

Techniques for Fault-Tolerant Coding in Embedded Systems

Jon Fick
Westford MicroSystems
Westford, Vermont, USA 05494

April 04, 2003

© Westford MicroSystems 2003

Introduction

This application note pertains to fault-tolerant coding that can augment the two main techniques used for graceful fault recovery: a watchdog timer coupled with robust hardware. The seven coding techniques herein generally apply to all embedded systems programming; some or all may be applicable to the Microchip PIC microcontrollers and the application at hand.

The Problem

Aberant code sequence and operation in an embedded system can be initiated by either software or hardware.

Software can run into unexpected states, unexpected data, endless loops, and other events from which recovery might not be possible if the code was not designed to respond such conditions. *There is no substitute for well-designed, robust software!*

Hardware can be susceptible to EMI hits and other unusual conditions that can throw a microprocessor and associated peripheral hardware into a “scrambled” condition. Faults can sometimes be initiated by voltage shifts caused by drawing large transient currents through power traces on a circuit board. The presence of a strong, localized RF magnetic field can also introduce faults in the wiring in and around the microcontroller. *There is no substitute for well-designed, robust hardware!*

In the face of such anomalies, the watchdog timer goes a long way toward providing graceful software recovery in embedded systems. However, there are conditions that will bypass those preventative measures. For instance, normal processing loops take longer than normal processing time, or the program counter gets hit and never returns to point to coherent code, or internal/external data memory gets hit and causes unexpected results, or the main loop is corrupted but interrupts continue, or an interrupt is corrupted but the main loop continues. The following techniques may be useful toward that end.

Fault identification and recovery techniques

Fault: Program counter points to unused program space

Unless all of EPROM memory is specifically initialized by the code, it defaults to an “erased” value. If that erased value is benign when executed as an instruction, there is no hazard in leaving it unchanged. If the program counter gets hit and begins to execute these benign bytes the program counter will finally fall through to address 0000, thus recovering on its own. An example of a benign scenario is where erased memory defaults to “FF” which is interpreted as a harmless “ADDLW FF” in many Microchip microcontrollers.

If unused ROM is not automatically initialized at something that is benign, or if the program counter cannot be guaranteed to synchronize to the beginning of a multi-word instruction such as in the Intel 8031 microcontroller, then the unused areas should be purposely filled with self-synchronizing code that jumps to a recovery location. A continuous series of “LJMP 0000” instructions (02-00-00) fills both requirements. If the program counter lands on the “02”, the LJMP is immediately executed. If it lands on either “00”, these are interpreted NOP’s and fall through to the next LJMP. If desired, the LJMP could branch to an error handling routine somewhere other than 0000. In the case where a checksum byte is written in the last EPROM address it must be considered unpredictable, and therefore unused EPROM should be filled with “LJMP 0000” prior to that checksum byte. An abbreviated method, useful where erased space is benign but has a checksum at the end, is to place a single “LJMP 0000” just prior to the checksum. The probability of the program counter missing the “02” and hitting either “00” and then executing the checksum byte is extremely low.

Fault: Internal registers get corrupted

There exists the possibility that internal registers, such as interrupt-related registers, could be corrupted and leave a main loop running but stop interrupt activity. Internal or external RAM variables could also be affected.

The solution is to reinitialize the registers and variables on each pass. For anything that can’t be reinitialized, check that values are within acceptable ranges and force a reset if not. If a multi-byte (i.e. integer) value can be modified by an interrupt routine, care must be given to disabling the interrupt while checking the value.

There is no way to ascertain the correct value of some variables such as the stack, but wrong behavior resulting from these variables might be detected by other of these methods.

Fault: Outputs get random data with wrong timing

The main loop or interrupt may be able to check if outputs are being changing in an invalid way, i.e. too fast, or more than one at a time, etc. This assumes that the main loop or interrupt is still running correctly.

