Using C libraries in your Modern C++ Embedded Project

EtherCAT Stack, C++17, and Modern C++ Idioms

ciere consulting

Michael Caisse

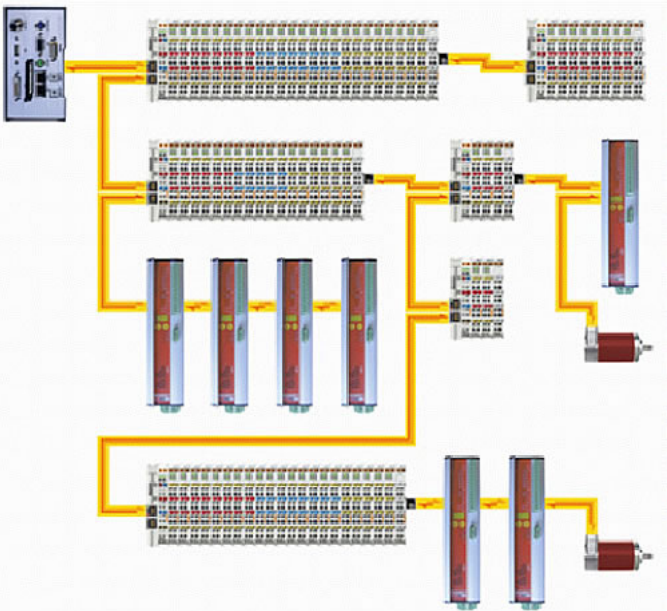
mcaisse@ciere.com

ciere.com

Part I

EtherCAT

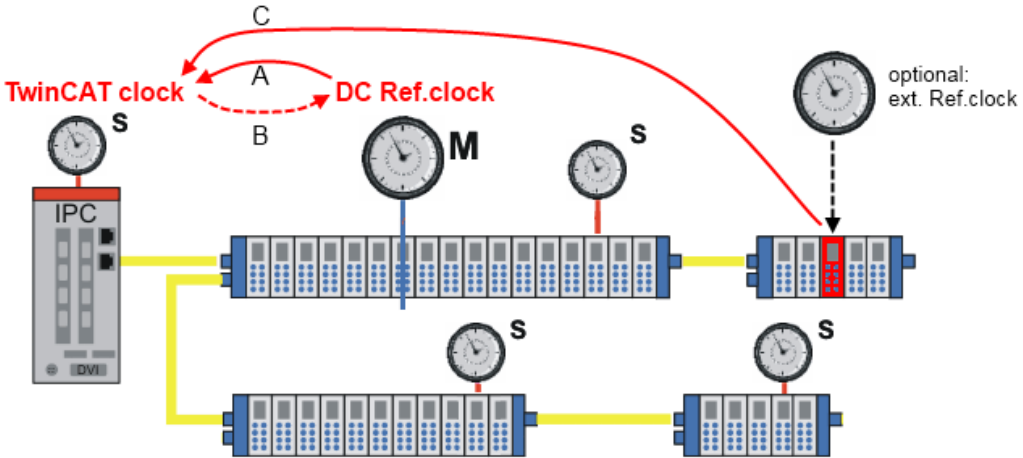
Topologies



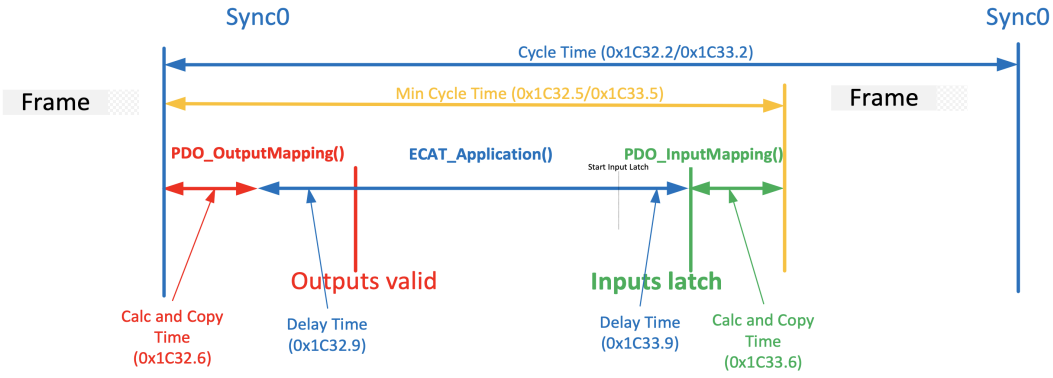
EtherCAT Functional Principal

EtherCAT Video

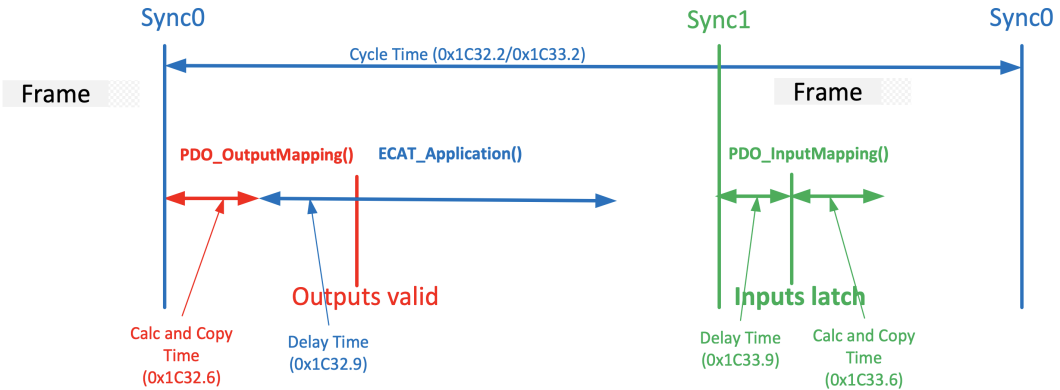
Distributed Clocks



Sync0


Sync0
Sync0
Cycle Time (0x1C32.2/0x1C33.2)
Frame
Min Cycle Time (0x1C32.5/0x1C33.5)
Frame
PDO_OutputMapping()
