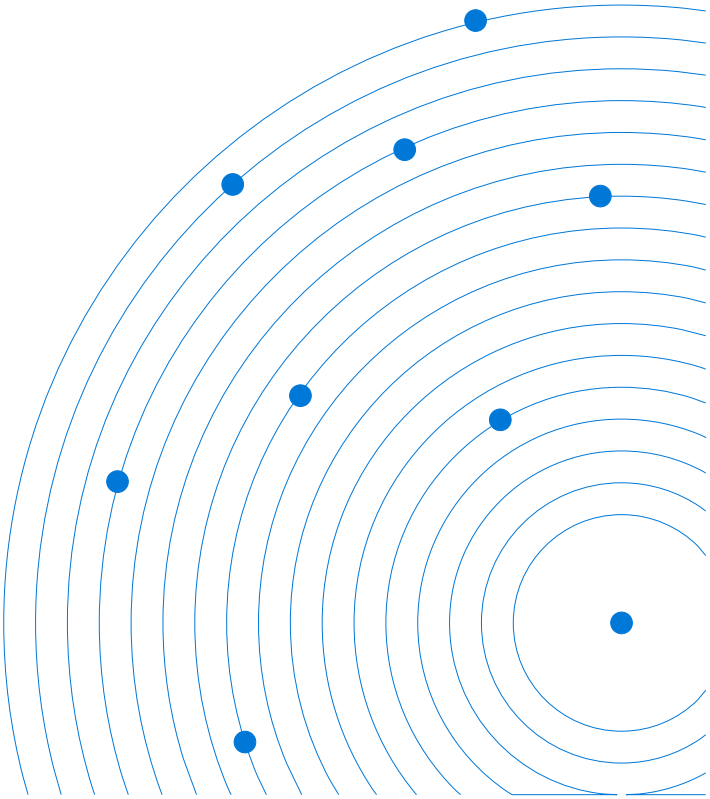
Consider the Source

Source code quality affects its usefulness and should be considered when evaluating an RTOS

