1999 Embedded Systems Conference
San Jose, CA, September 1999.

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

David B. Stewart

Software Engineering for Real-Time Systems Laboratory
Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

Email: *dstewart@eng.umd.edu*
Web: *http://www.ece.umd.edu/serts*

Abstract

The most common mistakes and pitfalls associated with developing embedded real-time software will be presented. The origin, causes, and hidden dangers of these mistakes will be highlighted. Methods ranging from better education to using new technology and recent research results will be 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 author identified the most common mistakes from his experience in reviewing the software designs and implementations of many embedded programmers, ranging from seasoned experts in industry to rookies just learning the material in college.

Introduction

Novices and experts alike, whether in a university or company, repeat the same mistakes when developing real-time software over and over again. This observation was made while reviewing and grading code in projects in academia, and as a consultant involved in numerous design and code reviews for industry.

Most real-time software developers are not even aware that their favorite methods are problematic. Quite often, experts are self-taught, hence they tend to have the same bad habits as when they first began, usually because they 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 presentation is to improve the awareness to common problems, and to provide a start towards eliminating mistakes and thus creating software that is more reliable and easier to maintain.

This presentation first began as the *ten* most common problems. But there were just so many common problems that the list grew, and even trying to keep it within twenty five is difficult. Despite the title, this paper actually lists thirty common mistakes.

For each problem, the misconception or source of the problem is stated. In addition, possible solutions or alternatives that can help minimize or eliminate the mistakes are given. If the reader is 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 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. It is thus encouraged that for each mistake listed, 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 or the potential for improved quality and robustness at no extra cost by modifying some of your current practices.

The thirty most common problems are now presented; problems that are higher on the list (where #30 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 is opinion. It is 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.

#30 *"My Problem is Different"*

Many designers and programmers refuse to listen to the experiences of others, claiming that their application is 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 the nuts and bolts of the real-time infrastructure are considered. 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 (DSP). In response, ask "what is different in the LCD display software in a cellular phone versus one on a temperature 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 by 256 image processed by a DSP algorithm might not be very different from graphical code for a LCD dot matrix display of size 320 by 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 other than the target, and many of the issues in developing software for a DSP also applies to developing software for a microcontroller.

The timing and volume of data is 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.

Perhaps if control systems and communication systems are similar, 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.

#29 *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 it, or because of a claim that appears 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 is 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 is the biggest does not mean it is 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 more people have unused versions of the tools sitting on the shelf after discovering they were not suited.

Promises of compatibility: Managers are especially influenced by a product because of promises of compatibility. So what if software is 100% POSIX compatible. What is its relevance? Is there a plan to change the operating system? Suppose there is a change to another POSIX-compatible 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. Considering standards such as POSIX have not been proven to be good for real-time systems, let alone the best. Therefore, do not 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. [4,5]

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 tools.

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

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

- They are extremely difficult to test, 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 either to under-utilizing the CPU, or possibilities of timing errors when the longest path is taken.
- The difficulty of structure code coverage testing grows exponentially with the number of branches, thus branches should be minimized.

Instead, computational methods can often provide an equivalent answer. 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 down 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, many programmers still toggle a boolean value with the if statement above.

#27 *Delays implemented as empty loops*

Real-time software often needs delays to ensure that data sent or received over an I/O port has time to propagate. These delays are often 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 (e.g. 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 are very sporadic.

Instead, use a mechanism based on a timer. Some real-time operating systems (RTOS) provide these functions, but if not, one can still easily be built. Following are two possibilities to build a custom *delay(int μsec)* 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 μsec per tick. Suppose the value is 2 μsec per tick, and a delay of 10 μsec is needed. Then the delay function busy-waits for 5 timer ticks. Suppose a different speed processor is used, the timer ticks are still the same. Or if the timer frequency changes, then the system variable would change, and the number of ticks to busy wait would change, but the delay time would remain the same.

If the timer does not support reading intermediate count-down values, then 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 μsec 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 we tested it with, without every having to change the code.

#26 *Interactive and incomplete test programs*

Many embedded designers create a series of test programs, each program testing a separate feature. Executing test programs needs to be done one at a time, and in some cases requires the user to type input (say, through a keypad or switch) and observe the output response. The problem with this method is that programmers tend to only test what they are changing. Since there is often interaction between unrelated code due to the sharing of resources, every time a change is made, the complete 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 anytime even the smallest change is made, a complete test can easily and quickly be performed. Unfortunately, this is much easier said than done, as 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 person creating test software, and not a side-thought with test code written only on an as-needed basis.