ECAT_Application()
PDO_InputMapping()
Start Input Latch
Outputs valid
Inputs latch
Calc and Copy Time (0x1C32.6)
Delay Time (0x1C32.9)
Delay Time (0x1C33.9)
Calc and Copy Time (0x1C33.6)

Sync0/Sync1


Sync0
Sync1
Sync0
Cycle Time (0x1C32.2/0x1C33.2)
Frame
Frame
PDO_OutputMapping()
ECAT_Application()
PDO_InputMapping()
Outputs valid
Inputs latch
Calc and Copy Time (0x1C32.6)
Delay Time (0x1C32.9)
Delay Time (0x1C33.9)
Calc and Copy Time (0x1C33.6)

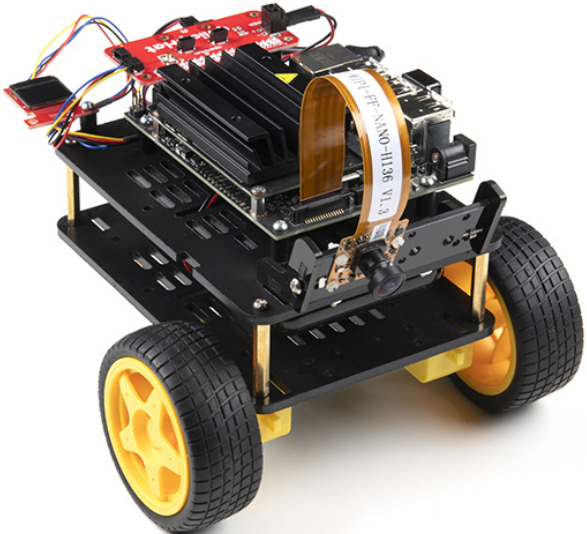
DC Jitter

System jitter is significantly less than 1us.

