Twenty-Five Most Common Mistakes with Real-Time Software Development

David B. Stewart

Executive VP and Chief Technology Officer
Embedded Research Solutions, LLC.
Columbia, MD 21046

Email: dstewart@embedded-zone.com
Web: http://www.embedded-zone.com

Abstract: *The most common mistakes and pitfalls associated with developing embedded real-time software are presented. The origin, causes, and hidden dangers of these mistakes are highlighted. Methods ranging from better education to using new technology and recent research results are discussed. The mistakes vary from problems with the high-level project management methodologies, to poor decisions on low-level technical issues relating to the design and implementation. The most common mistakes have been identified from experience in reviewing the software designs and implementations of many embedded programmers, ranging from seasoned experts in industry to rookies learning the material in college.*

Introduction

Novices and experts alike, whether in a university or corporation, repeat the same mistakes over and over again when developing real-time software. I have observed this while reviewing the software designs and implementations of many embedded programmers, ranging from seasoned experts in industry to rookies just learning the material in college.

Most real-time software developers are not even aware that some of their favorite methods are problematic. Other times the methods are satisfactory, but not the best. Quite often, experts are self-taught; hence they tend to have the same bad habits as when they first began, usually because they have never witnessed better ways of programming their embedded systems. These experts then train novices, who subsequently acquire the same bad habits. The purpose of this article is to improve awareness of common problems, and to provide a start towards avoiding and eliminating mistakes to create software that is both more reliable and easier to maintain.

This list first began as the 10 most common pitfalls, but there were just so many common mistakes and problems that the list grew. For each problem, I present the misconception or source of the problem. Then I offer possible solutions or alternatives that can help minimize or eliminate the mistakes. If you are not familiar with the details or terminology of the alternate solutions, then a quick library or Web search should yield additional literature on the topic. While there is usually agreement about most items being mistakes, some of the mistakes listed and the corresponding proposed solutions may be controversial. In such cases, simply highlighting that there is a disagreement as to what is the best way to alleviate these problems encourages designers to compare their methods to other approaches, and to reconsider if their methods are provably better.

Correcting just ***one*** of these mistakes within a project can lead to weeks or months of savings in manpower (especially during the maintenance phase of a software life cycle) or can result in a significant increase in quality and robustness of the application. If multiple mistakes are common and they are all fixed, potential company savings or additional profits can be in the thousands or millions of dollars. Thus I encourage you to review your current methods and policies, compare them to each of the reported mistakes and the proposed alternatives, and decide for yourself if potential savings exist for your company or project. Even if there are no direct savings, consider the potential for improved quality and robustness at no extra cost by modifying some of your current practices.

Here now are the most common mistakes; problems that are higher on the list (where #25 is lowest and #1 is highest on list) are either more common and/or have the most impact on quality, development time, and software maintenance. Naturally, the order represents my opinion. It's not so important that one mistake is listed higher on the list than another. What is important is that both are listed, thus both may be significant in your specific environment.

#25 "My problem is different"

Many designers and programmers refuse to listen to the experiences of others, claiming that their applications are different, and of course, much more complicated. Designers should be more open-minded about the similarities in their work. Even what seems like the most different applications are probably nearly identical when you consider the nuts and bolts of the real-time infrastructure. For example, communications engineers will claim their applications have no similarities to systems designed by control engineers because of the high volume of data and the need for special processors such as digital signal processors (DSPs). In response, ask "What is different in the LCD display software in a cellular phone vs. one in a traffic light controller? Are they really different?"

Comparing control and communication systems side-by-side, both are characterized by modules that have inputs and outputs, with a function that maps the input to the output. A 256 x 256 image processed by a DSP algorithm might not be that different from graphical code for an LCD dot matrix display of size 320 x 200. Furthermore, both use hardware with limited memory and processing power relative to the size of the application; both require development of software on a platform distinct from the target, and many of the issues in developing software for a DSP also apply to developing software for a microcontroller.

The timing and volume of data are different. But if the system is designed correctly, these are just variables in equations. Methods to analyze resources such as memory and processing time are the same—both may require similar real-time scheduling, and both may also have high-speed interrupt handlers that can cause priority inversion.

If control systems and communication systems are similar, perhaps so are two different control applications or two different communication systems. Every application is unique, but more often than not the procedure to specify, design, and build the software is the same. Embedded software designers should learn as much as possible from the experiences of others and not shrug off experience just because it was acquired in a different application area.

#24 *Delays implemented as empty loops*

Real-time software often uses delays to ensure that data sent or received over an I/O port has time to propagate. These delays are frequently implemented by putting a few no-ops or empty loops (assuming **volatile** is used if the compiler performs optimizations). If this code is used on a different processor, or even the same processor running at a different rate (for example, a 25MHz vs. 33MHz CPU), the code may stop working on the faster processor. This is especially something to avoid, since it results in the kind of timing problem that is extremely difficult to track down and solve, because the symptoms of the problem might be sporadic.

Instead, use a mechanism based on a timer. Some RTOS provide these functions, but if not, one can still easily be built. Following are two possibilities to build a custom **delay(int usec)** function.

Most count-down timers allow the software to read a register to obtain the current count-down value. A system variable can be saved to store the rate of the timer, in units such as microseconds per tick. Suppose the value is 2µs per tick, and a delay of 10µs is required: the delay function busy-waits for five timer ticks. Suppose a different speed processor is used—the timer ticks are still the same. If the timer frequency changes, then the system variable would change, and the number of ticks to busy-wait would also change, but the delay time would remain the same.

If the timer doesn't support reading intermediate count-down values, an alternative is to profile the speed of the processor during initialization. Execute an empty loop continuously and count how often it occurs between two timer interrupts. Since frequency of the timer interrupt is known, a value for the number of microseconds per iteration can be computed. This value is then used to dynamically determine how many iterations of the loop to perform for a specified delay time. In our custom RTOS with this implementation, the delay function was accurate within 10% of the desired time for any processor with which we tested it, without ever having to change the code.

#23 *Tools choice driven by marketing hype, not by evaluation of technical needs*

Software tools for embedded systems are often purchased based on the flashiness of the marketing, because a lot of other people are using them, or because of a feature that sounds appealing but really does not make a difference.

Flashiness. Just because one tool has a prettier graphical user interface than another does not make it better. It's important to consider the technical capabilities of each, relative to the needs of the application being built.

Number of users. Buying software from a vendor just because it's the biggest does not mean it's the best. Along with pitches that more people are using the software are probably hidden true stories that more people are paying for more than they really need, or that more people have unused versions of the tools sitting on the shelf after discovering the tools were not suited to their needs.

Promises of compatibility. Managers are especially influenced by a product because of promises of compatibility. So what if software is 100% POSIX-compliant? What is its relevance? Is there a plan to change the operating system? Suppose there is a change to another POSIX-compliant operating system-what is there to gain? Absolutely nothing, unless "extensions" are used. But if such extensions are used, compatibility is lost, hence the benefits are no longer there. Standards such as POSIX have not been proven to even be good for real-time systems, let alone the best. Therefore, don't assume that the product is better because of that promise. Portability and reusability can only be achieved if all the designers follow proven software engineering strategies for developing component-based software. [2,3]

When selecting tools, consider the needs of the application first; then investigate the dozens (or hundreds) of options available from a *technical* perspective, as they relate specifically to the application requirements. The best tools for a particular design or application are not necessarily the most popular.

