Consider The Source

John A. Carbone

Express Logic, Inc.

expresslogic

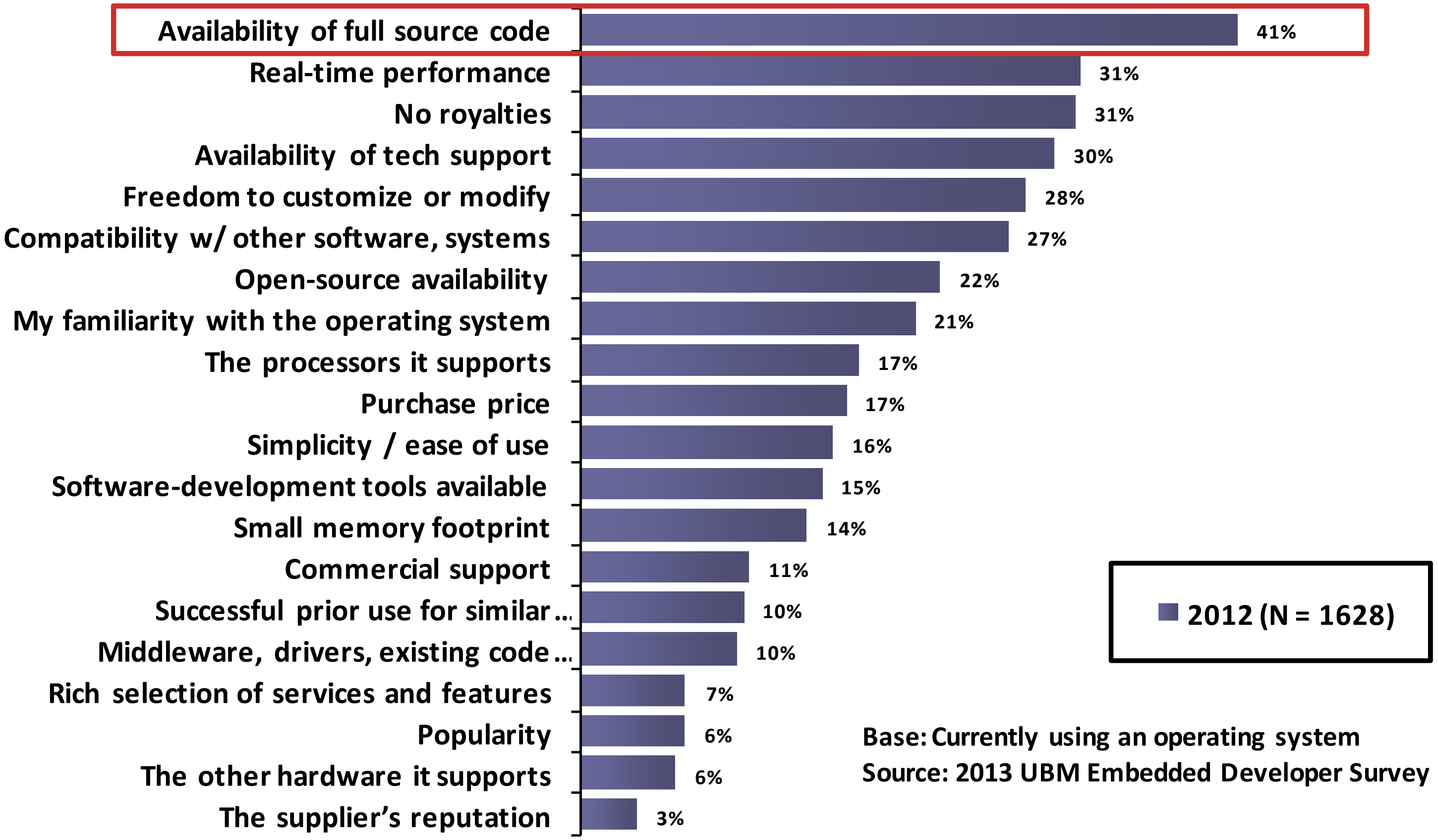
ARM 2013 TechCon
Where Intelligence Connects

Outline

- How do embedded developers value RTOS source code?
- Why is RTOS source code important?
- What constitutes Good Code?
- What makes a good RTOS API?
- Conclusions

Developers Value RTOS Source Code

- In 2012, what were the most important factors in choosing an operating system.



Base: Currently using an operating system
Source: 2013 UBM Embedded Developer Survey

Why is RTOS source code so important?