#25 *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 inter-dependencies with other code, such that if all of it is taken, there is more code than needed. If only part is taken, there is a need to dissect it like a surgeon, and increases the risk of cutting out something that is not needed without realizing it, or unexpectedly changing the functionality. If code is not designed for reuse, it is better to analyze what the existing code does, then re-design and re-implement the code as well-structured reusable software components [4] . 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.

Quite often, there is a misconception that because software is defined in separate modules, it is naturally reusable. This is a separate mistake on its own, related to creating software with too many dependencies. See more details in Mistake #18 Too many inter-module dependencies..

#24 *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 platforms. In such a case, it is not uncommon for the programmer to begin writing software for one of the platforms, but generalizing 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 of the different architectures. Intentionally choose 3 or 4 processors that are very different (e.g. from different manufacturers and using different architectures).

#23 *One Big Loop*

When real-time system code is designed as a single big loop, there is no flexibility to modify the execution time of various parts of the code independently. 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. This works, however, only if the multitasking features of an RTOS are used, or the code was developed based on a flexible custom or commercial real-time executive.

#22 *Over-designing the system*

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. It is a luxury for a software developer to write programs for a processor with more than enough resources. In some cases, however, this luxury is so costly that it can make the difference between a profit and bankruptcy! It is a software engineer’s duty to contribute towards minimizing the price and power consumption of an embedded system. If the CPU is only 45% utilized, a processor that is half the speed can be used instead, thus saving as much as four times the power and possibly one or more dollars per processor.

If the product is mass-produced, a $1 saving in 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 3 hours when using a *heavy* battery. A watch, however, has a lightweight, cheap battery, that can last three years! Although software is not 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. It is recommended to 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 can use more efficient hardware is more likely to evolve from this approach than from using a fast processor, then trying to cut corners to bring down the cost of the system.

#21 *No analysis of hardware peculiarities before starting software design.*

How long does it take to add two 8-bit numbers? 16-bit numbers? 32-bit numbers? What about 2 floats? What if an 8-bit number is added to a float? A software designer who cannot answer these questions off the top of their head for their target processor is not adequately prepared to design and code real-time software.

Here are sample answers to the above measurements for a 6 MHz Z180 (in microseconds): 7, 12, 28, 137, 308! Note how it takes 250% more time to do float+byte than float+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 33 MHz 68882, but *sin()* and *cos()* took same amount of time. This is because 68882 has the trigonometric functions built-in to hardware, while the floating point accelerator did those functions in software.

When code is implemented for a real-time system, it is important to be aware of the timing implications of every single line of code that is typed into the computer. Understand the capabilities and limitations of the target processor(s), and redesign an application that makes excessive use of long instructions. For example, for the Z180 it is better to do everything in float than to have only some variables float and others integers with lots of mixed-type arithmetic.

#20. *Fine-grain optimization during first implementation.*

The converse to problem #21 is also a common mistake. Some programmers foresee anomalies (some which are real, some are myth). An example of a mythical anomaly is that multiplication takes much longer than addition. Many designers would implement *3*x* as *x+x+x*. On many embedded processors, however, multiplication is less than twice as long as addition, so *x+x+x* would be slower than *3*x*.

A programmer who foresees all the anomalies may implement the first version of the code in a very unreadable manner so as to optimize the code; this is before knowing if optimization is even needed. As a general rule, do not perform fine-grained optimizations during implementation. Only optimize segments of code later 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 there remains a variety of places in the code where simple straightforward optimizations can be performed quickly.

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

Some programmers use the same workarounds over and over and over again, because there is a glitch in the system. A programmer's typical response would be that it always executes fine if the workaround is used.

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!

#18 *Too many inter-module dependencies.*

The dependencies between modules in a good software design can be drawn as a tree, as shown in Figure 1(a). 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 what other modules the node depends on. 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 bi-directional. 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 is easy to (1) identify what parts of the software can be reused, (2) create a strategy for incremental testing of modules, and (3) develop a method to limit error propagation through the entire system.

Each circular dependency (i.e. 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 there are too many cycles in the graph, 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 1(b) and (c) both include circular dependencies. If a circular dependency is inevitable, then (b) is much preferred over (c), since in (b) it is still possible to reuse some of the modules. The restriction in (b) is that modules *pqr* and *xyz* can only be reused together. In (c), however, it is not possible to reuse any subset of modules, as there are too many dependencies between modules. Furthermore, there is a major circular dependency, 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 more similar to (c) than to (a) or (b), 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 modules dependencies. If this convention is not followed, and there is an *extern* declaration 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 independent of the other module will be difficult.

