به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

گزارش تمرین ششم ابزار دقیق

رضا مومنی

810199497

Q1

به طور کلی برنامه نویسی میکروکنترلر ها به سه دسته زیر تقسیم می شود:

1. Bare metal
2. RTOS
3. OS

سیستم عامل‌های بلادرنگ RTOS (Real Time Operating System) انواع مختلفی دارند، از این بین سیستم عامل بلادرنگ FreeRTOS سیستم عاملی پر سرعت، بهینه، قابل اجرا بر روی میکروکنترلرهای ARM، نظیر STM32, LPC :و...، حتی میکروهای ساده 8 بیتی نظیر AVR :و پردازنده‌های بسیار زیاد دیگر است. در دنیای میکروکنترلرها و امبدد سیستم‌ها(Embedded System) ، سیستم عامل‌های بلادرنگ یک پاسخ قطعی برای پروژه‌های پیچیده امروزی و پروژه‌هایی که نیاز به انجام چند کار به صورت همزمان دارند هستند.

برنامه‌نویسی میکروکنترلرها به روش‌های متفاوتی صورت می‌گیرد. روش مرسوم Bare Metal است که تحت عنوان Superloop نیز شناخته می‌شود. این روش برای پروژه‌های نسبتا کوچک که وظایف محدودی دارند مناسب است، اما زمانی که با پروژه‌های بزرگتر و پیچیده‌تر که قرار است مداوم توسعه داده شوند و لازم است چند کار را به صورت همزمان انجام دهند مواجه هستیم و همینطور زمانی که پاسخ زمانی سیستم باید تضمین شود، استفاده ازRTOS ها بهترین و کم‌هزینه‌ترین راه ممکن برای برنامه‌نویسی میکروکنترلر و پیاده‌سازی پروژه است. در این روش برنامه بلوکه و ماژولار (Modular) می‌شود و هر بخش می‌تواند به صورت مستقل توسعه داده شود، به این ترتیب حتی چند برنامه‌نویس به صورت همزمان می‌توانند روی یک پروژه کار کنند، برنامه‌نویس درگیر لایه سیستم عامل نمی‌شود و صرفا به پیاده‌سازی کاربرد مورد نیاز پروژه می‌پردازد، همینطور انتقال برنامه به پردازنده‌ای دیگر راحت‌تر می‌شود، علاوه بر این زمان ارائه پروژه به بازار کاهش می‌یابد، به این مفهوم که محصول نهایی رقابت‌پذیرتر شده و شانس موفقیت پروژه در بازار بیشتر می‌شود.

روش برنامه‌نویسی RTOS امکان استفاده بهینه از توان پردازنده را فراهم می‌کند و روز به روز بیشتر مورد توجه شرکت‌هایی که کار طراحی انجام می‌دهند قرار می‌گیرد. با توجه به پروژه‌های پیچیده و گسترده امروزه که به طور عمده در حوزه IoT (Internet of Things) هستند، برنامه‌نویسی RTOS یک پاسخ قطعی است. به عنوان کاربردهای دیگر، پروژه‌هایی که شامل رابط گرافیکی هستند و با اینترنت تبادل اطلاعات دارند، به طور عمده با RTOS ها پیاده‌سازی می‌شوند.

مزیت ها:

- خوانایی برنامه را افزایش می دهد
- بهبود ثبات کد
- به روز رسانی برنامه را ساده تر می کند
- این باعث افزایش سرعت فرآیند برنامه نویسی می شود
- استفاده کارآمد از قدرت پردازنده
- کنترل بهینه مصرف انرژی پردازنده
- پروژه ها را به وظایف مجزا و مستقل تقسیم کنید
- کار را می توان به طور جداگانه و مستقل اعمال کرد
- اجرای قسمت های مختلف برنامه تاثیر چندانی بر یکدیگر ندارند
- امکان کار تیمی
- برنامه های کاربردی بلادرنگ را تضمین می کند.
- این در برنامه های LAN و انتقال داده از طریق اینترنت مهم است
- این در برنامه هایی که با نمایشگرهای LCD گرافیکی و لمسی کار می کنند و همچنین در طراحی رابط کاربری مهم است.

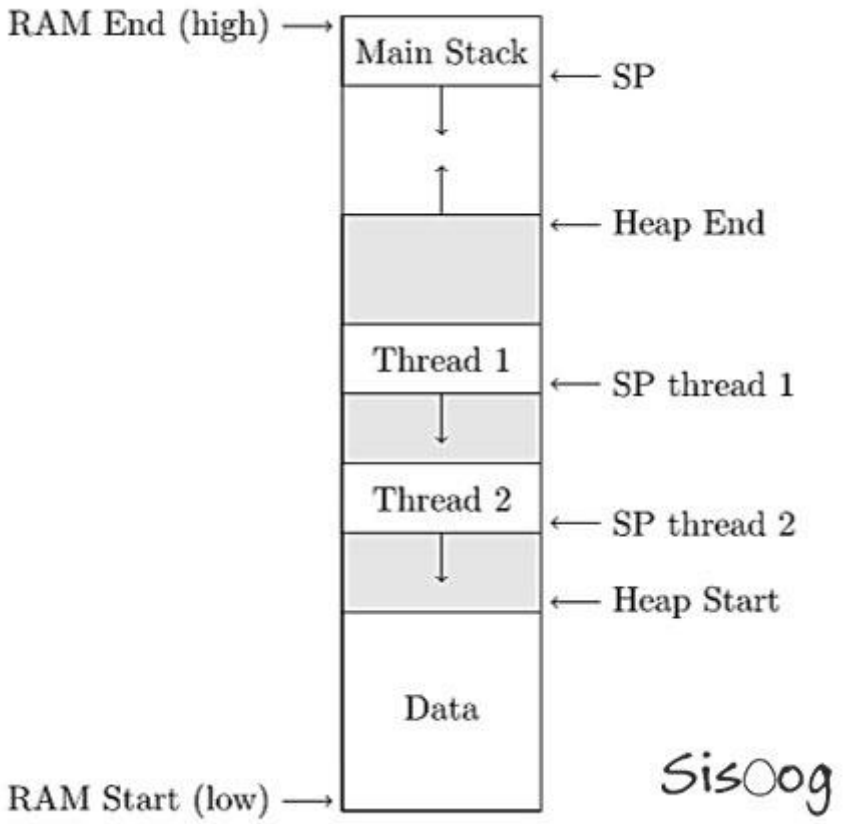
مزیت ها واضح هستند اما اگر بخوام مثال بزنم میشه گفت که زمانی که یک بخش از سیستم به درستی کار نکنه بخش دیگه به کار خودش ادامه میده یا اگر پروژه ای بزرگ باشه و نیازمند این هستیم که بدون اطلاعات زیادی از بخش های دیگه کار خودمونو جلو ببریم میشه از این سیستم عامل استفاده کرد. همچنین سیستم عامل هایی مثل ویندوز realtime نیستند و لینوکس هم برای خیلی از پردازنده ها سنگینه پس این سیستم عامل میتونه در عین سبک بودن آنی و بدون درنگ بودن رو تحقق ببخشه.

FreeRTOS

FreeRTOS فقط از یک زمانبندی رشته ای و یک TCP/IP تشکیل شده است و واقعاً حافظه مجازی ندارد. همچنین هیچ سیستم فایل و مدل امنیتی وجود ندارد. شما هسته FreeRTOS (بخش مرکزی) را مستقیماً در برنامه بارگذاری شده در MCU ایجاد می کنید.

انواع مختلفی از سیستم عامل های بلادرنگ (RTOS) وجود دارد که در میان آنها سیستم عامل بلادرنگ FreeRTOS یک سیستم عامل پرسرعت بهینه سازی شده است که می تواند بر روی میکروکنترلرهای ARM مانند STM32 و LPC پیاده سازی شود. میکروکنترلرهای ساده 8 بیتی مانند غیره، AVRو بسیاری از پردازنده های دیگر. در دنیای میکروکنترلرها و سیستم های تعبیه شده، سیستم های کنترل بلادرنگ پاسخی آشکار به پروژه های پیچیده و چند وظیفه ای امروزی هستند.

