Embedded C++14 in a "Super-loop" Firmware

Amazon Prime Air

Erik Rainey
Contacts

• Technical Questions:
  – Erik Rainey – erikrain@amazon.com

• Interested in working with us? For openings contact:
  – Matt Sigman – msigman@amazon.com
Public Information

• Drone is approximately 6ft across
• Flies upwards of 60mph
• Carries < 5 lbs package. Limited (but large) catalog of items.
• C++14 Firmware runs on the drone today
Outline

• Brief Intro to Firmware & the Environment
• Language Features for Embedded platforms
• Embedded C++14 Policies
• Class Design and Execution Model
• Communication between Devices
• Future C++17 and Reference material
Brief Intro

- This is to help level-set your expectations
- This is just what we think
What is Firmware?

- Everyone has a slightly different take
- Spectrum of complexity
- Used across broad categories
- Many caveats and special cases.
What is Firmware?

• *Usually*, though **not** always:
  – is software which is installed into the internal memory of a processor
  – Has 1 execution context plus interrupts
  – Runs as highly privileged (w.r.t. processor modes)
  – Intentionally difficult to update
  – Updates come as a single large “image” instead of piecemeal updates of individual parts
  – Limited memory environments
  – Limited memory protection mechanisms
Many Types of Firmware

In order of least complexity:

- Bare Metal
- Super Loop Firmware
- Real Time Operating Systems
  - FreeRTOS
- Reduced Functionality Operating Systems
  - ucLinux – Linux for micro-controllers
- Full Operating Systems, but with limited flexibility
  - Symbian OS
  - Some distributions of Linux for Routers (dd-wrt, tomato)
What is a ”Super-loop” Firmware

- No Threads, No Scheduler
- No Virtual Memory
- No Users, No Groups
- No Heap

How does it work?
What is a "Super-loop" Firmware

- It is not an OS.
- Is conceptually just a big for loop
- All executable “tasks” are visited once in the Super Loop then it repeats

```cpp
void TaskManager::run_all_tasks() {
    for (auto& task : get_task_list()) {
        task.execute();
    }
}
```
What is a "Super-loop" Firmware

- No software limit of execution time
- No guarantee of when next start time is
- No idle task
- Always at 100% processor usage
- Metrics (usually trailing window):
  - Min/Avg/Max Super Loop time
  - Min/Avg/Max per Task time
Why is that stuff missing?

• Eliminates some classes of runtime errors
  – No memory fragmentation error, out of memory conditions
  – No thread starvation, priority inversion, priority arrangement issues
• Less functionality means less things can go wrong
• Easier to reason about the system
• More deterministic, repeatable
• Reducing the testing burden
• Allows you to focus on solving the high level functionality
Why is that stuff missing?

• “Undefined” behavior – what happens to the OS if:
  – Your heap is full?
  – Your heap is fragmented but has enough free space?
  – Your kernel is deadlocked?
  – Your kernel itself had an exception?

• “Undefined” is not an option in a critical environment
• Certification burden
  – you have to know how your system will behave, even in severe error cases
  – If something has an ”undefined” failure mode, it should be removed
Software Environment

- **Stacks**
  - Normal Mode Stack
  - Privileged Mode Stack (optional)
- **System Calls** ($swi$)
- **Custom Linker Script**
  - If we “allocate” too much it won’t build.
- **Special “libc” (libnano)** which is intended for embedded processors which we further restrict
  - Still have printf but reduced functionality
  - Stub out functions like _sbrk $\leftarrow$ grows the top of memory (_end)
- **Standard Cross Compiler from ARM**
  - arm-none-eabi-g++ (10.x)
Hardware Environment

- Typically a micro-controller.
  - Like a Cortex M series
- These class of devices:
  - Do not have an MMU
  - Have small but “fast” SRAM (< 1MB) – some TCM too (1 cycle memory, even smaller)
  - Have small flash (< 4MB)
  - May or may not have any floating point ops
- Frequently they must interact with low level, slower busses used in Industry.
  - I²C, SMBus
  - SPI, QSPI
  - UART, RS-485
  - CAN, LIN
  - Ethernet (<=100Mbps)
- Devices need to have very fast boot times (<< 1 sec)
- Devices need to run for long extended periods with no issues
  - May have a low power mode.
Super Loop Firmware on the Drone

- What type of systems use this type of solution?
- Simple Bus to Bus message bridging applications
- Examples are leaves in a graph topology
  - Simple Sensors (Source)
    - SPI/I2C to CAN publisher
  - Data loggers (Sink)
    - Capture all CAN
  - Controllers (Source + Sink)
    - UART/CAN, PWM+ADC/CAN, I2C/CAN
Language Features for our Embedded Platforms
Language Features for our Embedded Platforms

What do these limitations translate to?