#17 *We have deadlines! I don't have time to take a break.*

Many programmers struggle for hours non-stop on a problem, only to hit dead-end after dead-end. They continue because they have a deadline, whether it be a product deadline at a company or a homework assignment at school. Many hours could be saved if the person simply takes a break after not making any progress for an hour. Relax, take a walk around a lake, go for a beer, take a nap, or anything.

With a clear mind that results from a bit of mental relaxation, it is much easier to analyze what is happening, and more quickly converge to a solution. A 2-hour break—even with a deadline looming—might save a day of work. A 10-minute coffee break *away from the computer* can sometimes save an hour of work.

#16 *Using message passing as primary inter-process communication.*

When software is developed as functional blocks, the first thought is to implement inputs and outputs as messages. Although this works well in non-real-time environments, such as for distributed networking, it is problematic in a real-time system.

There are three major problems that arise when using message passing in a real-time system:

- Message passing requires synchronization, a primary source for 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, there is the possibility of deadlock. Use a state-based system instead. For example, instead of saying "turn on the brake", revise the state so that it says "the brake should be on".
- Significantly more overhead as compared to using shared-memory. While messages may be required for communication across networks and serial lines, it is often inefficient when random-access to the data is possible, as is the case for inter-process communication on a single processor.

State-based communication is preferred in embedded systems to provide higher assurability. A state-based system uses structured shared memory, such that communication has less overhead. The most recent data is always available to a process when the process needs it. Streenstrup and

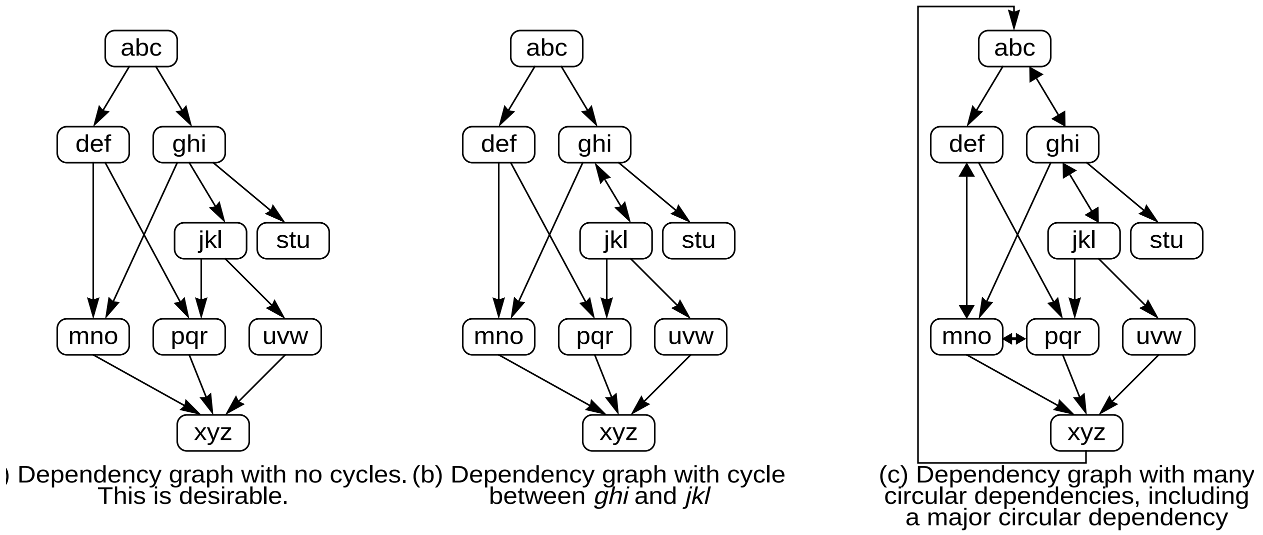
abc
def
ghi
jkl
stu
mno
pqr
uvw
xyz
) Dependency graph with no cycles. This is desirable.
(b) Dependency graph with cycle between *ghi* and *jkl*
(c) Dependency graph with many circular dependencies, including a major circular dependency

Figure 1: 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.

Arbib developed the port-automaton theory to formally prove that a stable and reliable control system can be created by only reading the most recent data [6] . 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 there is possibility of lost messages, if not all code executes at same rate, and if implemented using shared memory generates less operating system overhead. An example of an efficient state-based communication protocol is given in [4] .

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 input/output (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. This approach has high communication overhead, potential loss of messages if tasks execute at different frequencies, non-deterministic blocking, a separate copy of the message for every process, and there exists the possibility of deadlock. Due to the dependencies among processes, it creates a real-time system that is difficult to analyze and is not suitable for reconfigurable 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. Since processes are periodic, a schedulability analysis is easier. Processes only need to bind to a single element in the state variable table, thus eliminating direct dependencies between processes. Communication through shared memory also incurs less overhead as compared to a message-passing system.

