Centrale Nantes

MAC : 4^{th} Lab's Report
Timers — Stepper Motor control

1^{st} year Embedded Systems Engineering

By

EL KHAYDER Zakaria

SAOUTI Rayan

Professor

BRIDAY Mikael

December 07, 2023

Session

2023-2024

Made with LaTeX

Contents

List of Tables

Listings

1 Counter-Clockwise Rotation

1.1 Setup

```
void setup()
{
    // Setup Motor Output pins
    for (int i = 5; i <= 8; i++)
        pinMode(GPIOA, i, OUTPUT);
}
```

Listing 1: Motor output setup

1.2 Single step Counter-Clockwise

```
#define SEQ_LEN sizeof(seq) / sizeof(*seq)

void stepCCW()
{
    static uint8_t index = 0; // Static variable to preserve its state
    const unsigned char seq[] = {8, 0xC, 4, 6, 2, 3, 1, 9};

    GPIOA->ODR &= ~(0b1111 << 5); // Force all the 4 pins to 0
    GPIOA->ODR |= seq[index] << 5; // Set the appropriate bits to 1

    index++;
    index %= SEQ_LEN; // reset on 8
}
```

Listing 2: stepCCW()

1.3 Using a timer

1.3.1 Setup

We should start by setting up the timer, for this lab, we chose to use `TIM6`.
We chose a pre-scalar of 64000(-1), giving us an increment each 1 ms (1KHz). The chosen `PSC` value makes calculating the `ARR` easier with the simple equation:

$$\texttt{ARR} = \frac{1000}{f} - 1$$

In our first run, we take $f = 10$, but the equation is available for any equation between 1 and 999. We can inline this equation directly in code or as C++ Macro. We chose the latter. While at it, we might define a `Timer` const for `TIM6`.

```
#define Timer TIM6
#define ARR(f) 1000.0 / f - 1
```

For the rest of the setup function, we just copied (stole) it directly from the presentation slides.

```
void setup()
{
    ...

    // input clock = 64MHz.
    RCC->APB1ENR |= RCC_APB1ENR_TIM6EN;
    // reset peripheral (mandatory!)
    RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_TIM6RST;

    // Configure timer
    Timer->CNT = 0; // Reset CNT
    Timer->SR = 0; // Reset flags
    Timer->PSC = 64000 - 1; // Prescalar
    Timer->ARR = ARR(10);
    Timer->CR1 = 1; // Start the timer
}
```

Listing 3: Timer setup function

1.3.2 Loop

With the timer set up, we can now continuously check if the 0th bit of the `SR` flag is raised, if that is the case, we single-step the motor and reset the said flag bit.

```
int main(void)
{
    setup();

    while (1)
    {
        if (Timer->SR & 1U)
        {
            Timer->SR = 0; // Reset timer overflow flag
            stepCCW();
        }
    }

    return 0;
}
```

Listing 4: 10HZ Counter-clockwise stepping loop

1.4 Ditching the ODR and welcoming the cool BSRR

As you may have already noticed, we can't force both 1's and 0's using masking operations on the `ODR` register in a single operation. That's why, a group of people, way more smarter than us, created the `BSRR` register to bypass this limitation. With some tweaks on the `stepCCW` function, we can use the same already-defined `seq` for the `BSRR`.

```
void stepCCW()
{
    ...

//  GPIOA->ODR &= ~(0b1111 << 5); // Force all the 4 pins to 0
//  GPIOA->ODR |= seq[index] << 5; // Set the appropriate bits to 1
    GPIOA->BSRR = seq[index] << 5 | ~seq[index] << (16 + 5);

    ...
}
```

Listing 5: Migration from ODR to BSRR

2 Clockwise rotation

2.1 Single step Clockwise

This is easy, we just copy the same already defined function (`stepCCW`) and just decrement the index instead of incrementing it.

```
void stepCW()
{
    static uint8_t index = 0;
    const unsigned char seq[] = {8, 0xC, 4, 6, 2, 3, 1, 9};

    GPIOA->BSRR = seq[index] << 5 | ~seq[index] << (16 + 5);

    if (index == 0)
        index = SEQ_LEN - 1;
    else
        index--;
}
```

Listing 6: stepCW()

2.2 Flippity-Floppity-Floop

We can reuse the same code from the previous lab to get the Button state, and we can reverse the rotation direction accordingly.

2.2.1 Issue

Now that we have two separate functions, we need a way to share the current rotation command index, to avoid distant jumps of the motor. We found that creating a global variable named `RotationIndex` is the most straightforward.