January 2020



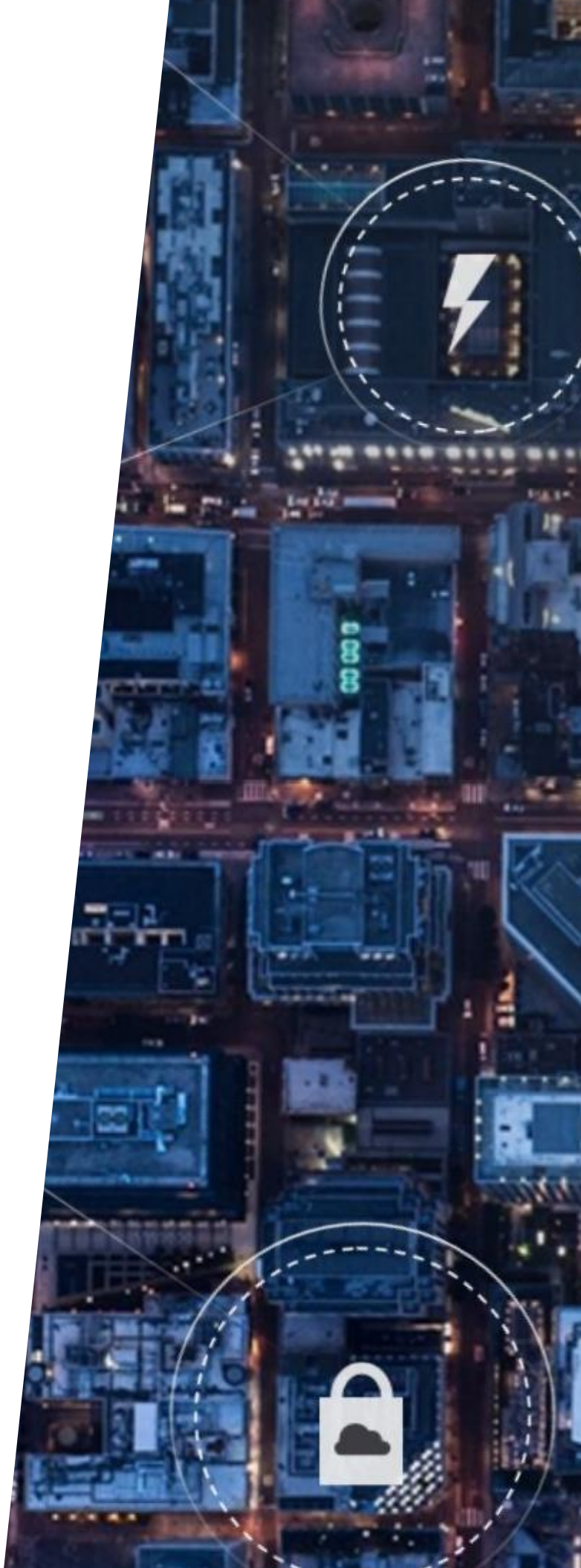
Microsoft

Contents

•

Introduction

Most embedded system developers recognize the value of having source code for the RTOS they use, but exactly why is having source code so important? And, is simply "having" the source code sufficient? Are there qualitative elements or attributes of source code, beyond its simple existence, that make it more or less valuable? These are serious questions that should be considered by embedded software developers when considering an RTOS. To answer these questions, we have to consider RTOS source code both externally and internally. An external examination focuses on the RTOS's Application Programming Interface, commonly referred to as the "API." This is the portion of the RTOS code that the application programmer uses to take advantage of RTOS services. The API is very important and can play a huge role in the RTOS's ease of use, and consequently, in the success of a project that is using the RTOS. Looking internally, developers must examine the implementation code for the RTOS's functions and assess that code against criteria that are meaningful for the success of the project.



Developers Care About Source Code

Each year, UBM conducts a survey of embedded developers and shares its findings with the industry. Various topics are covered, organizational, job title, hardware, software, and more. One of the questions UBM asks each year is "What criteria are most important in selecting an RTOS?" Each year, the answers cover the spectrum of RTOS features, but consistently, the "availability of source code" ranks high on the list. In the most recent survey, released in February 2013, the availability of source code ranked #1 among all criteria for selecting an RTOS, as shown in Figure-1.