#22 *Large if-then-else and case statements*

It's not uncommon to see large *if-else* statements or *case* statements in embedded code. These are problematic from three perspectives:

- Such statements are extremely difficult to debug, because code ends up having so many different paths. If statements are nested it becomes even more complicated.
- The difference between best-case and worst-case execution time becomes significant. This leads to either under-utilizing the CPU, or the possibility of timing errors when the longest path is taken.
- The difficulty of structured code coverage testing grows exponentially with the number of branches, so branches should be minimized.

Computational methods can often provide an equivalent answer. Performing Boolean algebra, implementing a finite state machine as a jump table, or using lookup tables are alternatives that can reduce a 100-line if-else statement to less than 10 lines of code.

Here is a trivial example of converting an if statement to Boolean algebra:

```
if (x == 1)
    x=0;
else
    x=1
```

Instead, a Boolean algebra computation would be the following:

```
x = !x;       // x = NOT x; can also use x = 1-x
```

Despite the simplicity, some programmers still toggle a Boolean value with the if statement above.

#21 *Documentation was written after implementation*

Everyone knows that the system documentation for most applications is dismal. Many organizations make an effort to make sure that everything is documented, but documentation isn't always done at the right time. The problem is that documentation is often done after the code is written.

Documentation must be done before and during coding, never afterward. Before implementation begins, start with the detailed specification and design documents. These become the basis for what will ultimately be the user and system documents, respectively. Implement the code exactly as in these documents; anytime the document is ambiguous, revise the document first. Not only does this ensure that the document remains up to date, but it ensures that the programmer implements what the document specifies.

Updating documentation during the implementation also serves as a review for the code. Programmers often find bugs in their code as they're writing about it. For example, the programmer may write, "Upon success, this function returns 1." The programmer then thinks, "if there is no success, what is returned?" He looks at his code and might realize that the lack of success scenario has not properly been implemented.

#20 *Interactive and incomplete test programs*

Many embedded designers create a series of test programs, each program testing a separate feature. Test programs need to be executed one at a time, and in some cases require the user to provide input (say, through a keypad or switch) and observe the output response. The problem with this method is that programmers tend only to test what they are changing. Since there are often interactions between unrelated code due to the sharing of resources, every time a change is made, the entire system should undergo testing.

To accomplish this, avoid interactive test programs. Create a single test program that goes through as much self-testing as possible, so that any time even the smallest change is made, a complete test can easily and quickly be performed.

Unfortunately, this is more easily said than done. Some testing, especially of I/O devices, can only be done interactively. Nevertheless, the principle of automated testing should be at the forefront of any attempt to create test software, and not a side-thought with test code written only on an as-needed basis.

#19 Software engineers not participating in hardware design

For new products where the hardware has not yet been defined, the software engineer should work closely with the hardware designer to select a system architecture that can minimize the overall cost. For example, the hardware designer may feel that a 32-bit processor is needed for a particular application. The software designer, however, may realize that close to 40% of the processor utilization would be spent busy-waiting or polling for just one of the I/O devices. Suppose the processor costs $10. That means $4 in parts is dedicated to polling the I/O device.

Instead, it might be possible to use a two-processor design that includes one 16-bit processor (cost $4) and one 8-bit processor (cost $1). The 8-bit processor handles the I/O device, while the 16-bit processor handles the rest of the workload. Although the system has increased to using two processors, which might result in a small increase in design time, the hardware component cost is cut in half, and can lead to a more cost effective design. The hardware and software designers should together decide on the best way to interface the processors, so as to minimize both hardware and software costs.

Often, the needed performance of hardware for new products is estimated based on an existing product. The software designer should provide accurate answers as to the utilization of the processor in the existing application.

If the processor and memory utilization are less than 90% on average and less than 100% peak, then the system has probably been over-designed. Writing programs for a processor with more than enough resources is a luxury for a software developer. In some cases, however, this luxury is so costly that it can make the difference between a profit and bankruptcy! Contributing towards minimizing the price and power consumption of an embedded system is a software engineer's duty. If the CPU is only 45% utilized, you can use a processor that operates at half the speed instead, thus saving on power consumption (yielding higher market share if the device is battery-powered) and possibly reducing the cost of the processor.

If the product is mass-produced, saving $1 on the processor could save a million dollars over the production span of the item. If the product is battery-powered, it will allow the battery to last much longer, thus increasing the marketing appeal of the product. As an extreme example of power consumption of computers, consider a laptop. Most have less than three hours of power when using a heavy battery. A watch, however, has a lightweight, cheap battery that can last three years. Although software isn't usually associated with power consumption, it does have a major role.

Fast processors and more memory than necessary tend to also lead to laziness in thinking about the design. Start embedded development with slower processors with less memory, and move up to the next level of processor only on an as-needed basis. Software that uses hardware more efficiently is more likely to evolve from this approach than from later trying to cut corners to bring down the cost of the system.

#18 No emulators of target application

Building an emulator of an embedded application can significantly improve development time. Implementing software for target platforms can be quite time-consuming. Compiling, downloading, then executing code on a target platform takes longer than doing the same on the host platform. Debugging is more difficult, because the ability to use simple techniques like *printf()*'s throughout the code is often not possible. While some modern systems improve debugging capabilities through use of techniques such as BDM (background debug mode) and give the ability to put breakpoints in the embedded software, using these techniques is significantly more time-consuming than using tools directly on the host workstation, such as the GDB debugger to step through code and Purify to search for memory leaks.

While in the later stages of implementation it will be necessary to execute and debug all code on the target platform, this is often not necessary in the early stages. In particular, before timing and synchronization is an issue, each module should be tested individually for its functionality. The integration of the modules should also be tested; regardless of the real-time needs of the software, there should not be any code or data integrity problems.

It is usually very worthwhile to build at least a simple simulator of the application's devices, so that most of the code can be developed, tested, and debugged in the host environment. In our experience, it might take a week to build this simulator, but it can easily save months of development time. For example, we have a simulator of an LCD text display running under UNIX. The interface to this simulator is similar to the interface of the real hardware, so that the display code we write is nearly identical in both the target environment and the host environment. To compare development environments, imagine you want to make a very simple change, like modifying one line of code from *y++* to *y+=2*. The entire cycle to edit/compile/download/execute/test on the target environment might take five minutes or more. This same cycle on a host workstation might only take one minute. You can then be five times more productive implementing code in the early stages.

The key to building these simulations is to use the multitasking features of the host operating system. The devices that you are simulating execute as one or more separate heavyweight processes (not threads). Each of these processes creates either a shared memory segment or a message queue (we use the System V IPC mechanisms). Use shared memory to emulate a memory-mapped device, and message passing to emulate a stream-oriented device, such as an embedded network.

The embedded code executes in its own heavyweight process, with this process emulating the target processor. If you are using multitasking in the target environment (e.g. by using an RTOS), use the lightweight threading features of a POSIX-compliant operating system for each thread. If the RTOS is also POSIX-compliant, you can use the same interface. Otherwise, built a simple middleware layer (e.g. using macros) that unifies the thread interface.