When transferring a stream of data between objects, then a producer/consumer-type buffer should be created in shared memory, such that a maximum amount of data can be processed during each periodic cycle.

#15 *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. Quite often, few others in the organization have their level of knowledge for this kind of programming, leaving the programmer to solve the problem on their own. Unfortunately, this misconception often leads to the downfall of projects or quality, as important solutions are never found. If there is nobody else with more expertise, teach the material to someone with less expertise to better understand the problem.

If there is nobody with more knowledge that can help, then ask someone with *less* knowledge to help. In particular, many organizations have new recruits who are willing to learn new things to gain experience. Explain to such an eager person how the program works, and what the problem is. They likely will not be able to fully understand the problem. However, their questions may expose an issue or problem that was overlooked, and may lead to a solution to the problem.

This approach also has a *very* important side effect. It doubles as training new people, so that when advanced programming techniques are required, there is more than a single person in the company who can contribute.

#14 *Only a single design diagram.*

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, like top view, side view, bottom view, detailed view, functional view, etc., despite being much simpler than a software project.

When designing software, it is essential to get the *entire* design on paper. The most commonly accepted methods are through the creation of software design diagrams. There are many different kinds of diagrams. Each is designed to present a different view of the system.

Furthermore, there exists the notion of *good* diagrams vs. 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. In order to create good software, it is essential that the diagrams representing the software designs are *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 can be either 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 such that a programmer can implement the details without ambiguity. Note that 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 a *process-oriented* or *data-oriented* design is being used.

- 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.

#13 *No legend on design diagrams.*

Even when someone has design diagrams, many times there is no legend. Such a diagram usually mixes data-flow and process flow blocks, and is marred by inconsistencies and ambiguities. Even any of the diagrams in Software Engineering textbooks have this problem!!!

A quick rule of thumb to determine if 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 types objects are visually distinguishable.

It does not matter whether diagrams are 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 that is needed by an applications.

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. This is the most common diagram at the architectural level. Unfortunately, most data-flow diagrams are poorly done. This is usually a result of inconsistencies in the diagram.

To create good diagrams, do the following: Create a convention, and *stick with it*! Always make a legend that explains the convention. *Minimize* the number of lines (hence 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 less 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.

Several examples of data flow diagrams for control systems are given in [4] .

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:

- rectangle is a procedure or computation
- diamond is a decision point
- circle is a begin, end, or transfer point
- 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:

- a single rectangle is a single field within a structure or class.
- a group of adjacent rectangles are all in the same structure or class.
- non-adjacent rectangles are in different structures or classes.
- arrow leaving a rectangle indicates a pointer; other side of arrow shows the structure or object being pointed to.
- solid line shows a relationship 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.

An example of a data structure diagram is shown in Figure 2 (from [1]). An example of a class hierarchy diagram with multiple relationships is shown in [3] .

#12 *Using POSIX-Style device drivers*

Device drivers are used to provide a layer of abstraction to hardware I/O devices, so that higher levels of software can access devices in a uniform, hardware-independent fashion. Unfortunately, UNIX/POSIX-style device drivers used in many commercial RTOS do not fulfill the needs of embedded system design.

Specifically, the *open(), read(), write(), ioctl()*, and *close()* interfaces used by existing systems were created for files and other stream-oriented devices. In contrast, most real-time I/O have sensors and actuators that are connected through I/O ports. I/O ports include parallel I/O bits, analog to digital converters, digital-to-analog converters, serial I/O, or special purpose processors such as a DSP filtering data from a camera or microphone. Trying to adapt the POSIX device driver application programmer interface to use these devices forces programmers to perform the undesirable practice of coding hardware-specific functions at the application level.

Consider the following example: real-time software controlling an electromechanical device must turn on two solenoids, which are connected to bits 3 and 7 of an 8-bit digital I/O output board, without affecting the values on the other 6 bits of the port. None of the POSIX interfaces allow a programmer to specify such functionality.

In practice, three common approaches are used to map the hardware into this device interface. One approach modifies the arguments of the *write()* routine, such that the third argument specifies which ports to write instead of specifying the number of bytes to transfer. By changing the definition of the arguments for a standard API, the ability to use both that driver – and the code that calls it – in a hardware independent manner is eliminated, since there is no guarantee that a different I/O device driver will specify arguments in the same manner. What happens if we want to specify bits 3 and 7 of port 4 on an 8-port I/O board? A different definition of the argument would be required for this board.

