High Reliability Systems

Lloyd Moore, President

Lloyd@CyberData-Robotics.com

www.CyberData-Robotics.com

Northwest C++ Users Group, June 18, 2014

Overview

Appropriate Use of This Presentation

Causes of Failures

Watchdogs

Memory Techniques

“Safer” Coding Practices

Safe Shutdown Practices

Summary

Appropriate Use of this Presentation

This presentation is intended to fill a "middle region" between normal software development practices and formal high reliability specifications such as: MISRA, DO-178B, PCI-DSS, IEC 62304 and many others.

IF THE PROJECT YOU ARE WORKING ON IS SUBJECT TO FORMAL RELIABILITY GUIDELINES AND/OR SPECIFICATIONS THIS PRESENTATION IS NOT FOR YOU – FOLLOW THE APPROPERATE GUIDELINES TO THE LETTER!

Ok if you are still reading then what is this presentation about? The above guidelines do not apply to every case and are too "heavy" for many projects. This presentation will cover techniques that can be used as needed to make a project better in terms of reliability, but without going so far as to increase the development cost of the project.

Causes of Failures

- Software bugs!!!
  - The program does exactly what you said to do, just not what you intended to do!
- Electrical and environmental noise
  - Both noisy power lines and stray induced magnetic fields can alter system state
  - ESD (static electricity) can alter memory and register contents randomly
- Operator abuses
  - They did WHAT?!?!?!?!?!?!?!?!?
- Resource limitations
  - Memory fragmentation, unexpectedly large inputs, unexpectedly long times
- Failures in other parts of the system
  - Mechanical changes and/or failures
  - Component failures which cascade, but do not fully disable the system
  - Networking and communications issues

The Basics

- Set the compiler to the most sensitive warning level, and ensure the code builds with ZERO warnings
  - Use of pragmas to clear warnings is VERY debatable
- Use "safe" libraries
  - No "naked" pointers
  - Use APIs with length checking to avoid buffer overruns
- Have an established development and test process
  - Use revision control, defect tracking, and whatever else you believe is a "best practice"
  - Do code reviews on ALL code – builds team understanding and finds bugs early
  - Unit test coverage should be 100% of critical code and as much of non-critical code as management will afford you
  - When there is a failure do a "root cause" analysis and update deficient processes
  - The key is to have SOMETHING in place to which improvements can be made over time

This slide is NOT complete as there are MANY talks and debates covering these topics!

Watchdogs

General Definition: Maintain surveillance over (person, activity, situation)

In our case refers to an independent piece of hardware which monitors the desired process and either shuts down or resets the desired process if some condition is not met.

Most common form is the Watchdog Timer, which is a dedicated piece of hardware which will reset the main processor if it does not see a specific activity happen within a specified time interval.

The activity is generally toggling an I/O line or writing one or more values to specific registers.

Two general forms of this: "on chip" and "off chip" – advantages and disadvantages to both and some feel quite strongly over which is better!

Desktop PC motherboards can also be purchased with watchdog hardware!

Watchdog Behavior

Single Stage Watchdog:

- Must toggle a line or write a specific value to a location every X mS to reset the watchdog timer
- If the watchdog reset event does not happen the watchdog resets the system / main processor

Windowed Watchdog:

- Must reset every X mS but not more often then every Y mS
- Protects against more cases than Single Stage Watchdog

Multi-stage Watchdog:

- Must toggle a line high then low, toggle multiple lines or write multiple specific values to specific locations at some predefined time interval(s)
- Specifics vary from device to device
- Key is that you can ensure multiple locations in your code are executing in the desired order

Typical Watchdog Program Pattern

```
void main()
{
   initialize_system();
      while(1)
      {
         read_sensors();
         reset_watchdog();
         write_outputs();
         sleep_until_next_cycle();
       }
}
```

General idea is that the watchdog gets reset once per main loop, will want the time out of the watchdog to be barely longer than the longest execution time of the main loop.

If using a multi-stage watchdog can position one call just before read_sensors() and another call just before write_outputs() – now you can verify that the sensors were read before the outputs were written.

Watchdog Gotchas!

DO NOT put watch dog resets into interrupt calls unless the only thing you care about is verifying that the interrupt is still running!

On-chip watchdogs are typically disabled when in debugging mode, off-chip watchdogs are not. Typical issue is you connect the debugger, hit a break point and your system resets!

- Recommended practice – have the reset signal trace on the board connected by default, but allow for a jumper location.
- On debug boards cut the trace and install the jumper, on production boards leave the trace alone, and no jumper.

As your code grows and changes the length of time for the watchdog timeout will also change – keep an eye on this as watchdog resets will look like system crashes when you are developing!!!

Memory (I/O) Lockout Regions

Specific regions in memory that are protected by some form of "lockout". These are typically assisted by dedicated hardware but can also be emulated with a MMU.

Goal is to prevent accidental writes to some type of critical control.

Various forms of this:

- Location can only been written X clock cycles after reset
- Location can only been written once after reset
- Location is protected by some other location which must have a "key value" currently written to it
  - Note in this case you may NOT want to have the "key access" and "protected value" access in a common routine!
  - Remember to always clear the "key value" when you are done updating!

Very commonly used to protect the on-chip watchdog timer, both in terms of configuring the timer and writing to the reset location.

On chips with FPGA style resources you can also build your own protection to do specifically what is needed.

