Type-safe modern embedded development with C++

A journey into year-long uptimes

embedded conference scandinavia

Jan Ypma

jan@ypmania.nl

Agenda

- The start of our journey
- Arduino's strengths and weaknesses
- Modern software development
- Embedded techniques
    - Unit testability
    - Continuous integration
    - Interrupt handlers
    - Time constants

A little bit about me

- By day
  - Software architect at Tradeshift, a platform for business interactions
  - 100K+ LoC code bases running on 100+ servers
  - CI/CD, Unit testing, Integration testing, Docker
- By night
  - Electrical engineering
  - Home automation

A journey starts

- Let's automate dimming the lights when watching a movie

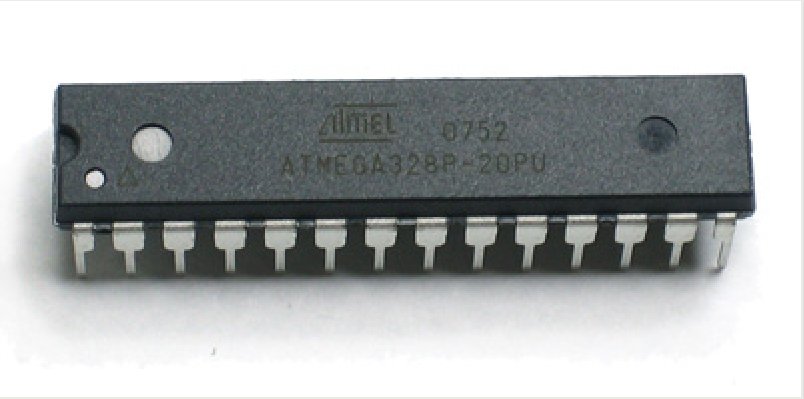
A journey starts

- **Kodi**
  - Media center software with JSON API

- **FS20**
  - Affordable home automation devices
  - Well-documented protocol on the 868MHz band
  - Simple on-off keying

- **JeeNode**
  - ATMega328 microcontroller
  - RFM12B 868MHz radio
  - Arduino software support
  - Existing library to transmit FS20 signals

Introducing our hero



- Atmel (now Microchip) **atmega328p**
    - 32 KB Flash program ROM
    - 2KB RAM
    - 1KB Flash EEPROM
- *"It's an old code, sir, but it checks out."*
    - 10 μA in sleep (~30 μW)
    - 8 mA when awake (~30 mW)

Arduino: blinking an LED

- Seems simple enough

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

Arduino: blinking an LED

- Seems simple enough

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

- However
  - What is `LED_BUILTIN`?
  - `digitalWrite(HIGH, LED_BUILTIN);`
  - Where is `main()`?

Arduino: blinking an LED

- Let's look it up
- In `pins_arduino.h`:

```
#define LED_BUILTIN 13
```

- In `Arduino.h`:

```
#define HIGH 0x1
#define LOW  0x0

#define INPUT 0x0
#define OUTPUT 0x1
```

- So, we're actually saying `digitalWrite(13, 1);`
- Wait, they said this was C++?

Arduino: blinking an LED

- Let's hunt down `main()`
- Ah, in `cores/arduino/main.cpp`:

```
int main(void)
{
  init();
  // [...]
  setup();

  for (;;) {
    loop();
    // [...]
  }
  return 0;
}
```

Arduino: blinking an LED

- And, `wiring.c`:

```
void init()
{
#if defined(TCCR0A) && defined(WGM01)
    sbi(TCCR0A, WGM01);
    sbi(TCCR0A, WGM00);
#endif

#if defined(__AVR_ATmega128__)
    sbi(TCCR0, CS02);
#elif defined(TCCR0) && defined(CS01) && defined(CS00)
    sbi(TCCR0, CS01);
    sbi(TCCR0, CS00);
#elif defined(TCCR0B) && defined(CS01) && defined(CS00)
    sbi(TCCR0B, CS01);
    sbi(TCCR0B, CS00);
#elif defined(TCCR0A) && defined(CS01) && defined(CS00)
    sbi(TCCR0A, CS01);
    sbi(TCCR0A, CS00);
#else
    #error Timer 0 prescale factor 64 not set correctly
#endif

  // 150 more lines of #define and direct register mangling
}
```

The arduino ecosystem

- Works well
  - Libraries available for any hardware you can imagine
  - They generally do work
  - Very broad community with good hardware tips
  - Useable defaults for AVR initialization
- Works not so well
  - AVR initialization isn't customizable
  - Libraries not necessarily work *together* (no HAL, no way to declare interrupt handlers)
  - Libraries have no unit tests
  - The core has no unit tests
- Code basically gets written, tested on hardware, and then "don't touch it"
- RFM12 Arduino library is a good example of the above
- Oh, and no `Makefile` or build system of any kind

We must be able to do better than this!

