NACKADEMIN

Rapport och dokumentation – Inbyggda system, arkitektur och design.

Nackademin – Mjukvaruutveckling, inbyggda system & IoT

2023-05-10

Jonas Larsson

jonas.larsson2@yh.nackademin.se

Innehållsförteckning

Inledning

Protokoll

Programkod

- UART.cpp
- UART.h
- Led.cpp
- Led.h
- Main.cpp

Funderingar och Slutsats

Källhänvisning

Inledning

Som innehållsförteckningen vittnar om så blir detta mer en redovisning och kortare rapport om den kunskap jag förvärvat under under kursen. Redovisningen simulerar funktioner och processerna av drivrutiner genom UART-protokollet med ett STM32F411x-chip. I och med att mycket av koden redan är kommenterat har jag lagt fokus på att skriva koden här och fördjupa mig där jag anser nödvändigt.

Jag ska gå igenom processen i inställningar och kommunikation som vi har gått igenom och främst fokusera på UART protokollet men även beröra lite av I2C och SPI protokollen.

Rapporten kommer vara mer av löpande text än i dagboksform, delvis på grund av att dagboksanteckningarna från flera dagar trillat bort.

Jag kommer utefter förmåga beskriva utvecklingen av kontroll och drivrutiner för ovan nämnda kort men också lite funderingar i slutsatsen kring andra kort (och protokoll) som vi har gått igenom under kursen.

Det blir främst fokus på att beskriva kunskaper och vissa frågeställningar. Att skriva drivrutiner och analysera datablad för specifika enheter känns i dagsläget som stort fält och på så vis känns det som jag har vissa luckor för att fullt ut förstå allt som kan behövas. Jag ämnar att fördjupa mig i områden där jag upplever att jag har vissa luckor. Vissa kommer jag att fylla i under rapportens gång andra luckor kommer jag att fortsatt studera och gå lite djupare i efter kursslut.

På vissa ställen i den här texten har jag skrivit in sidnummer som refererar till chippets datablad och dokumentation för att kunna gå tillbaka och läsa, främst för att arbetet men även för att kunna kolla på det senare för att kunna fräsha upp minnet.

Protokoll

De protokoll som vi berörde under kursen var I2C, SPI och UART. Till skillnad från I2C och SPI är UART Asynkront med det menas att protokollet ”stämmer av” genom start och stop bitar ,transmitter och reciever ställs in på timing på förhand.

Data i UART protokollet överförs genom frames som innehåller startbit, data frame, parity bit och stop bits. Parity bit(s) kontrollerar genom binär funktion om data framen stämmer överens med skickad data.

UART-protokollet är ett parallelt-till-seriellt protokoll som som skickar från en transmitter till en reciever (samt vice versa), det gör UART till ett tillgängligt och billigt protokoll att använda.

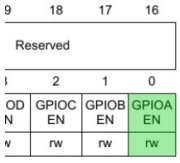
Programkod

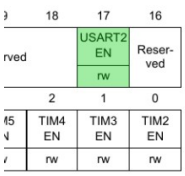
Uart.cpp

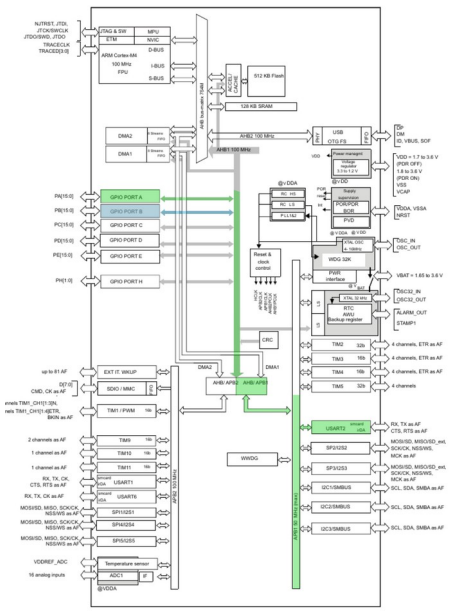
Uart.cpp börjar med en header som hämtar in andra headers samt funktioner. Genom funktionen USART_init initeras USART protokollet för chipet.

```
RCC-> APB1ENR |= 0x20000      // 0010 0000 0000 0000 0000
RCC-> AHB1ENR |= 0x01         //                0000 0001
```

RCC (Reset and Clock Control) sätter APB1ENR/USART2 (s 118) till PÅ genom att sätta 17:e biten till 1 och aktiverar på så vis UART-protokollet. Aktiverar klockan på AHB1ENR/GPIOA (s 117) genom att sätta bit 0 till 1, illustrerat på respektive bild. I koden ställs respektive bit in genom att med en pipa (| , OR operator) innan lika med tecknet så att övriga bitar innan, (0-16) ignoreras och inte ändras, som i annat fall hade nollats.