در برنامه‌های تکThread ، داده‌های برنامه و پشته اصلی نیز در ابتدا و انتهای RAM قرار دارند. با اجرای برنامه، پشته با رفتن به تابع‌ها به پایین رشد می‌کند و با خروج از تابع‌ها به بالا فشرده می‌شود. در برنامه‌های چندThread ی هرThread ، حافظه پشته‌ی خود را در RAM دارد. در شکل زیر این را می بینیم.



ناحیه داده در هنگام لینک کردن برنامه به شکل ایستا اختصاص می‌یابد. این ناحیه شامل متغیرهای عمومی و ایستاست. ناحیه بالای این ناحیه برای اختصاص دهی پویا به کار می‌رود. این ناحیه را ناحیه heapمی‌نامند و توسط تخصیص‌دهنده حافظه گسترش می‌یابد. پشته اصلی در انتهای حافظه قرار دارد و به پایین رشد می‌کند. در FREERTOS پشته‌هایThread ها به شکل بلوک‌هایی در ناحیه heapقرار دارند. آزمون‌هایی وجود دارد تا مطمئن باشیم که در هنگام اجرا ریسمان‌ها، فراتر از فضایی که در اختیار دارند، نروند.

در این سیستم عامل بهThread ها، Taskگفته می‌شود. یک وظیفه یک حلقه بی پایان است که کاری را انجام می‌دهد. به عبارتی تابع Task، بازگشتی ندارد.

```
void threadFunction(void *params){
  while(1) {
    // do something
  }
}
```

برای اجرای Task، باید آن را توسط یک تابع خلق کرد. برعکس کدهای معمولی که یک تابع، با فراخوانی اجرا می‌شود، در سیستم عامل، Taskبا ساختن، اجرا می‌شود. تابعی که فرایند ساختن یک Taskرا انجام می‌دهد به شکل زیر است.

```
portBASE_TYPE xTaskCreate(
       pdTASK_CODE pvTaskCode,
       const portCHAR * const pcName,
       unsigned portSHORT usStackDepth,
       void *pvParameters,
       unsigned portBASE_TYPE uxPriority,
       xTaskHandle *pvCreatedTask
);
```

در این تابع پارامترها به ترتیب عبارتند از نام تابع-Task نامی که به Task می‌دهیم) -مثلا در بالا نام آن ThreadFunction بود(، اندازه پشته، اشاره‌گر به پارامترهای انتخابی، اولویت و دستگیره ی Taskساخته شده.

در FreeRTOS یک Task به نام بیکار نیز وجود دارد که وقتی هیچ Task دیگری برای اجرا نباشد اجرا می‌شود. این Task کمترین مقدار الویت را دارد. ساختار کلی برنامه‌ای که چندThread ی نوشته شده باشد به شکل زیر است.

```
main() {

  // include misc.h

  NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );

  // Initialize hardware
     ...
  // Create tasks

  xTaskCreate( ... );
     ...
  // Start Scheduler

  vTaskStartScheduler();
}
```

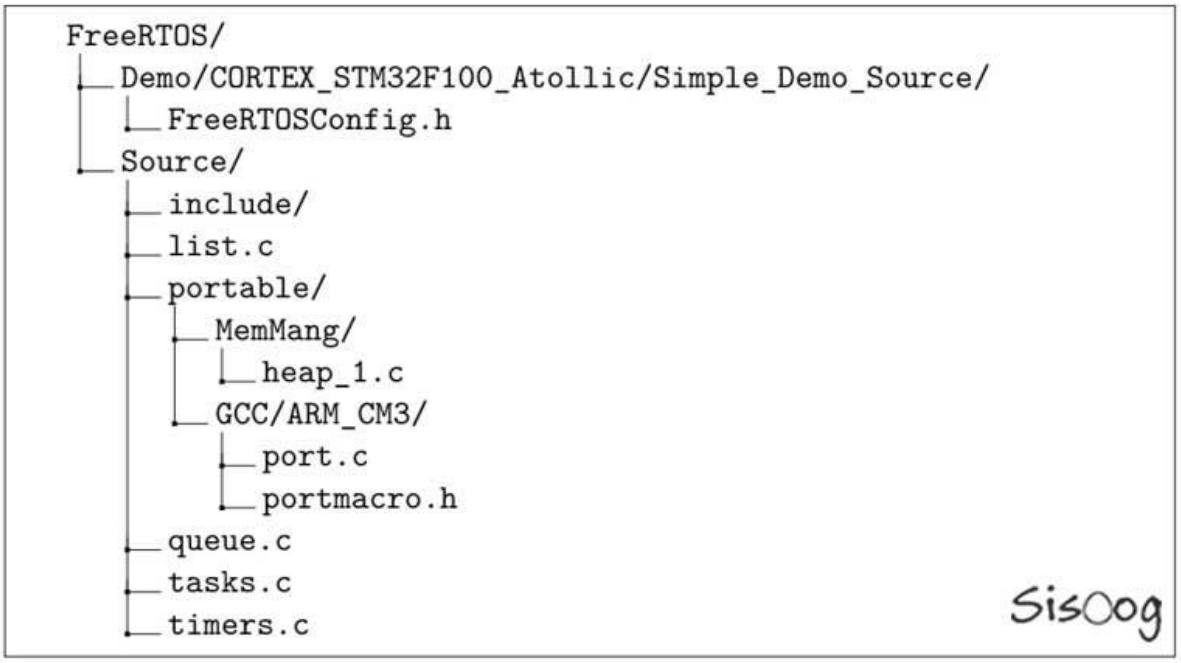
همان‌طور که دیده می‌شود ابتدا آغازسازی سخت افزار انجام می‌شود. برای راه اندازی این سیستم عامل روی STM32 باید وقفه‌های NVIC را با ۱۶ اولویت و بدون زیر الویت آغازسازی کرد. سپس وظایف ساخته می‌شود و در نهایت تابع زمان‌بند شروع به کار می‌کند. این تابع هیچ بازگشتی ندارد، اگرچه تابعی در API برای خروج از آن وجود دارد.

یک برنامه نمونه با دو Thread در شکل زیر دیده می‌شود. برنامه دو Thread دارد که دو LED را چشمک زن می‌کنند:

```
static void Thread1(void *arg) {
  int dir = 0;
  while (1) {
    vTaskDelay(300/portTICK_RATE_MS);
    GPIO_WriteBit(GPIOC, GPIO_Pin_9, dir ? Bit_SET : Bit_RESET);
    dir =  1 - dir;
  }
}

static void Thread2(void *arg) {
  int dir = 0;
  while (1) {
    vTaskDelay(500/portTICK_RATE_MS);
    GPIO_WriteBit(GPIOC, GPIO_Pin_8, dir ? Bit_SET : Bit_RESET);
    dir =  1 - dir;
  }
}

int main(void)
{
  // set up interrupt priorities for FreeRTOS !!

  NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );

  // initialize hardware

  init_hardware();

  // Create tasks

  xTaskCreate(Thread1,                    // Function to execute
        "Thread 1",                 // Name
        128,                        // Stack size
        NULL,                       // Parameter (none)
              tskIDLE_PRIORITY + 1 ,      // Scheduling priority
        NULL                        // Storage for handle (none)
        );
  xTaskCreate(Thread2, "Thread 2",  128,
        NULL, tskIDLE_PRIORITY + 1 , NULL);

  // Start scheduler

   vTaskStartScheduler();

  // Schedule never ends

}
```

برای دانلود این سیستم عامل می‌توانید به این آدرس مراجعه کنید. با نصب نرم افزارKail ، فایل‌های headerاین سیستم عامل نیز در پوشه‌های آن موجود است. راه دیگر استفاده از نرم افزار CubeMX است که به شکل گرافیکی تنظیمات اولیه و اختصاص پین‌ها را برای میکروکنترلر ST شما انجام می‌دهد. در این نرم‌افزار می‌توانید این سیستم عامل را نیز به پروژه خود اضافه کنید تا پروژه‌ای که ساخته می‌شود شامل سیستم عامل هم باشد. اگر قصد داشته باشید به شکل دستی این عملیات را انجام دهید باید تعدادی از فایل‌ها را در پروژه خود استفاده کنید.