• Should avoid pointers
  • In a micro-controller some real memory may or not exist at nullptr

• Must not use Exceptions
  • No throw/noexcept
  • No try/catch

• Must not use Run Time Type Info
  • typeid()

• Must not use Dynamic memory
  • No new/delete
  • No malloc/free
    • Pass the compiler/linker “-Wl,--wrap=malloc -Wl,--wrap=calloc”
Language Features for our Embedded Platforms

• Strongly should avoid forever for/while loops
  • Only 1 is allowed in the entire system in main().

• Should avoid while Loops without definite bounds
  • without definite, testable exit conditions these could lead to stuck execution
Language Features for Embedded Platforms

Should use:
- **for/while** loops with fixed, definite bounds
- References
- Statically defined memory
  - use the linker to check the size of named sections
  - Controlled *placement* `new`
- Const-ness
  - `constexpr`
  - `const`
  - `static_assert`
- Templates, but:
  - with parametric fixed sizes
Language Features for Embedded Platforms

• Complexity and Data Encapsulation
  – Well defined layers and APIs

• Inheritance

• Compile Time Polymorphism
  – Function Overloading
  – Operator Overloading

• Run Time Polymorphism
  – Function Overrides
  – Pure Virtual Interfaces
Embedded C++14 Policies
Embedded C++14 Policies

- Don’t invent your own basic types
- Use `<cstdint>`
- Cross-Compiler writers generally know better than you
- Use `uintptr_t`, not `uint32_t` to transport addresses
  - Unit testing is important, even in low level code!

```cpp
using my_uint8 = unsigned char; // is that true everywhere?
using my_uint16 = unsigned short;
using my_int32 = long; // is that really 32bits everywhere?

uint32_t get_register_address() {
    return reinterpret_cast<uint32_t>(ptr); // error on 64bit PC
}
```
Embedded C++14 Policies

- Use References *everywhere* you can
- Pointers should be *rare*
  - With no heap, `std::unique_ptr` is not usable!
  - low level exceptions
- With effort, their usage can be severally reduced, or isolated

```c++
// declare as a reference
Foo &foo = context.get_foo();
foo.bar();

// pass as a reference
void somefunc(Foo& foo) {
    foo.set_value(10);
}
somefunc(foo);
```
Embedded C++14 Policies

- **Use C++ namespaces**
  - Unlearn C namespacing for C++
- **Keep names DRY – Don’t repeat yourself.**

```cpp
// DO
namespace Project {
    namespace Component {
        enum class Mode { FreeRunning }
        class Peripheral {
            static void initialize(Mode mode);
        }
    }
}

// DO
Project::Component::Peripheral::initialize(
    Project::Component::Mode::FreeRunning
);
```
Embedded C++14 Policies

- Unlearning “C” namespacing is hard.
- Mental “Frames” are difficult to dislodge
- Commit to C++ namespacing 100%

```cpp
// DON'T half-way commit to C++ naming
namespace Project {
    enum class ComponentMode { ComponentModeFreeRunning };
    class ComponentPeripheral {
        static void initializePeripheral(ComponentMode mode) = 0;
    }
}

// DON'T
Project::ComponentPeripheral::initializePeripheral(
    Project::ComponentMode::ComponentModeFreeRunning
);
```
Embedded C++14 Policies

- Resource Acquisition is Instantiation (RAII)
  - A good practice in general C++
  - An essential practice in Embedded C++

```cpp
class DriverImpl {
    DriverImpl(volatile Peripheral * peripheral, ...) :
        m_periph(peripheral) {
            m_periph->configure(...);
            m_periph->enable();
        }
    ~DriverImpl() {
        m_periph->disable();
    }
};
```
Embedded C++14 Policies

