NACKADEMIN

Utveckling av UART-drivrutin för stm32f411x plattformen, med inkluderad LED-applikation

Frej Mellberg
frej.mellberg@gmail.com

Innehållsförteckning

Inledning

Detta är en rapport som syftar på att ge läsaren en insikt i hur det kan vara att utveckla enklare drivrutiner. I det här fallet gäller det att utveckla en drivrutin för kommunikation via UART-protokollet på en STM32F411x plattform.

Utöver det kommer även en drivrutin för kontroll av LEDs att utvecklas. Detta för att ge en ytterligare insikt i hur man kan använda UART-kommunikation på mikrokontrollern (STM32f411x) för att styra andra enheter.

Jag kommer att försöka ge en grundlig bild av hur processen och arbetet med de olika delar gått till. Från förarbetet med att sätta sig in i dokumentation och hårdvara till utvecklandet av koden och dess funktioner. Det är förstås vanligt och grundläggande att man också utför tester på den drivrutin man har utvecklat men den delen kommer att vara ytterst begränsad då ingen fysisk hårdvara varit tillgänglig.

I kombination med denna rapport kommer även all den kod som skrivits i samband med utvecklingen, samt dokumentation för STM32F411x-plattformen, att finnas tillgänglig på Github.

Dagbok

Dag 1

Då jag var en nybörjare när det kommer till utvecklingen av drivrutiner fanns det mycket som jag behövde lära in från grunden. Till att börja med var UART ett nytt protokoll för mig.

En UART-enhet tar emot parallell-kommunikation från en databuss och paketerar data-bitsen tillsammans med start, stop och, eventuellt, en parity bit. Paketet skickas sedan till en mottagande Uart enhet, via seriell-kommunikation, och översätts där sedan tillbaka till parallell-kommunikation. På detta sätt går det att skicka data med bara en data-line istället för till exempel åtta stycken. Det leder till enklare överföring över större sträckor. Man skulle även kunna skicka signalen trådlöst med UART via IrDA men det är inget som kommer användas under detta projektet.

Efter att ha bekantat mig med UART-protokollet gick jag vidare till att titta på dokumentationen för STM32F411x-plattformen. Här hade jag överlägset mest hjälp av referens manualen (Se hårdvaru-mappen för projektet). Det var här jag hittade alla olika register som jag behövde komma åt och information om vad olika bits i registren hade för funktion. Det var även hjälpsamt att studera block diagrammet i data sheet-dokumentet (figure 3). Här fick jag en översiktsbild på hur de olika delarna hänger ihop och kommunicerar med varandra. Till exempel var det lätt att se hur de olika GPIO-portarna satt på en annan databuss än USART2 och att de bussarna arbetar på olika frekvenser.

Dag 2

Efter att ha byggt en kunskapsgrund för att förstå vad jag skulle göra behövde jag utöka den till HUR jag skulle skapa drivrutiner. Här tog jag dels hjälp av några tutorials på nätet men till största del var det Ludwig Simonssons lektioner på Nackademin som jag använde mig av.
Efter att under morgon och förmiddag gått igenom ovan nämnda material hade jag ett möte med mina klasskamrater där vi diskuterade om vi hade samma uppfattning om vad som behövde göras och om vi missat eller missuppfattat något.

Dag 3

Sedan började jag med själva koden. Då jag i koden använder mig av utförliga kommentarer som förklarar vad som händer och varför kommer jag här vara relativt kortfattad.
Det första jag gjorde var att ta mig an funktionen som ska initiera och konfigurera UART-protokollet. På STM32f411x-plattformen kan UART även användas synkront och

kallas därmed USART. Vi kommer att använda USART2, som är kopplat till APB1-bussen, för vår funktion USART2_init(). För att aktivera USART2 måste den ges tillgång till systemklockan. Detta görs med “RCC->APB1ENR |= 0x20000;”, vilket betyder att bit 17 i APB1s peripheral clock-enable-register sätts till 1. Nu får alltså USART2 ström och kan kommunicera med resten av systemet. Vi aktiverar även klockan för GPIO-port A där PA2 och PA3 sitter, vilka är de pins som är förberedda som USART2s Tx och Rx.

När klockan är avklarad kan vi börja göra övriga konfigurationer. För att PA2 och PA3 ska fungera som vi vill måste vi först sätta deras MODE, via MODE-Registret, till “alternate function”. Det gör att vi kan ändra pin-funktionen till andra än input/output. Dessa alternativa funktioner är kopplade specifikt till vissa pins. I vårt fall vill vi ha USART2-tx och rx funktionalitet, vilket vi får på PA2 och PA3 genom att ställa in AF7 i GPIOAs AFRL-register.

Efter det är bara konfigurationen av själva USART2 kvar.
Vi börjar med att välja en baud rate på standardvärdet 9600bps. Att välja ett standardvärde kan vara en god idé då det högst får differentiera 10% i baud rate mellan två UART-enheter. Vi bestämmer också utformningen av paketet som ska skickas. 8 bitar data-frame, 1 stop-bit och ingen parity-bit.
Sedan säkerställer vi att Control Register 2 och 3 är nollställda, vilket bland annat betyder att IrDA är avstängt.
Med det är alla konfigurationer gjorda och vi sätter bit 13 i CR1 till 1 och USART2 är därmed aktivt och redo för kommunikation.