Introducing AvrLib

- An attempt to increase maintainability of C++ AVR code
- Let's blink an LED, again

```cpp
#include "HAL/Atmel/Device.hpp"
#include "Time/RealTimer.hpp"

using namespace HAL::Atmel;
using namespace Time;

auto LED = ArduinoPinD9();
auto timer0 = Timer0::withPrescaler<1024>::inNormalMode();
auto rt = realTimer(timer0);

int main() {
  LED.configureAsOutputLow();
  while (true) {
    LED.setHigh();
    rt.delay(1_s);
    LED.setLow();
    rt.delay(1_s);
  }
}
```

Type-safe registers

Let's enable the "input capture noise canceller" on timer 1, by setting the `ICNC1` bit:

```
TCCR1A |= (1 << ICNC1);
```

Type-safe registers

Let's enable the "input capture noise canceller" on timer 1, by setting the `ICNC1` bit:

```
TCCR1A |= (1 << ICNC1);
```

In Arduino, these are just numeric macros:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
#define TCCR1A _SFR_MEM8(0x80)

#define ICNC1 7
```

So we just actually wrote

```
  *(volatile uint8_t *)(0x80) |= (1 << 7);
```

Type-safe registers

Let's enable the "input capture noise canceller" on timer 1, by setting the `ICNC1` bit:

```
TCCR1A |= (1 << ICNC1);
```

In Arduino, these are just numeric macros:

```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
#define TCCR1A _SFR_MEM8(0x80)

#define ICNC1 7
```

So we just actually wrote

```
  *(volatile uint8_t *)(0x80) |= (1 << 7);
```

However, it turns out `ICNC1` is *actually* bit 7 on `TCCR1B`, **not** `TCCR1A`. Oops.

Type-safe registers

Let's try this again:

```
TCCR1A |= ICNC1;
// compile error: no match for ‘operator|=’
TCCR1B |= ICNC1;
// compiles fine
```

C++ operators and a bit of code generation to the rescue

```
using TCCR1A_t = Register8<0x80,
          ReadWriteBit,
          ReadWriteBit,
          ReservedBit,
          ReservedBit,
          ReadWriteBit,
          ReadWriteBit,
          ReadWriteBit,
          ReadWriteBit>;
constexpr StaticRegister8<TCCR1A_t> TCCR1A = {};
constexpr TCCR1A_t::Bit0 WGM10 = {};
constexpr TCCR1A_t::Bit1 WGM11 = {};
constexpr TCCR1A_t::Bit4 COM1B0 = {};
constexpr TCCR1A_t::Bit5 COM1B1 = {};
constexpr TCCR1A_t::Bit6 COM1A0 = {};
constexpr TCCR1A_t::Bit7 COM1A1 = {};
```

Type-safe pins

- Not all pins are created equal
  - **ATMega328**: Each pin has unique alternate functions
  - **STM32F030**: Alternate functions can sometimes be configured, but on fixed pin(s)

```
auto LED = ArduinoPinD9();
auto timer0 = Timer0::withPrescaler<1024>::inNormalMode();
auto rt = realTimer(timer0);

int main() {
  LED.configureAsOutputLow();
  while (true) {
    LED.setHigh();
    rt.delay(1_s);
    LED.setLow();
    rt.delay(1_s);
  }
}
```

- Pins are type-safe
  - Different AVR pins have different features, and each pin has its own class
  - Doing `myPin.comparator().setTarget(15)` on a non-PWM pin is a compile error

Type-safe time handling

```
auto LED = ArduinoPinD9();
auto timer0 = Timer0::withPrescaler<1024>::inNormalMode();
auto rt = realTimer(timer0);

int main() {
  LED.configureAsOutputLow();
  while (true) {
    LED.setHigh();
    rt.delay(1_s);
    LED.setLow();
    rt.delay(1_s);
  }
}
```

- Time is type-safe
  - Converting timer units to real time units is externalized to `RealTimer`
    - And, hence, can be unit tested
  - `RealTimer` works on any timer, and any prescaler
  - Conversion factors are compile-time known, so `delay(1_s)` compiles down to a constant
  - Compiler error if using too small or too large time constants

Note: Prefer using `periodic` or `deadline` instead of `delay`.

Encapsulation and testability

- Having all your code in `main.cpp` makes it kinda hard to unit test
- Write a class instead for your app