Part II

Philosophy

Priorities and trade-offs



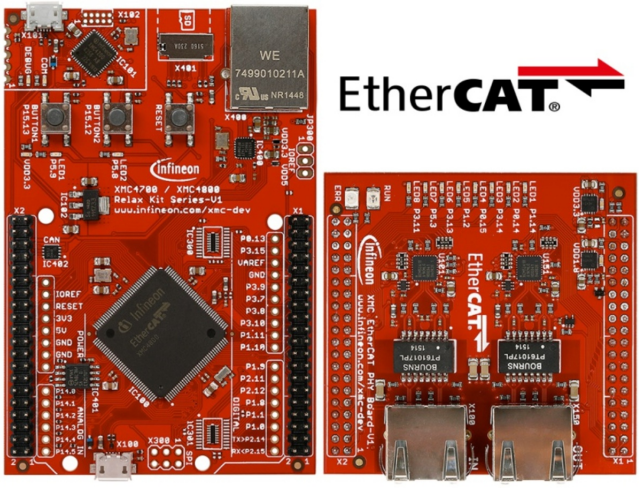
Pick one?

- Time
- Cost
- Resources

Part III

Pass 1

Infineon XMC



Outline

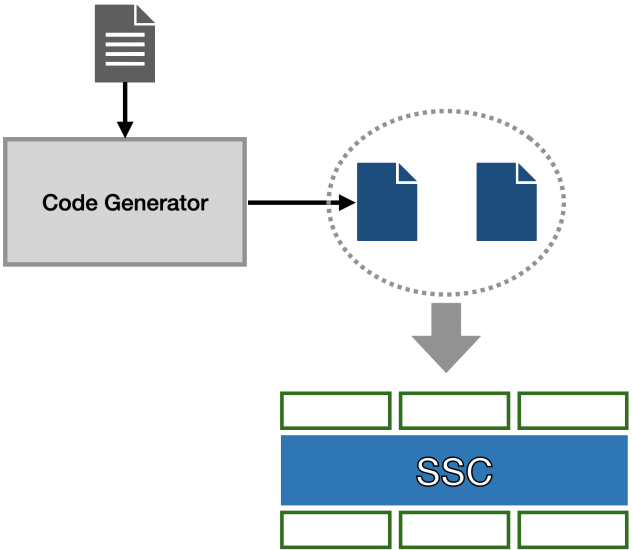
- Beckhoff SSC
- `extern "C"`

SSC

Beckhoff SSC - Slave Stack Code

“The EtherCAT Slave Stack Code (SSC) is a code written in ANSI C. Its modular and simple structure enables fast entry into slave development.”

SSC



Plugins

User provides functionality:

- Calls the main loop
- Provides call for process data handling
- Provides calls for interrupt control
- Calls interface on interrupts
- Hardware concrete to the abstraction

The Interface

Called in our main loop. The main loop processing for SSC

```
void MainLoop(void);
```

Called from SSC when cyclic data is available.

```
void process_cycle(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out);
```

Outline

- Beckhoff SSC
- `extern "C"`

Overloads

We can overload C++ functions.

```
int func();
int func(int);
float func(float);
float func(int, float);
```

Overloads

We cannot overload C functions.

```
int func();
int funci(int);
float funcf(float);
float funcif(int, float);
```

Can we use the C++ functions in C?

Overloads

We cannot overload C functions.

```
int func();
int funci(int);
float funcf(float);
float funcif(int, float);
```

Can we use the C++ functions in C?

Language Linkage

These have language linkage property:

- Function types
- Function names with external linkage
- Variable names with external linkage

```
int func1(int, float);
static int func2(int, float);  // internal linkage
```

Language Linkage

Language linkage property describes requirements to link such as

- Calling convention
- Name mangling

C available to C++

C Code:

```c
int get_radio_id() {
  /** ....  **/
}
```

C++ Code:

```cpp
extern "C" {
   int get_radio_id();
}

// ...
auto radio_id = get_radio_id();
```

C++ available to C

C++ Code:

```
StateMachineA sm_a;
StateMachineB sm_b;

extern "C" void UserLoop() {
  sm_a.step();
  sm_b.step();
}
```