Ovan(fig. 2 & 3) ; Exempel på bitar enligt Register maps.
TH (fig 1); chiparkitektur och och aktiverade portar/kanaler.

Till skillnad från pipan ovan så gör &=~ alla bitar till 0 genom en AND operator och unary operator som växlar bitarna. Bitarna 0-3 inverteras/växlas och bitarna 4-7 sätts till 1. Sedan sätts bitarna 5 och 7 till 1, detta ”öppnar” portarna till PA2 och PA3 i GPIO mode register till portkonfiguration 1:0 , ”Alternate funcion mode”.

```
GPIOA->MODER &= ~0x00F0;      // 0000 0000 1111 0000
GPIOA->MODER |= 0x00A0;       // 0000 0000 1010 0000
```

Genom ställa in AFR[0] (GPIOx_AFRL) (s 161) rensas först bitarna 8-15 med &= ~ sedan ställs desamma in enligt mönstret i den binära koden nedan.

```
GPIOA->AFR[0] &= ~0xFF00;     // 1111 1111 0000 0000
GPIOA->AFR[0] |= 0x7700;      // 0111 0111 0000 0000
```

För konfigureringen av UART så sätts baud-rate till 0x0683 vilket sätter bitar 9-10 i mantissa till 1 respektive 0-1 i fraktionen till 1, för 9600 bps.

CR1 (Control Register 1) så slås RX/RE (Reciver Enable) samt TX/TE(Transmitter Enable) på och sätts i 8 bitars läge. RX påslagen börjar lyssna till start bitar. CR2 samt CR3 nollas helt och sedan sätts bit 13 i CR1 till ett och slår på denna.

```
USART2->BRR  = 0x0683         // 0000 0110 1000 0011
USART2->CR1  = 0x000C         // 0000 0000 0000 1100
USART2->CR2  = 0x000          //      0000 0000 0000
USART2->CR3  = 0x000          //      0000 0000 0000
USART2->CR1 |= 0x2000         // 0010 0000 0000 0000
```

Här på följer två funktioner USART2_write som tar in en karaktär och USART2_read som skriver respektive läser in data. While-loopen läser om bit 7 (Transmit data registry empty) sätts till 1 har blivit skickad till shift register. Den rensas sedan genom att skrivas till dataregistret. (s 547)

Read-funktionen kontrollerar bit 5, RXNE(Read data registry not empty) om shift registret har överfört data till DR-registret, som sedan läser ut datan.

(s 548 & 550)

```
int USART2_write(int ch){
while (!(USART2->SR & 0x0080)){}    //0000 0000 1000 0000

USART2->DR = (ch & 0xFF);           //          1111 1111
      return ch;
}
int USART2_read(void){

  while(!(USART2->SR & 0x0020)){} // 0000 0000 0010 0000
  return USART2->DR;
}
```

UART.h

En headerfil med definitioner som inkluderar hänvisning till headerfil stm32f4xx.h, det specifika kortet som hämtar källkod och hänvisningar, samt standard in-, output hänvisning för C.

Här finns även funktionen UART2_init som beskrivs ovan och som används i main.cpp. Samt en test_setup som inte används i main funktionen senare.

Led.cpp

Startar med att inkludera headern för led.h's funktioner. Sedan följer en konstruktor som tar in två värden från led.h som färg och tillstånd. Två pekare definierar color respektive state. AHB1ENR startar klockan för en alternativ funktion och sätter bitarna för LED_PORT_CLOCK.

En switch tar in respektive färg och ställer beroende på case som sätter motsvarande bit i GPIOB->MODER registret. I fallet RED skulle den som i Led.h är definerad RED att flyttas 28 steg till vänster.

En if-sats som kontrollerar status OFF/ON. I fall av tillståndet/state är ON och skulle således genom ODR(Output Data register) sätta bit 14 till 1. I fall av OFF skulle den slå av bit 14 och slå av LEDen. Respektive färg i switchen behandlas på samma sätt.

```
switch(_color){

    case RED:
      LED_PORT->MODER |= LED_RED_MODE_BIT;
      if(this->state == ON){

        LED_PORT->ODR |= LED_RED_PIN;
      }
      else{
        //Stänga av LED
        LED_PORT->ODR &= ~LED_RED_PIN;
      }
      break;
...
```

Led.h

Headern för den körbara Led-filen innehåller hänvisningar till UART.h samt en header till producentens/chipets specifika header. Sedan följer definitioner för vilken GPIO port som ska användas i vårat fall är det B porten, GPIOB.