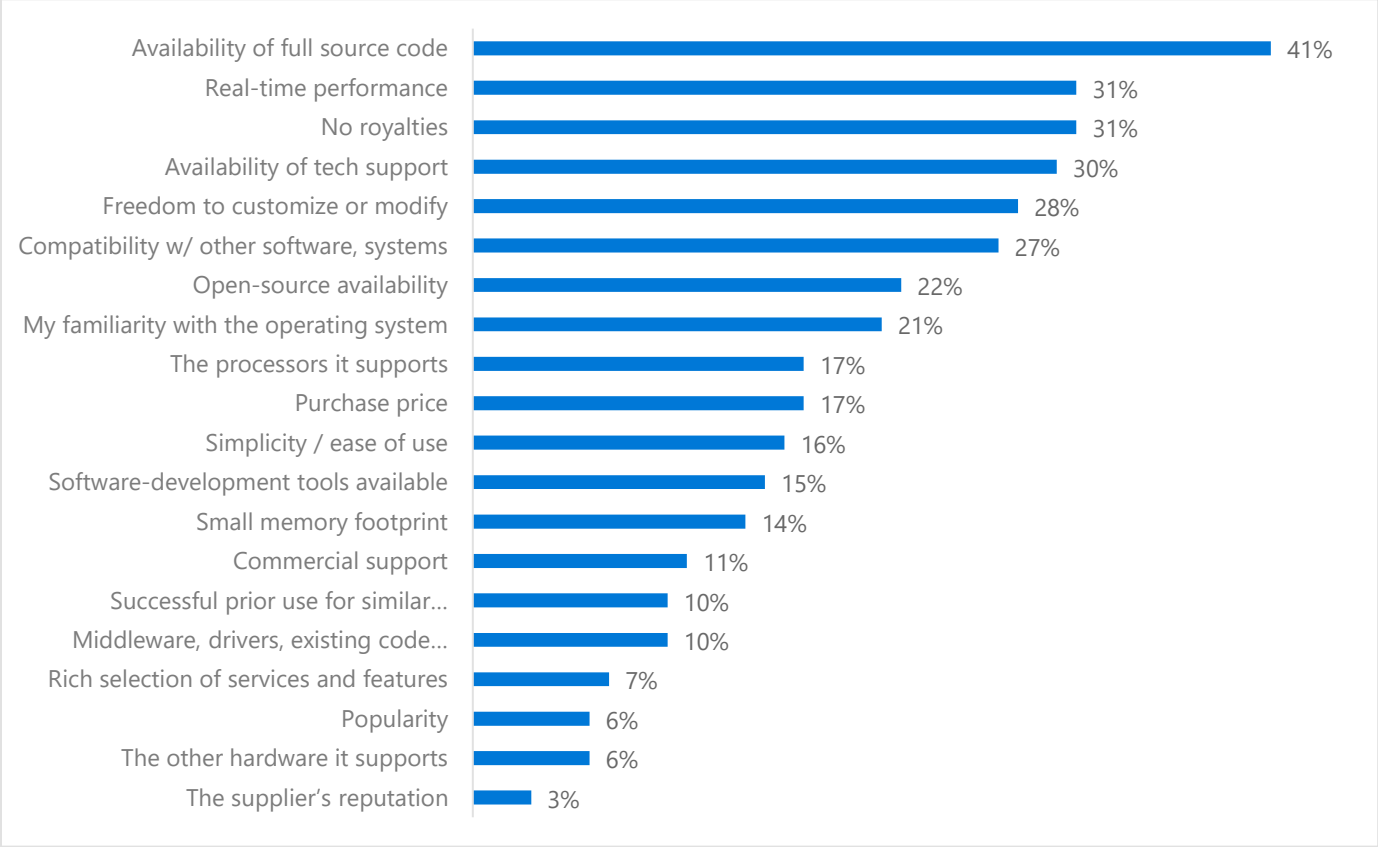


Figure 1 UBM Survey of Embedded Developers, February 2013

As shown in Figure-1, the availability of source code wasn't just #1, it was #1 by a wide margin over the next most highly valued RTOS characteristic. This underscores just how strongly developers feel that source code is important. But why is RTOS source code so important? There are several reasons why developers value having source code, and each of them is important:

1. **Source code helps developers *understand* exactly how the RTOS performs its services.** Developers using the RTOS can examine the source code and see every step as the RTOS performs a given function. Such examination often reveals subtleties not described in the User Guide, or other documentation. Without the source, developers are blind to the implementation details, and totally dependent on the User Guide, which might not provide all the information the developer would like to have. Beyond just reading and understanding the code, the actual C instructions used by the RTOS enable single stepping in C, while retaining the option to step through the machine code if desired. When debugging, stepping-into an RTOS function can explain how a function came up with its final result. Without source code single- stepping is at the machine instruction level - often too low a level

for the intent of the debugging. Source code can make debugging far more direct, and can isolate a problem to a specific instruction, which would be much more difficult without access to the source.

2. **Source code enables developers to *build* the RTOS using various compile-time options.**
   All RTOSes have to be compiled and linked with application code to produce an executable image of the application. Without source code, an RTOS must be used in a "pre-built" mode, as provided by the vendor. This means that all compile-time options must be specified by the vendor, and the RTOS that is produced will have these settings and no others. With source code, developers can build the RTOS themselves, and make their own decisions regarding how they select from all available options. There are many compile-time options found in most RTOSes, including:

   - Compiler Optimization options for size and speed
   - #define settings to enable or disable certain features
   - Symbols, and other debug Information
   - Error-Checking activation
   - Event Trace activation
   - Statistics Collection activation

3. **Source code provides *security* in the event the supplier is unable to support the product**
   Developers protect their ability to provide their customers with product support regardless of the RTOS supplier's help or lack of help. Developers become self-sufficient, and able to fix bugs, make upgrades, and port to new processors themselves. While most developers would find these responsibilities unattractive, it's far less attractive to try to maintain an RTOS without source code!

4. **Source code enables developers to *customize* the RTOS to meet their needs**
   While modifying the code of an RTOS generally is not recommended, there are situations where it can be very helpful. For instance, it may be necessary to achieve compatibility with previously developed application code, or to remove unneeded functionality, or perhaps to add proprietary features/technology.

5. **Source code enables developers to get their safety-related products *certified***
   Most safety-related certifications require submission of complete source code for the product's software, including the RTOS if one is used. Unless the RTOS vendor can provide certification assurance, source code is essential to achieving safety-related certification.

When considering each of these reasons why source code availability is so important to embedded developers, it's clear that each benefit can be achieved to varying degrees. What the source code actually, looks like can make each of the 5 benefits mentioned above either easier or harder to achieve. With these benefits in mind, let's see what characteristics of source code would make that code better able to achieve those benefits.

Characteristics of Source Code

First of all, it's clear that all source code is not alike. Anyone who has ever looked at code from various sources, either co-workers, consultants, vendors, or open source, would notice the differences among various bodies of code, and would likely find some "better" or "cleaner" than others. Generally, "clean" code reduces the total cost of ownership of the code, whether as an author or user. "Bad" code only gets worse, becomes difficult to maintain, and ultimately begs to be re-designed and "cleaned-up."