C Code:

```
UserLoop();  /** somewhere deep in the C code **/
```

Dealing with `const`

See Jason's C++ Weekly Episode 283

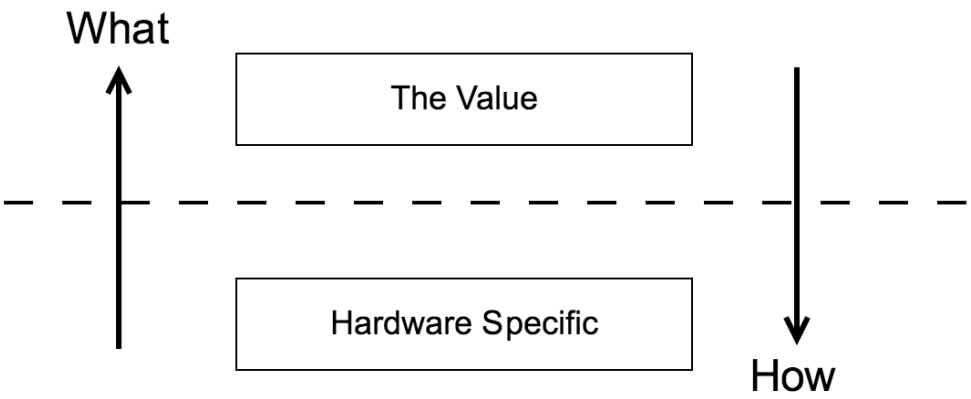
Main loops all the way down

```
extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
     // UGLY THINGS WE DON'T PUT ON SLIDES
}
}

extern "C" {
// This is the main loop in the SSC stack.
void MainLoop(void);
}

int main() {
    init();

    while (1U) {
        // The SSC main loop handles the EtherCAT state machine
        MainLoop();

        /**  UGLY CODE THAT WORKS BUT LACKS DIVISTION  **/
    }
}
```