A second approach uses *ioctl()*. The request and value are supplied as arguments. Unfortunately, there are no standards for the request, and every device freely selects its own set of supported requests. An example of the problems that result is with setting an serial port to 9600 bits per second. Different device drivers use different bit-mapped request structures to implement that function. Consequently, no compatibility exists between devices that should have the same hardware abstraction layer. Therefore, the application programs that use these devices become device dependent, and are not usable in a reconfigurable environment. Furthermore, *ioctl()* is primarily for use during initialization, as compared to *read()* and *write()*, because *ioctl()* has significantly more overhead in deciding what is being requested, and converting the arguments to a form suitable for the request.

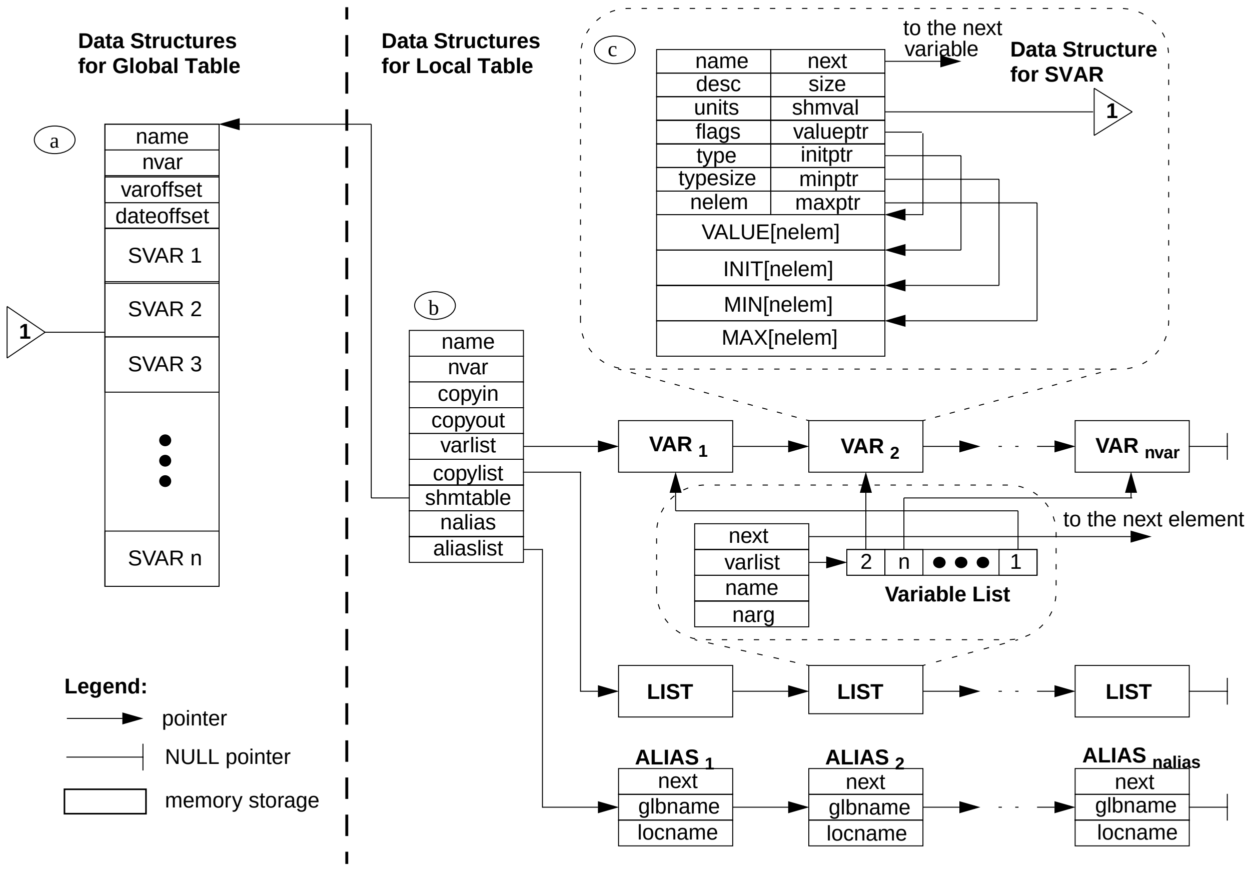

Figure 2: Example of a data structure diagram. The diagram is from [1] , and shows the relationship between multiple data structures for a global state variable mechanism [4]

A third, quite popular approach, is to use *mmap()* to map the registers of the devices. This method allows the programmer to directly access the device registers. Although this method provides the best performance, it defeats the purpose of using a device driver to create a hardware-independent abstraction of the device. Code written in this manner is non-portable, usually difficult to maintain, and cannot be used effectively in a reconfigurable environment.

