C and C++ Embedded Software Nuggets- April 2018

Matthew Eshleman
covemountainsoftware.com

Background - Matthew Eshleman

- 20+ years of embedded software engineering, software architecture, and project planning

- Independent consultant and contractor focused on embedded software and firmware development, design, architecture, and analysis.

- Learn more: http://covemountainsoftware.com/consulting

Agenda

- Share and discuss as many C and C++ “nuggets” of embedded software as we can in about 1 hour.

volatile qualifier (C or C++)

- Writing firmware for a microcontroller? Better understand the use of ‘volatile’!
- volatile specifies a variable whose value may be changed by some process outside the current program.
  - Therefore the compiler must avoid optimizations for any variable qualified with ‘volatile’
- i.e: “Dear compiler: please do not make any assumptions about this variable. Please!”
- Declare any memory mapped hardware registers as volatile
- **NOTE**: many custom or older microcontroller specific C compilers may expect volatile to be used for ISR/thread safe variable updates. This is strictly speaking wrong. But might be correct on your compiler. Know your compiler!
- example:

```
volatile uint32_t a_volatile_var;
volatile uint32_t * ptr_to_volatile = &a_volatile_var;
```

Accessing HW register: https://godbolt.org/g/XkRiXC

atomic data types (C11 or C++11)

- “atomic … provides components for fine-grained atomic operations allowing for lockless concurrent programming. Each atomic operation is indivisible with regards to any other atomic operation that involves the same object. Atomic objects are free of data races.”

- Enforces “memory barriers”

- Available in newer compilers: C11 or C++11

- This is the data type to use if a variable is accessed in multiple threads or interrupt service routines! (especially for system with multiple cores or internal processors)

```c
//C11 example
#include <stdatomic.h>

atomic_int a_atomic_int;
```

```cpp
//C++11 example
#include <cstdint>
#include <atomic>

std::atomic<uint32_t> some_atomic_variable;
```

atomic example with ISR/bare metal

https://godbolt.org/g/EzSgZQ

volatile and std::atomic examples: godbolt!

- Using the online ‘godbolt’ compiler explorer, we can see what an ARM compiler does with volatile and/or atomic types, all in one example

- https://godbolt.org/g/BknDt6

memory mapped registers (C or C++)

- Nearly all embedded devices will have some set of memory mapped ‘registers’ which control some portion of the microcontroller or hardware.

- For example (STM32F405, random number generator):

24.4.4 RNG register map

Table 116 gives the RNG register map and reset values.

Table 116. RNG register map and reset map

Offset	Register name reset value	Register size																													
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	RNG_CR 0x0000000	Reserved																												IE	RNGEN	Reserved	
0x04	RNG_SR 0x0000000	Reserved																									SEIS	CEIS	Reserved		SECS	CECS	DRDY
0x08	RNG_DR 0x0000000	RNDATA[31:0]																															

map the registers to a data struct

```cpp
#include <cstdint>

struct StmRngRegisters
{
    volatile uint32_t CR;
    volatile uint32_t SR;
    volatile uint32_t DR;
};

static volatile StmRngRegisters * m_RNG =
                    (StmRngRegisters*)0x50060800;

uint32_t GetRandomData()
{
    return m_RNG->DR;
}
```

24.4.4 RNG register map

Table 116 gives the RNG register map and reset values.

Table 116. RNG register map and reset map

Offset	Register name reset value	Register size																													
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	RNG_CR 0x0000000	Reserved																												IE	RNGEN	Reserved	
0x04	RNG_SR 0x0000000	Reserved																									SEIS	CEIS	Reserved		SECS	CECS	DRDY
0x08	RNG_DR 0x0000000	RNDATA[31:0]																															

https://godbolt.org/g/32YfTc

static assert (C or C++)

- A static assert is an assert that occurs at “compile time” i.e. when actually building the executable.

- I **love** static asserts: because I love catching myself making mistakes before loading and running the firmware. Saves time. Prevents confusion.

- C11 and C++11 make this easier with standardized static assert methods.

- Your future self or a future maintainer of the code will love you the first time a static assert saves them from injecting an error into the code base.

static assert examples

- catch a change in an enum
- ensure the compiler is packing a struct as expected
- catch changes in third party data structures that impact your code
- confirm a lookup table has all the necessary elements

static assert examples

https://godbolt.org/g/oUmJvF

Tip: Avoid malloc/new

- Most embedded systems have little-to-no RAM
- Some have zero heap (‘heap’ is the section of memory where ‘malloc’ and ‘new’ manage and allocate memory from)
- Use static allocation (i.e. allocate everything at compile time) when possible
- Why?

static allocation continued…

- Why?
  - Avoids the memory overhead of allocations (i.e. the heap overhead)
  - Avoid the CPU overhead of allocations (i.e. CPU usage managing the heap)
  - Gives precise “always right” memory allocations at build and link time. The linker will throw an error if the device RAM is full.
- When to break the rule?
  - If you need precise control over when a C++ object comes into existence
  - If the RTOS requires you to break the rule (older FreeRTOS). In this case, just allocate once. Avoid ongoing allocations and deletes.
  - Embedded Linux device with lots of RAM. Go for it! But keep an eye on `/proc/meminfo` for leaks or instrument the application for leak detection.