Since it is often not possible to emulate the I/O devices exactly, we have found that we can write two device drivers, one for the simulator, the other for the real I/O device. When switching from simulation to the target environment, only the device drivers need to change. The rest of the code remains *identical*. If your RTOS does not provide device drivers, or you have found the POSIX-style *open/read/write/close* interface for device drivers inadequate, consider developing each device driver as a separate thread, as we describe in [1].

Using a simulator has the further advantage that it can be replicated easily. Suppose three programmers need to share the target hardware. There can be lots of human synchronization needed for each one to get a turn using the hardware. With the simulator, however, two of them can use their own copy of the simulator, and the third person the real hardware. The resource conflicts are minimized, and all three of the programmers are more productive.

The simulator can also be used by the customer to obtain customer feedback even before the hardware is ready, especially with regards to the user interface if the application has an embedded display. The customer can try out the menu system, key inputs, and provide feedback about what might be missing, what is displayed but confusing, or suggestions for shortcuts that can greatly improve usability of the product.

#17 *Error detection and handling are an afterthought and implemented through trial and error*

Error detection and handling are rarely incorporated in any meaningful fashion in the software design. Rather, the software design focuses primarily on normal operation, and any exception and handling are added after the fact by the programmer. The programmer either puts in error detection everywhere, many times where it's unnecessary but its presence affects performance and timing; or does not put in any error handling code except on an as-needed basis as workarounds for problems that arise during testing. Either way, the error handling isn't designed and its maintenance is a nightmare.

Instead, error detection, or related issues such as fault tolerance, should be incorporated into the design of the system, just as any other state. Thus, if an application is built as a finite state machine, an exception can be viewed as an input that causes action and a transition to a new state. Unfortunately, the best way to implement a complete design that incorporates all error detection and handling is still a major research topic.

#16 *Generalizations based on a single architecture*

Embedded software designers may have the need to develop software that is intended to run on a variety of processors and platforms. In such a case, it's not uncommon for the programmer to begin writing software for one of the platforms, but generalize anything and everything in preparation for porting the code at a later time.

Unfortunately, doing so usually causes more harm than good. The design will tend to over-generalize items that are very similar on very different architectures, while not generalizing some items that are different, but that the designer did not foresee as different.

A better strategy is to design and develop the code simultaneously on multiple architectures, generalizing only those parts that are different in the different architectures. Intentionally choose three or four processors that are very different (for example, from different manufacturers and using different architectures).

#15 *Optimizing at the wrong time*

Coarse-grain optimizations are usually global in nature, and involve major design decisions that need to be considered before implementation begins. Fine-grain optimizations are localized, and can be performed during the later stages of implementation.

Both coarse-grain and fine-grain optimizations require a knowledge of the hardware peculiarities. Thus, an analysis of the hardware should be performed even before implementation begins.

For example, how long does it take to add two eight-bit numbers? What about two 16-bit or 32-bit numbers? What about two floats? What if an eight-bit number is added to a float? Without an answer to these questions, a software designer is not prepared to design the real-time software, because decisions such as whether to perform all operations in floating point or as scaled integers cannot be made. If a design is implemented assuming one of those, it will be very difficult later to change.

Here are answers to the above questions for a 6MHz Z180 (in microseconds): 7, 12, 28, 137, and 308. Note that it takes 250% more time to do float plus byte than float plus float, due to the long conversion time from byte to float. Such anomalies are often the source of code that overloads the processor.

In another example, a special purpose floating-point accelerator did floating-point addition/multiplication 10 times faster than a 33MHz 68882, but **sin()** and **cos()** took the same amount of time. This is because the 68882 has the trigonometric functions built into its hardware, while the floating point accelerator did those particular functions in software.

When code is implemented for a real-time system, being aware of the timing implications of every single line of code is important. Understand the capabilities and limitations of the target processor(s), and redesign an application that makes excessive use of slow instructions. For example, for the Z180, doing everything in float is better than having only some variables float and lots of mixed-type arithmetic.

On the other hand, this information should not be used to perform fine-grain optimizations. These are optimizations such as replacing 3*x with x+x+x. This not only affects readability, but in some cases will actually slow down code, because one multiplication is faster than two additions.

A programmer who optimizes every line of code from the beginning may implement the first version of the code in an unreadable manner. This could severely increase the testing and debugging time, at possibly unnecessary cost since it is not even known yet if the code needs to be optimized.

As a general rule, do not perform fine-grained optimizations during implementation. Only optimize segments of code later during the debugging phases if it proves necessary to get better performance. If optimization is unnecessary, then keep the more readable code. If the CPU is overloaded, it is nice to know that a variety of places remain in the code where simple, straightforward optimizations can be performed quickly.

#14 *Reusing code not designed for reuse*

Code that is not designed for reuse will not be in the form of an abstract data type or object. The code may have interdependencies with other code, such that if all of it is taken, there is more code than needed. If only part is taken, it must be thoroughly dissected, which increases the risk of unknowingly cutting out something that is needed, or unexpectedly changing the functionality. If code isn't designed for reuse, it's better to analyze what the existing code does, then redesign and re-implement the code as well-structured reusable software components. From there on, the code can be reused. Rewriting this module will take less time than the development and debugging time needed to reuse the original code.

A common misconception is that because software is defined in separate modules, it is naturally reusable. This is a separate mistake on its own, and related to creating software with too many dependencies. See more details in mistake #3.

#13 *Using blocking forms of message passing*

When software is developed as functional blocks, the first thought is to implement inputs and outputs as messages. Although this type of inter-process communication (IPC) works well in non-real-time environments—such as for distributed networking—it's problematic in a real-time system.

Several major problems arise when using blocking forms of message passing in a real-time system:

- Message passing requires synchronization, a primary source of unpredictability to real-time scheduling. Functional blocks end up executing synchronously, and thus analysis of the system's timing is difficult, if not impossible.
- In systems with bi-directional communication between processes or any kind of feedback loop, deadlock is a possibility.
- Message passing incurs significantly more overhead as compared to shared memory. While messages may be required for communication across networks and serial lines, it's often inefficient when random-access to the data is possible, as is the case for IPC on a single processor.

Simply replacing message passing with shared memory buffers and guarding access by using semaphores is not a good solution, as the semaphore mechanism has the same problems as blocking in message passing.

Research literature shows thousands of papers that describe IPC mechanisms for almost every scenario imaginable. While it might take some time to find just the right solution for the problem at hand, finding a solution that addressed the above problems could greatly improve the robustness and reliability of the real-time system.

For example, in our component-based control systems, we use state-based communication to provide higher assurability [2,3]. In this scheme, the most recent data is always available to a process when the process needs it. Steenstrup and Arbib developed the port-automation theory to formally prove that a stable and reliable control system can be created by only reading the most recent data [4]. Costly blocking is eliminated by creating local copies of shared data, to ensure that every process has mutually exclusive access to the information it needs. Using states instead of messages also provides robustness if the possibility of lost messages exists, if code does not all execute at the same rate, and if implementing with shared memory generates less operating system overhead.