2.2.2 Reading the button state

```
#define Button GPIOB, 1

void setup()
{
    ...

    pinMode(Button, INPUT_PULLUP);
}
```

Listing 7: Button setup

```
enum class ButtonState
{
    Released,
    Pushing,
    Pushed,
    Releasing,
};

ButtonState getButtonState()
{
    static ButtonState state = ButtonState::Released;

    switch (state)
    {
    case ButtonState::Released:
        if (digitalRead(Button) == 0) // The button is active low
            state = ButtonState::Pushing;
        break;

    case ButtonState::Pushing:
        state = ButtonState::Pushed;
        break;
```

```
23
24      case ButtonState::Pushed:
25          if (digitalRead(Button) == 1)
26              state = ButtonState::Releasing;
27          break;
28
29      case ButtonState::Releasing:
30          state = ButtonState::Released;
31          break;
32      }
33
34      return state;
35  }
```

Listing 8: Button state

2.2.3 Loop

We update the main loop to check if the button is clicked, it rotates clockwise if that is the case, counter-clockwise otherwise.

```
1  int main(void)
2  {
3      ...
4
5  //  stepCCW();
6      if(getButtonState() == Button::Pushed)
7          stepCW();
8      else
9          stepCCW();
10     ...
11 }
```

Listing 9: Rotation direction based on button state

3 Rotation Speed

3.1 Reading the potentiometer

Thanks to the provided `adc.h/c`, the task is easy enough.

```
1  void setup()
2  {
3      ...
4
5      ADCInit(); // Setup ADC
```

```
}
```

```
int main(void)
{
    ...

    uint16_t value = ADCRead();
}
```

Moving along...

3.2 Updating rotation frequency

The potentiometer value is coded on 12 bits, meaning it goes from 0 to 4095. We can use a simple mathematical formula to map this range from 0/4095 to 10/500

$$f = (500 - 10) \cdot \frac{v}{4096} + 10$$

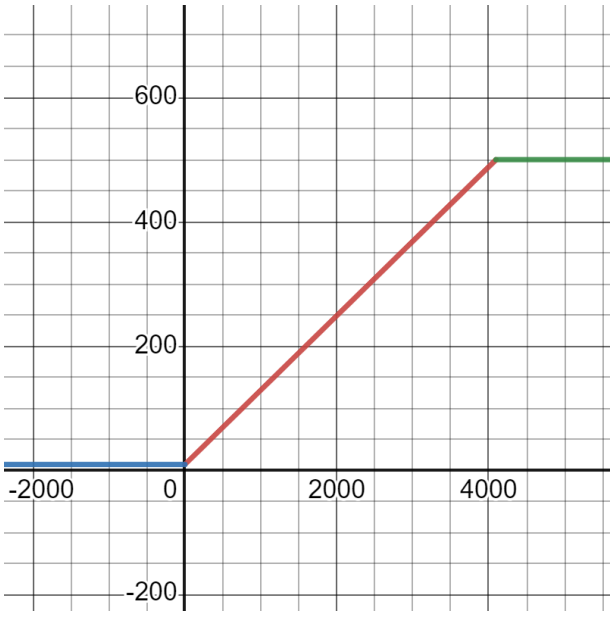

Figure 1: Rotation frequency (f) $\propto$ Potentiometer value (v)

This can be easily translated to code in a single line

```
int main(void)
{
    ...

    while(1)
    {
```

```
        ...

        Timer->ARR = ARR(10. + (500. - 10.) * (float)ADCRead() / 4096.);
    }

    ...
}
```

Listing 10: f $\propto$ v

3.3 Flippity-Floppity-Floop: The sequel

We will need a way to save the current rotation direction, and then instead of checking if the button is `Pushed`, we check if it is `Pushing`.

```
enum class Direction
{
    Clockwise,
    CounterClockwise
};

int main(void)
{
    ...

    Direction direction = Direction::Clockwise;


    while(1)
    {
        if (getButtonState() == ButtonState::Pushing)
        {
            // Invert rotation direction
            if (direction == Direction::Clockwise)
                direction = Direction::CounterClockwise;
            else
                direction = Direction::Clockwise;
        }

        if (Timer->SR & 1U)
        {
            ...

            if (direction == Direction::Clockwise)
                stepCW();
            else // direction == Direction::CounterClockwise
                stepCCW();

```

```
            ...
        }
    }

    ...
}
```

4 Tour de France

There are 64 steps for each round, but there is also a reduction factor of 64. Meaning, that for each rotation, we should step $64 \cdot 64 = 4096$ steps.

We will need a way to keep track of the remaining steps to do. We will create a variable that we increment each time we do a step, and once it reaches the max amount (4096), we do not execute the stepping code.

Now each time we click the button, besides only inverting the rotation direction, we should also reset the steps counter to 0

```
#define FullRoundSteps 64 * 64

int main(void)
{
    ...

    int rotatedSteps = 0;

    while (1)
    {
        ...

        if (getButtonState() == ButtonState::Pushing)
        {
            // Reset remaining steps
            rotatedSteps = 0;

            ...
        }

        if (Timer->SR & 1U)
        {
            ...

            if (rotatedSteps < FullRoundSteps)
            {
                ...

```

```
                rotatedSteps++;
            }
        }
    }

    return 0;
}
```

Listing 11: What goes around... comes around

Resources

The code files, and this report's source code, are available on this GitHub repository: elkhayder/sec1-tp-mac