More Pitfalls for Real-Time Software Developers

The countdown continues, and the problems get worse. This month we present the top 15 problems real-time software developers face.

Last month, I presented the first half of the 30 most common mistakes made by real-time software developers. As I continue this month, I'll state the misconception or source for each problem. In addition, I'll provide 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, a library or Web search should yield additional literature on the topic.

While there is usually agreement about most of these items being mistakes, some of the mistakes listed and the corresponding proposed solutions may be controversial. In such cases, simply highlighting this disagreement can encourage designers to compare their methods to other approaches, and to reconsider whether their methods are provably better.

I'll now present the top 15 problems. Problems that are higher on the list (where #15 is lowest and #1 is highest) are either more common and/or have more impact on quality, development time, and software maintenance. Naturally, the order is based on my opinion, and it isn't so important that one mistake is listed higher on the list than another. What is most important is that any item on the list may be significant in your specific environment.

#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. 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.

Most software systems are designed such that the entire system is defined by a single diagram (or, even worse, none!).

Unfortunately, this misconception often leads to the downfall of projects or quality, 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. Explain to such eager people 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.

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.

#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—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- 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 relationships 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 this 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`.

#13 No legend on design diagrams

Even when someone has provided design diagrams, they often haven't 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

BEN FISHMAN

FIGURE 1 Many software developers will define data structures, as shown in (a). The design of the software and interrelationship of the data structures is not obvious unless the code is accompanied by a data structure diagram, as shown in (b).

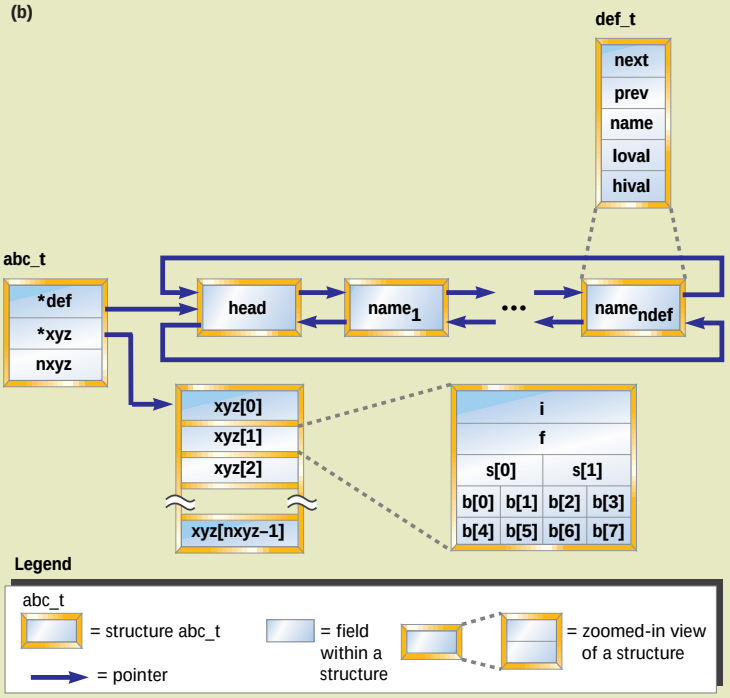
(a)

```
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 {
  long 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;
```

(b)



to a standard such as UML or based on a custom set of conventions developed by the company. What's 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 make 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.

#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 RTOSes 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/Os 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, and special purpose processors such as a DSP filtering data from a camera or microphone. Trying to adapt the POSIX device driver API 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 eight-bit digital I/O output board, without affecting the values on the other six 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, because 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 eight-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, no standards exist for the request, and every device freely selects its own set of supported requests. An example of the problems that result is with setting a serial port to 9,600 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.

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 nonportable, usually difficult to maintain, and cannot be used effectively in a reconfigurable environment.

The alternative is to encapsulate the device driver in its own thread. The data from the device is transferred through shared memory (and not via message passing, as in mistake #16).[1] The device driver is then a single process that can be executed whenever the device is present and needed, and otherwise not executed.

#11 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 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. 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 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.