Converting control systems from message-based communication to state-based communication is generally straightforward. For example, an intelligent train control system has independent control of every brake to maximize train handling. To minimize stopping distance when coming to a full stop, all the brakes on the train must be applied together. The I/O logic for each brake is handled by a separate process; the control module must inform each brake module to turn on the brakes. When using a message-based system, the controlling unit sends a message, "apply brake," to every brake process. Due to the dependencies among processes, it creates a real-time system that is difficult to analyze and has the potential for unbounded blocking or deadlocks, thus it is not suitable for real-time systems. In contrast, in a state-based communication mechanism, each brake module executes periodically and monitors the brake variable to update the state of its own brake I/O. For example, instead of the "apply brake" message, revise the state of the brake variable so that it says, "the brake should be on." Since processes are periodic, a schedulability analysis is easier. Processes only need to bind to a single element in the state table, thus eliminating direct dependencies between processes. Communication through shared memory also incurs less overhead when compared to a message-passing system.

There are various other IPC mechanisms that are suitable for real-time systems too. The important part is to use solutions that take timing into consideration, and that do not result in a task blocking at arbitrary times because a queue is empty or full.

#12 *No memory analysis*

The amount of memory in most embedded systems is limited. Yet most programmers have no idea what the memory implications are for any of their designs. When they're asked how much memory a certain program or data structure uses, they are commonly wrong by an order of magnitude.

In microcontrollers and DSPs, a significant difference in performance may exist between accessing ROM, internal RAM, and external RAM. A combined memory and performance analysis can aid in making the best use of the most efficient memory by placing the most-used segments of code and data into the fastest memory. A processor with cache adds yet another dimension to the performance.

A memory analysis is quite simple with most of today's development environments. Most environments provide a .map file during compilation and linking stages with memory usage data. A combined memory/performance analysis, however, is much more difficult, but is certainly worthwhile if performance is an issue.

#11 *Improper use of Global Variables*

Global variables are often frowned upon by software engineers because they violate encapsulation criteria of object-based design and make it more difficult to maintain the software. While those reasons also apply to real-time software development, avoiding the use of global variables in real-time systems is even more crucial.

In most RTOS, processes are implemented as threads or lightweight processes. Processes share the same address space to minimize the overhead for performing system calls and context switching. The side effect, however, is that a global variable is automatically shared among all processes. Thus, two processes that use the same module with a global variable defined in it will share the same value. Such conflicts will break the functionality; thus, the issue goes beyond just software maintenance.

Many real-time programmers use this to their advantage, as a way of obtaining shared memory. In such a case, however, care must be taken and any access to shared memory must be guarded as a critical section to prevent undesirable problems due to race conditions. Unfortunately, most mechanisms to avoid race conditions, such as semaphores, are not real-time friendly, and they can create undesired blocking and priority inversion. The alternatives, such as the priority ceiling protocol, use significant overhead.

Global variables should never be used as a substitute for passing arguments. While this may seem like a good way to reduce the overhead of passing arguments to functions, it prevents any form of scalability or reuse of software modules for multiple functions, because the functions or modules that use the global variables cannot be replicated. Instead, use an abstract data type (ADT) to encapsulate all the data that you would like to make global. Pass a pointer to this ADT as the first argument to any function. The subroutine call overhead from passing a single argument is minimal, yet this technique enables code to be replicated, since each instance of a module can have its own ADT. For larger applications for which overhead is less of a concern, objects can replace ADTs to eliminate the need for most global variables.

There are times when it is acceptable or necessary to use global variables. For example, if every instance of a module needs to share a data item, then a global variable might be in order. In this case, however, appropriate synchronization or mutual exclusion is needed to conserve the integrity of the shared data.

Exchanging data with interrupt handlers often forces the need for global variables, since the interrupt handler executes in a different context than the rest of the code. As with global variables used for shared memory, extra care needs to be taken to avoid race conditions.

#10 *Indiscriminate use of interrupts*

Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements. The reason for this delay is that interrupts preempt everything else and aren't scheduled. If they preempt a regularly scheduled event, undesired behavior may occur. An ideal real-time system has no interrupts.

Many programmers will put 80% to 90% of the applications's code into interrupt handlers. Complete processing of I/O requests and the body of periodic loops are the most common items placed in the handlers. Programmers claim that an interrupt handler has less operating system overhead, so the system runs better. While it's true that a handler has less overhead than a context switch, the system doesn't necessarily run better for several reasons:

- Handlers always have high priority and can thus cause priority inversion;
- Handlers reduce the schedulable bound of the real-time scheduling algorithm, thus counteracting any savings in overhead as compared to a context switch;
- Handlers execute within a different context and force the use of shared global variables to exchange data with the rest of the application;
- Handlers are difficult to debug and analyze because few debuggers allow the setting of breakpoints or performing user I/O within the handler.

Instead, minimize the use of interrupts when possible. For example, program interrupts so their only function is to signal an aperiodic server. Or convert handlers from periodically interrupting devices to periodic processes. If you must use interrupts, use only real-time analysis methods that take into account the interrupt handling overhead. Never assume that overhead from interrupts and their handlers is negligible.

#9 *Poor software design diagrams*

Most software systems are designed such that the entire system is defined by a single diagram (or, even worse, none!). Yet a physical item like a chair or table would have several more diagrams—for instance, top view, side view, bottom view, detailed view, functional view, and so on-despite the fact that a chair can be much simpler than a software project.

When designing software, getting the entire design on paper is essential. The most commonly accepted methods are through the creation of software design diagrams. Many different kinds of diagrams exist. Each is designed to present a different view of the system.

Of course, there are good diagrams and there are poor diagrams. A good diagram properly reflects the ideas of the designer on paper. A poor diagram is confusing, ambiguous, and leaves too many unanswered questions. To create good software, the diagrams representing the software designs must be good.

Common techniques for presenting designs through good diagrams include the following:

- An *architectural design diagram* shows the top-down decomposition of a large project. It is usually a data flow diagram that shows relationships between objects, modules, or subsystems based on the data exchanged between them
- Each element in an architectural design should be represented by a *detailed design diagram*. This diagram provides enough detail for a programmer to implement the details without ambiguity. In a multi-level decomposition, the detailed design at one level may become the architectural design for the next lower level. Therefore, the same diagramming techniques are applicable to both kinds of diagrams

The software designers must be sure to distinguish whether they're using *process-oriented* or *data-oriented* designs. A process-oriented design, as typically used in many control and communication systems, should include data flow diagrams (such as for control system representation), process flow diagrams (also called flow charts), and finite state machines representations. A data-oriented design, as used in knowledge-based and database applications, should consist of relationship diagrams, data structure diagrams, class hierarchies, and tables.

An *object-oriented* design is a combination of process-oriented and data-oriented design, and should contain diagrams that represent all of the different views.

As an example of the need for diagrams, consider the data structures shown in Figure 1a. If you have an application with lots of structures defined, but no diagrams to show the relationship between them, you would need to spend hours (or days) going through the code or relying on comments (which may or may not be there) to figure out the relationships.

