Reliable Software for Real-Time Mission Critical Systems

Chris Turiano
Mesa State College
Grand Junction, CO 81501

Abstract
A discussion of the concepts involved in creating and implementing reliable software for use in real-time critical systems. Among others, topics will include program design, fault-tolerance, error handling, and error recovery. To help demonstrate the ideas, a program, written in Ada95, mimicking a simple real-time system will accompany the discussion.

I. Introduction
The software requirements of today’s real-time embedded systems are vastly different from normal personal computer programs such as Microsoft Office or AutoCAD. In many situations, failure of the software in an embedded system results in loss of life, such as an airplane’s navigation system or a submarine’s oxygen sensing equipment.

With such dire consequences, failure of these systems is not an option and in the odd situation that they do fail, there must be back up measures in place to prevent catastrophe. To increase the reliability of such systems requires a multi-tiered methodology of software engineering including: fault prevention, fault detection and fault reaction.

The previous paragraphs have introduced several terms that require context-specific definition:
1. Reliability: a quantitative measure of how well a system conforms to its specified behavior
2. Failure: when a system does not conform to its specified behavior
3. Fault: the internal or algorithmic cause of a failure
4. Real-time system: a system that produces a time-constrained output into the physical world based on input from the same physical world
   a. Hard real-time system: system failure occurs for any missed deadline
   b. Soft real-time system: system is tolerant of missed deadlines

II. Threads
Most real-time systems have the potential for parallel processing; meaning, the software gives the illusion of more than one thing occurring at the same time. If implemented on a multi-processor machine, multiple computations would actually be occurring at the same time. Parallel processing is necessary in real-time systems because the nature of the real world is such that needed datum often occur at the same time. For instance, a weather station must collect wind velocity, temperature and humidity data, all of which are constantly changing and occur at the same time.

Fundamental to real-time systems is the concept of a thread. At its simplest, a thread is a section of code, usually incorporating an infinite loop, which is able to run at the same time as other threads. To achieve parallel processing, real-time systems may use several threads as potentially autonomous, sequential programs. These threads are allocated and de-allocated CPU time by a Real-Time Operating System (RTOS). The RTOS schedules which threads may run on which CPU at what time and how long each thread may run.

It is important to consider the types of interaction between threads when discussing real-time reliability. There are three types of interaction: independent, cooperating and competing.

Independent threads do not interact in any way with each other. Each thread has its own memory space and shares no variables with other threads.

Cooperating threads do communicate and synchronize with each other to achieve some common goal or operation. Often this communication comes in the form of shared memory or common resource.
If more than one thread may be vying for a common resource at a given time, the threads are competing. Competing threads must communicate and synchronize so that each thread may use the resource without interfering with the other.

It is when threads communicate or compete that real-time systems begin to fail. Often, this is due to the RTOS, which may interrupt a thread while it is accessing a shared resource and run a different thread.

As an example, let us take two instances of the same thread, A and B. Thread A begins execution and has gathered some data that it needs to write to the next available memory location. First, it discovers that the next location is x and records x in its variable called 'nextLoc'. However, just before it can write to that location, the RTOS interrupts it and begins thread B. Thread B has also gathered some data that it needs to write to the next available memory location. It discovers that the next empty location is x and writes its data to it. The RTOS begins thread A where it previously stopped and A writes its data to location x and the data that thread B had written is lost. In Ada95, the code for this thread would be:

```ada
Task MyThread is
    nextLoc : integer := 0;
    someData : integer := 0;
Begin
    While true Loop
        GetNextLoc(nextLoc);
        GetSomeData(someData);
        WriteSomeData(someData, nextLoc);
    End Loop;
End MyThread;
```

To overcome situations such as the example above, languages have incorporated concepts such as protected objects, monitors, semaphores, mutual exclusion, atomicity and critical sections. Critical sections are sequences of statements that have to execute without interruption by the RTOS. It is common to place access to shared resources in critical sections, thereby protecting the resource from multiple threads accessing it at the same time. This indivisibility of execution is atomicity and is crucial to the concept of real-time reliability. A key point to note about atomicity and critical sections of code are to keep execution time as short as possible so that the RTOS can more efficiently schedule CPU time.