- Helps developers **understand** exactly how the RTOS performs a given service
  - Developers using the RTOS can see every step as the RTOS performs a given function, and often reveals subtleties not described elsewhere.
  - 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, no symbols may be available, and single-stepping is at the machine instruction level - often too low a level for the intent of the debugging.

Why is RTOS source code so important?

- Enables developers to **build** the RTOS using various compile-time options and optimizations, as befit the intended use at that point in the development process;
  - Examples include:
    - Optimizing for speed
    - Optimizing for code size
    - Varying optimizations by routine
    - Enabling full debug symbols
    - Run-time error checking
    - Run-time trace, etc.

Why is RTOS source code so important?

- 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
    - Bug Fixes
    - Upgrades
    - Ports to new processors

Why is RTOS source code so important?

- Enables developers to **customize** the RTOS to meet their needs
  - Perhaps for compatibility with previously developed application code
  - To remove unneeded functionality
  - To add proprietary features/technology

Why is RTOS source code so important?

- Enables developers to get their safety-related products **certified**
  - Providing source code
  - Providing unit test results
  - Providing detailed documentation
  - Correcting shortcomings

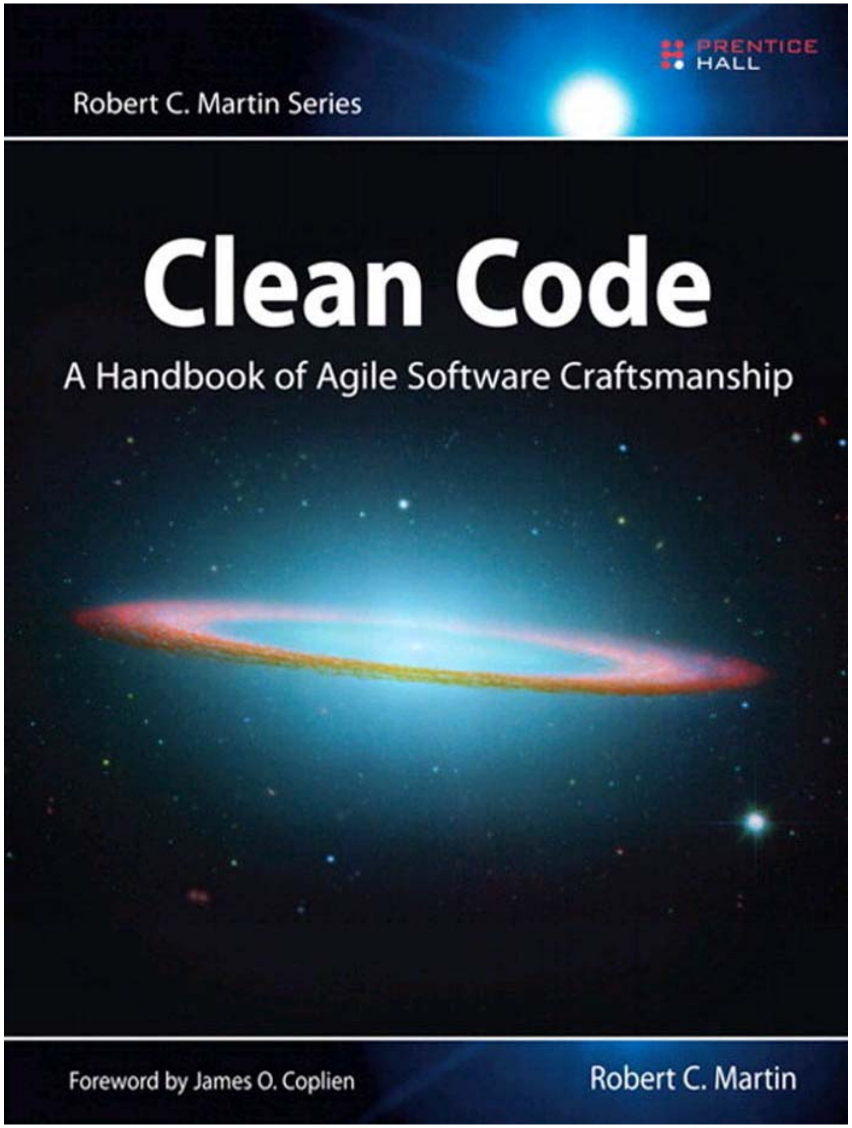
Good Code vs Bad Code

- All source code is not alike
- “Good” 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 demand source code, best to assess the quality of the code – “Consider the Source”

Good Code or Clean Code?

