Sean Beatty

Sean Beatty is a Principal with High Impact Services in Indianapolis. He holds a BSEE from the University of Wisconsin - Milwaukee. Sean has worked in the embedded systems field since 1986, developing and testing software for the consumer, industrial, medical, and automotive markets. Recently, he noticed a serious lack of information available to engineers in the area of Testing Embedded Systems Software. So since February, 2000, he has been presenting on this topic at various technical conferences. Sean has also published articles in "Embedded Systems Programming" and "Test and Measurement World". He often teaches what he's learned via seminars, consulting, and hands-on training. Sean enjoys discussing embedded systems issues and can be reached at smbeatty@highimpactservices.com.

Finding Firmware Defects
Class T-18
Sean M. Beatty

Embedded systems have many characteristics that differ from other software-based applications. They tend to perform very specific computations, in contrast to general purpose computing. Another example: whereas desktop applications may use floating-point math exclusively, many embedded systems used fixed-point (scaled integer) math. This provides an order of magnitude increase in the speed of a given computation, and requires much less of the fixed memory resources of the embedded microprocessor. It can also make the programming of the computations much more complicated, which leads to a greater probability of errors.

Embedded systems have limitations when it comes to handling errors. Rarely is it acceptable (or possible) to display a dialog box stating "An error has occurred…" Since embedded systems tend to be relied on continuously, the best recourse to encountering a critical error is often simply resetting the microprocessor – which has many significant consequences. Thus, errors become more critical in embedded systems, since the ability to respond to them and take corrective action is limited.

This article begins an exploration of how the nature of embedded systems software, also called "firmware", affects the software testing process. *The testing methods commonly applied to other types of software are inadequate to thoroughly test critical firmware-based systems.*

Embedded systems are characterized by their close association to hardware – the electronics that responds to and affects the physical environment. Sensors are used to get data about the environment in which the system is operating. The data coming from a sensor may be contaminated by noise in the environment. Noisy data in turn can cause problems in the algorithms it feeds, so sensor data must often be conditioned to get rid of noise.

PC application programs are launched only after the computer has "booted up". In contrast, embedded systems code must handle everything from the time power is applied to the system to the moment it's shut off. In addition, many battery powered systems have low power modes which they enter in order to extend battery life. The transitions in and out of these various power modes are managed by the firmware. The added complexity is another source for potential errors, especially for those mode changes that are infrequently exercised.

In order to recover from catastrophic errors, most embedded systems use a watchdog timer. If this timer is not accessed within a designated time it will reset the microprocessor. This implies that the watchdog timer must be accessed before it times out for all correctly operating modes of the system. If a rare sequence of events occurs that takes longer than expected to complete, the watchdog may timeout and force a system reset. In this unfortunate event, the very device intended to enhance system reliability becomes a source of critical errors.

Most embedded systems do not have virtual memory. If any critical memory resource is exhausted, the system may crash, reset, or worst yet, fail in an unpredictable manner. In particular, enough stack space must be allocated for the worst possible situation. The path through the code that consumes the most stack space is generally not obvious. In addition, if third-party libraries are used, their stack needs must also be considered. Failure to correctly determine the worst case stack depth and allocate enough memory for it results in a system which fails under certain, often difficult to duplicate, conditions.

There are many types of software defects that are common to all classes of computer software:

- Errors, omissions, and ambiguities in requirements and designs
- Errors in the logic, mathematical processing, and algorithms
- Problems with the software's control flow: branches, loops, etc.
- Inaccurate data, operating on the wrong data, and data-sensitive errors

- Initialization and mode change problems
- Issues with interfaces to other parts of the program: subroutines, global data, others …

In addition to all these, real-time embedded systems have potential problems in many additional areas:

- Exceeding the capability of the microprocessor to perform needed operations in time
- Spurious resets caused by the watchdog timer
- Power moding anomolies
- Incorrect interfacing to the hardware peripherals in the system
- Exceeding the capacity of limited resources like the stack, heap, others
- Missing event response deadlines

These issues combine to make embedded systems more challenging to test than many other types of software.

The software quality assurance activities often described as "testing" fall into one or more of the following three categories:

- Reviews, inspections, walkthroughs: *checking*
- Functional, structural, integration, and regression testing: *demonstrating*
- Timing and other analyses: *proving*

All the activities designed to locate potential software defects either check it against a list of desired characteristics, demonstrate it performs correctly under defined conditions, or prove it will not exhibit certain types of failure under all possible conditions. Over the years, certain types of testing have become popular. Almost all projects incorporate a number of different methods of testing, since they each have different strengths and weaknesses.

Code reviews have become popular as a way of identifying potential software defects early in the development process. They are adept at discovering logical and mathematical problems, as well as syntax, typing, and "cut-and-paste" type errors. A rigorous inspection is sometimes employed to verify certain non-testable requirements or paths through the code. Due to the detailed nature of a review or inspection, they are typically performed on individual units of the code, such as a single function or small group of related functions. This makes them ineffective at identifying problems involving the larger scope of the program, such as event timing, watchdog usage, and other system issues.

Functional testing is performed to insure that the system performs its expected behavior. It is effective at identifying missing or ambiguous requirements. Problems among the various interfaces of the systems components are also often illuminated. Functional testing tends to focus on the most critical features of the system and those used most frequently. It is practically impossible to exercise every path through the code using this technique. In particular, those paths that respond to error conditions or confluences of events are difficult. This results in many potential problems going undetected.

In contrast, structural (also referred to as "white box") testing is excellent at forcing execution of every possible path through the code. It tends to find more software defects than functional testing does. Due to the close scrutiny of the software required, it also finds many of the same types of errors which inspection uncovers: math, logic, structure, and data problems. Also like inspection, this focus on a function or unit of the software in isolation of the rest of the system misses errors with timing, interfaces, and system-level requirements.