To achieve atomicity, mutual exclusion protects the critical section. The concept of mutual exclusion states that only one thread may be in its critical section at a given time. The implementation of mutual exclusion is different for different languages. At its simplest, mutual exclusion can be a Boolean variable that prevents multiple threads from entering their critical sections at the same time. Prior to entering its critical section, a thread tests the variable: if false, the thread sets the variable to true and enters its critical section; if true, the thread enters either a busy-wait loop or sleeps until the variable returns to false. It is possible to extend this sort of atomicity to producer-consumer buffer situations as well as shared resources. Modifying the 'MyThread' code to incorporate mutual exclusion and a critical section, we have:

```ada
Task MyThread is
    nextLoc : integer := 0;
    someData : integer := 0;
    nextLocLock : Boolean := false;
Begin
    While true Loop
        If nextLocLock Then
            GetNextLoc(nextLoc);
            GetSomeData(someData);
            WriteSomeData(someData, nextLoc);
        Else
            Delay(5.0);
        End If;
    End Loop;
End MyThread;
```

Unfortunately, because testing the variable is not atomic with the critical section it is trying to protect, this simple solution is ineffective. To achieve mutual exclusion using only flag variables and busy
wait loops requires three variables. Gary L. Peterson first proposed his Mutual Exclusion Algorithm in 1981 and it involves two threads (A and B), one shared variable (called ‘turn’) and two independent variables (called ‘flagA’ and ‘flagB’, control of which is only by the owner thread). Each thread first sets its own flag to true, and sets ‘turn’ to the value of the other thread. If the other thread’s flag is true and ‘turn’ is set to the value of other thread, this thread must wait. If the other thread’s flag is false or ‘turn’ is the value of this thread, then this thread may enter its critical section. After leaving its critical section, each thread sets its flag to false. The Peterson algorithm written in Ada95 looks like:

```
turn : Character;
Task MyThreadA is
  flagA : Boolean := false;
  nextLoc : integer := 0;
  someData : integer := 0;
Begin
  While true Loop
    flagA := true;
    turn := B;
    While GetFlagB() = true And turn = B Loop
      null;
    End Loop;
    flagA := false;
    GetNextLoc(nextLoc);
    GetSomeData(someData);
    WriteSomeData(someData, nextLoc);
  End Loop;
End MyThread;
```

While the Peterson algorithm guarantees that only one thread may enter its critical section at a time, the busy wait loop is inefficient and wasteful of CPU time. Fortunately, it is possible to suspend a thread, and then awaken the thread when the conditions are correct for execution. The suspend/resume cycle uses CPU time much more effectively.

The suspension of a thread involves the RTOS stopping the execution of the thread and placing its thread identification (PID) in a suspended thread queue. The RTOS then chooses another thread to execute which will eventually execute a resume and the RTOS will suspend that thread and choose a thread off the suspended thread queue. While the queue could be a FIFO queue, most real-time systems would implement a priority queue with the most important thread always on top of the queue.

Although it is possible to implement effective synchronization of threads using the Peterson algorithm and suspension/resumption, most languages have built-in abstractions that make synchronization and communication simpler. One such abstraction is the semaphore.

A semaphore is a natural number variable with two procedures that act upon it: ‘wait’ and ‘signal’. Both procedures are atomic. A thread wanting to enter its critical section calls ‘wait’, while ‘signal’ is used when exiting a critical section. Procedure ‘wait’ tests the semaphore for equivalence with zero: if zero the calling thread must wait; if greater than zero, the procedure enters its critical section and decrements the semaphore. Procedure ‘signal’ increments semaphore allowing another thread to enter its critical section. The initial value of semaphore limits the number of threads allowed to enter their critical sections. A mutex (mutual exclusion semaphore) is a semaphore with an initial value of one and allows only one thread into its critical section.

Care must be taken to avoid the situation where thread A is waiting on a resource that thread B owns and thread B is waiting on a resource that thread A owns. This situation results in a ‘deadlock’ where both threads wait forever and the system eventually fails. ‘Starvation’ is another situation to avoid. It occurs when thread A has a resource it will not give up and thread B wants that resource but can never gain access to it.

Abstracting synchronization further, it is possible to incorporate mutual exclusion and semaphores into the objects of object-oriented programming. Monitors are objects that guarantee all procedure calls execute in mutual exclusion. Hiding shared variables inside a monitor allows access only through the monitor, which eliminates the complexity of multiple semaphores scattered throughout the program. In a monitor, the implementation of mutual exclusion is by a condition variable, which, while effective, does have some drawbacks. To omit a condition variable, for instance, the monitor will not function as intended and deadlock will eventually occur.
A step up from a monitor is the ‘protected object’ of Ada95. While similar to monitors in guaranteeing all procedure calls execute in mutual exclusion, protected objects implement this mutual exclusion in a different way. Instead of using condition variables, protected objects use a higher level of synchronization called ‘barriers’. ‘Barriers’ are Boolean expressions that must evaluate to true before access is allowed.