• Layers interact through Pure Virtual Interfaces
• Enforces good separation of data and responsibilities

```cpp
// interface
class Foo {
public:

    virtual uint64_t get_count(void) const = 0;
};

// implementation
class Bar : public Foo {
public:

    uint64_t get_count() const override { ... };
};
```
Embedded C++14 Policies

• Use `final` when appropriate
• This controls what can be subclassed and what can not.
• Customers need to know what is flexible and what isn’t.

```cpp
// implementation has an interface and can not be subclassed
class ControllerImpl final : public IController { ... };

// an attempt to subclass
class MyController : public ControllerImpl { // error!
```
Embedded C++14 Policies

• Only inline simple methods

class Foo {
public:
    inline size_t get_count() { return m_count; }
Protected:
    size_t m_count;
};
Embedded C++14 Policies

• **Be as const as you can stand!**

```cpp
class Foo {
public:
    const char * const bar(const char * const, …) const;
    inline const size_t &get_count() const { return m_count; }

Protected:
    size_t m_count;
};

// the methods invoked on the object must be const too
bool somefunc(const Foo& obj) { ... };
```
Embedded C++14 Policies

- Methods which use `const` truly should have no side-effects!
  - “Read and Clear” register operations should not be marked `const`
  - Unless they capture the lost information (and have a manual clear!)

```cpp
class SimulatedStatusRegister {
public:
  uint32_t get_status() const { // acts like a Read and Clear Register
    auto tmp = m_value;
    m_value &= ~0x80; // lowers status bits!
    return tmp;
  }
protected:
  mutable uint32_t m_value;
};

enum class StatusCode {};

class DriverImpl : public Driver {
public:
  StatusCode get_status() const { // problematic to call this const!
    return static_cast<StatusCode>(m_periph.get_status() & 0x7F);
  }
};
```
Embedded C++14 Policies

- Avoid `mutable` unless you absolutely need it
  - Metrics collectors
  - Locking mechanisms (between ISR/Task)

```cpp
class Foo {
public:
  void bar() const {
    // …
    m_call_count++; // mutable
  }
  const size_t &get_call_count() const {
    return m_call_count;
  }
protected:
  mutable size_t m_call_count;
};

class Baz {
public:
  bool check(void) const {
    m_lock.lock();
    // …
    m_lock.unlock();
  }
  void alter(...) {
    m_lock.lock();
    // …
    m_lock.unlock();
  }
protected:
  mutable Lock m_lock;
};
```
Embedded C++14 Policies

• Ownership mechanisms

```cpp
template <typename UNIT>
class OwnedBuffer {
public:
    using Owner = void *; // for ease of use with "this".

    // For use with "const" methods
    void acquire(const Owner owner, const UNIT* &buffer) const;
    void release(const Owner owner) const;

    // For use with "non const" methods
    void acquire(const Owner owner, UNIT* &buffer);
    void release(const Owner owner, size_t length);

protected:
    mutable Owner m_owner;
};
```
Embedded C++14 Policies

- Use `std::atomic<>` to share data between ISR and Tasks
- Critical Sections tend to be implemented using atomics
- READ the docs. `std::atomic<>` has some gotchas!

```cpp
class CriticalSection {
public:
    ...
    bool lock() {}
    bool unlock() {}
protected:
    std::atomic<uint32_t> m_count;
};

// See atomics video from CppCon!
```
Embedded C++14 Policies

- Data structures must be sized at compile time.
  - Usecases must be thought out ahead of time.

```cpp
template <typename UNIT, size_t COUNT>
class Foo : public OwnedBuffer<UNIT> {
  public:
    /// ...
  protected:
    std::array<UNIT, COUNT> m_memory;
    /// ... Other tracking information
};

// declared in global space or within Class
Foo<
type8_t, 128u> buffer;
```
Embedded C++14 Policies

- **static** Memory definitions can be used
- Allows object Construction phases to be controlled.

```cpp
/// Somewhere in global memory
static std::aligned_storage<sizeof(Foo), alignof(Foo)>::type storage;

...
void func(void) {
    // object initialization happens in preallocated global memory
    Foo &foo1 = *new(&storage) Foo{...};

    // object initialization happens on stack
    Foo foo2{...};
}
```
Embedded C++14 Policies