```cpp
#define auto_var(name, expr) decltype(expr) name = expr

template<typename led_t, typename timer_t>
class Blink {
  led_t * const LED;
  timer_t * const timer;

  auto_var(rt, realTimer(*timer));
public:
  Blink(led_t &l, timer_t &t): LED(&l), timer(&t) {
    LED->configureAsOutputLow();
  }

  void loop() {
    LED->setHigh();
    rt.delay(1_s);
    LED->setLow();
    rt.delay(1_s);
  }
};
```

Interrupts

- Original problem: interrupt handlers are global-scope `"C"` style functions in `avr-gcc`
- Solution: framework takes ownership of these handlers, delegating to user class member functions
  - A bit of macro, a lot of `type_traits`

```
class MyApp {
  auto_var(button, ArduinoPinD8());
  void onButton() { /* handle button press */ }

public:
  typedef On<MyApp, typename button::INT, &MyApp:onButton> Handlers;
  void loop() { /* main application loop */ }
};

RUN_APP(MyApp)  // declares main() and interrupt handlers
```

- Handlers are known at compile time, so optimizer can fully inline them
- Handlers can be composed, e.g.

```
typedef On<MyApp, typename button::INT, &MyApp:onButton,
        Delegate<MyApp, decltype(blink), &MyApp::blink>> handlers;
```

Testing at any level

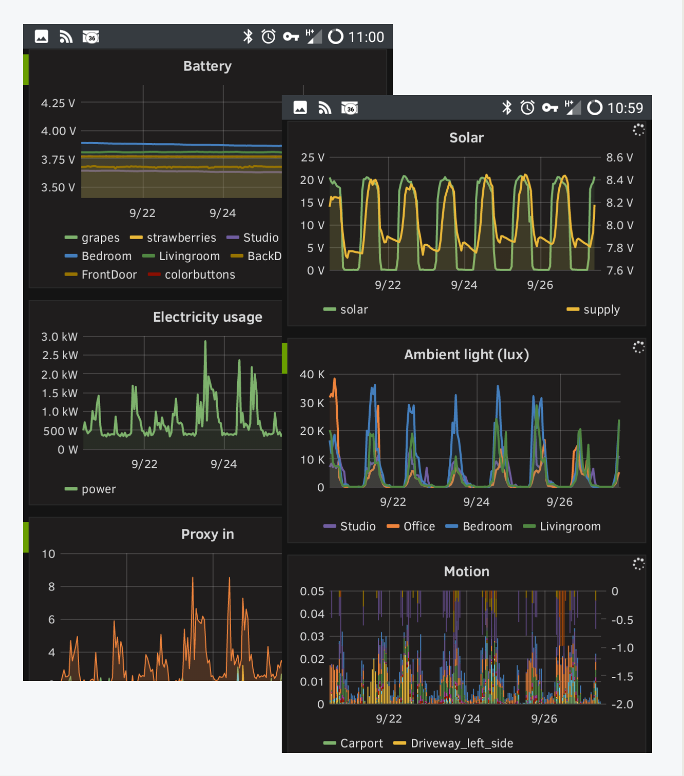
- Let's take a closer look at `RealTimer`
  - Unit testable, since there are no direct dependencies on avr-libc
  - Tests using Google Test
  - Implementation

Continuous integration

- GCC (and avr-gcc) is a particularly troubled piece of software
  - Most major upgrades I've tried hit internal compiler errors.
    - 7.1.1: 81074
    - 9.2.1: 91925
  - Currently, avr-gcc 5.4.0, 7.2.0 and 8.3.0 build correctly. 9.2.1 has issues, promised fixed in 9.3
- Solution: docker container with working version
- Build `AvrLib` on Travis CI using `Makefile`: build passing

Status

- Powering about 100 devices: door sensor, doorbell, room sensor, heating
- Tests pay off: if devices fail, it's usually hardware
- Streams library with Protobuf support
- Drivers for RFM12B radio, ESP8266 in AT mode, RS-232, IR decoding, temperature sensors, and more
- Future work
  - Integrate `AvrLib` into platformio
  - Move to ARM and/or Rust?



Source: https://github.com/jypma/AvrLib/

Demos: https://github.com/jypma/AvrLibDemo/tree/master/apps

embedded conference scandinavia