It's not enough to just demand the availability of source code, it's best to assess the quality of the code – and to look for code that will best help achieve the 5 benefits we're looking for. Dave Thomas, founder of OTI, godfather of the Eclipse strategy, uses "clean" in the more general sense of "high quality code" and says:

> *Clean code[1] can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.*[1]

Some developers prefer to use "Clean" in a narrower, appearance sense, and "Good" in the more general sense. I tend to agree with that, but the difference is semantic and largely irrelevant. Both terms refer to the quality and value of source code, and they can be used pretty interchangeably.

"Good" source code exhibits several characteristics, that make it "better" than "bad" code. These are some things to look for in source code, that represent qualitative differences in the code's ability to achieve desired benefits:

- Clean
- Clear
- Commented
- Consistent
- Correct

Clean

In the narrower sense - code should be neatly formatted, evenly spaced, for best readability. Use blank lines to separate different sections of code. Surround operators with spaces. Indent to clearly show hierarchy of code and color-code various items (auto-formatting editors handle this nicely).

```
int main(void)
{
 /* Initialize the Demo */
  Demo_Init();

  while (1)
  {
    /* If SEL pushbutton is pressed */
    if(SELStatus == 1)
    {
      /* External Interrupt Disable */
      IntExtOnOffConfig(DISABLE);

      /* Execute Sel Function */
      SelFunc();

      /* External Interrupt Enable */
      IntExtOnOffConfig(ENABLE);
      /* Reset SELStatus value */
      SELStatus = 0;
    }
  }
}
```

Figure 2 Example of "Clean" Code

[1] Clean Code, a Handbook of Agile Software Craftsmanship, Prentice Hall, © 2009

Clear

Code should be easily readable, and easily understood by a reviewer who didn't write it, but who must examine it and/or support it. Use intention-revealing names for functions and variables. A long, descriptive name is better than a long descriptive comment, or a short vague name. Use pronounceable names. Use underscore to separate words in a name. Use consistent nouns and verbs to describe the same thing in different routine names. (eg: thread). Use full words, don't abbreviate. See the abbreviated names highlighted in Figure-3. Use searchable names (eg: tx_Object_Operation). Avoid being "cute." You might save a few cycles, but unless they're more important than maintaining the code, stick to the basics.

```
int main(void)
{
 /* Initialize the Demo */
  Demo_Init();

  while (1)
  {
    /* If SEL pushbutton is pressed */
    if(SELStatus == 1)
    {
      /* External Interrupt Disable */
      IntExtOnOffConfig(DISABLE);

      /* Execute Sel Function */
      SelFunc();

      /* External Interrupt Enable */
      IntExtOnOffConfig(ENABLE);
      /* Reset SELStatus value */
      SELStatus = 0;
    }
  }
}
```

Figure 3 Example of Unclear Code - Names

Commented

Explain, in simple English (or the language of the reader), what each line of code is intended to do. Comments can explain intent, clarify an operation, warn of consequences. Avoid comments that are mumbling, redundant, misleading, or stale, are used instead of a clear variable or function name. Don't disable code. Use SCCS instead to retain old code.

Dave Thomas says comments are a failure to use descriptive names in code. We disagree; comments should be at a higher level than the code. Comments should help explain the code-one comment for each line of C code.

Comments should not simply describe what the code does. For example:

```
/* Set detect flag to 1. */ detect_flag
= 1;
```

The above comment doesn't mean anything more than the actual code, so it is a bad or worthless comment. Better this:

```
/* We found the file so set the detect flag to indicate that. */ detect_flag =
1;
```

This comment describes why the code does what it does. Comments complement the documentation, at the lowest possible level.

Consistent

Code should use consistent terminology, style, structure, and formatting, to make it more easily readable and understood. Multiple sections of code, each perhaps "good" in its own right, might be difficult to understand when combined. Consistency makes the learning experience at least singular. Use consistent naming, formatting, commenting, headers, algorithms.