- No “diamond” virtual inheritance patterns
- Compilers stops overt bad behavior
- Don’t try to make it work, redesign your Hierarchy or Models

class Foo {
public:
    virtual size_t get_bar() const = 0;
};
class Bar : public Foo {
public:
    size_t get_bar() const override { ... }
};
class Baz : public Foo {
public:
    size_t get_bar() const override { ... }
};
class Gaf : public Bar, public Baz {};

Gaf g{};
// where is it from?!
g.get_bar();
Embedded C++14 Policies

- **Avoid #define and #ifdef as much as possible**
- **Be as constexpr as possible**
  - Use the *compiler* to test your code

```cpp
constexpr uint32_t foo(uint32_t) { ... }

namespace {
    constexpr uint32_t foo_test[] = {foo(X), foo(Y)};
    static_assert(foo_test[0] == 0x10, "Must be true or something is wrong");
} // anonymous
```
Embedded C++14 Policies

- Static Assert as much as you can.
- `sizeof()`, `offsetof()`, `alignof()` are useful in stating assumptions and having the cross-compiler check them.

```cpp
struct RegisterDefinition { ... };
static_assert(sizeof(RegisterDefinition) == 0x80, "Definition is incorrect"); // from the Datasheet!

static_assert(offsetof(RegisterDefinition, CTRL) == 0x10, // From the Datasheet!
  "Must be at the correct offset");

struct FancyDataType { ... };
static_assert(alignof(FancyDataType) == 8, "Must has specific alignment!");
```
Embedded C++14 Policies

- Use `alignas(x)` for data types which need to understand the Data Cache / Bus

```cpp
struct alignas(8) CacheAlignedDataType {
    constexpr CacheAlignedDataType(uint16_t _x,
        uint16_t _y,
        uint16_t _z,
        uint16_t _w)
        : x{_x}, y{_y}, z{_z}, w{_w} {}
    uint16_t x, y, z, w;
} // would naturally be alignof(2)

static_assert(alignof(CacheAlignedDataType) == 8, "Must has specific alignment!");

// prealigned preinitialized constants which don't span cache lines
static const CacheAlignedDataType g_value1{1,2,3,4};

// constexprs may not even be put into global memory
constexpr CacheAlignedDataType g_value2{5,6,7,8};
```
Embedded C++14 Policies

- Use `enum class` to be strict about types for parameters, etc
- Use "underlying" template wrapper to explicitly get the value

```cpp
class Peripheral {
    enum class Mode : uint8_t { FreeRunning = 0, SingleShot = 2, ... };
};

void DriverImpl::set(Peripheral::Mode value) {
    m_periph->MODE = static_cast<typename std::underlying_type<RegisterValue>::type>(value);
}

Driver& foo = context.get_foo();
foo.set(Peripheral::Mode::FreeRunning);
```
Embedded C++14 Policies

- **Use volatile to describe Memory Mapped Registers**

```cpp
namespace Chip {
  struct Peripheral {
    uint32_t CONTROL;
    uint32_t MODE;
    uint32_t STATUS;
  };

  class DriverImpl : public Driver {
    public:
      DriverImpl(volatile Chip::Peripheral& ref) : m_periph(ref) {}
  }

  protected:
    volatile PeripheralRegisters& m_periph;
  }

  // in the context
  Chip::Peripheral& periph0 = *new(PERIPH_ADDRESS)Chip::Peripheral{};
  Chip::Driver driver0{periph0};
```
Embedded C++14 Policies

- Rule of 5 + 1
  - Default, Move, Copy Constructors and Move, Copy Assignment should be defined as needed. Plus destructor.

```cpp
class Foo {
public:
    // Rule of 5
    Foo() = delete;
    Foo(const Foo&) = delete;
    Foo(Foo&&) = delete;
    Foo& operator=(const Foo&) = delete;
    Foo& operator=(Foo&&) = delete;
    // Rule of 5 + 1
    ~Foo() = default;

    Foo(…); //!< Can only be constructed w/ Params
};
```
Embedded C++14 Policies

- Peripherals, Drivers, Services, Applications should never be moved or copied in a Statically defined environment

```cpp
class DriverImpl : public Driver {
public:
    // Rule of 5
    DriverImpl() = delete;
    DriverImpl(const Foo&) = delete;
    DriverImpl(DriverImpl&&) = delete;
    DriverImpl& operator=(const DriverImpl&) = delete;
    DriverImpl& operator=(DriverImpl&&) = delete;
    // Rule of 5 + 1
    ~DriverImpl() = default;

    DriverImpl(...); // May only be constructed w/ Params
};
```
Embedded C++14 Policies