För att överföring av data ska ske måste vi dock ha två funktioner till.
Den första är USART2_write(int). Den tar en integer som parameter och gör det möjligt att skicka data via USART.
Den andra funktionen är USART2_read() som gör det möjligt att ta emot data via USART.

En headerfil, uart.h, skapades också för att hålla stm32f4xx.h-biblioteket (som innehåller de nödvändiga register-definitionerna mm), samt stdio.h och USART2_Init(void)-deklarationen.

Dag 4

Efter att ha skapat drivrutiner för USART2 så skapades även drivrutiner för LEDs. Detta skulle kunna kombineras med USART för att styra LED:sen via terminalen. Här behövdes först en headerfil, Led.h, vilken dels definierar macros för de pins/bits som ska användas och även skapar typedef enum typer för LedColor_Type och LedState_Type. Detta ger namn åt de olika typerna(färger/states) utan att kompilatorn behöver hantera strings och gör även koden mer läsbar för utvecklare. Sist definieras en klass med LedColor_Type och LedState_Type som attribut och tre member funktioner.

Funktionerna är “Led(LedColor_Type _color,LedState_Type _state)”, konstruktor för led-objekt.
“setState(LedState_Type _state)”, Hanterar LED-objekts state.

“LedState_Type getState() const”, kontrollerar state på Led-objekt.
Dessa member funktioner definieras sedan i Led.cpp.

I Led.cpp inkluderas först Led.h så att vi får tillgång till alla macros mm. Sen börjar vi utveckla själva funktionerna. Det första vi gör är en konstruktör för att skapa LED-objekt. Den kommer, som vi såg i led.h, att ta parametrarna LedColor_Type _color och LedState_Type _state. Det första vi dock måste göra är att ge GPIOB tillgång till klockan. Detta så att de objekt vi skapar kan få ström och kommunicera med systemet.
Med en switch-funktion väljs sedan de konfigurationer som behöver göras för att varje led-objekt ska kopplas till rätt pin, samt om den ska vara på eller ej. Detta i enighet med våra i led.h definierade Macros.
Vi definierar även Led::setState(LedState_Type _state) funktionen, så att vi för varje konstruerat objekt kan byta state.
Sist tar vi också LedState_Type Led::getState(void)const. Med denna funktion kontrolleras och returneras ett led-objekts state.

Jag hade lite problem med precis hur olika typer av constructors fungerade, och även med klasser i allmänhet. För att lösa detta sökte jag information på nätet, främst Geeksforgeeks.org. Där hittade jag svar på mina frågor.

Det kan vara viktigt att nämna att hela LED-delen här förstås inte behöver gälla just LEDs. Denna kod skulle ju funka för alla enheter som bara behöver output. Att vi skriver LED och färger är bara för att göra ett tydligt exempel. Det vi egentligen väljer, när vi skriver color, är bitar i register som representerar Pins och funktioner för dessa.

I main.cpp inkluderas bara Led.h. Detta eftersom vi i den headern inkluderar alla andra bibliotek som vi har använt i övriga filer. Sen testas i main.cpp filen de olika drivrutiner vi har utvecklat. USART2_Init() kallas på och flertalet Led-objekt skapas, på lite olika sätt med olika minnes-allokeringar.

Resultat

Ingen direkt testning har gjorts av denna kod tyvärr. Detta då jag saknar hårdvaran för att göra detta möjligt. Saknar även mjukvaran för att köra simuleringar. Dock har koden körts i CubeIDEs miljö och kompilerat utan problem. Eller utan problem när alla bibliotek som programmet frågat efter inkluderats.

Det vi har lyckats göra är grundläggande drivrutiner för UART på stm32f411x-plattformen. Vi har även gjort drivrutiner för att kunna använda LEDs(eller liknande enheter) via plattformen.

Ytterligare funktionalitet bör läggas till för att kunna kombinera dessa drivrutiner och till exempel skicka instruktioner via UART från terminalen till våra olika LEDs. Eller få ut informationen om deras status.

Poängen med detta projekt har dock inte bara varit att skriva fungerande drivrutiner. Det har även varit ett sätt att lära sig mer om hur man programmerar hårdvara, öka förståelsen för register, klockor, bussar, portar, pins och hur dessa samverkar. Och ser man lärandet som det främsta målet har resultatet definitivt varit gott.

Referenser

Ludwig Simonssons lektioner på Nackademin 2023.

Dokumentation för STM32f411-plattformen (inkluderad i hårdvarumappen)

Geeksforgeeks hemsida, https://www.geeksforgeeks.org/

Youtube:

Robert Feranec,https://www.youtube.com/watch?v=dnfuNT1dPiM&t=2572s

WeeW, https://www.youtube.com/watch?v=LvfjFihNSgE&t=332s

Microtronics Academy, https://www.youtube.com/watch?v=EwTk5eLeXz0&t=570s