فایل های کلید این سیستم عامل در شکل زیر دیده می‌شود:



ابزارهای همزمان‌سازی

تصور کنید دو Thread به یک منبع مشترک نیاز داشته باشند. برای مثال دو Thread قصد دارند از USARTکاراکتری دریافت کنند. کدی که برای این کار استفاده می‌شود به شکل زیر است.

```
int getchar(void){
  while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
  return USARTx->DR & 0xff;
}
```


اینThread ها باید رجیستر وضعیت USART را بخوانند تا هر وقت خالی نبود و کاراکتری دریافت شده بود، آن را از رجیستر داده بخوانند. اگر Thread ۱ رجیستر وضعیت را بخواند و بفهمد که رجیستر داده خالی نیست ولی اینThread ، در همین زمان، توسط Thread ۲ قبضه (preempt) شود، در این صورت2 Thread ، رجیستر وضعیت را بررسی می‌کند و داده را می‌خواند. سپس Thread ۱ از سرگیری می‌شود و اطلاعات آن از رجیستر وضعیت دیگر معتبر نیست و اگر داده‌ای را از رجیستر داده بخواند نادرست خواهد بود.

برای حل این مشکل می‌توان USART را کاملا در اختیار Thread ۱ قرار داد. یک راه جلوگیری از قبضه کردن، غیرفعال کردن وقفه هاست که کاری است به وضوح اشتباه، چون ممکن است تا مدتی

کاراکتری نیاید یا با تاخیر بیاید. چیزی که نیاز است، ساز و کاری است کهThread هایی که برای رقابت به USART شرکت دارند، بلوکه شوند. این راه حل، با استفاده از سمافورها (Semaphore) قابل انجام است. سمافور در واژه به معنی جفت پرچم های کوچکی است که برای علامت دادن و ارتباط از راه دور به کار می‌رفته است. از همین کاربرد، در مفهوم سیستم عامل استفاده شده است.

```
xSemaphoreHandle uartRcvMutex;

uart_init(...){
  ...
  uartRcvMutex = xSemaphoreCreateMutex();
}

int getchar(void){
  int c;
  if (xSemaphoreTake(uartRcvMutex, portMAX_DELAY) == pdTRUE)
  {
    while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
    c = USARTx->DR & 0xff;
    xSemaphoreGive(uartRcvMutex);
  }
  return c;
}
```

سمافور یک نخستینه همزمان‌سازی استاندارد است که برای ساختن سیستم عامل نیاز بود. این‌ها برای دسترسیِ ایمنِ Thread به منابع مشترک به کار می‌روند. یک Thread که نیاز به منبع دارد یا اجازه دسترسی به آن را می‌یابد یا توسط زمان بند scheduler بلوکه می‌شود تا منبع رها شود. وقتی یک Threadیک منبع را رها می‌کند ممکن است به عنوان یک اثر جانبی یک Thread را از بلوکه در آورد.

این سیستم عامل سه نوع سمافور را پشتیبانی می‌کند که عبارتند از موتکس ها، سمافورهای باینری و سمافورهای شمارنده. در اینجا از یک موتکس استفاده می‌کنیم. موتکس یک بلیت token دارد. اگر موتکس آزاد باشد با دستور take آن را می‌گیریم اگر هم آزاد نباشد تابعی که درخواست token کرده است بلوک می‌شود و در فهرست انتظار قرار می‌گیرد. دستور give توکن را بازیابی می‌کند. تفاوت اصلی بین موتکس‌ها و سمافورهای شمارنده در این است که دومی می‌تواند چند token داشته باشد.

می‌توان یک موتکس دیگر برای حفاظت تابع putchar استفاده کرد تا از رقابت‌های داده مانند بالا جلوگیری کرد. اما این نتیجه مطلوبی نخواهد داشت. تصور کنید چند Thread تابع putchar را از طریق فرایندی به نام putstring که در زیر مشاهده می‌کنید صدا بزنند. در این حالت ممکن است دو Threadهمزمان روی خروجی بنویسند.

```
void putstring(char *s){
  while (s && *s)
    putchar(*s++);
}
```