Tip: Let your compiler help!

- Humans are error prone. Let the compiler help!
- It may hurt at first, but just do this:
- gcc: -Wall -Werror
- And if really being aggressive: gcc: -Wextra

Tip - show the size!

- Tidbit: when the build has completed, add a makefile call to print out **size** information on your firmware.

- Keep an eye on it! If there is a large sudden increase… check the code!

```
> arm-none-eabi-size firmware.elf
   text    data     bss     dec     hex
  42784    3004    9540   45339    b11b
```

struct named member init (C99 only.)

- Sadly… not in C++.
- See this style frequently in the Linux kernel source code.
- Example:

```c
#include <stdint.h>

struct Foo {
    uint16_t header;
    uint32_t command;
    uint32_t data;
};

int main()
{
    struct Foo myFoo = {
        .header = 3,
        .command = 4,
        .data = 5
    };


    return (int)myFoo.header;
}
```

RAII: C++ only

- RAII: Resource acquisition is initialization (RAII)
- A class has a constructor and a destructor. We can use that feature creatively to do things like:
  - automatically close a file handle
  - automatically free some memory or object
  - automatically release a mutex
  - automatically measure the performance of some scope of software

RAII Example

- Create a class to measure the performance of some scope of software

- https://godbolt.org/g/9USs8P

Units are hard

- rockets explode, probes crash. millions are lost.

- https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure

- Document units and enforce them as much as possible. C++ can help.

- Small tidbit: make the units clear AND readable

- Include units in variable names: `double bandwidth_Hz;`

Units: C++11 User defined literals can help

```
#define ALERT_TONE_TRIGGER_VELOCITY    35.7632f
    // or
constexpr MetersPerSecond ALERT_TONE_TRIGGER_VELOCITY = 80_MPH;
```

```
constexpr MetersPerSecond operator"" _MPH ( long double value )
{
    return MetersPerSecond(value * 0.44704);
}
constexpr MetersPerSecond operator"" _MPH ( unsigned long long value )
{
    return MetersPerSecond(value * 0.44704);
}
```

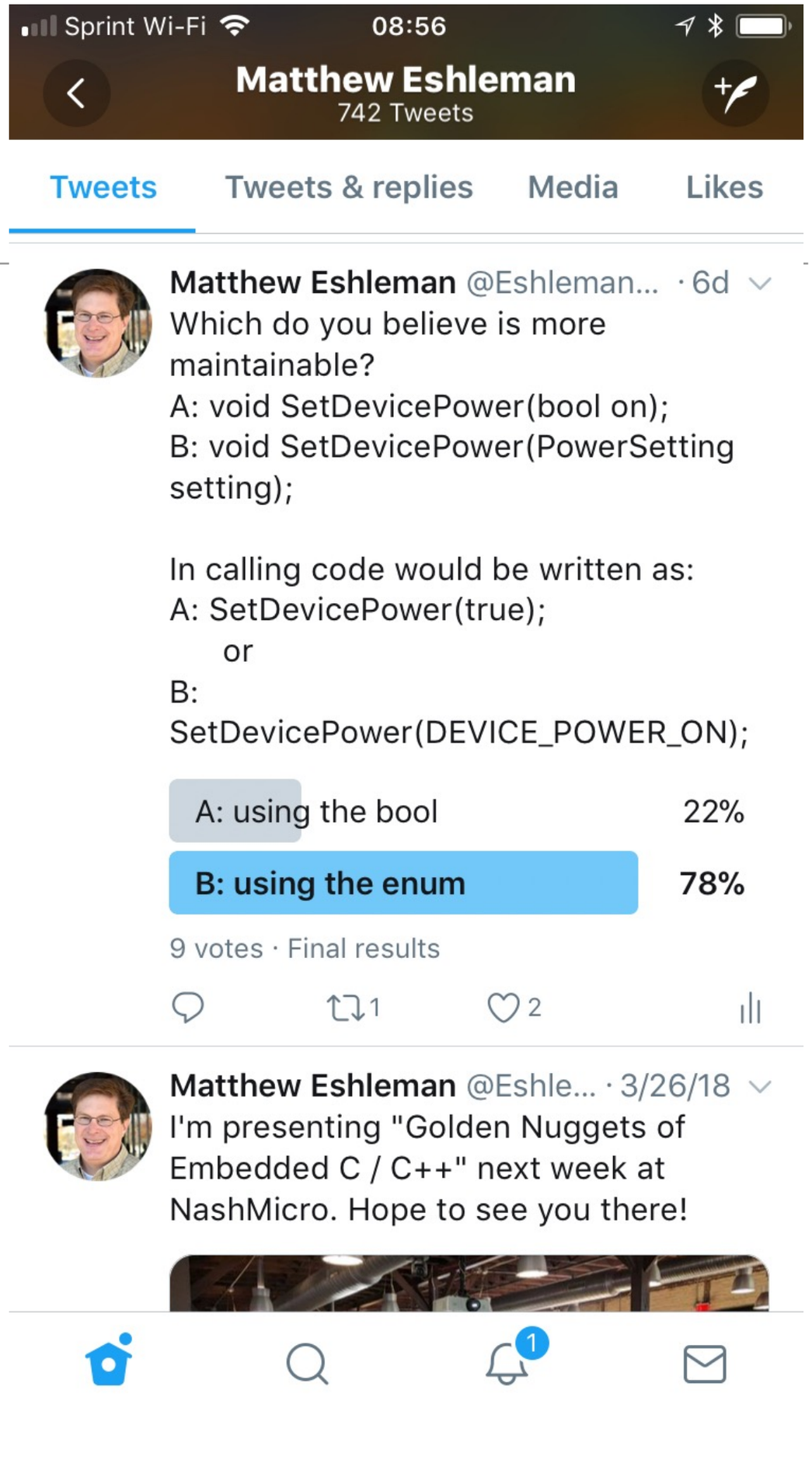
Learn more: https://www.embeddedrelated.com/showarticle/1013.php

