulating each error checking mechanism with a well-defined error condition (e.g. a test pulse) and verify whether the error-detecting mechanism actually detected the error. The system should be put in a special state so that it does not report the error as being genuine. In this state, failure to report the error causes an error log.
- **Task checking:** although multithreaded architectures are typically frowned upon in safety-critical systems, it is still possible that systems resort to these architectures. If this is the case, extra attention should be payed to risks introduced by the context switching mechanism (which also must be done in the presence of interrupts). If not provided by an RTOS, kernel or library, periodic checks against deadlock and starvation must be implemented. Variations on the progress checking mechanism described earlier can be used. Better yet is to avoid deadlock by banning the use of semaphores in preference of dedicated resource handling task driven by a queuing mechanism. The latter is heavier on data copying but eliminates deadlocks.
- **Scheduler checking:** periodic checks should verify that the scheduler does not remain disabled for too long. It should also check whether all tasks are invoked within their allotted timeframes.
- **Interrupt checking:** critical sections disable interrupts. Hazardous situations can occur when interrupts remain disabled for too long. Periodic checks should verify that the interrupts are not disabled for too long. It should also check whether all interrupt handlers are invoked within their allotted timeframes. The non-maskable interrupt (if available) could be considered a candidate for this task.

- **Shared data checking:** when sharing data between different tasks and interrupt handlers, maintaining multiple copies of the same data can mitigate against unintended side-affects. When changing the data, multiple identical deliberate write operations should be executed. Data should be accessed through an interface routine that compares both copies before passing their value along.
- **Heartbeat:** providing a heartbeat mechanism (typically a blinking LED, a dedicated signal to another system) to the outside world can assure users that the system is working. This heartbeat pulse can be the result of successfully gathering satisfactory status information from all error mitigation activities in a general health checking task.
- **Communication session management:** when different seperatable components need to establish a vital communication session, mitigations can be included in the protocol to ascertain a valid communications link. A software component should only be in an operational state when a valid communication session provides a constant update of valid commands. Techniques include:
    - A predetermined number of identical commands need to have been received before a command is accepted as valid
    - Commands need to be received at a regular interval. Failure to receive a command message in a given period can break the connection.
    - Each command packet should have a timestamp – the receiver should check subsequent timestamps. They should be different but can not be too far apart.
    - Each command packet should be CRC protected
    - Each command packet should have an embedded sequence number – packets can not be missed and have to arrive in sequence.
    - Each command packet can be equipped with a pseudorandom number. Subsequent numbers should be different.
    - clock synchronization messages should be periodically exchanged
    - acknowledge packets can help establish a bidirectional assurance of the integrity of a valid communication link.

An example vital implementation of a relay control circuit

Following is a description of a vital relay output (VRO) circuit. This particular design is patented by Safetran Systems. Purpose is to mitigate against a single-point of failure to cause unwanted energizing the output. In its most simple form, a relay can be driven by a single GPIO line feeding a driver circuit that is capable of energizing a relay. However, several reasons exist that can drive this line unwanted (e.g. transient pulses, singl-bitstuck events, programming errors, …). In safety-critical systems, several lines of defense need to be in place to countercheck any possible failure.

At Safetran Systems, a VRO component needs to drive a standard railway vital relay used for safety-critical application (driving a gate-arm, warning lights, switch, …). These systems are designed such that losing power, de-energizes the relay, which is set by default to the safest condition, i.e. when anything goes wrong, de-energize the relay. A general guideline applies: make it easy to get in a safe-state, make it hard to get in an unsafe state, i.e. you have to do several independent steps, which themselves are non-trivial, to allow the system get switch to an unsafe state.

The design in "Figure 1 - vital relay output" is an example of such a system. Let's dissect it to get a better understanding of what makes a vital circuit.