III. Faults

There are three types of faults from four general sources. The types of faults are transient, permanent and intermittent, which occur in the following four sources: system specification, software design, hardware issues and communication issues.

Transient faults occur, persist for a time, then later disappear and are usually due to some sort of outside interference such as radiation or electro-magnetic interference. After this sort of fault disappears, the manifested failure may or may not also disappear.

Permanent faults occur and persist until reparation of the cause. Mechanical and software malfunctions often cause this sort of fault.

Intermittent faults are recurring transient faults. For example, if thread A overfills a buffer triggering a fault and then thread B drains the buffer, returning the system to a correct state.

In a real-time system, the delivery of output must be within the expected timeframe or a failure has occurred. Therefore, three possible failures exist in respect to timely delivery: too early, too late or never. Incorrect values, as well as unexpected services, are other possible types of failures.

Combining these types of failures, allows us to speculate as to how a system might fail. For instance, a system may deliver the correct value but always late; or it may produce random errors in either value or time or both. The goal is a system that always produces correct values in a timely fashion.

Two different approaches to handling the faults that cause these failures are fault prevention and fault tolerance. The first is obviously preventing the fault before it occurs and the latter is enabling the system to function (perhaps with less functionality) in the presence of faults. Using both approaches results in the most reliable system possible.

A. Fault Prevention

Fault prevention begins during design specification. By using appropriate specifications for a given system, an engineer can eliminate most hardware and software malfunctions including component communication faults and faults due to outside interference. Software fault prevention stems from using fault avoidance techniques such as proven design methodologies, strongly typed languages with data abstraction and modularity and tools such as integrated development environments, design reviews, program-verification software and well-defined time constraints for communication. Fault avoidance helps prevent the introduction of faults into a system.

After choosing an appropriate design specification, the next step of fault prevention is the removal of faults introduced by design errors in hardware and software. The most significant part of this step is system testing. Extensive system testing is the best method to improve the reliability of a program. However, since exhaustive system testing is impossible, many faults could remain. The major shortcomings of testing are the difficulty proving correctness, adequate realism and late stage manifestations of incorrect early assumptions.

B. Fault Tolerance

The second approach to handling faults is fault tolerance and has three levels: full fault tolerance, graceful degradation and fail safe. The level of fault tolerance depends on the nature of the system, but most mission-critical systems have full fault tolerance as a goal.

A full fault tolerant system retains all functionality in the presence of faults, although perhaps for a limited time. Graceful degradation is a system that operates with less functionality until repair of the fault. A fail-safe system records a prior safe state so it can return to that state after a temporary shutdown.

While a system may target full fault tolerance, as more components fail, the design of the system should allow regression to graceful degradation and finally to fail-safe if no other options are available. In this way, data recovery is possible and a system restart may restore functionality if the hardware is operational.

To achieve fault tolerance, an engineer adds additional components (software and hardware) to detect, handle and recover from faults. Ironically, the more redundant a system, the more complex the
system, the more likely it is to contain faults. Therefore, an engineer must strike a balance between redundancy, the risk of introducing new faults and the constraints of system memory and weight. To minimize the complexity created by redundancy, it is advisable to separate the fault-tolerant components from the essential functionality of the system. Using a language that supports modularity and data-abstraction make it easier to separate the redundant components.

There are two types of redundancy: dynamic and static. Dynamic redundancy is comparable to an error detection system. An example is the "exception" block of the Ada95 language. Static redundancy is a method of masking errors by comparing the output of multiple components and using either an average or a majority-rule algorithm to select the correct value. Static redundancy is known as n-version programming and n-modular redundancy in the software and hardware industries, respectively.

Dynamic redundancy has four stages: error detection, error diagnosis, error recovery, and fault treatment. During program execution, a fault will manifest itself as an error in most cases. If this error goes undetected, the software can do nothing to prevent system failure. As such, error detection is the most important part of dynamic redundancy. Error diagnosis assesses the extent of damage and attempts to limit the spread of that damage. Error recovery attempts to treat the damaged data while fault treatment fixes the cause of the error and restores the system to full service.