The alternative is to encapsulate the device driver as its own thread. The data from the device is transferred through shared memory (and not via message passing as discussed in Mistake #16 Using message passing as primary inter-process communication.). The device driver is then a single process that can be executed whenever the device is present and needed, or not executed otherwise. This method has proven to enable the creation of device drivers for reconfigurable systems, as discussed in [4] .

#11 *Error detection and handling is an after-thought, and implemented through trial and error.*

Error detection and handling is rarely incorporated in any meaningful fashion in the software design. Rather, the software design focuses primarily on normal operation, and any exception and handling is added after the fact by the programmer. The programmer either (1) puts in error detection everywhere, many times it is added where it is unnecessary but its presence affects performance and timing; or (2) does not put in any error handling code except on an as-needed basis as workarounds for problems that arise during testing. Either way, there is no design of the error handling and its maintenance is a nightmare.

Instead, error detection should be incorporated into the design of the system, as just another 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. The best way to accomplish this is still a topic of research in academia.

#10 *No memory analysis*

The amount of memory in most embedded systems is limited. Yet most programmers have no idea about the memory implications for any of their designs. When they are asked how much memory a certain program or data structure uses, it is not uncommon for them to be wrong by an order of magnitude.

In microcontrollers and DSPs, there may be a significant difference in performance 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 [2] .

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.

#9 *Configuration information in #define statements*

Embedded programmers continually use *#define* statements in their C code to specify add such as register addresses, limits for arrays, and configuration constants. Despite this practice being very common, it is undesirable, as it prevents on-the-fly software patches for emergency situations, and it increases the difficulty of reusing the software in other applications.

The problem arises because a #define is expanded everywhere in the source code. The value might therefore show up at 20 different places in the code. If that value must change in the object code, it is not easy to pinpoint a single location to make the change.

As an example of an emergency patch, suppose a client discovers that for their application, a hard-coded 64 millisecond timeout period is insufficient, and it needs to be changed to 256 milliseconds. If #defines were used, then entire application must be recompiled, or every instance of that value being used in the machine language code must be patched.

On the other hand, if this information is stored in a configuration variable (possibly stored in a non-volatile memory) then it is simple to change the value in just one place. The code does not have to be recompiled; at worse there is a need to reset or reboot the system. The need for recompilation prevents in-the-field updates, as users generally do not have the means to recompile and download the code. Instead, the designers must make the change and distribute an entirely new revision of the software.

As an example of software reusability, suppose that code for an I/O device is implemented, with every the address of each register *#define*d. That same code cannot be reused if a second identical device is installed in the system. Instead, the code must be replicated, with only the port addresses changed.

Alternately, a data structure that maps the I/O device registers can be used. For example, an I/O device with an 8-bit status port, an 8-bit control port, and 16-bit data port at addresses 0x4080, 0x4081, and 0x4082 respectively can be defined as follows:

```
typedef struct {
    uchar_t    status;
    uchar_t    control;
    short      data
} xyzReg_t;

xyzReg_t *xyzbase = (xyzReg_t *) 0x4080;
    :
xyzInit(xyzbase);
etc.
```

Adding a second device at address 0x7E0 is as easy as adding another variable; e.g.:

```
xyzReg_t *xyzbase2 = (xyzReg_t *) 0x7E0;
    :
xyzInit(xyzbase2);
```

If an emergency patch needs to be made for this value, then the address of the variable *xyzbase* can be retrieved from the symbol table, and thus the precise memory location that needs to be modified is already known.

#8 *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 is wanted. At some point, Eureka! It works. Once it works the programmer removes all the debug code, and puts that code into the module for good. Never shall that code ever 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. It is definitely an important step, because it is much easier to improve a working system, than to get the system to work in the first place. However, improving the answer once the first answer has been achieved seems to rarely be done, especially for parts of the code that seem to work fine. Indirectly, however, a poor design that stays 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, then other experts should be consulted with to obtain alternate designs.

#7 *#include "globals.h"*

A single *#include*d file with all of the systems constants, variable definitions, type definitions, and/or function prototypes is a sure sign of code that is not reusable. During a code review, it only takes five seconds to spot code that cannot be reused, if such a file exists (see Mistake #25 Reusing code not designed for reuse.). 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.

A programmer will claim that it makes their life 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 (see #18) 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 is not exported. More details on enforcing strict modular conventions are given along with Mistake #2 No naming and style conventions!.

#6 *Documentation 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 is not 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 after***. Before implementation begins, begin 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 says!