#9 Configuration information in `#define` ***statements***

Embedded programmers continually use `#define` statements in their C code to specify register addresses, limits for arrays, and configuration constants. Although this practice is common, it is undesirable because 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, pinpointing a single location to make the change isn't easy.

As an example of an emergency patch, suppose a client discovers that for an application, a hard-coded 64ms timeout period is insufficient, and it needs to be changed to 256ms. If `#define`s were used, the entire application would have to be recompiled, or every instance of that value being used in the machine language code patched.

On the other hand, if this information is stored in a configuration variable (possibly stored in nonvolatile memory), then changing the value in just one place is simple. The code does not have to be recompiled—at worst, resetting or rebooting the system is necessary. The need for recompilation

prevents in-the-field updates, as users generally don't 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 address of each register `#defined`. That same code can't 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 eight-bit status port, an eight-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. For example:

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

#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 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.

#7 `#include` "globals.h"

A single `#included` file with all of the system's constants, variable definitions, type definitions, and/or function prototypes is a sure sign of nonreusable 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 (see mistake #18)[2] 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 along with mistake #2.

#6 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, "What if there is no success… then what is returned?" He looks at his code and might realize that the lack of success scenario 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, 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 also 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.

#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 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 the wrong context and for the use of global variables to pass data to the processes
- Handlers are difficult to debug and analyze because few debuggers allow the setting of breakpoints within an interrupt 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.

#3 Using 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 RTOSes, 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.

#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 #5, 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. 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.

Following is an excerpt of the naming conventions that are enforced in the Software Engineering for Real-Time Systems (SERTS) Laboratory at the 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 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 writ-

ten and distributed to all developers, and they are strictly adhered to by all programmers.

SERTS naming conventions. Simply creating modules to achieve a higher level of software maintainability is not sufficient. One of the biggest costs in maintaining software is spending time trying to figure out what some other programmer did in his or her code. To alleviate this problem, an entire organization must adhere to strict style and naming conventions. Table 1 gives one such set of naming conventions that have already proven themselves to work well.

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 order of "big to small" for compounded function names, and not in the order 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) on which the verbs act.

This convention makes it obvious that `xyzFile` is a sub-structure 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` substructure and corresponding functions 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 isn't followed, revising all of the names for use in the new module would take much longer.

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, you might always use `xyzInit` as the initialization code for module `xyz`, rather than `xyzInitialize`. Or you might use either `snd` and `rcv`, or `send` and `receive`, but you shouldn'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 abbre-

TABLE 1 SERTS naming conventions to improve software maintainability for C-language programs

Symbol	Description
xyc.c	File that contains code for module 'xyz'
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.
xyz_t	Primary data type for module xyz. Defined in xyz.h
xyzAbcde_t	Secondary type "Abcde" for module xyz. Defined in xyz.h.
xyzAbcde()	Function "Abcde" that applies to items of type xyz_t.
XYZ_ABCDE	Constant for module XYZ. Must be defined in xyz.h.
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 variables should be avoided!
_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"
abcde	Local variable. Must be defined inside a function.Fields within a structure are also defined using this convention.
_abcde	Internal global variable. Must be defined as "static" at top of xyz.c.
_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.
_abcde_t	Internally-defined type. Must be defined at top of xyz.c.
abc_e	An exported enumerated type for module abc. Each entry of the enumerated type should defined using the same conventions as a constant.

viations 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 example, 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.

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

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*.

Evaluate your methods

Over two issues I have presented the 30 most common problems in real-time software development. 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. **esp**

Dr. David B. Stewart is an assistant professor at the University of Maryland. He earned his PhD in computer engineering from Carnegie Mellon University. He now teaches and consults in the area of real-time embedded software. He has led large student projects, including the Pinball Machine Project that was demonstrated in Las Vegas, and the Computer-Controlled Electric Train project that received an honorable mention in the 1998 Motorola University Design Contest. His research focuses on next generation RTOS and middleware technology to support the rapid design and analysis of dynamically reconfigurable real-time software. You can reach him by referencing his home page at www.ece.umd.edu/~dstewart.

References

1. From Part 1, October 1999, p. 40: *#16 Using message passing as primary inter-process communication*
2. From Part 1, October 1999, p. 38: *#18 Too many inter-module dependencies*