Mini-Mapper Software 1: Motor prototype board

Ian Ross

November 30, 2020

These notes describe software development for the motor prototype board for the Mini-Mapper robot. This is intended as a platform to test motor driver and motor encoder setup, and to develop motor early control algorithms (particularly constant-speed PID control and soft start).

I'm intending to develop all the software for the Mini-Mapper as "bare metal" C++. Although I'll probably end up using an RTOS later on (almost certainly FreeRTOS), I'm going to start with a simple event loop without an RTOS. I'll use a Nucleo board with an STM32F767ZI microcontroller on it for development experiments, and will choose a suitable STM32 processor to put on the final Mini-Mapper boards based on experiences using the Nucleo board. I'm not going to use any HAL libraries from ST Micro because I'm interested in writing device drivers myself.

The basic requirements I want to follow are:

- **Language** Firmware written in C++, maybe doing some experiments with Rust later on.
- **Libraries** Use only low-level CMSIS hardware definitions, writing my own C++ HAL.
- **Build system** Use Make for building, possibly adding Meson on top of this later on.
- **IDE** Use Emacs as an IDE, setting up the necessary tools for comfortable development.
- **Communications** Develop a command shell to run on the Nucleo board, talking to a PC over a USB serial port. This will make it easy both to do ad hoc experiments directly talking to the shell with Minicom, but also to write GUI front-ends for experiments and data collection using Python GTK or something similar. (This is the approach I used for my Teensy Load project, and I find it really useful.)
- **STM32 features** Clocks; power; USB serial (USART + DMA + interrupts); GPIOs and timers for PWM motor control; ADC for motor coil current sensing; GPIO input, interrupt and timer for motor encoder pulse detection.

I want to cover the following topics in this prototyping phase:

1. Platform setup: clocks, power, etc.
2. CMSIS familiarisation.
3. Blinky: platform test, clock setup, GPIO setup.
4. USART setup with interrupts for RX and DMA for TX.
5. USB serial command shell.
6. PWM demo: more GPIO setup, timer setup, USB serial control, view output using AD2.
7. First pass experiment GUI: commanding of motor state (on/off, forward/backward, PWM duty cycle), monitoring of motor coil current and motor encoder pulses; data saving; real-time graphing.
8. Encapsulation of motor abstractions: `Motor::Driver` for PWM driver, `Motor::Torque` for motor coil current measurement for torque estimation, `Motor::Encoder` for the encoder disk measuring motor rotation and `Motor::Controller` for the overall control of all motor functions.
9. Constant speed PID control, including independent optical speed measurement.
10. Soft start/ramp control.

This should be enough for the first "direct control" milestone.

1 Initial setup

1.1 Platform and tools

- I'm using the `arm-none-eabi` version of GCC, which is pretty standard.

1.2 CMSIS

- The only files taken from the ST Micro-supplied code are the header files `stm32f767xx.h`, `stm32f7xx.h` and `core_cm7.h` (and some compiler support files that they include), the startup file `startup_stm32f767xx.s`, the linker script `STM32F767ZITx_FLASH.ld`, and `system_stm32f7xx.c`, which is a small file including a couple of support functions to help with clock setup.
- The header files provide register-level access to all the core functionality of the Cortex-M7 processor (`core_cm7.h`) and the peripherals provided on the STM32F767ZI (`stm32f767xx.h` and `stm32f7xx.h`).

- The `core_cm7.h` also provides some basic functions for dealing with some core processor features, like cache control, the interrupt controller and the SysTick timer.
- I'm basically going to use anything in these files freely, and I'm happy to look at examples in the STM32 HAL code to work through any problems I have with my own HAL code, but I'm going to write more or less everything starting at the bare metal "write to registers" level, since this seems like the best way to get a good understanding of how things really work.

1.3 Blinky

2 Software architecture

3 USB serial shell

3.1 USART setup

3.2 Command shell

4 PWM demonstration

5 Experiment front-end GUI

6 Microcontroller setup

The Nucleo board has an STM32F767ZI processor on it.

6.1 Motor driver

- The Toshiba motor driver has an application note where they seem to suggest using $f_{\mathrm{PWM}} = 30\,\mathrm{kHz}$.
- For a 10-bit resolution PWM duty cycle, this gives a clock frequency of $1024 \times 30\,\mathrm{kHz} = 30.72\,\mathrm{MHz}$.
- The clock input on Nucleo board is 8 MHz by default. To use this, the STM32 should be set to run on the HSE clock (High-Speed External) with external bypass (`RCC_CR:HSEBYP` = 1, `RCC_CR:HSEON` = 1).
- The main clock PLL should be set up with `RCC_PLLCFGR:PLLSRC` = 1 (HSE), `RCC_PLLCFGR:PLLM` = 4 (VCO input: 2 MHz), `RCC_PLLCFGR:PLLN` = 128 (VCO output: 256 MHz), `RCC_PLLCFGR:PLLP` = 4 (PLL output: 64 MHz), and enabled with `RCC_CR:PLLON` = 1.