- Symantics
- Dave Thomas, founder of OTI, godfather of the Eclipse strategy, uses “clean” in the more general sense of “high quality code”
  - *Clean code 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.*
- We use “Good” in the more general sense, and “Clean” in a narrower, appearance sense

Dave Thomas on Clean Code


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Characteristics of Good Code

- “Good” source code exhibits several characteristics, that make it “better” than “bad” code. These are some things to look for in source code, and things that represent qualitative differences in code:
  - Clean
  - Clear
  - Commented
  - Consistent
  - Correct

Clean

- Clean - In the narrower sense - It 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; Indent consistently
- Color-code various items
- Auto-formatting editors handle this nicely

Example of “Clean” Code

```
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;
    }
  }
}
```

Clear

- Clear- It should be easily readable, and easily understood by the 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.
  - 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.

Example of Unclear Code - Names

```c
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;
    }
  }
}
```

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.
  - 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 their own right, might be difficult to understand when combined
- Consistency makes the learning experience at least singular
- Consistent naming, formatting, commenting, headers, algorithms

Example - Consistent Names

- Use same verb for same action:
  - tx_thread_**create**
  - tx_semaphore_**create**
  - tx_queue_**create**
- Use Same Noun for same object:
  - tx_**thread**_sleep
  - tx_**thread**_relinquish
  - tx_**thread**_suspend
  - tx_**thread**_priority_change

Correct

- Almost goes without saying – almost!
- The code must work under all system conditions
- It must match the object code
  - 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 API is the Application Programming Interface
  - It’s the part of the RTOS that developers actually touch every time they use the RTOS
  - Generally, a set of C-callable functions, with parameters
- The API can make the RTOS easy to use, if it is:
  - Intuitive
  - Understandable
  - Well Documented
  - Consistent
  - Efficient
  - Platform-Independent

Intuitive

- Function names should be easily recognizable
  - Eg: tx_queue_send
  - Should be full words, not abbreviations
  - Eg: tx_qsnd
- Parameter names should be meaningful
  - Eg: tx_queue_performance_messages_sent_count
  - Not: tx_QueueCount
- The goal is to be understandable, without having to go to the User Guide
- Makes it easier to write code that uses RTOS
- Makes it easier to understand code that uses RTOS

Understandable

- Function names and parameters should convey meaning, and reflect their role in the function
  - No cryptic abbreviations or “cute” names
  - Eg: tx_performance_preemptions_count
  - Rather than: tx_preemptions, or tx_perf_cnt
- Constants should describe their meaning, not their value
  - Eg: TX_WAIT_FOREVER = 0xffffffff
  - Used to control pending
  - Other options might be “TX_NO_WAIT”, etc…
- Minimize need to consult User Guide

Well Documented

- User Guide should be written first, as the definition of the functions and the API.
  - This helps achieve a user point of view
  - It also helps achieve 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
  - Header description
  - Code comments
- Examples showing actual application code using each API

Consistent

- All APIs should follow the same structure
- Eg: “tx_queue_send (…..)”
  - Where “tx” identifies the RTOS
  - Underscores separate elements for better readability
  - “queue” specifies the RTOS object being controlled, or the noun representing what is being referenced
  - “send” specifies the action being performed with the object – the verb
- Enables alphabetical grouping of RTOS functions apart from application functions
  - Groups all services for each object, making them easy to find in User Guide
- Makes understanding new functions more intuitive

Efficient

- One API that offers several modes of operation, based on the parameters, rather than multiple APIs for these variations

- The API should enable common operations to be performed with a single call, rather than requiring a combination

```
/* Send message to queue 0.  */

status =  tx_queue_send(&tx_obj.queue_0,
          &thread_1_messages_sent, TX_WAIT_FOREVER);
```

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
  - Avoid machine-dependent types

Conclusion

- RTOS source code is valuable
- All source code is not of equal benefit to a user
- The RTOS API is critical, and can aid ease of use
- Availability of source code is not all a developer should look for
- Consider the source!

John Carbone

VP, Marketing,

Express Logic, Inc.

jcarbone@expresslogic.com

expresslogic

ARM 2013 TechCon
Where Intelligence Connects

ARM® 2013 TechCon™

Where Intelligence Connects

UBM Tech