On the other hand, the data structure diagram shown in Figure 1b clearly shows the relationship. For example, it now becomes obvious that structure *def_t* is a doubly linked circular list with a header node; there are *nxyz* instances of the structure *xyz_t*, defined as an array; and structure *abc_t* points to both the header node of *def_t* and to the first element in *xyz_t*.

Even when someone has provided design diagrams, they often have not provided a legend. Such a diagram usually mixes data flow and process flow blocks, and is marred by inconsistencies and ambiguities. Even many of the diagrams in software engineering textbooks have this problem!

A quick rule of thumb to determine whether a diagram has flaws is to look at the legend and make sure that every box, line, dot, arrow, thickness, fill color, or other marking on the diagram matches the function specified in the legend. This simple rule serves as a syntax checker, allowing developers and reviewers to quickly identify problems with the design. Furthermore, it forces every different type of block and line and arrow to be drawn differently, so that different objects are visually distinguishable.

Diagrams can be drawn according to a standard such as UML or based on a custom set of conventions developed by the company. What is important is that for every design diagram there is a legend, and that all diagrams of the same type use the same legend. Consistency is the key.

Following are guidelines for creating consistent data flow, process flow, and data structure diagrams. Similar guidelines should be established for any other kind of diagram required by an application.

Data flow diagrams. These diagrams show the relationship and dependencies between modules based on the data that is communicated between them. These diagrams are most often used in the modular decomposition phases. The data flow diagram is the most common diagram at the architectural level;

but most data flow diagrams are poorly done, usually a result of inconsistencies in the diagram.

To create good diagrams, create a convention and stick with it. Always include a legend that explains the convention. Minimize the number of lines (and therefore, data items) that flow between processes or modules. Note that each block in this diagram will become a module or process, and each line will be some form of coupling between module or communication between processes. The fewer lines, the better. Some typical conventions for data flow diagrams include the following:

- Rectangles are data repositories such as buffers, message queues, or shared memory
- Rounded-corner rectangles are modules that execute as their own process
- Directed lines represent data that flows from the output of one process or module to the input of another process or module

Process flow diagrams. These diagrams generally show the details within a module or process. They are most often used during the detailed design. As with data flow diagrams, create a convention, stick with it, and make a legend that explains the conventions. Some typical conventions for process flow diagrams include:

- Rectangles are procedures or computations
- Diamonds are decision points
- Circles are begin, end, or transfer points
- Directed lines represent the sequence to execute code
- Ovals represent interprocess communication
- Parallelograms represent I/O
- Bars represent synchronization points

Data structure diagrams and class hierarchies. Data structure diagrams and class hierarchies show the relationship between multiple data structures or objects. Such diagrams should contain enough detail to directly create a struct (if using C) or class (if using C++) definition in a module's *.h* file.

Some typical conventions for these diagrams include:

- A single rectangle is a single field within a structure or class
- Groups of adjacent rectangles are all in the same structure or class
- Non-adjacent rectangles are in different structures or classes
- Arrows leaving a rectangle indicate pointers; the other side of the arrow shows the structure or object being pointed to
- Solid lines show relationships between classes. A legend should indicate the type of relationship(s) shown in the graph. Each different type should be represented by a line of a different width, color, or type

For example, Figure 1b is a data structure diagram.

#8 *"It's just a glitch."*

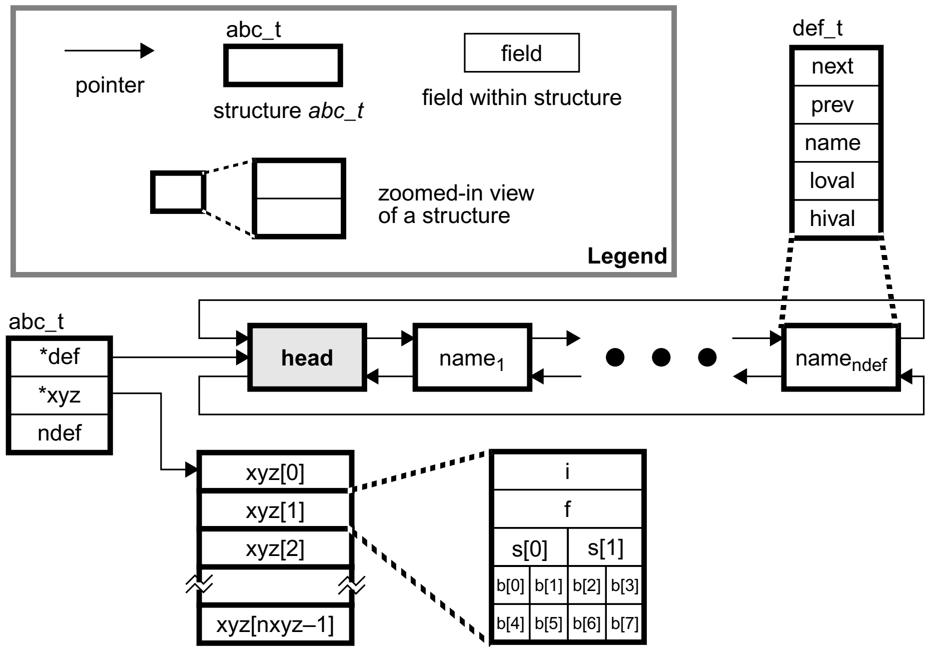
Some programmers use the same workarounds over and over again because the system has a glitch. A programmer's typical response is that it always executes well if the workaround is used.

```
typedef struct _def_t {
  struct _def_t *next;
  struct _def_t *prev;
  char name[8];
  short loval;
  short hival;
} def_t;

typedef struct _xyz_t {
  int i;
  float f;
  short s[2];
  unsigned char b[8];
} xyz_t;

typedef struct _abc_t {
  def_t *def;
  xyz_t *xyz;
  short ndef;
} abc_t;
```

(a)



(b)

Figure 1: Example of a data structure diagram. Suppose that only the data structures in (a) are provided. The lack of the diagram in (b) would make it extremely difficult to understand the true structure of the design. Providing the diagram in (b) significantly improves the ability to visualize the design.

Unfortunately, the same errors that force a workaround are likely to resurrect themselves later in a different form. Anytime there is any "glitch," it means something is wrong! Make sure appropriate steps are taken to understand the problem. A workaround may be valuable to ensure that a product is shipped on time, but immediately after the deadline, take a bit of extra time to identify the problem, to ensure it does not show up again—such as during the next big demo.

The most common problems associated with glitches that cause the program to crash or perform incorrectly just once every few days or weeks are a race condition, memory corruption, deadlock, or priority inversion. Each of these problems is extremely difficult to track down, because they occur randomly and leave very little evidence as to the true cause.

The solution is two-fold. For new applications, take precautions during the design and implementation of the application to minimize the possibility of any of these happening. A formal code review might identify the problems whereas testing will not (see Mistake #6). For addressing race conditions, this means minimizing the number of global variables or shared memory segments, minimizing preemption, and minimizing the number of interrupt handlers. For memory corruptions, use a tool such as Purify to identify all possible problems. Since such a tool might not be available in the target environment, use it in the emulator environment to at least eliminate most problems. To avoid deadlocks, consider using IPC mechanisms that provide deadlock-free solutions, such as the state-based communication we described in #13 or the priority ceiling protocol. To minimize the chance of priority inversion, don't use interrupts, and use proper real-time scheduling methods.