Inspection is too myopic to uncover many of the potential defects in firmware. Testing is unlikely to trigger the conditions necessary to expose them. Only an exhaustive ***analysis*** *of the software will uncover certain types of potential critical errors.*

Analysis finds problems that testing and inspection miss, due to its focus on the details of the implementation as they relate to the entire scope of the system. This analysis can be very tedious, so it is generally focused only on those areas that cannot be verified any other way. Potential problems with stack depth, watchdog timer usage, shared resources, and timing are illuminated in the light which a detailed analysis sheds on the code.

Insuring that adequate stack has been allocated for the program is a good example of the need for analysis. Maximum stack depth is a property of the entire system, not of any particular function or part of the code. Yet it is not a requirement that can be easily tested. Function calls and their parameters, temporary variables, interrupts, and tasks all need stack space to operate. Until the code has been completed, the maximum stack depth cannot be determined. Nor is it generally obvious which particular path (or concurrent paths) will produce the greatest need for stack at a particular instance of the system's operation. For these reasons, the code must be thoroughly analyzed, as follows:

- Build a call tree for each task
- Determine the stack used by each function
- Determine the worst case stack depth for each task's call tree
- Sum the stacks of each task
- Add the stack needed for each level of interrupt
- Add any stack used by initialization or an RTOS

Once the maximum stack depth has been determined, the appropriate amount of memory can be allocated for its use.

Other potential defects which analysis uncovers include data sharing violations. Embedded systems often use global data to some extent. Almost all embedded systems use interrupts. Since interrupts cannot return any values, the interrupt service routine typically writes its results into a global data store. The sharing of data between interrupt service routines and the rest of memory must be carefully managed. Like the stack depth analysis, a shared data analysis examines the minute details of the implementation as they interact with the entire system. In addition to data shared between interrupts and other parts of the system, data shared between tasks of different priorities in a multi-tasking environment must also be understood and carefully managed. Failure to do so results in data being overwritten or corrupted.

In the example in Listing 1, an external port F is read. The value is scaled and an appropriate offset is added. Then this final result is written to the global variable `result`, which is later used by an interrupt service routine. Since the interrupt that uses this data could occur at any time, it's important to put all the preliminary calculations into a temporary variable, which has no scope outside this function. If the write to `result` on line 9 is done in a single microprocessor instruction, it cannot be interrupted, and the interrupt service routine will always read a correct value from `result`. However, if this code is running on an 8-bit microprocessor which only writes 8 bits at a time to memory, the write would take two microprocessor instructions. An interrupt could potentially occur in between these two instructions, and the interrupt service routine would then be using data in which one byte was old and the other byte was new. This could cause serious problems. Disabling interrupts around this write prevents the error.

Listing 1 Safely modifying global data

```
Extern int result;
void Update_result(int offset, int scale)
{
       int temp;
       temp = read_port_F()
       temp *= scale;
       temp += offset;
       disable_interrupts();
       result = temp;       /* line 9 */
       enable_interrupts();
}
```

A shared data analysis carefully examines all data used between interrupt service routines and other areas of memory, and between tasks of different priorities, to insure these types of problems are prevented.

Additions to the common software quality assurance test techniques used to find software defects are necessary to adequately test embedded systems. Here are some recommendations:

Perform a system-level inspection of the software, checking specific items:

- Insure all peripheral registers are understood and accessed correctly.
- Insure that the watchdog timer is enabled. (Many times, engineers keep it disabled while the code is being developed to ease debugging.)
- Insure all digital inputs are de-bounced appropriately.
- Insure that the turn-on delay for analog to digital converters (and digital to analog converters) is handled appropriately.
- Carefully examine power-up and power-down behavior, and the entrance and exit from any low power modes.
- Insure an appropriate interrupt vector is defined for every interrupt, not just those that are expected.

During structural (white-box) unit testing:

- Determine the worst-case stack usage for each function. This will be used later.
- Determine the longest execution time for any path through the function (from call to return). This will also be used later.
- Look for any non-deterministic timing structures, such as recursive routines, waiting for hardware signals or messages, etc. These can make the subsequent timing analyses inaccurate.

Perform the following Critical Analyses on the software:

- *Stack Depth* - Determine the worst-case stack depth and insure adequate stack has been allocated.
- *Watchdog Usage* - Using the timing data obtained in the unit tests, find all places where the watchdog timer is reset and insure that all correctly operating modes of the system will always reset the watchdog timer before it expires.
- *Shared Data* - Locate all data that is accessed by interrupt service routines and other areas of the code. If a multi-tasking design is used, also locate all data that is shared by tasks operating at different priority levels. Verify that adequate protection measures are used on the data to insure it never becomes corrupted.
- *Deadlock* - Build an allocation graph of all the data shared among different tasks that could end up causing a deadlock. Insure that deadlock is prevented by the design of the system and the order in which shared data items are locked.
- *Utilization* - Using the timing information obtained in the unit tests, determine the worst-case execution times of all tasks in the system. Verify that the microprocessor can complete all tasks by their deadlines, at their maximum rate of occurrence, under worst-case situations.
- *Schedulability* - Using the worst-case task execution times obtained earlier, determine if all the tasks can be scheduled and meet their deadlines under all situations. Use a scheduling algorithm that follows the design of the system (e.g. RMA).
- *Timing* - Insure all other timing requirements of every task are met under worst-case situations. For example, release-time jitter, end-to-end completion, etc.

By performing these additional checks and analyses the embedded systems software can be released with much greater likelihood that it will be free of critical defects.