Part IV

Pass 2

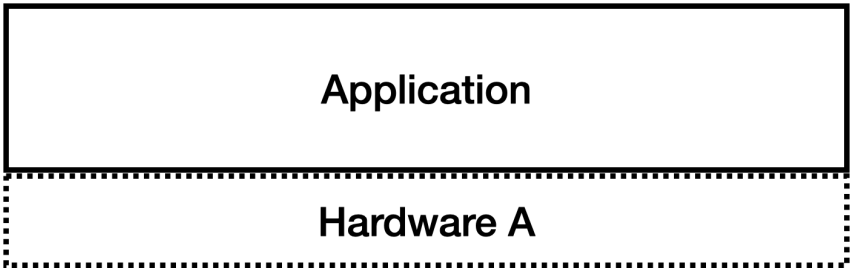
Declaritive Divisions



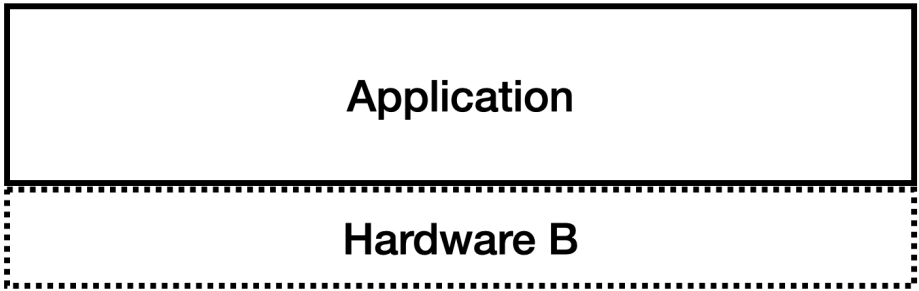
Some Goals

- Testability
- Simulation
- Hardware agnostic

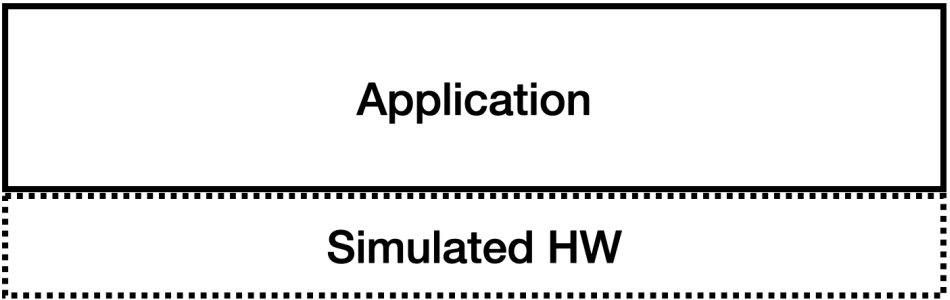
Layered



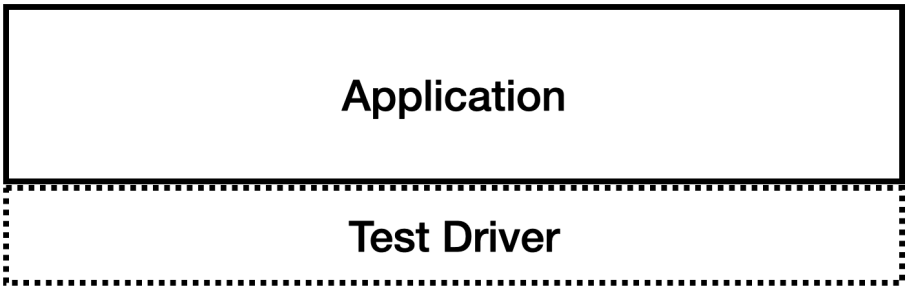
Layered



Layered



Layered



main things

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

main things

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

main things

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

main things

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

main things

```
ecat_data::config_in to_config_in(TOBJ7000 *pdo) {
    return ecat_data::config_in
        {  .freeze_boost        = (uint32_t(pdo->Freeze_boost_h) << 16) |
                                   uint32_t(pdo->Freeze_boost_l),
           .outside_temperature = int32_t(pdo->Outside_temperature),
           .do_clean            = bool(pdo->Do_clean),
           .flash_sign          = bool(pdo->Flash_sign)
        };
}
```

```
extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
```

main things

```
ecat_data::config_in to_config_in(TOBJ7000 *pdo) {
    return ecat_data::config_in
        {  .freeze_boost         = (uint32_t(pdo->Freeze_boost_h) << 16) |
                                    uint32_t(pdo->Freeze_boost_l),
           .outside_temperature = int32_t(pdo->Outside_temperature),
           .do_clean             = bool(pdo->Do_clean),
           .flash_sign           = bool(pdo->Flash_sign)
        };
}
```

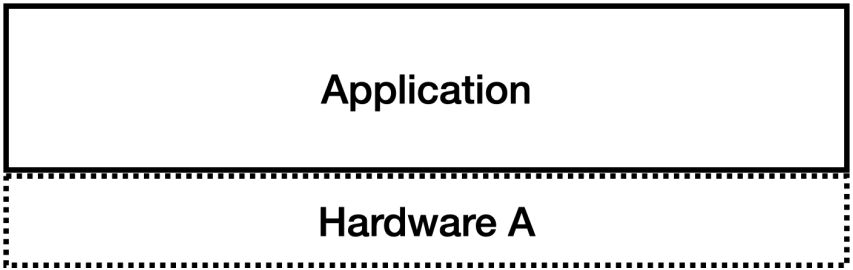
```
extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
```

main things

```
ecat_data::config_in to_config_in(TOBJ7000 *pdo) {
    return ecat_data::config_in
        {  .freeze_boost        = (uint32_t(pdo->Freeze_boost_h) << 16) |
                                   uint32_t(pdo->Freeze_boost_l),
           .outside_temperature = int32_t(pdo->Outside_temperature),
           .do_clean            = bool(pdo->Do_clean),
           .flash_sign          = bool(pdo->Flash_sign)
        };
}
```

```
extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
```