- The APB prescalers need to be set to make the low-speed prescaler give a frequency of less than 45 MHz: `RCC_CFGR:PPRE1` = 4 (divide by 2 to give 32 MHz).
- The system clock needs to be set to use the PLL: `RCC_CFGR:SW` = 2.
- Using a 64 MHz system clock and a 11-bit resolution PWM duty cycle yields a PWM frequency of 31.25 kHz.

6.2 Motor torque measurement

6.3 Motor encoder

The motor encoder uses a photointerrupter and an encoder disk on the motor shaft. The motor shaft rotation rate is reduced by a 48:1 gearbox to drive the wheels.

Here are some basic questions and notes on the software approach for capturing the edge transitions from the encoder.

How to do edge capture? Can you do capture on multiple pins using the same timer? Can you do both rising and falling edges at the same time?

- Timers with input capture are basically all timers except TIM6 and TIM7 basic timers.
- Timers with multiple input capture channels are TIM9, TIM12 (2 channels); TIM2, TIM3, TIM4, TIM5 (4 channels); TIM1, TIM8 (4 channels).
- Typical times we'll want to measure:
  - Wheel turning at 90 rpm (max, equivalent to 28.3 cm/s linear speed) => encoder turning at 90 x 48 = 4320 rpm = 72 Hz.
  - Disk has 12 teeth => tooth frequency = 72 x 12 = 864 Hz.
  - If teeth/gaps are same size, edge freq. = 864 x 2 = 1728 Hz. (Teeth and gaps aren't same size, but this is indicative.)
  - So time between edges will be 578.7 s.
- If we have a timer with a prescaler set to give a count frequency of 1 MHz, then a 16-bit counter will be able to measure times of up to 65.536 ms. A 32-bit counter will be able to measure times of up to 4294.97 s (i.e. 1 hr 11 min 34.97 s).
- At 1 mm/s linear speed, we see 1 / (60) * 48 * 12 * 2 = 6.11 edges per second, i.e. an inter-edge time of 163.6 ms. Seems like a 32-bit timer might be easier in this case...
- If we use a 32-bit timer, we can also reduce the prescaler to get a higher count frequency and corresponding better timing accuracy.

- The 32-bit timers are TIM2 and TIM5, both of which have four capture channels.
- GPIO inputs for capture channels have to be assigned as specific alternate functions. There seem to be enough redundant assignments to make this easy enough to do.
- Can either get an interrupt on the capture event, or get the capture timer count transferred via DMA and get an interrupt on DMA transfer complete. (For our purposes, an interrupt on each capture event is probably simplest to deal with, because we need to look at the GPIO level to know whether we have a rising or falling edge. In the ISR for the capture, we can check the GPIO level, bundle that up with the capture count and the channel we're looking at, and emit an event carrying all that information for later processing.)
- Capture edge polarity is controlled by the CCxP and CCxNP fields in the TIMx->CCER register: setting both of these bits captures both rising and falling edges.

What information do we want to make available? Ultimately we want to provide a speed measurement, possibly with different kinds of smoothing (none: latest instantaneous value, boxcar smoothing over sample time, exponentially weighted smoothing?), but we also probably want access to raw time values as well, i.e. times of the last N edge transitions.

- For a single encoder, we expect to get a sequence of alternating rising and falling edges. We'll store the edge type (rising or falling) and the timer count when the edge occurred. We'll keep the last N edge times, retiring eges when they get too old or we need to reuse storage space.
- We want to be able to retrieve time differences between edges and to convert them into rotation rates and linear speeds, both on an instantaneous basis and smoothed over the collected data (for the last 500 ms or something similar?).
- To start with, let's provide functions to retrieve time differences between adjacent edges (i.e. of opposite polarity) and between adjacent edges of the same polarity.

What sort of time resolution do we need? And what to do about the really slow cases when the wheel is more or less stopped? Do we need some sort of rollover counter to time longer periods?

- A single edge corresponds to 60 / 48 / 12 / 2 = 0.16 mm (assuming symmetry of rising and falling edges).

- Using a 32-bit timer running at 1 MHz, we can get 1 s resolution for each edge. At maximum speed (90 rpm for the wheel, i.e. 60 * 1.5 = 282.74 mm/s), 1 s corresponds to a linear distance of 0.28 m. At lower speeds, the linear distance resolution is correspondingly finer.
- As noted above, with a timer count frequency of 1 MHz, a 32-bit counter is able to measure times of up to 4294.97 s (i.e. 1 hr 11 min 34.97 s).
- Both of these factors indicate that a 1 MHz 32-bit timer is pretty huge overkill... So let's go with that then!