Despite all the precautions, the problems might still arise, and they require debugging. To pinpoint the problems, several techniques can be used.

To track down a randomly-occurring problem that might be a result of a race condition, put a “sleep()” command before and after every access to a shared data item. This will obviously slow down the code tremendously, so be sure you are executing code but with minimal power to the rest of the application (e.g. turn off power to motors). If a race condition does exist, it is much more likely to occur, since you will be forcing the context switches to occur during the critical sections.

A common memory problem is stack overrun. To check for this, initialize all of the memory allocated for the stack with a non-zero and non-0xFF value. For example, put 0x55 everywhere. Run the code. Next time the glitch occurs, check to see if there are still 0x55 in the memory. If not, then a stack overrun did occur. You can even monitor the stack regularly in this way to find the best size for the application.

For any type of random glitch, debugging might need to be done over an extended period of time. When the glitch occurs, make a note of the scenario in as much detail as possible. Add some debugging code that might help zero in on the problem. But since the problem only recurs occasionally, after the debugging code is in, go back to whatever else you are working on. If that glitch occurs again, look at the debug output for clues, and repeat by adding more debugging information. If the embedded system has extra memory, debug output can simply be copied into memory, and looked at only if necessary. If there is very little memory but a spare digital I/O port is available, the debugging output can be copied to that I/O port, and captured by a logic analyzer. If the glitch occurs, look at the logic analyzer output for clues.

There is no doubt that finding the cause of random glitches is one of the hardest debugging tasks for an embedded system programmer. Keep this in mind from the beginning, so as to reduce the number of such errors that enter the system, and ultimately reducing the debugging time to find the cause of the errors.

#7 *The first right answer is the only answer*

Inexperienced programmers are especially susceptible to assuming that the first right answer they obtain is the only answer. Developing software for embedded systems is often frustrating. It could take days to figure out how to set those registers to get the hardware to do what they want. At some point, though, it works. Once this happens, many programmers will remove all the debug code and put that code into the module for good. Never shall that code change again. Because it took so long to debug, nobody wants to break it.

Unfortunately, that first success is often not the best answer for the task at hand. That step is definitely important, because improving a working system is much easier than getting the system to work in the first place. But improving the answer once the first answer has been achieved is rarely done, especially for parts of the code that seem to work fine. Indirectly, however, a poor design that remains unchanged might have a tremendous effect, like using up too much processor time or memory, or creating an anomaly in the timing of the system if it executes at a high priority.

As a general rule of thumb, always come up with at least two designs for anything. Quite often, the best design is in fact a compromise of other designs. If a developer can only come up with a single good design, other experts should be consulted to obtain alternate designs.

#6 *No code reviews*

Many programmers, both novices and experts, guard their code with the same secrecy that inventors guard patentable ideas. This practice, unfortunately, is extremely damaging to the robustness of any application. Usually, programmers know they have messy code; hence they fear others seeing and commenting on it. As a result, they hide it the same way that children hide messy rooms from their parents.

To guarantee robustness, formal code reviews (also called software inspections) must be performed. Code reviews should be done regularly for every piece of code that goes into the system. A formal review involves several people looking over code and tracing it by hand on paper. Software engineering studies have shown that more bugs can be found in a day of code reviews than a week of debugging.

The programmer should also get into the habit of doing self-reviews. Many programmers write code, run it, and see what happens—and if it does not work, they start debugging it, without ever tracing it on paper. Spending one day hand-tracing the code can save days or weeks of agonizing debugging.

Code reviews have the additional positive side effect of increasing the number of people who understand the code, thus preventing total reliance on a single employee.

#5 *Nobody else here can help me*

As most any teacher will confirm, you learn more about a topic by teaching it.

Real-time programmers often feel helpless when they encounter obstacles (which happens all the time) such as an I/O device not working as described in the documentation. Often, few others in the organization have the level of knowledge required for this kind of programming, leaving these programmers to solve the problem without assistance. Unfortunately, this misconception that nobody else can help often leads to the downfall of projects or quality of the application, as adequate solutions might never be found. If no one else has more expertise, the programmer should teach the material to someone with less expertise, so that both the teacher and the student can arrive at a better understanding of the problem.

Many organizations have new recruits who are willing to learn new things to gain experience. The expert should explain to such eager people how the program works and what the problem is. The new person likely will not be able to fully understand the problem. However, their questions may expose an issue or problem that was overlooked by the expert and may lead to a solution.

This approach also has an important side effect. It doubles as a training technique so that when advanced programming knowledge is required, there are more programmers qualified to contribute.

#4 *One Big Loop*

When real-time software is designed as a single loop, there is no flexibility to modify the execution time of individual parts of the code. Few real-time systems need to operate everything at the same rate. If the CPU is overloaded, one of the methods to reduce utilization is to selectively slow down only the less critical parts of the code.

Real-time systems should be implemented as concurrent applications. For lower-end processors, use a flexible multi-rate executive. For higher-end applications, use an RTOS with limited or full preemption and appropriate tools to guard against critical sections. In either case, the small overhead that is added by the executive or RTOS is easily reclaimed by executing each part of the code only at the rate it needs, rather than executing all code at the fastest rate.

Developing software as a collection of small loops instead of one big loop has the further advantage of providing good modular decomposition into tasks, with data exchanges between the modules being explicit. With proper design, it may be possible to reuse some of the modules in other applications. In the one big loop scenario, however, it is doubtful that any part of the software can be reused. Identifying the design of software modules for reuse is described next.

#3 *Too many inter-module and circular dependencies*

The dependencies between modules in a good software design can be drawn as a tree, as shown in Figure 2a. A dependency diagram consists of nodes and arrows, such that each node represents a module (such as one source code file), and the arrows show dependencies between that node and other modules. Modules on the bottom-most row are not dependent on any other software module. To maximize software reusability, arrows should always point downwards, and not upwards or bidirectionally. For example, module *abc* depends on module *def* if it has a *#include* "*def.h*" in the code, or an *extern* declaration in the file *abc.c* to a variable or function defined in module *def.c*.

The dependency graph is a valuable software engineering aid. Given such a diagram, it's easy to identify what parts of the software can be reused, create a strategy for incremental testing of modules, and develop a method to limit error propagation through the entire system.

Each circular dependency (a cycle in the graph) reduces the ability to reuse the software module. Testing can only occur for the combined set of dependent modules, and errors will be difficult to isolate to a single module. If the graph has too many cycles, or a major cycle exists where a module at the bottom-most level of the graph is dependent on the top-most module, then not a single module is reusable.

Figure 2b and Figure 2c both include circular dependencies. If a circular dependency is inevitable, Figure 2b is much preferred over Figure 2c, since in Figure 2b reusing some of the modules is still possible. The restriction in Figure 2b is that modules *pqr* and *xyz* can only be reused together. In Figure 2c, however, reusing any subset of modules isn't possible, as too many dependencies exist between modules. Furthermore, a major circular dependency exists, where module *xyz*-which should not be dependent on anything because it is at the bottom of the graph-is dependent on *abc*. *It only takes one such major cycle to make the entire application non-reusable*. Unfortunately, most existing applications are more similar to Figure 2c than to Figure 2a or Figure 2b, hence the difficulty in reusing software from existing applications.