• Rule of 16???
  – There a ton of new and delete overrides when compiling unit tests on non-target hosts.
  – Be complete! Understand your usecase!

```cpp
static void* operator new(std::size_t) = delete;
static void* operator new(std::size_t, const std::nothrow_t& noexcept = delete;
static inline void* operator new(std::size_t size, void* ptr noexcept { return ::operator new(size, ptr); } // routes to other deleted
static void* operator new[](std::size_t) = delete;
static void* operator new[](std::size_t, const std::nothrow_t& noexcept = delete;
static void* operator new[](std::size_t, void*) noexcept = delete;
static inline void operator delete(void*) {}
static void operator delete(void*, const std::nothrow_t& noexcept = delete;
static inline void operator delete(void*) noexcept = delete;
static void operator delete[](void*) = delete;
static void operator delete[](void*, const std::nothrow_t& noexcept = delete;
static void operator delete[](void*, void*) noexcept = delete;
static void operator delete[](void*, std::size_t noexcept = delete;
static void operator delete[](void*, std::size_t, const std::nothrow_t& noexcept = delete;
static void operator delete[](void*, std::size_t) noexcept = delete;
static void operator delete[](void*, std::size_t, const std::nothrow_t& noexcept = delete;
```
Embedded C++14 Policies

- **Operator Overrides**
  - Implement all in a group
  - `==`, `!=`
  - `>`, `>=`, `<`, `<=`
  - `+`, `-`, (on itself) `/`, `*`, `%` (by scalars)
  - `operator bool`, `!`
  - Avoid member access oddities `->`, `->*`, `&`, `*a`, `[]`

- Keep it regular, avoid weirdness and unexpected behavior

```cpp
class Foot {
public:
    Foot& operator*(const Foot & other); // Logical Error! Foot x Foot = Area!
    Foot& operator*(long double v); // OK! Scaled value
};
```
Embedded C++14 Policies

- Quote Operator – useful for defining constants with a type.
- Put these in a literals namespace!
- Useful for defining max/mins for Sensor data types

```cpp
class furlong {
public:
    constexpr furlong(long double);
};

namespace literals {
    furlong operator""_fl(long double x);
}

using namespace literals; // enables access to quote operator
constexpr auto track_lengths[] = {2.0_fl, 5.0_fl, 6.5_fl, 9.0_fl};
```
Embedded C++14 Policies

- Favor the {} (brace initialization) Constructors
- This warns about narrowing operations!

```cpp
class Foo {
public:
    Foo(short a) : b{a} {}  
protected:
    short b;
};

unsigned long long value = 20092837409893ull;  
// or
int value = 2;

Foo f{value}; // narrowing warning!
```
Embedded C++14 Policies

- & is one of the most “overloaded” terms in C++
- Reduce the mental burden of reading these by using some new terms and, or, xor, not for logical operations

```cpp
// Address of
void *ptr = &thing;

// Reference
Foo& foo = …;

// Bit And
uint8_t thing = (a & b);

// Logical And
bool thing = (a && b);

// Move Semantics
Foo(Foo&&) = delete;

// Custom Overrides
Foo& operator&(…);

// logical replacement
bool foo = …;
bool bar = …;

if (foo and bar) {
}

if (not foo) {
}

if (foo or bar) {
}

if (foo xor bar) {
}

// bitwise words, bit_and, bit_or, bit_xor, bit_not also exist
```
Execution Model and Class Design
Execution Model and Class Design

- What not to do
- What Problems can arise
- Boot to Main()
- Class Hierarchy
- Asynchronous State Charts
What not to do

• There is 1 rule of Super Loop Firmware
• Do not write blocking functions!
  – Never, Ever
  – EVAR!1!

smol angrz pupper
What problems can arise?

- If you block, you prevent other Tasks from executing
- If you block long enough the Watchdog will reset you
- Two types of Blocking
  - Waiting on Peripheral (HW)
  - Waiting on other SW component
What problems can arise?