Tip: prefer enums (C or C++)

- Lets say we need to control the power on an external device. We could create an API such as:
  - void SetDevicePower(bool on);
  - or
  - void SetDevicePower (PowerSetting setting)
  - https://godbolt.org/g/sMRv4Y

And the crowd has spoken

- Wow. 9 votes.
- I would agree: using the enum results in more readable and more **maintainable** software.



Thank you very much!

- Any Golden Nuggets from NashMicro attendees?
- Bonus material?

Thank you!

Any questions?
matthew@covemountainsoftware.com

- Resources:
  - http://icecube.wisc.edu/~dglo/c_class/const_vol.html
  - https://en.wikipedia.org/wiki/Volatile_(computer_programming)
  - https://en.wikipedia.org/wiki/Memory_barrier
  - C11: http://en.cppreference.com/w/c/atomic
  - c++11: http://en.cppreference.com/w/cpp/atomic
  - http://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf
  - https://mcuoneclipse.com/2013/04/14/text-data-and-bss-code-and-data-size-explained/
  - https://covemountainsoftware.com/2018/02/13/why-i-love-c11-for-embedded-software-and-firmware-or-c14-or/
  - https://www.embeddedrelated.com/showarticle/1013.php
  - https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization
  - https://covemountainsoftware.files.wordpress.com/2016/08/event-driven-software-and-statecharts.pdf

- Bonus Material as time allows….

Lookup tables (C or C++)

```
////////////////////////////////////////
//  Private, internal to module
typedef struct KeyBehavior
{
    Key           abstracted_key;
    uint16_t      repeat_rate_ms;
    const char * debug_key_name;
} KeyBehaviorT;

//Internal key behavior look up table
static constexpr KeyBehaviorT m_KeyBehaviors[] = {
    { Key::KEY_VOLUME_UP,   100, "VolumeUp"   },
    { Key::KEY_VOLUME_DOWN, 100, "VolumeDown" },
    { Key::KEY_SELECT,      0,   "SelectKey"  },
    { Key::KEY_UP,          250, "UpKey"      },
    { Key::KEY_DOWN,        250, "DownKey"    },
};
```

Learn more: https://www.embeddedrelated.com/showarticle/1009.php

Event Driven Software - C or C++

- I’m a big fan of event driven software
    - with an exception: true safety oriented firmware. Think brake controller for a locomotive, etc.
- Prior presentation
- The software is designed to process events in order, to completion. The architecture is “event centric.”
- No polling loops.
- No global variables/flags that can change in the middle of logic.

Event driven summary

- Event sources: buttons, network status, battery status, sensor updates, etc.

- Event framework: a way for sources to ‘inject’ or send an event. Can be just as simple as a single queue. Or as complex as a full publish/subscribe framework.

- Event receiver: the main loop or a thread that receives events via the queue and executes code based on that event.

    - Highly recommend the event receiver process events using a state machine approach.