1. Error Detection

Most languages include some basic mechanism for single-threaded error detection. Java includes the 'try…catch' block, for instance, while Ada uses the 'exception' block, which functions more or less as a switch depending on the specific exception that occurs:

```java
package body SomeThing is
    SomeException : exception;
begin
    procedure SomeProc (param : OUT integer) is
        ProcException : exception;
    begin
        <the internal workings of SomeProc>
    exception
        when ProcException => ProcExceptionHandler(param : IN integer);
    end SomeProc;

    exception
        when SomeException => SomeExceptionHandler(param : IN integer);
        when Others => OtherExceptionHandler();
end SomeThing;
```

This model of exception handling provides excellent protection and fault tolerance at different levels of abstraction. At the most specific level, our object (SomeThing) has a method (SomeProc) which can raise and handle its specific exception (ProcException) while any other error that occurs propagates to the statically scoped parent, SomeThing. SomeThing can handle its own exception (SomeException) as well as a general catch-all (Others) for any other exception that may occur. Any exception SomeThing does not catch will propagate to the object’s parent. This hierarchical propagation continues until a handler handles the exception, or, if unhandled, causes the program to terminate. In this way, error detection occurs everywhere while error diagnosis and error recovery occur where the specific error is best handled. As mentioned earlier, error detection is the most important part of fault tolerance and this model provides for very extensive detection capabilities.

While the ‘try…catch’ or ‘exception’ block model works well in most situations it does have a few shortcomings. If, for instance, an error occurs that corrupts the data in a thread’s stack, the exception handlers have no pointer to the error handling code and no pointer to the parent thread, thus go unhandled. If this error manages to propagate to the parent thread, the same problem could occur and would eventually cause a failure of the program. The other shortcoming is in the case of a logic error resulting in an infinite loop. The problem here is the exception event never fires and the infinite loop continues, unhindered until program failure.

Because the exception handling mechanism fails in these areas, the designer must build in other mechanisms to handle them. In real-time systems, with multiple threads communicating and competing, the timeliness of interaction becomes important. This raises the detection issue of how to handle deadline errors that occur if a thread enters an infinite loop or if a thread’s stack is corrupted. There are two approaches to checking timing: watchdog timer thread and RTOS scheduling. RTOS scheduling is an inherent part of the underlying system and its effectiveness is part of the design specification.
A watchdog timer thread is usually a priority queue with priority based on the next communication to expire. Prior to executing a communication, a thread logs its thread id and an error handler in the priority queue of the watchdog thread. If the thread does not remove this queued entry before the expected time of completion, the watchdog thread executes the error handler. As an example, imagine a parent and teenager. The teenager wants to take the family car to the movies, so she tells her parent that she will arrive at the movie theater in 20 minutes at which time she will call the parent. If the teenager is late, the parent calls the police, knowing an accident has happened. If the teenager calls within the 20 minutes, the parent knows everything is okay and disregards the deadline.

2. Error Diagnosis

Error diagnosis involves preventing spread of erroneous data and examining all aspects of a system to which the error could have spread. To speed this step, it is helpful to know what error occurred in which component. To confine erroneous data spread, good design practices are necessary. Using an object-oriented design throughout the system naturally limits the interactivity between components. All access from outside the object is well defined and the object hides its data preventing ambiguous access. Another useful design practice for confinement of erroneous data is atomicity. Keeping all access to shared resources indivisible, the introduction of outside data to the thread is unlikely.

3. Error Recovery

There are two types of error recovery: forward and backward. Backward error recovery provides a means of undoing any state changes made during a critical section after an error has occurred. This allows threads to resume from a certain known point if an error occurs during an atomic action. With inter-thread communication and competition, if an error occurs it propagates in all threads participating in the communication and all threads roll back to the recovery point. Since all threads roll back regardless of the error, there is no need for the system to know which error occurred. A drawback to this sort of error recovery is the potential creation of an infinite loop: threads enter atomic action and each creates a recovery point, then an error occurs rolling all thread back to the same state they just left.

Forward error recovery is similar to backward error recovery in that both roll all participating threads back to a known recovery point by propagating the error. However, while backward error recovery deals mostly with inter-thread communications, forward/backward error recovery handles errors raised in conjunction with the environment. As such, there is a need for the environment to know which error occurred.

IV. Conclusion

Due to the nature of embedded systems involving the lives of humans, the reliability of the software and hardware used must be 100%. By using sound design specifications, fault prevention techniques such as extensive testing, strongly typed languages with built-in abstractions for mutual exclusion, engineers can avoid introduction of faults into a real-time system. It is possible to create software that is able to function in the face of faults by using fault tolerance techniques such as exception handling, watchdog timers, and forward/backward error recovery.

Combining the techniques of design specification, fault prevention and fault tolerance, the software written for mission critical real-time systems, such as is used in today’s missiles and spacecraft, can be truly reliable.

V. References