To best use dependency graphs to analyze the reusability and maintainability of software, write code that makes it easy to generate the graph. That is, all *extern* declarations for exported variables in functions in a module *xxx* should be defined in file *xxx.h*. In module *yyy*, simply looking at what files are *#include*d allows determination of that module's dependencies. If this convention is not followed, and an *extern* declaration is embedded in *yyy.c* instead of *#include*-ing the appropriate file, then the dependency graph will be erroneous and an attempt to reuse code that appears to be independent of the other module will be difficult.

A leading cause of circular dependencies is a single *#include*d file with all of the system's constants, variable definitions, type definitions, and/or function prototypes is a sure sign of non-reusable code. During a code review, it takes only five seconds to spot code that cannot be reused, if such a file exists. The key to spotting these problems almost immediately is the existence of an *include* file, often called *globals.h*, but other common names are *project.h*, *defines.h*, and *prototypes.h*. These files include all of the types, variables, *#defines*, function prototypes, and any other header information that is needed by the application. Programmers will claim that it makes their lives much easier because in every module all they need to do is include a single *.h* file in every one of their *.c* files. Unfortunately, the cost of this laziness is a significant increase in development and maintenance time, as well as many circular dependencies that make it impossible to use any subset of the application in another application.

The right way is to use strict modular conventions. Every module is defined by two files, the *.c* and the *.h*. Information in the *.h* file is only what is exported by the module. Information in the *.c* file is everything that isn't exported. More details on enforcing strict modular conventions are given next.

#2 No naming and style conventions

For non-real-time system development, this mistake is #1.

Creating software without naming and style conventions is equivalent to building homes without any building codes. Without conventions, each programmer in an organization does his or her own thing. The problems arise whenever someone else has to look at the code (and if an organization properly does code reviews as in mistake #6, this will be sooner, not later). For example, suppose the same module is written by two different programmers. The code of one programmer takes one hour to understand and verify, while the same code by the other programmer takes one day. Using the first version instead of the second is an 800% increase in productivity!

Naming and style conventions are the primary factors that affect readability of code. If strict naming conventions are followed, a reader will know what the symbol is, where it is defined, and whether it is a variable, constant, macro, function, type, or some other declaration just by looking at it. Such conventions must be written, just as a legend must appear on a design diagram, so that any reader of the code knows the conventions.

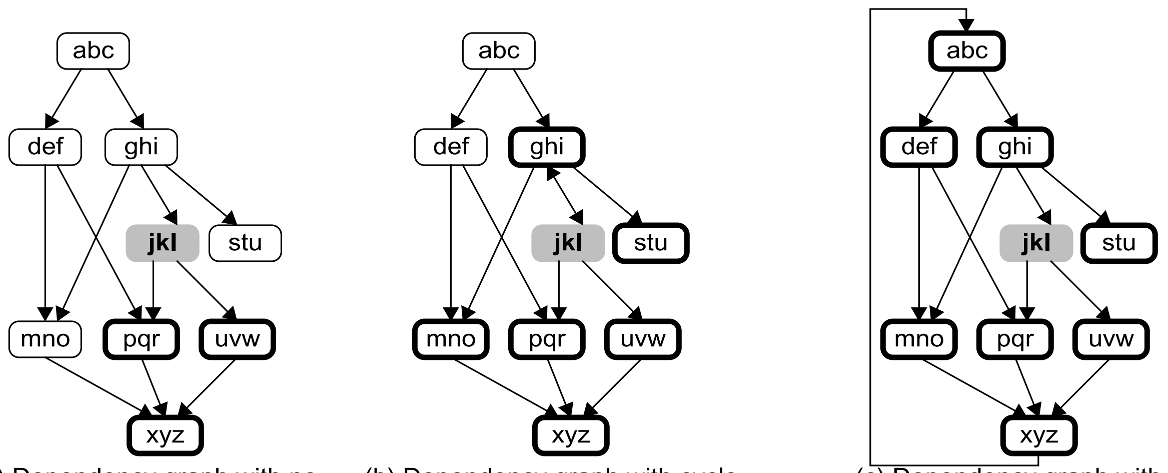
An organization should insist that all programmers use the naming conventions in all parts of their projects. Part of a code review should include checking for adherence to the conventions. If necessary, a company can hold back merit raises from programmers who do not follow the conventions; it may seem like a silly reason to refuse to grant a raise, until you take into consideration that a programmer not following the conventions may cost the company $50,000 the following year due to all of the extra labor expended by other employees to understand and modify the code. If employees prefer to use their own conventions, that's their tough luck. Just as architects must follow strict guidelines to get their designs approved by the building inspector, a software engineer should follow strict guidelines as established by the company to get their programs approved by the quality assurance department.

The most fundamental questions with respect to software maintainability are the following:

- If a customer reports a software error, how quickly can it be found?
- If a customer requests a new feature, how quickly can it be added?
- Once the error is identified, how many lines of code must be changed to fix it?

Obviously, answers to the above questions depend on the specific application and nature of the problems. However, given two pieces of code that have the same functionality and need the same fix, which program's conventions will help do the job more quickly? These criteria help to evaluate software maintainability, and should be used when comparing not only designs, but also styles and conventions.

Table 1 shows an excerpt of the C naming conventions that are enforced at our organization. Researchers and engineers who have learned these conventions quickly appreciate the



(a) Dependency graph with no cycles. This is desirable.

(b) Dependency graph with cycle between *ghi* and *jkl*

(c) Dependency graph with a a major circular dependency

Figure 2: Examples of dependency graphs, without and with cycles. An objective in developing good software is to decompose code into modules to minimize or eliminate circular dependencies. In each diagram, the modules with thick borders should modules that are directly or indirectly dependent on module *jkl*, and would be needed to test or reuse *jkl*, and could be affected by any problem in *jkl*.

more readable code they produce, especially after they are forced to read code written by someone else who does not follow any written convention.

Whether an organization favors these conventions or its own doesn't matter; what is important is that the naming conventions can be backed by a good reason why each specific convention was selected, they are written and distributed to all developers, and they are strictly adhered to by all programmers.

Functions should always be given names such that each exported function has a converse, as shown in Table 2. Two important benefits are gained by defining functions in pairs. It forces the designer to ensure completeness and allows the designer to create the two portions simultaneously, using each part to test the other. It also ensures that pairings are consistent; for example, that the converse of *send* is not *read*, and that the converse of *create* is not *finish* (see Table 2). If a designer is creating the code for reading and writing at the same time, both pieces of code can be tested by writing from one process and reading from the other.