Sisoog

- منبع: فرادرس و sisoog

Q2

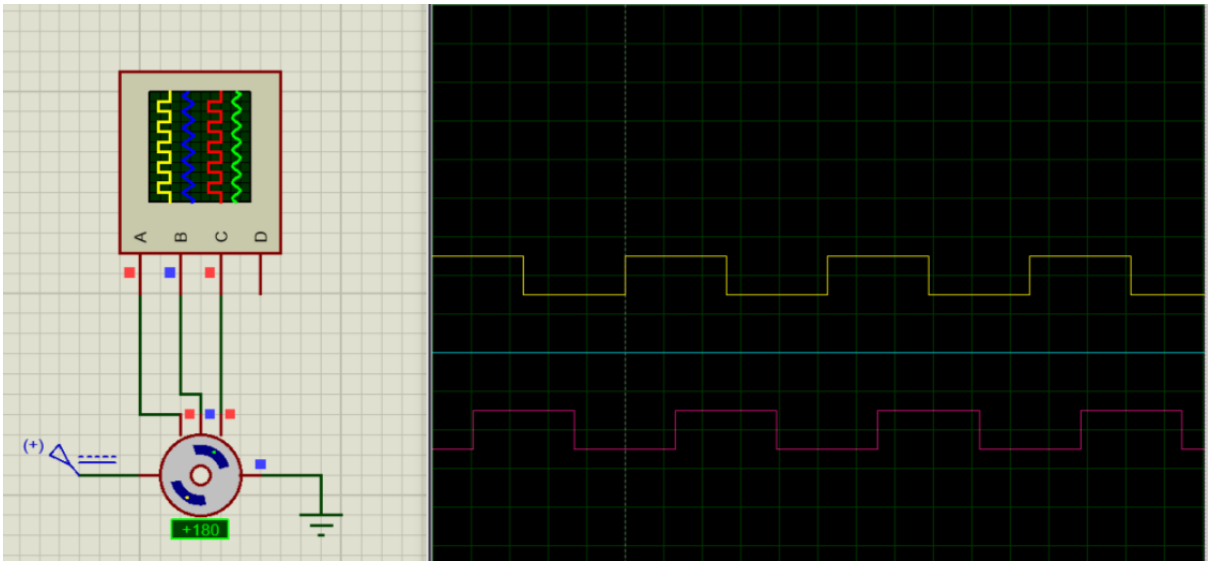
MOD(SN,100)=97

Pulse per rev = 220+97 = 317

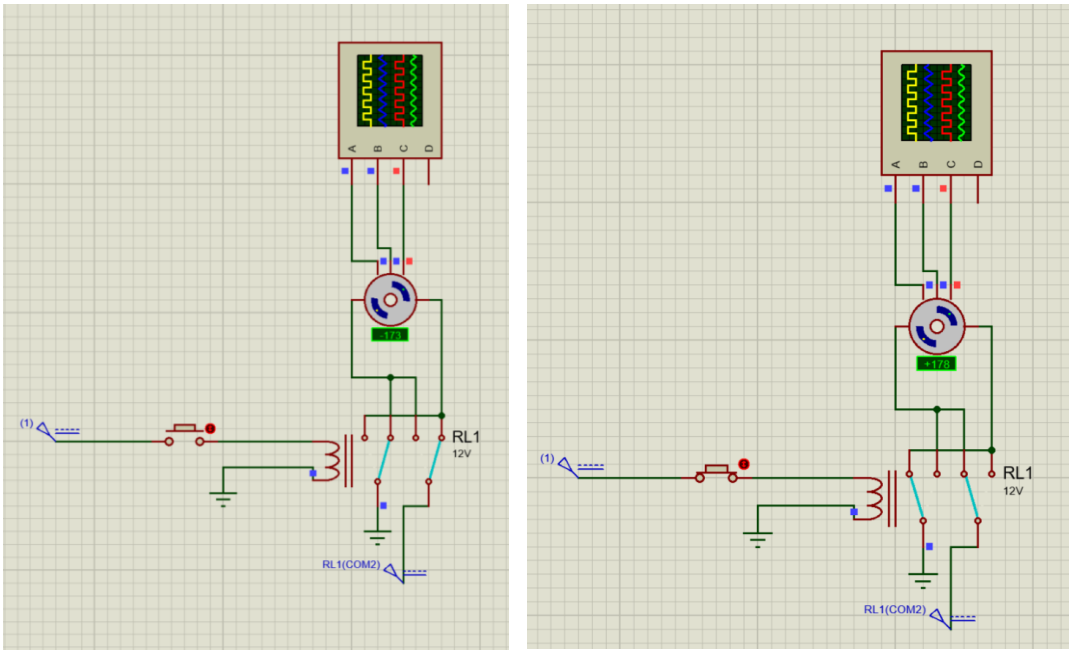
کلاک = 26 مگاهرتز

کلاک تایمر انتخاب شده: 10 کیلوهرتز

1

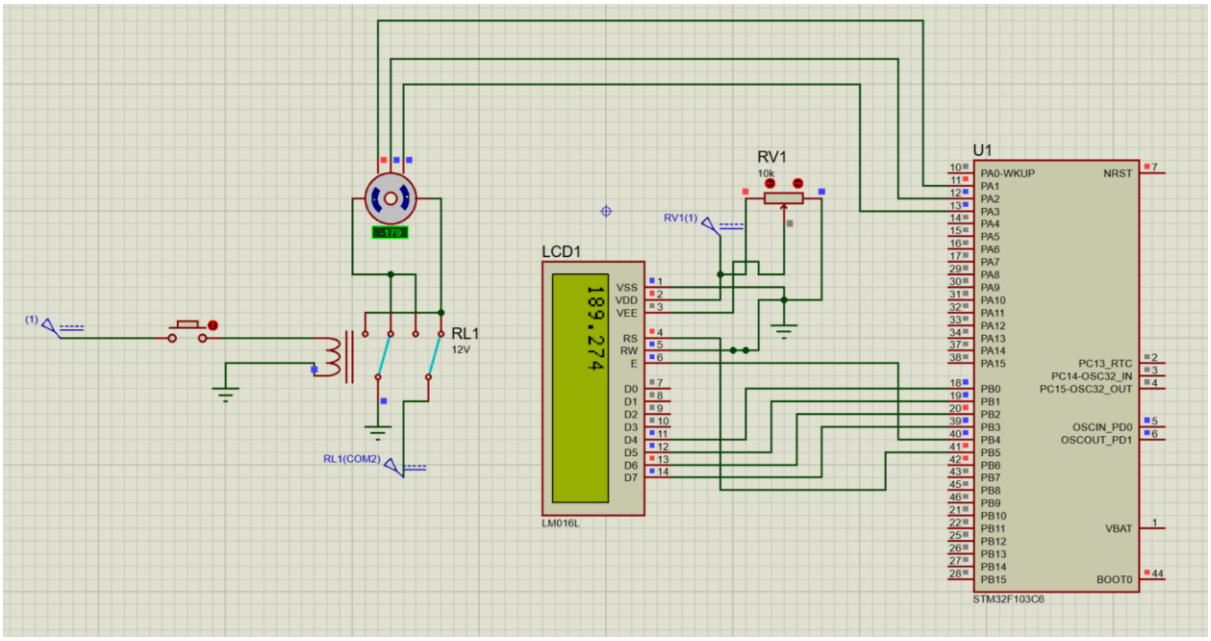


2



از یک منبع ولتاژ 12 ولت برای کلید دو سر رله استفاده میکنیم و مدار را مانند شکل بالا میبندیم.

3

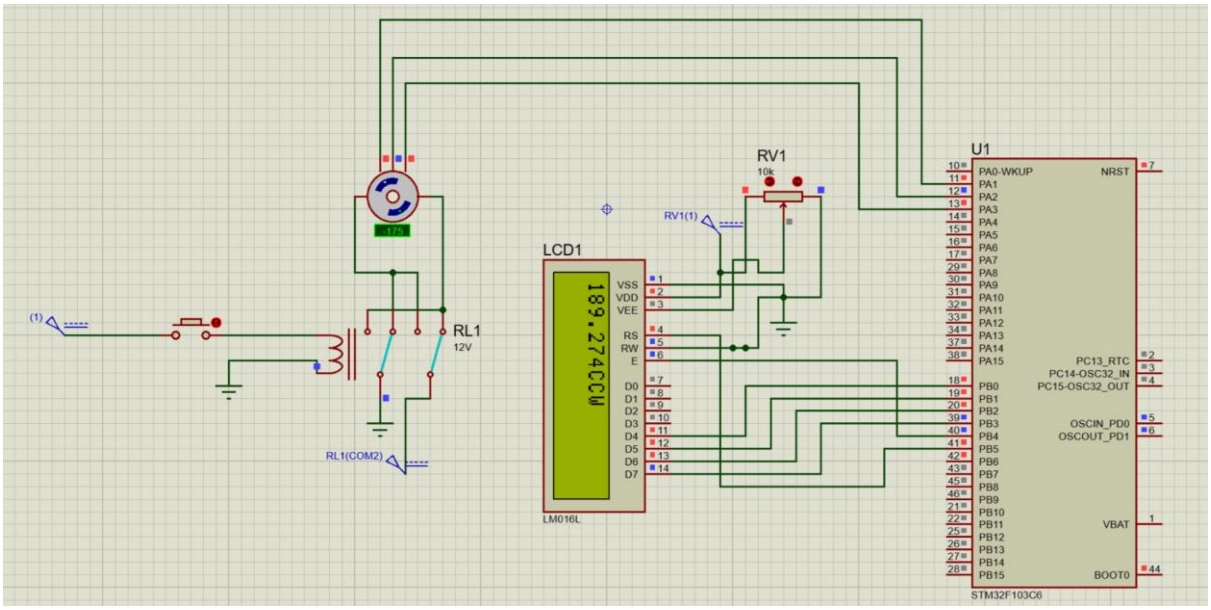


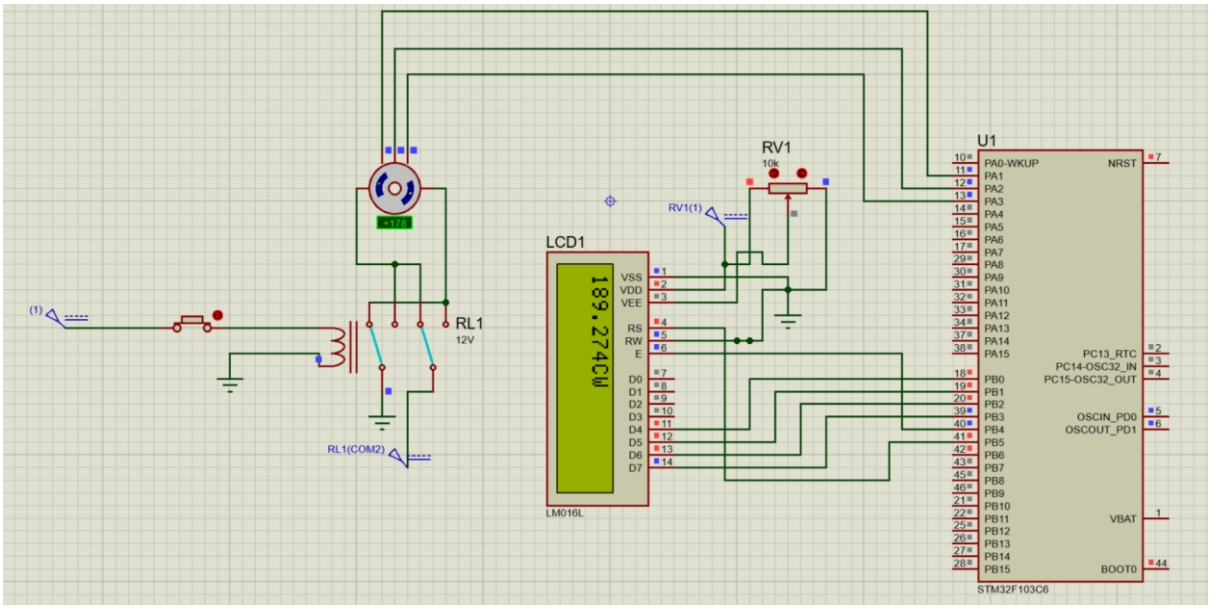
```
75 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
76 {
77     if (GPIO_Pin == A_Pin){
78         pre = now;
79         now = cnt_time;
80         one_pulse_time=(now-pre);
81         speed= one_pulse_time*0.317;
82         speed = 1/speed;
83         speed = speed *60;
84     }
85 }
86
87 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
88 {
89   cnt_time++;
90 }
```

کد نمایش روی ال سی دی مثل تمارین گذشته است.