Memory Allocation Patterns

In long running systems memory fragmentation becomes a big issue.

Many embedded systems run for years without a reboot and don't have any virtual memory system to "hide" fragmentation.

In these cases dynamic memory allocation becomes a source of instability!

Potential solutions:

- Use only static allocations – memory usage known at compile time
  - For embedded microcontrollers actually a very desirable solution
- Use only automatic allocations – everything will end up on the stack
  - Watch your stack space here – trades fragmentation for stack overflow
- Use dynamic memory allocation but only once at system startup
  - If using C++ may want to disable new() and delete() to prevent "hidden" allocations
  - Will also preclude using portions of the standard library!
- Use dedicated heap(s) and re-initialize it every so often

Data Integrity Issues

Data values themselves can change OUTSIDE OF program control!

Most of us are familiar with "overwrite" type problems, however in some systems this isn't the only issue. Memory can also be affected by:

- Bad memory locations
- Electrical noise
- Electrostatic discharge (special form of electrical noise)
- Environmental radiation

Note that this may not happen very often but it does happen! Electrostatic discharge is a particularly common event in many areas, particularly in low humidity conditions.

Some systems address this issue with ECC memory.

Data Validation Methods

General principle is to keep critical data in a common data structure. Now you can operate on the data as a set and this gives you some advantages:

- Data can be checked easily
- Sentinel values can be placed into the data structure and tested at regular intervals – these are constant values through the life of the program
- Whole data structure can be checksum / CRC validated at key points
- Data structure can be "mirrored" and again validated at key points

These techniques work best when the program duty cycle is low, and checking is done during the idle times.

Can also incorporate this into the watchdog reset routine such that the watchdog is only reset if the data validation tests pass.

Memory "Munging"

Embedded microcontrollers will typically have extra memory which is not used by the application. System reliability can be improved by properly filling the memory with specific data:

- Unused RAM can be filled with a given pattern, and that pattern verified as described in the data validation slide
  - Particularly useful to do this just beyond the maximum expected stack
- Unused flash/ROM memory can be filled to trigger a reset or halt if the program ever jumps out of the defined program region
  - This is a processor dependent technique
  - Fill memory with NOP instructions – will cause most processors to loop around to start of memory just like a reset (beware of memory lockouts!)
  - Fill memory with "reset" instructions – some processors have this others don't
  - Fill memory with jumps to a common safe halting or reset routine
    - Note: Generally DO NOT want to fill memory with HALT instructions! If you just halt you don't know the system is in a "safe" state

State Tracking

This is a technique where each major step of the program verifies that it was called from the correct location.

The idea is to abort if any segment of code is called from an unexpected path. By necessity this will make your program very rigid!

Typically involves some type of check at the beginning of each critical routine. For a state machine this could simply be checking the prior state as a precondition to executing the current state.

In the most general case this would be checking the call stack to make sure the caller is one of an expected set.

Validation Boundaries

The general idea here is to have specific boundaries in your program where you fully check parameters being passed and/or overall system state. Very similar in concept to threat modeling called "trust boundaries".

1. Divide your application to specific layers and modules (should be doing this anyway!)
2. Anytime flow crosses from one layer or module to another any data being passed gets "sanity checked"
3. Extreme version of this is checking at the entry to EVERY function call – may not be feasible due to knowledge or time limitations

Has the benefit of pushing "sanity checks" to the various module APIs of the application where they are most easily accomplished and most easily verified by code review to exist!

Time Calculations

Time calculations are a frequent source of "one time" errors, specifically:

- Leap year events
- End of year events
- The 49 day roll-over event (and similar) (32 bit int used as mS timer)

Recommendations:

- Always use full date / time values for calculations
- Use standard libraries for time manipulation, do not invent your own!
- Scale simple counters to have a lifetime of exceeding the maximum possible lifetime of your program execution
  - Battery powered devices, at least 2x expected battery life, assuming batteries come out and force a reset
  - Other devices just use a 64 bit int! Typically gives millions of years – good enough!

Techniques to Avoid

Recursion

- Great for solving some types of problems but in general will lead to stack overflows which are dependent on data values

Threading

- Again great for certain types of problems but getting threading correct is HARD!!
- In many embedded applications threading can be simulated by the use of interrupts
  - Hardware guarantee of priority
  - On some processors only one can be "in flight" at any time

To Halt or To Reset?????

This is a VERY application dependent question – and can go either way!

Error conditions will generally appear to happen at random times, therefore the state of your system when an error conditions occurs should not be assumed.

- Motors could be on moving machinery
- Heating / cooling elements can be on
- Communication transaction could be in process

General recommendation here is to have ONE routine which places the system into a "safe" condition for your application. This routine is called at startup and also called anytime an error condition is detected. Note that this also means the "safe" routine gets tested regularly!

Question of halting or resetting now becomes one of desired behavior – do you want the process to continue without human intervention?

Can also use "scheduled resets" to improve system reliability.

Summary

- In real world application errors can come from sources other than programming bugs.
- Make sure you are following all the “basics” of good coding practices
- Watchdog timers are the most common form of error detection used on systems, however to get maximum benefit the watchdog needs to be used correctly.
- Most non-bug related failures come as a result of environmental influences corrupting memory and there are several techniques available to detect this condition without having to resort to ECC memory.
- Knowing the expected flow of your program opens up further opportunity for validation.

Questions?