To create software so that further decomposition can be done quickly if it's required, put names in an order that decomposes the module or ADT into sub-parts, each of which is described by a known, and a single verb in the name as the last word in the compounded function name. Do not order the words in the way that they would naturally be read. For example, if module *xyz* has a secondary structure *xyzFile_t*, then functions that operate on that structure should be named the following:

```
xyzFileCreate
xyzFileDestroy
xyzFileRead
xyzFileWrite
```

and not

```
xyzCreateFile
xyzDestroyFile
xyzReadFile
xyzWriteFile
```

Note that the last word for any function name should be the verb that represents the action performed by the function. The middle words are typically nouns that represent the object(s) within the module on which the verbs act.

This convention makes it obvious that *xyzFile* is a sub-set of the *xyz* module. Furthermore, if the module xyz grows and the designer decides to further decompose it, it's easy to move the entire *xyzFile* subset to a separate module-say, *xyzfile*. A global search and replace of *xyzFile* to *xyzfile* would result in all the necessary changes, and within a few minutes, the decomposition would be complete. If this naming convention is not used, then trying to perform the same task of renaming all the symbols when the *xyzFile* subset of *xyz* is placed in another file would be very tedious.

While having a short cryptic module name is acceptable because the name serves as a prefix to everything, you should only use obvious abbreviations for function names. If an obvious abbreviation isn't available, use the full name. If an abbreviation is used, use it everywhere for the project.

For example, always use *xyzInit* as the initialization code for module *xyz*, rather than *xyzInitialize*. Or use either *snd* and *rcv*, or *send* and *receive*, but don't mix the two. Examples of other common abbreviations include *intr* for interrupt, *fwd* for forward, *rev* for reverse, *sync* for synchronization, *stat* for status, and *ctrl* for control. An abbreviation like *trfm*, on the other hand, supposedly short for transform, is not recommended because the abbreviation isn't obvious and readability is therefore compromised. In such a case, the function name without abbreviation, *xyzTransform()*, would be a better choice. Uncommon abbreviations are difficult to follow when reviewing the code. Using the slightly longer names is much better and avoids confusion as to what the function does.

#1 *No measurements of execution time*

Many programmers who design real-time systems have no idea of the execution time of any part of their code. For exam-

Table 1: Naming conventions to improve software maintainability for C-language programs

Symbol	Description	Symbol	Description
xyz.h	File that contains header info for module 'xyz'. Anything defined in this file MUST have an *xyz* or XYZ prefix, and must be something that is exported by the module.	*xyc.c*	File that contains code for module 'xyz'
xyz_t	Primary data type for module xyz. Defined in xyz.h	*_abcde_t*	Internally-defined type. Must be defined at top of *xyz.c*.
xyzAbcde_t	Secondary type "Abcde" for module xyz. Defined in xyz.h.		
xyzAbcde()	Function "Abcde" that applies to items of type xyz_t.	*Abcde()*	Internal function. Must be defined as static. Prototype at top of *xyz.c*. Function declared at bottom of xyz.c, after all the exported functions have been declared.
XYZ_ABCDE	Constant for module XYZ. Must be defined in xyz.h.	*abcde*	Local variable. Must be defined inside a function.Fields within a structure are also defined using this convention.
XYZ_abcde	Constant for module XYZ within an enumerated type.	*ABCDE* *_ABCDE* *_ABCDE_FGH*	Local constant internal to module. Must be defined at top of *xyz.c*. The third version allows the use of multiple words. For example, _ABCDE_FGH. If just "ABCDE_FGH", is used, it implies module "abcde"
XYZ_ABCDE()	#define'd macro for module XYZ. Must be defined in xyz.h.		
xyz_abcde	Exported global variable defined in module *xyz*. Must be defined in xyz.c, and declared as *extern* in xyz.h. Global	*_abcde*	Internal global variable. Must be defined as "static" at top of *xyz.c*. Note that even thouse these are internal,

Table 2: Examples of always defining functions in pairs.

xyzCreate	↔	xyzDestroy	xyzInit	↔	xyzTerm	xyzStart	↔	xyzFinish	xyzOn	↔	xyzOff
xyzAlloc	↔	xyzFree	xyzSnd	↔	xyzRcv	xyzRead	↔	xyzWrite	xyzOpen	↔	xyzClose
xyzStatus	↔	xyzControl	xyzNext	↔	xyzPrev	xyzUp	↔	xyzDown	xyzStop	↔	xyzGo

ple, my colleagues and I were asked to help a company identify occasionally erratic behavior in its system. From our experience, this problem is usually a result of a timing or synchronization error. Thus our first request was simply for a list of processes and interrupt handlers in the system, and the execution time in each. The list of names was easy for them to generate, but they had no measured execution times; rather, only estimates by the designers before the code was implemented.

Our first order of duty was to measure the execution time for each process and interrupt handler. We quickly discovered that the cause of the erratic behavior was system overload. Engineers at the company replied that they already knew that. But they were surprised to hear that the idle process was executing over 20% of the time. (When measuring everything, you must include the idle task.) The problem was that their execution time estimates were all wrong. One interrupt handler, with estimated execution time of a few hundred microseconds, took six milliseconds!

When developing a real-time system, measure execution time every step of the way. This means after each line of code, each loop, each function, and so on. This process should be continuous, done as often as testing the functionality. When execution time is measured, correlate the results to the estimates; if the measured time doesn't make sense, analyze it, and account for every instant of time.

Some programmers who do measure execution time wait until everything is implemented. In such cases, there are usually so many timing problems in the system that no single set of timing measurements will provide enough clues as to the problems in the system. The operative word in real-time system is time.

One obstacle that many engineers face with timing code is not knowing where are the starting and end points of each process. If code is implemented in such a way that the start and end points are not obvious, then the code must be redesigned. It is an indication of poor decomposition and likely many circular dependencies. While it may seem extreme to immediately suggest rewriting the code, consider how much time can be wasted in making the real-time system work if it is not possible to accurately measure time!

Summary

I have presented the 25 most common problems in real-time software development, from my perspective as both an industry consultant and an academic professor. Correcting just one of these mistakes in a project can lead to weeks or months of savings in manpower (especially during the maintenance phase of a software life cycle) or can result in a significant increase in the quality and robustness of an application. If many of your mistakes are common ones, and you can find and fix them, potential company savings or additional profits can be in the thousands or millions of dollars.

For each mistake listed, I encourage you to ask yourself about your current methods and policies, compare them to the reported mistakes and the proposed alternatives, and decide for yourself if there are potential savings for your project or company. I expect you'll find potential for improved quality and robustness at no extra cost, just by modifying some of your current practices.

References

[1] M. Moy and D. B. Stewart, "An engineering approach to determining sampling rates for switches and sensors in real-time systems," in *Proc. of Real-Time Applications Symposium*, Washington DC, June 2000.
http://www.embedded-zone.com/bib/conf/rtas2000.shtml

[2] D. Stewart, "Designing Software Components for Real-Time," *Embedded Systems Programming*, v.13, n.13, pp. 100-138, December 2000.
http://www.embedded-zone.com/bib/mags/esp2000.shtml

[3] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. on Software Engineering*, v.23, n.12, December 1997.
http://www.embedded-zone.com/bib/journals/tse97.shtml

[4] M. Steenstrup, M. Arbib, and E.G. Manes. Port Automata and the Algebra of Concurrent Processes, *Journal of Computer and System Sciences*, v. 27, n.1, pp. 29-50, August 1983.