- **Waiting on Peripheral HW**

  // Obvious problem, may never be true!  
  void DriverImpl::wait_until_ready(void) {  
      while (m_periph.STATUS != 0x1) {}  
  }

  // Counter may not be enabled, may count too slowly  
  void DriverImpl::wait_until_value(uint32_t count) {  
      while (m_periph.COUNTER < count) {}  
  }

  // Best choice  
  bool DriverImpl::is_value(uint32_t count) {  
      return (m_periph.COUNTER == count);  
  }
What problems can arise?

• **Waiting on Software Component**
• **If the loop is blocked, the other component will never run!**

// Obvious problem, may never be true!
void ServiceImpl::wait_until_ready(void) {
    while (not m_driver.is_ready()) {}
}

// Parameterized wait will never give any other answer!
void ServiceImpl::wait_until_complete() {
    while (m_driver.get_count() < m_complete_count) {}
}

// Best choice
bool ServiceImpl::is_ready(uint32_t count) {
    return (m_driver.is_ready() && m_initialized);
}
What problems can arise?

- Memory Exhaustion
  - Too much static RAM allocation will cause a failure to link
  - DMA section has a cap in the linker script

```cpp
ASSERT (_dma_buffers_size < LIMIT, "DMA Section has been overfilled!")
__attribute__((used, section(".dma_buffers"))) alignas(uint32_t) static std::array<uint8_t, 1024> dma_buffer;
```
What problems can arise?

- **ISR/Task concurrent data corruption**
  - Avoid R/W patterns on both ends. One writes, the other reads.
  - Locking in an ISR can be dangerous, costs time.

- **Hardware Exceptions**
  - Divide by Zero, `nullptr_t` dereference, bus faults, alignment faults
Boot to main()

- This is a simplified reduction, reality is a bit more nuanced
- Know your hardware!
- Vector Table is located via linker to the correct physical address
- CMSIS – A set of Cortex specific setup code used by many vendors to help initialize your hardware. “C” centric.

```c
// the Linker defines where the stack starts in memory
extern void *stack_start;

// Crude C version that most CMSIS offers
extern "C" __attribute__((used, section(".vectors")))
void *vector_table[] = {
    _stack_start,
    &reset_handler,
    &nmi_handler,
    &hard_fault_handler,
    ...
};
```
Boot to main()

// Declared in Global Memory
Watchdog g_watchdog;

// called by Hardware during boot
__attribute__((used, naked, noreturn))
void reset_handler() {
    init_early_hardware(); // power, stack, critical items
    crt0_init(); // "libc/libc++" runtime init + global ctors are called here
    // now Globals have been initialized!
    init_later_hardware(); // pmumx
    jump_to_main();
}

// this is called before we can formally use instantiated classes
void init_early_hardware() {
    // C++ can be used here
    // just be sure to only use static class methods!
    // no GLOBALS have been initialized yet! constexpr (immediates) only!
    Power::boot_configure();
    Memory::ecc_initialize();
    Watchdog::configure(500_ms);
    Clocks::configure();
    Memory::enable_mpu(); // constant defined regions
}

// this is called after we can use instantiated classes
void init_later_hardware() {
    // call the post ctor setup methods on the drivers and services
    init_drivers();
    init_services();
}
Boot to main()

// constructed during crt0 initialization.
GlobalContext context; // contains most Drivers and Services

static std::aligned_storage<sizeof(App), alignof(App)>::type storage;

//[[noreturn]]
void main(void) {
    Watchdog &wd = get_watchdog();
    TaskManager &taskmgr = context.get_task_manager();
    App &app = *new(&storage) App(context);
    absp_assert(app.setup(), "Record fatals");
    while (true) {
        taskmgr.run_all_tasks();
        app.execute();
        wd.reload();
    }
    absp_abort("Main ended!");
}
Classes

- **Core Classes**
- **Communications between Layer**
  - **Peripheral**
    - Maps to Device Registers (see CMSIS)
  - **Context**
    - Hold the instances of all Drivers and Services
  - **Driver**
    - Has a reference to a specific instance of a Peripheral (memory mapped)
    - Can be a Task
  - **Service**
    - Has a reference to a Driver
    - Can be a Task
  - **Application**
    - Has many Services, occasionally a few Drivers
    - Can be a Task
Classes

• Core Classes
  – TaskManager
  – Watchdog
  – GPIOs
  – System Timer
  – Context
  – Fatal Error Log
Core Classes