Layers



hardware layer

```
namespace xmc::hardware {

struct hardware {
    static void init();

    static void tick();

    static bool send_can_msg_obj(uint8_t const* /*data*/);
    static std::optional<can_msg_pack_t> get_can_msg_obj();

    static uint32_t get_debug_status_word();

    static void status_in_idle();
    static void status_in_op();
    static void status_have_error();
    static void status_clear_error();
    static void status_have_ecat_msg();
};

}
```

the app

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

the app

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

the app

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

the app

```
slushie_app<xmc::hardware::hardware> app;


extern "C" {
void process_app(TOBJ7000 *pdo_in, TOBJ6000 *pdo_out) {
    auto config = to_config_in(pdo_in);
    app.process_cyclic_data(config);

    auto status = app.get_cyclic_status();
    fill_pdo_status(status, pdo_out);
}
}
extern "C" {
void MainLoop(void);
}

int main() {
    app.init();

    while (1U) {
        MainLoop();
        app.app_loop();
    }
}
```

app

```
template<typename HardwareLayer>
class slushie_app {
  public:
   void init();
   void process_cyclic_data(ecat_data::cmd_in const& cmd);
   ecat_data::status_out get_cyclic_status();
   void app_loop();

  private:
   void process_new_ecat_msg();
   void pre_app_loop();
   void post_app_loop();

};
```

app

```
template<typename HardwareLayer>
class slushie_app {
  private:
    ecat_data::cmd_in last_ecat_cmd_ = {};
    ecat_data::cmd_in new_ecat_cmd_ = {};
    ecat_data::status_out current_status_ = {};

    uint8_t status_toggle_ = 0x01;

    uint16_t app_loop_count_ = 0;
    uint16_t app_loop_count_shadow_ = 0xdead;

    enum class bus_cycle_result : uint8_t { waiting, have_msg, invalid_cycle };
    bus_cycle_result process_ecat_state_ = bus_cycle_result::waiting;

    using ecat_slushie_machine_t = ecat_slushie_machine::machine<HardwareLayer>;
    boost::sml::sm<ecat_slushie_machine_t> ecat_slushie_sm_;
};
```

```
void init() {
    HardwareLayer::init();
}
```

app

```
void app_loop() {
    pre_app_loop();

    switch (process_ecat_state_) {
        case bus_cycle_result::waiting:
            ecat_can_sm_.process_event(ecat_can_machine::tick{});
            break;

        case bus_cycle_result::have_msg:
            process_new_ecat_msg();
            last_ecat_cmd_ = new_ecat_cmd_;
            break;

        case bus_cycle_result::invalid_cycle:
            ecat_can_sm_.process_event(ecat_can_machine::invalid_ecat_msg{});
            break;

        default:
            ecat_can_sm_.process_event(ecat_can_machine::tick{});
            break;
    }

    process_ecat_state_ = bus_cycle_result::waiting;
    post_app_loop();
}
```

```
void process_cyclic_data(ecat_data::cmd_in const& cmd) {
    if (cmd.mode == last_ecat_cmd_.mode) {
        process_ecat_state_ = bus_cycle_result::invalid_cycle;
    } else {
        new_ecat_cmd_ = cmd;
        process_ecat_state_ = bus_cycle_result::have_msg;
    }
}
```

```
ecat_data::status_out get_cyclic_status() {
    using ecat_data::as_status;

    // See if we have a new incoming CAN message object
    auto can_msg_in = HardwareLayer::get_can_msg_obj();
    if (can_msg_in) {
        return as_status(toggle_status(ecat_data::status_mode::can_msg),
                         *can_msg_in);
    }

    uint32_t hw_status_word = HardwareLayer::get_debug_status_word();
    ecat_data::status_mode status_mode = ecat_data::status_mode::hw_status1;

    using boost::sml::operator""_s;
    if (ecat_can_sm_.is("safety"_s)) {
        status_mode = ecat_data::status_mode::can_error;
    }

    return as_status(toggle_status(status_mode),
                     app_loop_count_shadow_,
                     hw_status_word);
}
```

```
ecat_data::status_out get_cyclic_status() {
    using ecat_data::as_status;

    // See if we have a new incoming CAN message object
    auto can_msg_in = HardwareLayer::get_can_msg_obj();
    if (can_msg_in) {
        return as_status(toggle_status(ecat_data::status_mode::can_msg),
                         *can_msg_in);
    }

    uint32_t hw_status_word = HardwareLayer::get_debug_status_word();
    ecat_data::status_mode status_mode = ecat_data::status_mode::hw_status1;

    using boost::sml::operator""_s;
    if (ecat_can_sm_.is("safety"_s)) {
        status_mode = ecat_data::status_mode::can_error;
    }

    return as_status(toggle_status(status_mode),
                     app_loop_count_shadow_,
                     hw_status_word);
}
```

```
ecat_data::status_out get_cyclic_status() {
    using ecat_data::as_status;

    // See if we have a new incoming CAN message object
    auto can_msg_in = HardwareLayer::get_can_msg_obj();
    if (can_msg_in) {
        return as_status(toggle_status(ecat_data::status_mode::can_msg),
                         *can_msg_in);
    }

    uint32_t hw_status_word = HardwareLayer::get_debug_status_word();
    ecat_data::status_mode status_mode = ecat_data::status_mode::hw_status1;

    using boost::sml::operator""_s;
    if (ecat_can_sm_.is("safety"_s)) {
        status_mode = ecat_data::status_mode::can_error;
    }

    return as_status(toggle_status(status_mode),
                     app_loop_count_shadow_,
                     hw_status_word);
}
```