Updating documentation during the implementation also serves as a review for the code. Often, the programmer finds bugs in their own code as they are writing about it. For example, "upon success, this function returns 1". The programmer then thinks, what if no success, what is returned? They look at their code, and realize that lack of success has not properly been implemented.

#5 *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, the programmer knows they have messy code, hence their fear of others seeing and commenting on it; as a result, they hide it the same way that people with messy rooms hide it from their parents.

To guarantee robustness, formal code reviews (also called *software inspections*) code must be performed. Code reviews should be done regularly, for every piece of code that goes into the system. A formal review involves multiple 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 will type code into the computer, run it, and see what happens, and if it doesn't work, start to debug it, without ever tracing it on paper. Spending one day hand-tracing the code can also save days or weeks of agonizing debugging.

#4 *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 is that interrupts preempt everything else, and are not scheduled. If they preempt a regularly scheduled event, undesired behavior may occur. In an ideal real-time system, there are no interrupts.

Many programmers will put 80 to 90 percent 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 thus the system runs better. While it is true that a handler has less overhead than a context switch, the system does not necessarily run better for two main reasons:

- Events cannot be scheduled to provide execution guarantees, as is possible with the rate monotonic or earliest deadline first real-time scheduling algorithm.
- The handlers can only exchange data through global variables; the next mistake explains why this is bad.

Instead, whenever possible make each thread periodic. Avoid aperiodic events. If using an RTOS with fixed priority scheduling, then use the rate monotonic algorithm to assign priorities to each thread. If using an RTOS with dynamic scheduling, use the earliest-deadline-first algorithm. If no RTOS is present, the non-preemptive earliest-deadline-first algorithm, as described in [4] , can be used.

#3 *Using Global Variables!*

Global variables are often frowned upon by software engineers, as it violates encapsulation criteria of object-based design, and makes it more difficult to maintain the software. While those reasons also apply to real-time software development, there is even *more* crucial to avoid the use of global variables in real-time systems.

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 an way of obtaining shared memory. In such a case, however, care must be taken, as any access to it 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. The alternatives, such as the priority ceiling protocol, use significant overhead.

Critical sections and race conditions are described in any operating systems textbook.

#2 *No naming and style conventions!*

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

Creating software with a lack of naming and style conventions is equivalent to building homes without any building codes. Without conventions, every programmer in an organization does their 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 #5 No code reviews., this will be sooner, not later). For example, suppose the same module is written by two different programmers. The code of one programmer takes 1 hour to understand and verify, while the same code by the other programmer takes 1 day. Using the first version instead of the second is an 800 percent increase in productivity!

The primary factor that affects readability of code are the naming conventions. If strict naming conventions are followed, simply looking at a symbol quickly tells the reader what the symbol is, where it is defined, and whether it is a variable, constant, macro, function, type, or some other declaration. 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.

Following is an excerpt of the naming conventions that are enforced in the Software Engineering for Real-Time Systems (SERTS) Laboratory at University of Maryland. Researchers who have learned these conventions quickly appreciate the 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 their own does not 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.

SERTS Naming Conventions

It is not sufficient to simply create modules in order to achieve a higher level of software maintainability. One of the biggest costs in maintaining software is spending time trying to figure out what some other programmer did in their code. To alleviate this problem, strict style and naming conventions must be adhered to in an entire organization. This appendix gives one such set of naming conventions that have already proven themselves to work well.

An organization should insist that all programmers *must* use these naming conventions in all parts of their projects. Part of a code review should check for adherence. If necessary, a company can keep back merit raises from programmers who do not follow the conventions; it may seem like a silly reason to not give a merit raise, until it is taken into consideration that a programmer not following the conventions may cost the company $50,000 the following year. If employees prefer using their own conventions, then tough luck. Just like architects must follow strict guidelines in order 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 criteria for software maintainability is 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 need to change to fix it?

Obviously the above questions are very dependent on the specific on the application and nature of the problems. However, given two pieces of code side-by-side that have the same functionality, and need the same fix, which program's conventions will help do the job more quickly? That is a primary criteria that helps to evaluate software maintainability, that should be used when comparing not only designs, but also styles and conventions.

Naming conventions are *extremely* important. The maintainability of software is directly related to the naming conventions that are used. By looking at any symbol name, it should be possible to immediately distinguish between constants, variables, macros, and functions, as well as whether something is local, global, internal, or external. Using the naming conventions in Table 1 consistently will yield all that information, simply from the symbol name.

Functions should always be given names such that each exported function has a converse, as shown in Table 2. By defining functions as pairs there are two important benefits. It forces the designer to ensure completeness, and allows a designer to create the two portions simultaneously, and use each part to test the other part. Also make sure that pairings are consistent; for example, make sure the converse of send is not read, and that the converse of create is not finish, if the conventions shown in Table 1 are used. For example, if 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. It is also worthwhile to always use the same conventions for the same components in different modules.