Sedan sätts klocksignalen för porten och definitioner för de olika LED-pinsen. Genom att skapa definitioner i headern blir det enklare att använda genom mer lättläsliga variabler i den körbara koden.

Definitionen för LED_RED_PIN adresserar variabeln genom att flytta en unsigned bit 1 till vänster motsvarande angivet antal steg.

```
#define LED_RED_PIN (1U<<14)        // 14 steg vänster
#define LED_GREEN_PIN (1U<<12)      // 12 steg vänster

..etc.
```

I definitionerna av MODE_BIT flyttas bitarna respektive steg till vänster med samma mönster för Port Mode Register. Här definieras PB 14 till [1:0] till Alternate function mode.(s 157)

```
#define LED_RED_MODE_BIT (1U<<28)
...etc.
```

Typedef enum ger varje färg på LED numrering och följer mönstret att om RED = 0 så blir nästföljande 1 etc. På så vis är det lätt att ange LedColor_Type GREEN blir lika med LedColor_Type 1. På samma vis följer en enum för LedState_Type med OFF/ON motsvarande 0/1.

```
typedef enum {
  RED = 0,
  GREEN,
  YELLOW,
  BLUE
}LedColor_Type;
```

Slutligen skapas det en klass för att kunna konstruera ett Led-objekt. Klassen infattar Led som tar in LedColor_Type och ledState_Type. Samt setState och getState.

Main.cpp

I main-filen initieras tre variabler som tar OFF/ON status från LedState_Type. Ett Led-objekt som är röd och påslagen.

Sedan följer main-funktionen som startar USART2. Skapar ytterligare ett objekt, Led2 som ställs till Blå och tillståndet på, följt av objekt led3 som skapas med dynamiskt allokerad minnesplats som är gul och påslagen med new operatorn.

Sedan hämtas tillståndet som är (ON) och sedan slås led1 (OFF), båda gångerna med metoderna get och set.

Innan en evig loop körs så raderas led3 från minnet. Det på grund av att dynamiskt allokerat minne riskerar minnesläckage på små enheter. Funktioner med dynamisk minnesallokering så som malloc och calloc bör undvikas på inbyggda system och microdatorer.

Funderingar och Slutsatser

Vi har bollat lite, i gruppen från tidigare kurser, förståelsen av protokollen vi använt, då i synnerhet UART och dess funktioner i koden. För min del blev det lite av att ”polletten trillade ner” när vi har olika LCD-displayer, varav min har en I2C-konverterare påkopplad på min LCD-modul, medans en gruppmedlem saknar en sådan I2C-konverterare.
Förhoppningen var att vi skulle hinna laborera med olika kort och moduler för att testa men dessvärre hann vi inte med det under kursperioden.

Vi har diskuterat om att sätta oss och laborera med olika kort, jag beställde ett STM32 ”black pill”, men jag missade att beställa en ST-link (UART-USB konverterare vilket verkar nödvändigt).

En annan från gruppen beställt ett STM32 ”Nucleo”. Ambitionen är att vi, och eventuellt fler från klassen tar några dagar att utforska med olika labbar och förhoppningsvis kan vi göra det under sommaren.

Jag har dessvärre inte riktigt förstått (eller tagit mig tid) att gå igenom möjligheterna till att optimera koden så som vi gick igenom på sista lektionen. Möjligen är det något vi i gruppen kan titta på om vi ses och labbar på egen hand.
Jag tycker det varit en intressant kurs, som efter kursen om ellära, gav lite ytterligare substans och gjorde kursen ellära lite mindre abstrakt. Att fortsätta labba ytterligare med Arduino och STM-kort på egen hand känns mycket mer intressant nu efter denna kurs.

Källhänvisning

Utöver föreläsningar och material från lärare samt datablad har jag tillgodosett mig information från dessa källor.

1) https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html
2) https://www.electronicshub.org/basics-uart-communication/

3) http://www.modemhelp.net/faqs/8n1.shtml
4) https://blog.mbedded.ninja/programming/languages/c/dynamic-memory-allocation/#memory-allocation-on-embedded-systems