Example - Consistent Names
Use the same verb for the same action for different objects:

tx_thread_**create**
tx_semaphore_**create** tx_queue_**create**

Use the same noun for various actions for the same object:

tx_**thread**_sleep
tx_**thread**_relinquish
tx_**thread**_suspend
tx_**thread**_priority_change

Correct

It almost goes without saying – almost! The code must work under all system conditions. It must match the object code. It must be able to be compiled and produce the exact same binary. Of course, when using the same compiler, and options. Question: "Better ugly code that works or clean code that

doesn't?" Clean code can be fixed, then you have working clean code. Ugly code cannot as easily be made clean ….. up to a point of course

The RTOS API

The RTOS API is the Application Programming Interface. It's the part of the RTOS that developers actually touch every time they use the RTOS. Generally, it's the function prototypes for a set of C- callable functions, with parameters. The API is critical to the correct and easy use of the RTOS. The API must be well-designed and implemented to be:

- Intuitive
- Understandable
- Well Documented
- Consistent
- Efficient
- Platform-Independent

Intuitive

Function names should be easily recognizable, using full words, not abbreviations:

Eg: tx_queue_send Rather
than: tx_qsnd

Parameter names should be meaningful:

Eg: tx_queue_performance_messages_sent_count Rather
than: tx_QueueCnt

The goal is to be understandable, to convey meaning, and avoid the need to go to the User Guide. Intuitive naming makes it easier to write application code that uses the RTOS. It also makes it easier to understand code that someone else will have to read.

Understandable

Function names and parameters should be easily recognized and understood. They should reflect their role in the function. Developers should avoid cryptic abbreviations or "cute" names that do are not quickly understandable without reference to the User Guide:

Eg: tx_performance_preemptions_count Rather
than: tx_preemptions, or tx_perf_cnt

Constants should describe their meaning, not simply reflect their value Eg:

TX_WAIT_FOREVER
Rather than: 0xffffffff

Again, this minimizes the need to consult the User Guide.

Well Documented

Rather than being an afterthought, the User Guide should be written first, as the definition of the functions and the API. This helps the RTOS and its API to achieve a user point of view, and all desired functionality. It also helps minimize deviations from intended look and feel, not only in the User Guide, but also in the code itself. The documentation should include the User Guide, and also any relevant Release Notes for particular versions of the RTOS. The User Guide should include:

- Function Name
- Description
- Examples showing application code use of each function

Consistent

All APIs should follow the same structure.

Eg: tx_queue_send

- "tx" identifies the function as one from the RTOS, to distinguish it from application functions;
- "queue" specifies the RTOS object being controlled;
- "send" specifies the action being performed on the object;
- Underscores separate elements for better readability.

This RTOS_NOUN_VERB design enables alphabetical grouping of RTOS functions apart from application functions, groups all services by object, making them easy to find in User Guide, and makes understanding new functions more intuitive. Consistency must be consistent - in other words, a RTOS_NOUN_VERB structure should be used in ALL functions, for ALL RTOS components (Kernel, Network Stack, USB, File System, GUI, etc.). Otherwise, using multiple components together will be confusing, if they have been written with different structure and style.

Efficient

One API that offers several modes of operation, based on the parameters, rather than multiple APIs for these variations, reduces the number of functions needed, and eliminates duplication of common code for multiple functions. The API should enable common operations to be performed with a single call, rather than requiring a combination. For example,

/* Send message to queue 0. */

status = tx_queue_send(&tx_queue_0, &thread_1_message, TX_WAIT_FOREVER);

This function call specifies that the RTOS should suspend the calling thread if no elements are available for it in the referenced queue. Moreover, the parameter "TX_WAIT_FOREVER" indicates (intuitively)

that the suspension should be as long as necessary, with no time limit ("forever"). The same function can be called with a maximum wait time that is finite, simply by specifying that wait time as the last parameter. Some RTOSes might use multiple function names, depending on the action to be taken if the queue is empty.

Platform-Independent

The API should not require change when the application changes target platform. Nothing platform-dependent should enter into the API. Platform dependencies should be isolated to separate modules, irrelevant to the API. Also, compiler-independent. Avoid compiler special features, unless worth the trouble. Avoid in-line assembly and machine-dependent types

Conclusion

RTOS source code is valuable, developers cite it as the single most important criteria in the selection of an RTOS. But, all source code is not of equal benefit to a user. If the source code is essential, it makes sense that it's expected to provide benefits, such as those we've outlined, and that not all code will achieve those benefits to the same degree. In particular, the RTOS API is critical, and can aid ease of use, even apart from the benefits of the internal, implementation code. Availability of source code is not all a developer should look for. Make sure the code will provide the benefits for which it is justifiably considered so critical. This requires examining it and comparing code from the RTOSes the developer is considering using. Consider the source!