To create software in such a way that if further decomposition is needed, it can be done quickly, put names in order of "big to small" for compounded function names, and not in order that it would naturally be read. For example, if module *xyz* has a secondary structure *xyzFile_t*, then func-

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

Symbol	Description	Symbol	Description
xyc.c	File that contains code for module 'xyz'	*xyz_abcde*	Exported global variable defined in module *xyz.* Must be defined in xyz.c, and declared as *extern* in xyz.h. Global variables should be avoided!
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.	*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_t	Primary data type for module xyz. Defined in xyz.h	*abcde*	Local variable. Must be defined inside a function.Fields within a structure are also defined using this convention.
xyzAbcde_t	Secondary type "Abcde" for module xyz. Defined in xyz.h.	*_abcde*	Internal global variable. Must be defined as "static" at top of *xyz.c.*
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_t*	Internally-defined type. Must be defined at top of *xyz.c.*
XYZ_ABCDE()	#define'd macro for module XYZ. Must be defined in xyz.h.	*abc_e*	An exported enumerated type for module *abc.* Each entry of the enumerated type should defined using the same conventions as a constant.

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

tions 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 the middle words are typically nouns to represent the object(s) on which the verbs act.

This convention makes it obvious that *xyzFile* is a substructure of the *xyz* module. In the second case, it is not at all obvious. Furthermore, if the module *xyz* grows and the designer decides to further decompose it, then it is easy to move the entire *xyzFile* sub-structure and corresponding functions to a separate module, say it is called *xyzfile*. Then a global search and replace of *xyzFile* to *xyzfile* would result in all the necessary changes, and within a few minutes, the decomposition is complete. If this naming convention is not followed, it would take much longer to revise all of the names for use in the new module.

While it is acceptable to have a short cryptic module name because the name serves as a prefix to everything, only use obvious abbreviations for function names. If an obvious abbreviation is not available, then use the full name. If an abbreviation is used, then use it *everywhere* for the project.

For example, it can be a convention to always use *xyzInit* as the initialization code for module *xyz*; and never to use *xyzInitialize*. As another example, use either *snd* and *rcv*, or *send* and *receive*, but don't intermix the two. Examples of other common abbreviations include *intr* for interrupt, *fwd* for forward, *rev* for reverse, *sync* for synchronization, *stat* for status, *ctrl* for control. On the other hand, an abbreviation like *trfm*, supposedly short for transform, is not recommended, because the abbreviation is not obvious and thus readability is compromised. In such a case, the function name without abbreviation, *xyzTransform()*, would be a better choice. As another example of over-using abbreviations, consider the difference between *xyzFileCreate()* and *xyzFilCrt()*. The second one uses uncommon abbreviations, and difficult to follow when reviewing the code during the software maintenance phase. It is much better to use the slightly longer names, and to not have the confusion as to what the function does.

#1 *No measurements of execution time*

Many programmers are designing real-time systems, yet they have no idea of the execution time of any part of their code. For example, we were asked to help a company identify occasionally erratic behavior in their system. From our experience, this 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 was easy for them to generate, they had it readily available. But they had no measured execution times, 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. It was quickly discovered that the cause of the erratic behavior was that the system was overloaded. The company says "yeah, well we know that." But what they were surprised to hear was that the idle process was executing over 20% of the time. (When measuring everything, this includes the idle task). The problem was that their execution time estimates were all wrong. Even more so, one interrupt handler, with estimated execution time of a few hundred microseconds, took 6 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, etc. This should be a continuous process, done as often as testing the functionality. When execution time is measured, correlate the results to the estimates; if the measured time does not 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 problems in the system. The operative word in "real-time system" is *time*!

References

[1] M. Hassani and D. Stewart, "A Mechanism for Communicating in Dynamically Reconfigurable Embedded Systems," in *Proc. of High Assurance Systems Engineering Workshop*, Washington DC., August 1997.

[2] B. L. Jacob, "Cache design for embedded real-time systems." *Embedded Systems Conference Summer*, Danvers MA, June 1999.

[3] S. Shlaer and S. J. Mellor, "Recursive design of an application-independent architecture," IEEE Software, v.14, n.1, pp. 61–72, Jan/Feb 1997.

[4] D. Stewart, "Designing Software Components for Real-Time Applications," in *Proc. of Embedded Systems Conference*, San Jose, CA, September 1999.

[5] 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, Dec. 1997.

[6] 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, Jan. 1983.