در اینجا با تعریف یک اکسترنال اینتراپت برای سیگنال A هر بار که پالسی دریافت میشود، زمانی را که با یک شمارشگر در تایمری که تعریف کردیم را در متغیر now ریخته و از زمان پالس قبلی کم میکنیم. عدد بدست آمده را در دوره تناوب تایمر که 1 میلی ثانیه است ضرب میکنیم(درواقع ضرب را در خط پایینی نشان میدهیم(0.317)). سپس در 317 ضرب میکنیم که تعداد پالس در یک دور است. حالا با یک تناسب ساده میتوان سرعت را برحسب rpm بدست آورد.

4





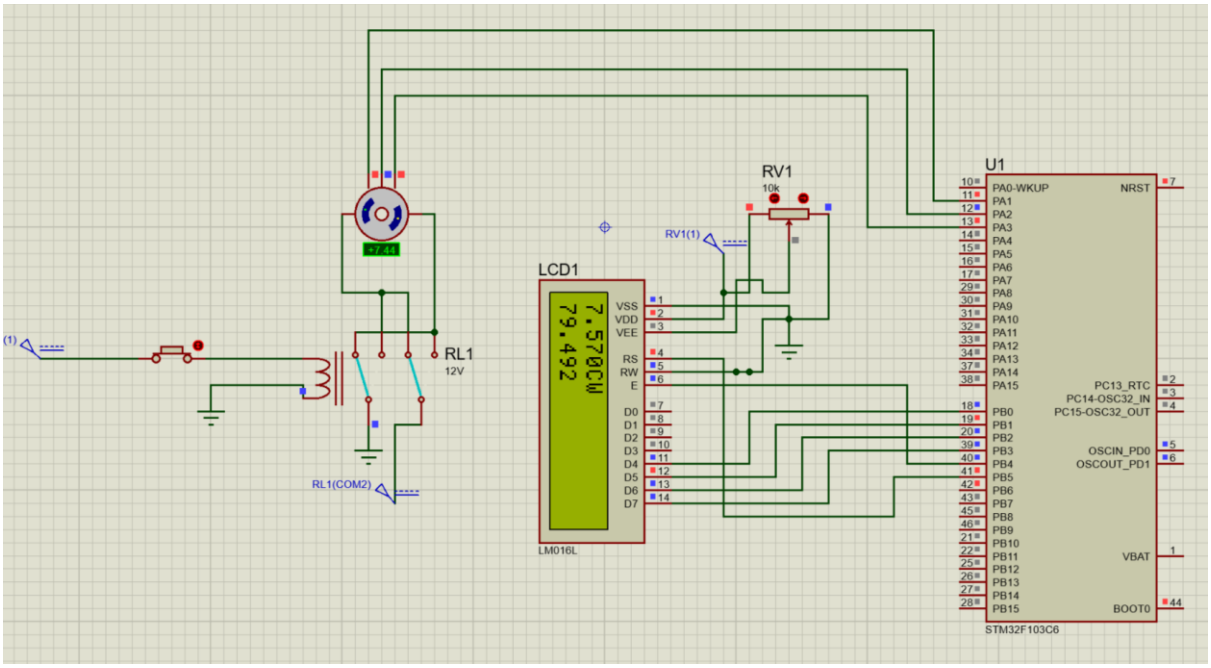
همانطور که مشاهده میشود جلوی سرعت در LCD، CW یا CCW بودن دور موتور نوشته شده است.

```
84
85          if (HAL_GPIO_ReadPin (GPIOA, B_Pin) == 0){
86              dir = "CW";
87          }
88          else{
89              dir = "CCW";
90          }
91      }
```

این قطعه کد درون اکسترنال اینتراپت نوشته شده است.

ابتدا با خواندن مقدار پین B زمانی که rising edge پین A خورده میشود چک میکند که اگر در آن پین B دارای value نیست پس CW است و در غیر این صورت CCW.

5

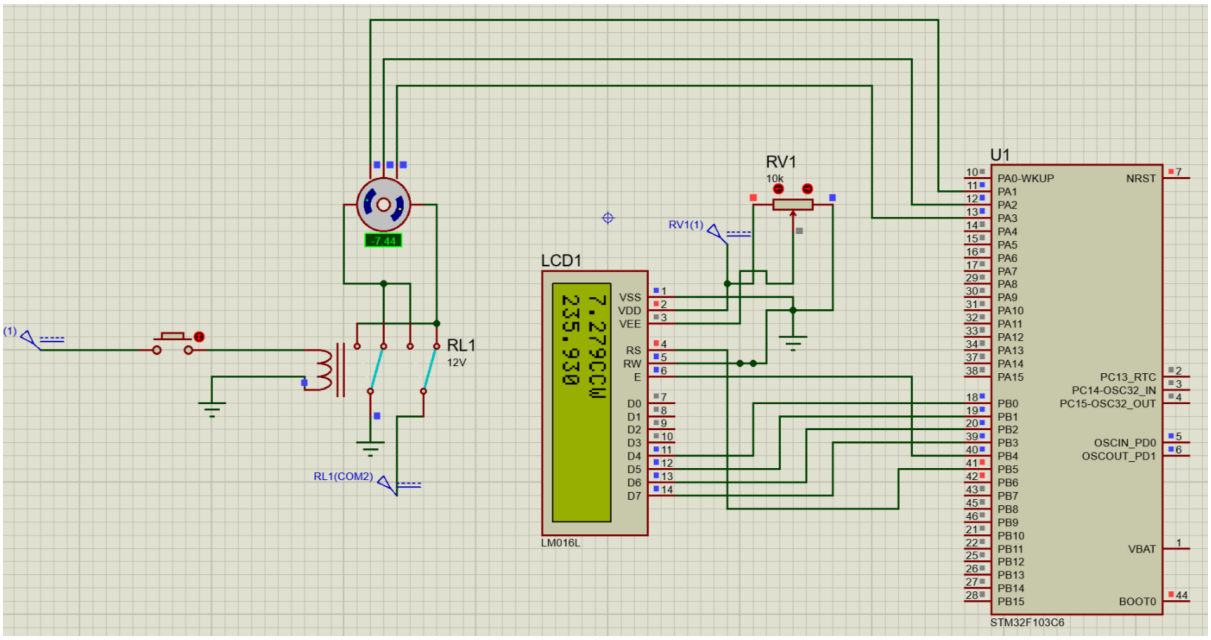


زاویه در خط دوم نمایش داده شده است.

```
97          //degree
98              if (strcmp(dir,"CW") != 0){
99                  cnt_deg++;
100             }
101             else{
102                 cnt_deg--;
103             }
104             deg = abs(cnt_deg) *1.1356;
105             // 360/317 = 1.1356
106     }
107     if(GPIO_Pin == Z_Pin){
108         cnt_deg=0;
109     }
```

ابتدا یک counter برای زاویه تعریف میکنیم که با هر لبه بالارونده سیگنال A با توجه به جهت حرکت موتور مقدارش تغییر میکند. همینطور زمانی که لبه بالارونده سیگنال Z مشاهده شود مقدار شمارشگر زاویه را صفر میکنیم.