machine

```
template <typename HardwareLayer>
struct machine {
  auto operator()() const {
     using namespace boost::sml;
     return make_transition_table(
        // ....
     );
  }

   void reset_all_counters() { /* ... */ }

   void move_to_safety()
   {
      hw_.send_can_msg_obj(slurp_motor::can::motor_off);
      hw_.status_have_error();
   }

   bool safety_enabled_ = true;
   uint32_t tick_count_ = 0;
   uint32_t invalid_count_ = 0;

   HardwareLayer hw_;
};
```

Transition Table

```
 //------------------+-----------------------+------------------+-------------
 //  Source State    | Event [Guard]         | Action           | Dest State
 //------------------+-----------------------+------------------+-------------
*"idle"_s            + event<tick>           / [](){},
 "idle"_s            + event<ecat_can_msg>   / send_can_msg     = "waiting_msg"_s,
 "idle"_s            + event<disable_safety> / safety_off,
 "idle"_s            + sml::on_entry<_>      / enter_idle,

 "waiting_msg"_s     + event<ecat_can_msg>   / send_can_msg     = "waiting_msg"_s,
 "waiting_msg"_s     + event<ecat_idle_msg>  / send_idle_can_msg = "waiting_msg"_s,
 "waiting_msg"_s     + event<invalid_ecat_msg> [too_many_invalid]  = "safety"_s,
 "waiting_msg"_s     + event<invalid_ecat_msg> / send_idle_can_msg = "waiting_msg"_s,
 "waiting_msg"_s     + event<reset>                                = "idle"_s,
 "waiting_msg"_s     + event<tick> [too_many_ticks]                = "safety"_s,
 "waiting_msg"_s     + event<tick>           / increment_tick,
 "waiting_msg"_s     + sml::on_entry<_>      / tick_reset,

 "safety"_s          + event<reset>                                = "idle"_s,
 "safety"_s          + sml::on_entry<_>      / safety_mode
 //------------------  ----------------------+------------------+-------------
```

events

```
namespace ecat_can_machine {
// ---------------------------------
// events
struct tick {};
struct timer_expired {};
struct reset {};
struct ecat_can_msg {
    uint8_t const* data;
};
struct ecat_idle_msg {};
struct invalid_ecat_msg {};
struct disable_safety {};
}
```

guards

```
constexpr auto too_many_ticks =
    [](auto const& /*event*/,
       auto& sm,
       auto const&,
       auto const&)
    {
       return sm.safety_enabled_
          &&  (sm.tick_count_ > sm.max_tick_count_);
    };
```

actions

```
constexpr auto send_can_msg =
    [](auto const& event,
       auto& sm,
       auto const&,
       auto const&)
    {
       sm.hw_.send_can_msg_obj(event.data);
    };
```

entry

```
constexpr auto enter_idle =
    [](auto const& /*event*/,
       auto& sm, auto const&,
       auto const&)
    {
       sm.hw_.send_can_msg_obj(slush_motor::can::motor_off);
       sm.hw_.status_in_idle();
       sm.reset_all_counters();
    };
```

Part V

Pass 3

The Next Step

Protocols made simple

Questions

Questions?