• TaskManager
  – Contains the reference to the list of Tasks
  – it’s just a for loop (see way above)
Core Classes

• **Watchdog**
  – Controls access to the hardware which will reset the MCU unless it is “reloaded” or “pet” every so often (usually < 1 sec)
  – NO Task can exceed that time in 1 cycle!
Core Classes

• **GPIO**
  – Controls the pinmux on bootup
  – Used to detect raw input/output on pins
  – Used to configure pin settings
Core Classes

• System Timer
  – Allows the system to measure time in a highly precise way (microseconds).

namespace Time {
class duration_us {}; // contains a 64 bit value of microseconds
class Timer {
public:
    virtual duration_us get_time_us() const = 0;
};
Core Classes

• Context
  – Contains all of the configured Drivers and Services for this board.
  – Instantiates implementations, exposes interfaces

```cpp
class Context {
public:
  UART::Driver& get_uart0() { return m_uart0; }
  SPI::Driver& get_spi1() { return m_spi1; }
  I2C::Driver& get_i2c0() { return m_i2c0; }
protected:
  internal::UartDriver m_uart0;
  internal::SpiDriver m_spi1;
  internal::I2CDriver m_i2c0;
};
```
Core Classes

• Fatal Error Log
  – Captures faults in the internal memory just before reset of the processor.
  – Asserts and Aborts route to this
  – Captures location in code without `__FILE__` or `__LINE__`
    • (which help make reproducible builds)

// [[noreturn]]
void absp_assert(bool condition, enum class reason, enum class id);
Core Classes

Tasks
- Tasks are for maintaining state which has to happen each cycle of the Super Loop
- Drivers need to check to see if a bus transaction has completed
- Services need to know if Driver transaction returned requested data.
- Applications need to know latest data from Services.

class Task {
public:
    virtual void execute(void) = 0;
...}
}
Core Classes

- Each of our Tasks is generally a wrapper around a state machine
- Context for each Task is given once at construction

```cpp
class DriverImpl : public Task, public Driver {
public:
    DriverImpl(volatile Peripheral * ptr) : ...
    { /* init start chart */ }

    void execute(void) {
        if (m_statechart.isActive()) {
            // input signals based on environment
            m_statechart.runCycle();
            // check for output signals
        }
    }

protected:
    StateChart m_statechart;
};
```
Class Layers

- Application
- Service
- Chip Driver
- Bus Driver
- Peripheral
- Hardware

- Data Centric API
- Async API
- Data Type Transactions
- Memory Transactions

External Communication
Communication Between Layers

• We’ll work out way bottom up
Communication Between Layers

- Driver ↔ Peripheral
  - Peripherals are memory mapped hardware
  - loads/stores w/ offsets
  - Know your assembly to debug!
Communication Between Layers

- **Chip Drivers ↔ “Bus” Driver**
  - Chip drivers understand the state of a chip at the end of a bus. Like an EEPROM or Temperature Sensor.
  - Bus Drivers take work as “Transactions”
    - Command Read/Write + Data Buffer
  - Data Buffers have to be formatted per Chip datasheets
Communication Between Layers

- Service ↔ Chip Drivers
  - Services use the “Async API” of the Chip Driver to start/check on work
  - Remember, no blocking!
  - Assuming some singular functionality of chip here (Temp Sensor)
  - Purpose built API per Chip
  - Driver has an internal StateChart
  - Driver may deal w/ calibration, etc without Service knowing or with directly intervention

```cpp
class Driver : public Task {
    ...
    virtual Status start_read(void) = 0;
    virtual Status is_read_complete(void) = 0;
    virtual Status get_read_status(void) = 0;
    virtual Value get_value(void) = 0;
    virtual Status start_calibration(void) = 0;
    ...
};
```
Communication Between Layers

- Applications ↔ Services
  - Services use timers or other external triggers to call lower layer from Application
  - Applications just want to capture latest values and get timestamps
    - Shouldn’t care about bus ordering or time multiplexing
  - Service has internal StartChart, timer based, etc

```
class Service : public Task {
...
    virtual Value get_latest_value(Time::duration_us& ago) = 0;
};
```
Communication Between Layers