6



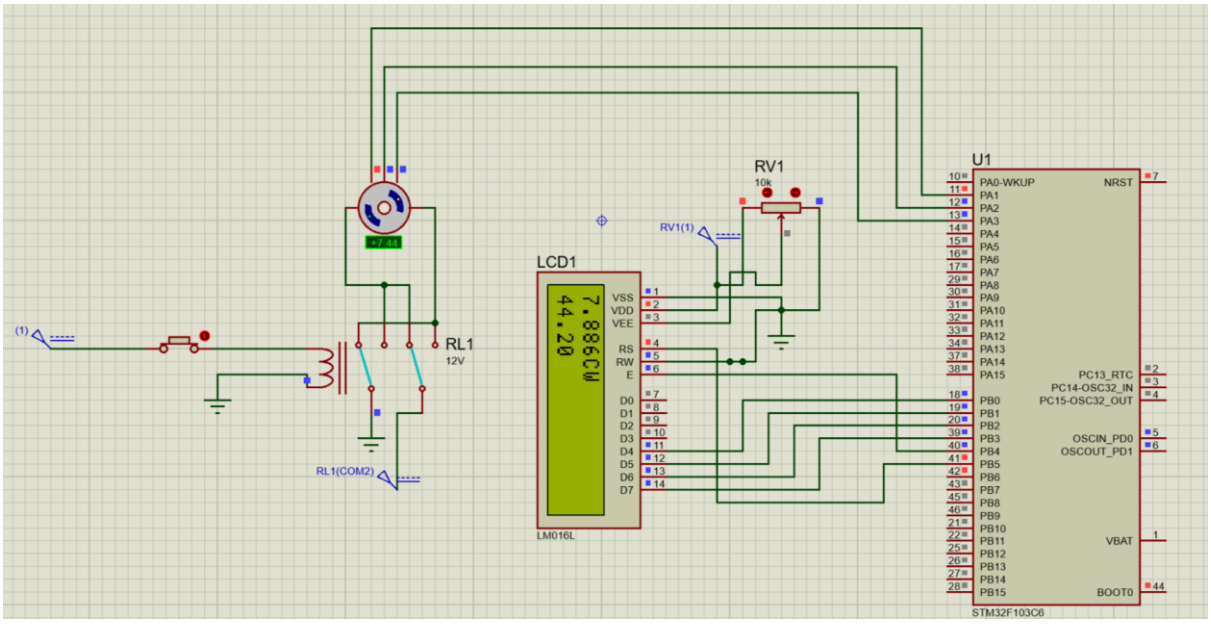
زاویه در خط دوم نمایش داده شده.

```
78 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
79 {
80     if (GPIO_Pin == A_Pin){
81 
82         //speed
83         pre = now;
84         now = cnt_time;
85         one_pulse_time=(now-pre);
86         speed= one_pulse_time*0.634;
87         speed = 1/speed;
88         speed = speed *60;
89 
90         //direction
91         if (HAL_GPIO_ReadPin (GPIOA, B_Pin) != HAL_GPIO_ReadPin (GPIOA, A_Pin)){
92             dir = "CW";
93         }
94         else{
95             dir = "CCW";
96         }
97         //degree
98             if (strcmp(dir,"CW") != 0){
99                 cnt_deg++;
100            }
101            else{
102                cnt_deg--;
103            }
104            deg = abs(cnt_deg) *0.56578;
105            // 360/634 = 0.56578
106    }
107    if(GPIO_Pin == Z_Pin){
108        cnt_deg=0;
109    }
```

با توجه به اینکه ما در حال حاضر از دو لبه بالارونده و پایین رونده یک سیگنال استفاده میکنیم بنابراین درواقع ما یک دور موتور را به اندازه دو برابر استپ های قبلی تقسیم کردیم یعنی 634 =2*317. حالا باید در کد مربوط به سرعت نیز به جای 0.317، 0.634 را جایگزین کنیم و شرط جهت را عوض کنیم و به مقایسه مستقیم دو سیگنال در آن بپردازیم. سپس مانند قبل با توجه به جهت حرکت شمارشگر را تغییر میدهیم.

در نهایت با استفاده از شمارشگر زاویه را بدست می آوریم.

7



زاویه در خط دوم نمایش داده شده.

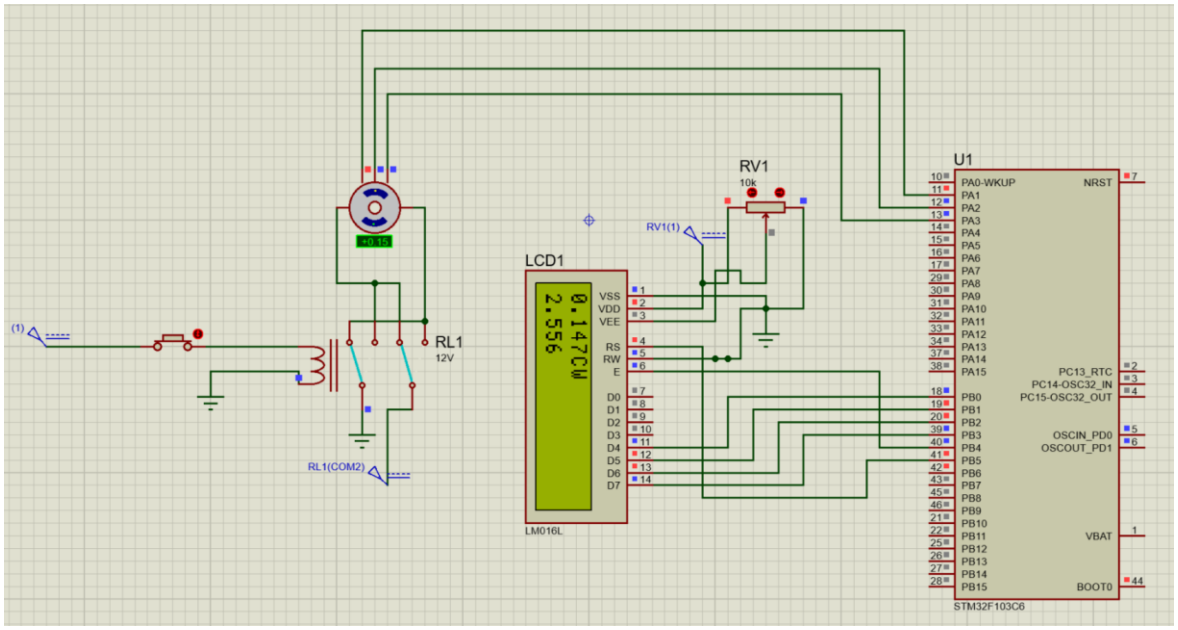
```
    if(GPIO_Pin == B_Pin){
        //speed
        pre = now;
        now = cnt_time;
        one_pulse_time=(now-pre);
        speed= one_pulse_time*1.268;
        speed = 1/speed;
        speed = speed *60;
        //direction
                if (HAL_GPIO_ReadPin (GPIOA, B_Pin) == HAL_GPIO_ReadPin (GPIOA, A_Pin)){
                    dir = "CW";
                }
                else{
                    dir = "CCW";
                }
                //degree
                    if (strcmp(dir,"CW") != 0){
                        cnt_deg++;
                    }
                    else{
                        cnt_deg--;
                    }
        }

    if(GPIO_Pin == Z_Pin){
        cnt_deg=0;
    }
```

همانطور که مشاهده میشود بعد از قرار دادن اینتراپت A در حالت بالا رونده و پایین رونده همینکار را با سیگنال B میکنیم.