Fault: Loop time increase

If the code sequence is corrupted, it's possible that loops can take longer than normal to execute. Normal feeding of the watchdog timer is usually to provide an overabundance of watchdog timer resets so as to guarantee that the watchdog timer won't cause a software reset in for normal software operating conditions. If, however, watchdog resets are judiciously generated in a single main loop, at a period somewhat close to the watchdog timeout period, then if the loop time increases, the watchdog resets may exceed the timeout threshold and successfully force a code reset. This method depends on an absolutely predictable code sequence. Apply with care to avoid false fault conditions.

Caveat: the loop execution time must be predictable to avoid unwanted watchdog code reset. Also, the timeout period vs. thermal conditions must be taken into consideration for the watchdog chip. For instance, with the Microchip PIC microcontrollers consideration must be given to the thermal operating environment because the watchdog timer, being clocked by an RC oscillator, increments at a rate that is heavily temperature dependant.

Fault: Interrupt failure

Implement a watchdog-like counter in the application software whereby the main loop code checks and decrements a counter during it's normal loop operation. Set up a timed interrupt to constantly preset that counter to 100 or some other appropriate value. As long as the interrupt is running, the main loop will never find the counter at zero. If the counter does reach zero it means the interrupt has failed, and a software reset can be initiated from the main loop code.

Caveat: the interrupt must be periodic and must preset the counter to a value that is high enough to prevent the shortest possible main loop from counting it down to zero before the interrupt can preset it again.

Fault: Main loop failure

(The reciprocal of "Interrupt failure", above)

Implement a watchdog-like counter in the application software whereby the interrupt code checks and decrements a counter. The main loop constantly presets that counter to 100 or other appropriate value. As long as the main loop is running, the interrupt will never find the counter at zero. If the counter does reach zero it means the main loop has failed, and a software reset can be initiated from the interrupt code.

Caveat: the main loop must be periodic enough and must preset the counter to a value that is high enough to prevent the shortest possible interrupt repetition from counting it down to zero before the main loop can preset it again.

Fault: Code sequence is corrupted

It's possible for code sequence to be corrupted and yet have some function calls operating when they are crossed. But when such functions execute their return instruction, control will return to the incoherent code that was being executed prior to the function call.

A solution is to initialize a single-byte variable at the beginning of a main loop or interrupt and then check the variable for that correct value at sequential locations in the code. The variable is incremented each time it is checked. This is done after every several steps in code, especially around function calls, or more often if execution time is not constrained. If the execution gets out of sequence, the variable will have the wrong value when it is checked and a software reset can be forced.

A function is declared:

```
void Checkpoint ( UCHAR cX )
{
    if ( cCheckpointML++ != cX )
    {
        reset_cpu();
        /* or "#asm GOTO 0000 #endasm" */
    }
}
```

and then used at strategic points in the main code as follows:

```
.
.
Checkpoint ( 1 );
.
.
Checkpoint ( 2 );
.
.
Checkpoint ( 3 );
.
.
```

Conclusion

Identification that a fault occurred depends on a multitude of techniques; no one method works for all faults. Regardless of the fault identification method, recovery may be done by one or more of the following:

- Stopping the watchdog feeding and waiting until the watchdog timer forces a reset.
- Doing a software reset by jumping back to address 0000.
- Doing a software reset to an error handler routine, but this depends on application software.

It should be noted that some devices and code cannot be fault tolerant. An example is a real-time clock in which the time and date are stored. If corrupted, there may be no way of recovering these values without outside intervention.

This document originally described software techniques that were successfully applied in software that ran on an embedded system containing an 8031 microprocessor with separate program ROM, data RAM, and associated circuit board traces (antennas!) Refer also to [Solving the Software Safety Paradox](#), Embedded Systems Programming, December 1998, Doug Brown.

(And, from many years of software debug in embedded systems, I have a sneaking suspicion that software can actually overheat and data can drift, thereby causing it to fail, but I've yet to confirm that! Also, there may be credence to the saying, "All electronic devices run on smoke. If you let the smoke out, the device ceases to work.")