• External Communication
  – Applications process messages using other Services/Drivers/Peripherals
  – CAN, Ethernet

```cpp
class Application : public Task {
...
  void messageSubscriber(const Message& msg);

  void execute(void) {
    ...
    m_service.publish(outmsg);
    ...
  }
};
```
Asynchronous Non-blocking State Machines

- We use a tool to generate a C++ class from a StateChart.
- This is a simple StateChart which just blinks an LED
Asynchronous Non-blocking State Machines

• Generated C++ looks similar to this:

```cpp
class BlinkSm {
public:
    class Callback {
        virtual void turn_on(void) = 0;
        virtual void turn_off(void) = 0;
    };
    // output events
    bool isRaised_on_led_on(void);
    // if it supported input events it would have
    // void raise_xxxxxx(void);
};
```
Asynchronous Non-blocking State Machines

- Your class must implement callbacks

```cpp
class DriverImpl : public BlinkSm::Callbacks {
public:
    DriverImpl(...) : ...
        m_startchart.registerCallbacks(this);
    void turn_on(void) override {}
    void turn_off(void) override {}
protected:
    StateChart m_statechart;
};
```
StateChart Execution Model

• Have state themselves
  – Active – has not reached Final
  – Final – can not further process, must be restarted
• Can be timer driven
  – The time difference since last execution must be known
• Internally they:
  – Determine if any exit guards are true
  – Execute any exit rules of the current State
  – Execute along the path until the next State
  – Execute the entry rule on the next State
StateChart Execution Model

- StateCharts need input and have output

```c
// overrides
void DriverImpl::turn_off(void) {
    m_led.inactive();
}
void DriverImpl::turn_on(void) {
    m_led.active();
}
void DriverImpl::execute(void) {
    // check the environment to detect state to send into the SM
    if (m_statechart.isActive()) {
        m_statechart.runCycle();
    }
    // check the outputs of the SM to inform other components
    if (m_statechart.isRaised_on_led_on()) {
        printf("Blink!\r\n"); // debugging
    }
}
Communication between Devices
Communication between Devices

- Communication between Devices is over CAN (FD) or Ethernet (<=100Mbps)
- Protocols like:
  - DroneCAN (was UAVCAN v0) work on CAN
    - droncan.github.io
  - Cyphal (was UAVCAN v1) works on CAN or Ethernet (special versions)
    - opencyphal.org
  - In these examples it will be referred to using UAVCAN v0 style.
Topology

Shared Physical Bus

Node ID X

Node ID X+1

Node ID X+2

Node ID N

…
Topology

- Nodes are all connected to a single bus
- Protocol does not explicitly have “routing” across multiple subnets
- Each Node has a unique ID on the bus (1-126), 0 and 127 are off limits
- ”Lower” Nodes have priority on the bus
- Each message has a built in CRC
- To publish or subscribe you have to be a Node on the bus
Messages

• Message definitions are written in a DSDL
  – (domain specific description language)
• A transpiler is used to convert the .dsdl files to .hpp files
• Allows for arbitrary sized field (3 bits, etc)
Example Message

tuavcan.protocol.NodeStatus (v0)
uint32 uptime_sec
uint2 health
uint3 mode
uint3 sub_mode
uint16 vendor_specific_code
Example Message

This generates a C++ header which you just include and then subscribe to:

```cpp
#include <uavcan/uavcan.hpp>
#include <uavcan/protocol/NodeStatus.hpp>

using Message = uavcan::protocol::NodeStatus;

// simplification
m_service.subscribe(&listener);

void listener(const Message& msg) {
    if (msg.health == ...) {...}
}
```
Future C++17 and Reference
Moving to C++17

- Use `explicit` keyword on ctors
  - Avoid C++ conversion magic from some types to others.
- Use `[[fallthrough]]` when you need to (rarely)
- Use `[[deprecated]]` when you need to
  - Good customer warning!
- `if constexpr` – removes many `#ifdef` conditions
- `std::variant<>`
  - Replaces tons of legacy custom struct/union/enum code!
  - Safe in a heap-less system!
- `std::optional<>`
  - A simpler form of variant
Interesting CppCon Talks

• Handling a Family of Hardware Devices with a Single Implementation - Ben Saks
• Memory Mapped Devices as Objects – Dan Saks
• Stop Teaching C
• Extern c: Talking to C Programmers about C++
• C++ Atomics from, basics to Advanced. What do they really do?