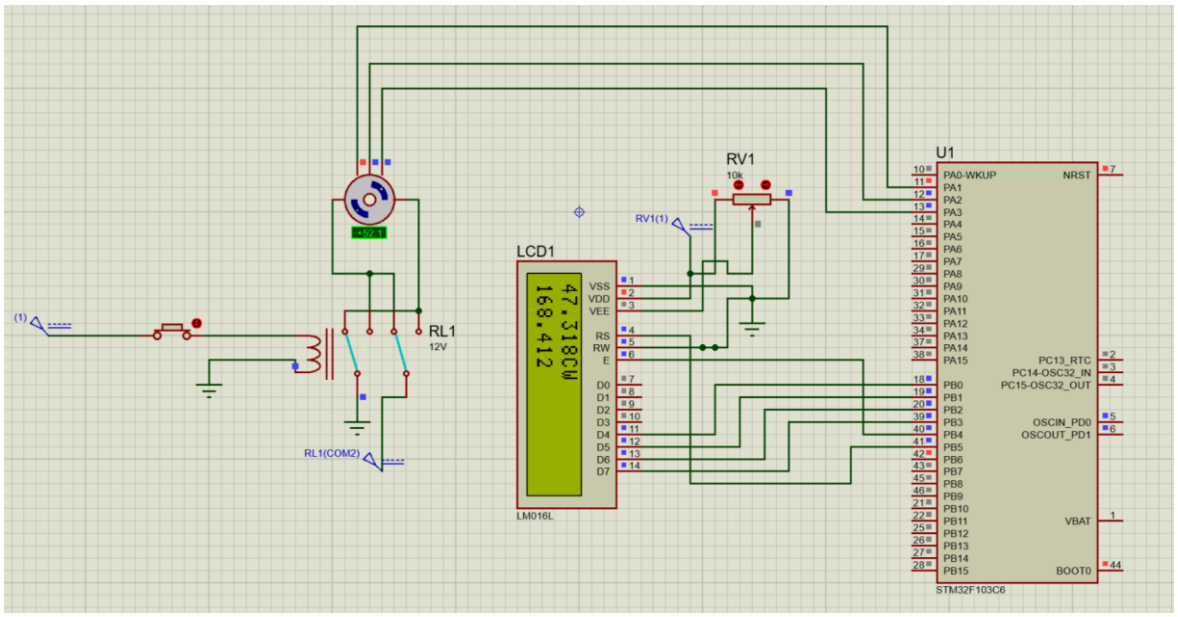
با توجه به اینکه ما در حال حاضر از دو لبه بالارونده و پایین رونده دو سیگنال استفاده میکنیم بنابراین درواقع ما یک دور موتور را به اندازه دو برابر استپ های قبلی تقسیم کردیم یعنی 1268 = 2*634. حالا باید در کد مربوط به سرعت نیز به جای 0.634، 1.268 را جایگزین کنیم و شرط جهت را در اینتراپت B عوض کنیم و به مقایسه مستقیم دو سیگنال در آن بپردازیم. سپس مانند قبل با توجه به جهت حرکت شمارشگر را تغییر میدهیم.

در نهایت با استفاده از شمارشگر زاویه را بدست می آوریم.

8



کمترین مقداری که تونستم اندازه گیری کنم: 0.15rpm



بیشترین مقداری که تونستم اندازه گیری کنم: 47.3rpm

9

دقت اندازه گیری زاویه در روش **1x meaturement**:

360 درجه با 317 قسمت تقسیم کردیم:

$$360/317 = 1.135$$

دقت 1 درجه است.

دقت اندازه گیری زاویه در روش **2x meaturement**:

360 درجه با 634 قسمت تقسیم کردیم:

$$360/634 = 0.5678$$

دقت 0.1 درجه است.

دقت اندازه گیری زاویه در روش **3x meaturement**:

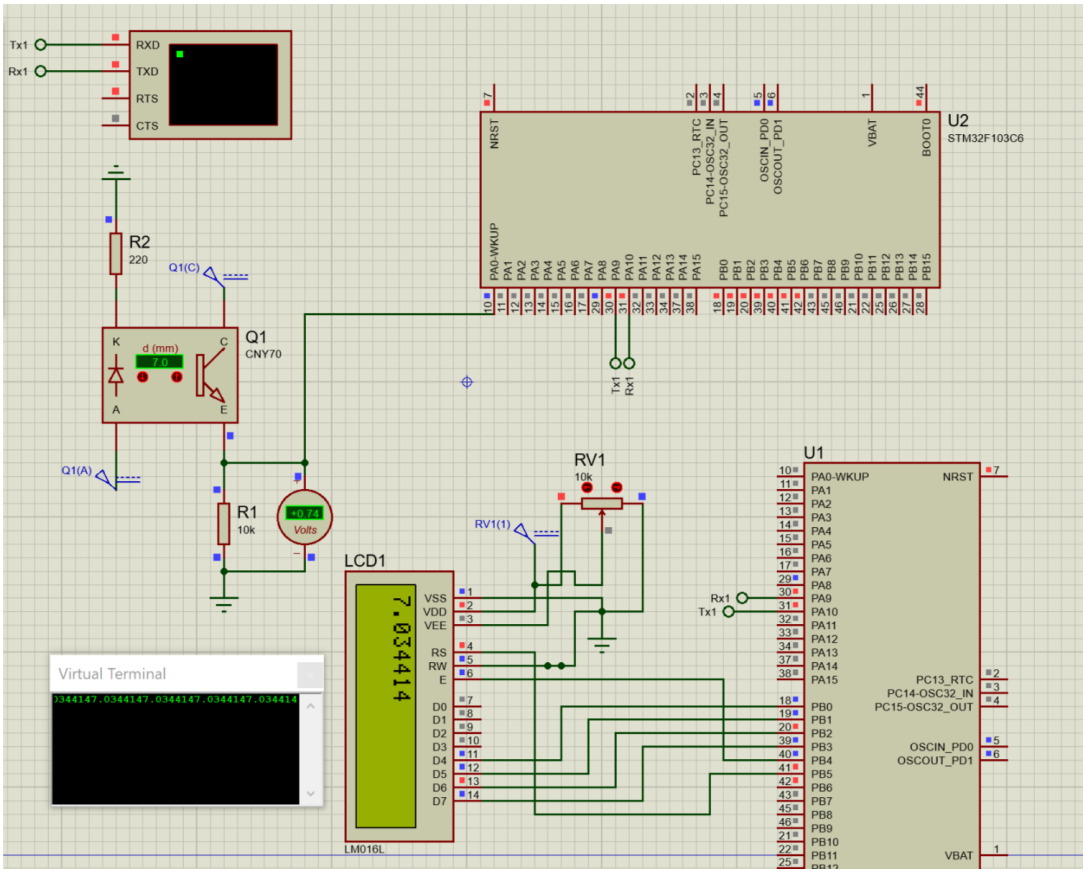
360 درجه با 634 قسمت تقسیم کردیم:

$$360/1268 = 0.2839$$

دقت 0.1 درجه است.