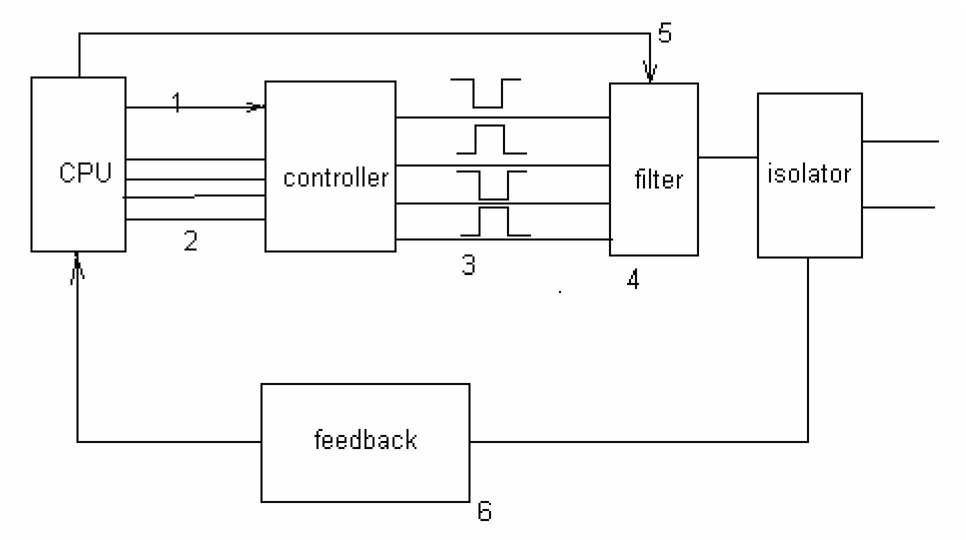

Figure 1 - vital relay output

Purpose is for the CPU to make the isolator energize it outputs on the far right. Between the CPU and the isolator are a controller, a filter and a feedback mechanism. The controller in our case is a simple PIC chip running a small program.

The first step is to drive the IRQ line into the controller (1) down. This negative pulse must be repeated periodically (e.g. every 30 ms). The controller track this interrupt for consistency (not too close to each other, not too far apart). If anything goes wrong here, the system shuts down.

The second step is an independent secondary command that tells the controller the same thing, but in a different form. In this design, instead of using a single line, 4 lines are used (2) which carry a constantly changing codeword (nibble), generated by a pseudorandom generator. The controller constantly executes the same algorithm. Both are seeded by the same clock frequency.

When the controller detects an incoming nibble-stream, which matches its own generated stream, together with a valid IRQ (1), it will continue generating an output signal to drive the relay.

Again, instead of using a single output line, 4 lines (3) drive a bandwidth filter. Each line carries a pulse train at a well specified frequency. The filter will only accept it if the frequency is in a well-defined range. Two lines are the exact mirror of the two other lines. The filter circuit (based on FET's) is designed such, that any drift in any of the lines compared to the others, causes the filter to shutdown, i.e. the controller needs to put a significant effort in keeping the output lines in sync.

On top of that, the CPU drives an ENABLE signal (5) to the filter. Only then will the filter drive the isolator to an energized state.

In good safety-critical fashion, nothing can be trusted. That's why a feedback mechanism (6) returns the actual output of the isolator to the CPU. If the CPU sees an output, it never commanded, other actions can be done, like resetting the system.

In order to check for dormant failures that can prevent a de-energize command to take place (e.g. an output can be driven to energized state for a very long time. During that time, any malfunctions can prevent a de-energization to take place when it is commanded). During energized state, a short test pulse is sent periodically on the ENABLE line (5). The feedback mechanism must detect a response from the isolator. If this is not the case, the system is reset. Since this circuit is driving a high-power railway relay, a short test pulse (in the order of microseconds) will not cause the relay to drop.

Conclusion

Safety-critical systems are harder and more costly to implement. Extra mitigations against potentially hazardous situations need to be included. However, in safety-critical design activities, there is no free lunch. These extra mitigations themselves increase the complexity and can contain errors themselves.

Extra design steps, some of which are unique to safety systems, need to be undertaken (e.g. fault tree analysis, hazard analysis, FMEA, ..) to ensure the design is safe. These activities alone will not catch all problems. Any given potential design activity (e.g. unit testing, integration testing, system testing, code reviews, design reviews, formal inspections, requirements modeling and analysis, prototypes, simulators, …) is capable of catching certain classes of problems. However, not every technique, will find them all, and not any technique can find all the problems, if any, that the other technique can find. Even when all techniques are rigorously applied, even then will not all problems be found.