Q3

بخش 1

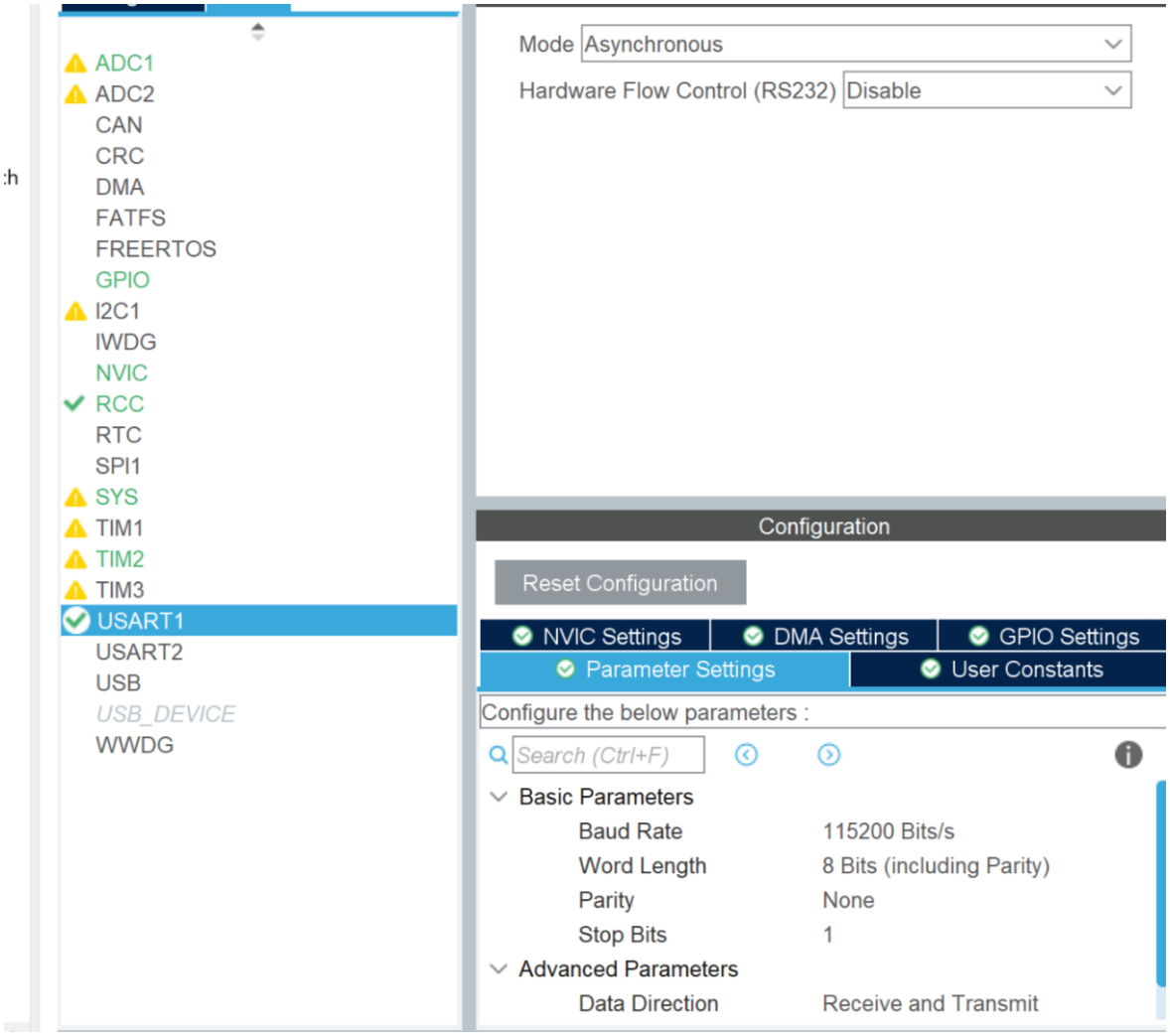



```
if (adc_valid == 1){
    v = adc_result*slope;
    result = 0.37*(v*v) -4.181*v + 9.943;
    sprintf(res_str, "%f", result);
  HAL_UART_Transmit(&huart1, (uint8_t*)res_str, 16, 100);
                    }
```

با این دستور داده ها را از طریق UART میفرستیم.

```
/* USER CODE BEGIN 3 */
HAL_UART_Receive(&huart1, buff, 16, 200);
//        sprintf(res_str, "%f", buff);
    LCD16X2_Init(MyLCD);
            LCD16X2_Clear(MyLCD);
```

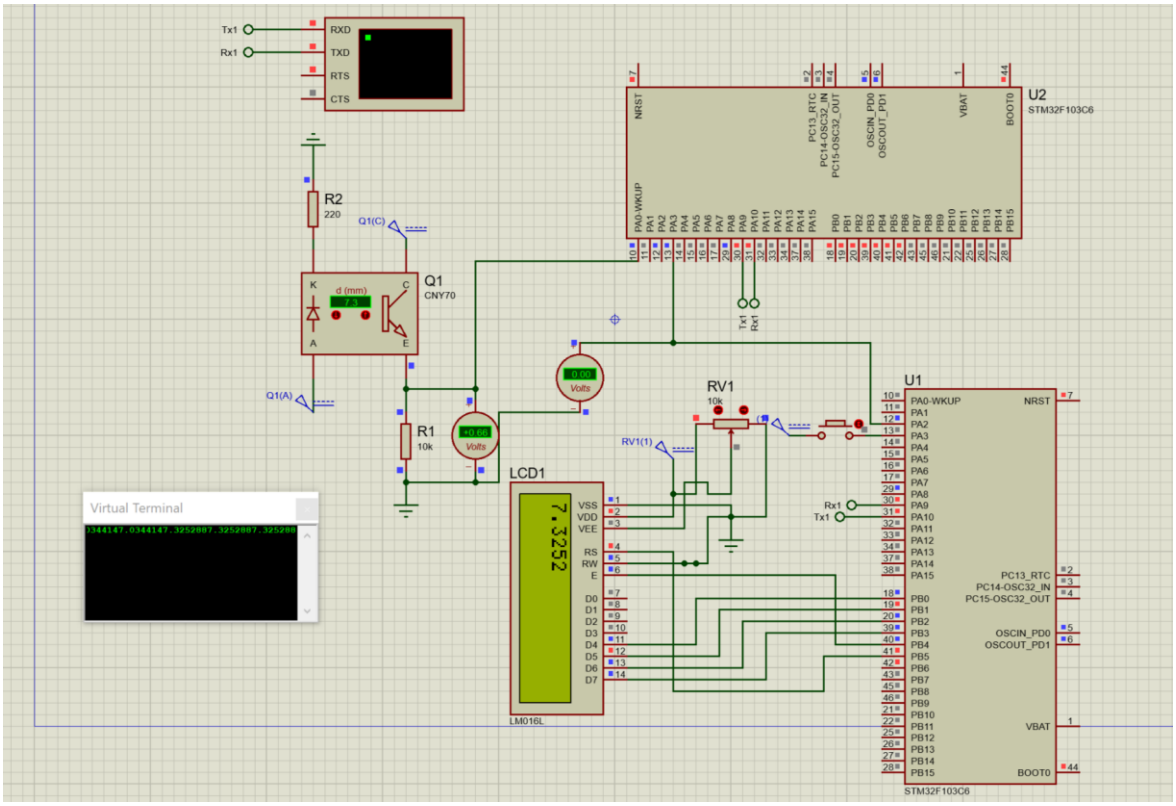
با این دستور داده های فرستاده شده توسط UART را میخوانیم



تنظیمات ioc. پهنای باند و آسنکرون بودن پروتوکل مشخص است.

برای استفاده از ترمینال مجازی در پروتئوس نیز از همین پهنای باند استفاده میکنیم.

بخش 2



با تعریف پین A3 بعنوان اینتراپت و پین A1 بعنوان خروجی دیجیتال این سیستم را راه اندازی میکنیم.

ابتدا با اینتراپت بر روی میکروی گیرنده و پوش باتن متوجه درخواست انتقال داده میشویم. سپس با استفاده از پین خروجی دیجیتال به میکروی فرستنده فرمان میدهیم که داده را بفرست.

شرط فرستادن داده در میکروی فرستنده

```
373   /* USER CODE BEGIN 3 */
374     if ((adc_valid == 1) && ( send ==1)){
375                 v = adc_result*slope;
376                 result = 0.37*(v*v) -4.181*v + 9.943;
377                 sprintf(res_str, "%f", result);
378               HAL_UART_Transmit(&huart1, (uint8_t*)res_str, 16, 100);
379               adc_valid=0;
380               HAL_ADC_Start_IT(&hadc1);
381               send=0;
382                           }
```

```
/* Private user code ---------------------------------
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
send=1;
}
```

اینتراپت درون فرستنده

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == but_Pin){
HAL_GPIO_WritePin(GPIOA, send_Pin, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOA, send_Pin, GPIO_PIN_RESET);
  }
}
```

اینتراپت درون گیرنده که یک رایزینگ اج ایجاد میکند.

بقیه کد مانند بخش یک است.