Inbyggda System, arkitektur och design

Hantera en LED med UART-protokoll

Bam **Mozafar**
bam.mozafar@yh.nackademin.se

Inledning

UART (Universal Asynchronous Receiver-Transmitter) är en viktig kommunikationsenhet inom inbyggda system, som möjliggör datautbyte mellan mikrokontrollrar och andra komponenter eller enheter.

Syftet med detta projekt är att utforma och implementera en UART-drivrutin för STM32F411x mikrokontrollerplattformen, med målsättningen att förbättra förståelsen för både UART-kommunikation och mikrokontrollerprogrammering.
För att uppnå detta, kommer rapporten först att beskriva bakgrunden och teorin bakom UART-kommunikation, inklusive viktiga begrepp som överföringshastighet, start- och stoppbitar, och paritet. Detta kommer skapa en grundläggande förståelse av hur UART fungerar och varför det är en så pass populär kommunikationsmetod inom inbyggda system.

Därefter kommer rapporten att presentera den utvecklingsmiljö och verktyg som har använts under projektets gång, såsom STM32F411x utvecklingskort, STM32CubeIDE och nödvändiga bibliotek. Detta kommer att ge en överblick över de tekniska resurserna som krävs för att utföra arbetet och hur de samverkar.

Slutligen kommer rapporten att beskriva den faktiska implementationen av UART-drivrutinen i detalj, inklusive hur de olika funktionerna och inställningarna konfigureras. Detta kommer att illustrera hur teorin bakom UART-kommunikation kan tillämpas i praktiken och hur mikrokontroller programmeras för att hantera dataöverföringar.

Vidare kommer exempel på användning av drivrutinen att presenteras, samt en diskussion om eventuella förbättringar och vidareutveckling av projektet.

Bakgrund & Teori

UART-kommunikation

Universell asynkron mottagare-sändare (UART) är en hårdvarukomponent som används för att överföra och ta emot data seriellt mellan två enheter. UART är populärt inom inbyggda system på grund av dess enkelhet och låga krav på systemresurser. Det är en fullduplex, asynkron kommunikationsmetod, vilket innebär att sändare och mottagare inte behöver en gemensam klocka för att synkronisera dataöverföringen.

Den här sektionen kommer att förklara grundläggande koncept och terminologi för UART-kommunikation.

Baud Rate

Baudrate är ett mått på överföringshastigheten i bitar per sekund (bps) och bestämmer hur snabbt data skickas över UART-gränssnittet. En högre baudrate innebär snabbare överföring, men ökar också risken för fel i kommunikationen. Vanliga baudrates inkluderar 9600, 19200 och 38400 bps.

Några faktorer som kan påverka baudrate inkluderar:
Kvalitet på kommunikationskanalen – Där en högkvalitativ kanal kan hantera högre baudrate och ge snabbare dataöverföring. Omvänt kan en lågkvalitativ kanal begränsa baudrate och därmed dataöverföringshastigheten.
Avstånd – Där ju längre avståndet mellan enheterna som kommunicerar, desto lägre blir baudrate. Detta beror på att signalen försvagas när den färdas över längre avstånd.
Störningar – Där elektromagnetisk störning och andra störningar kan påverka överföringshastigheten och därmed baudrate.
Hårdvarans kapabilitet – Där de enheter som kommunicerar måste vara kapabla att hantera den önskade baud rate för att uppnå en effektiv och korrekt kommunikation.

Start- och Stoppbitar

UART-kommunikation använder start- och stoppbitar för att indikera början och slutet av varje dataenhet som skickas. Start biten är alltid en logisk nolla, medan stoppbiten är en logisk etta.

När en enhet (sändaren) vill skicka data, omvandlar den data från parallell till seriell form och lägger till en start bit (vanligtvis 0) och en eller flera stoppbitar (vanligtvis 1) för att bilda ett datapaket.
Därefter skickas datapaketet sekventiellt, en bit i taget, över den seriella linjen (TX - sändare) till mottagaren (RX - mottagare).

På mottagarsidan övervakar UART-mottagaren (RX - mottagaren) den seriella linjen för start biten. När start biten detekteras, läser mottagaren resten av bitarna i datapaketet enligt den förinställda baud raten och dataformatet. Efter att ha läst alla bitar, omvandlar mottagaren seriella data till parallella data och tar bort start- och stoppbitarna.

Paritet

Många protokoll inkluderar felkontrollsmekanismer, såsom paritetsbitar eller checksummor för att upptäcka och eventuellt korrigera fel som kan inträffa under överföringen. Paritet är en metod för att upptäcka fel i dataöverföringen. Paritet kan vara jämn, udda eller ingen. En extra bit läggs till varje dataenhet för att säkerställa att antalet ettor är jämnt (jämn paritet) eller udda (udda paritet). Om ett enskilt bit fel inträffar kommer pariteten att ändras och mottagaren kommer att veta att det finns ett fel.

Sändaren räknar antalet 1: or i databitarna och bestämmer om paritetsbiten ska vara 0 eller 1 för att uppnå önskad paritet (jämn eller udda).
Sändaren lägger till paritetsbiten till databitarna och skickar den resulterande bitsekvensen till mottagaren.

Mottagaren tar emot bitsekvensen och räknar antalet 1: or, inklusive paritetsbiten. Mottagaren kontrollerar om antalet 1: or motsvarar den förväntade pariteten (jämn eller udda).

Om pariteten stämmer antar mottagaren att dataöverföringen är korrekt.
Om pariteten inte stämmer innebär det att det har inträffat ett fel under överföringen, och mottagaren kan begära att sändaren skickar om dataenheten.

STM32F411x mikrokontroller
STM32F411x är en serie av mikrokontrollers som utvecklats av STMicroelectronics.
De är baserade på ARM Cortex-M4 kärnan och har en klockfrekvens på upp till 100 MHz.
Dessa mikrokontrollers är kraftfulla, energieffektiva och kostnadseffektiva, vilket gör dem lämpliga för en mängd olika inbyggda systemapplikationer.

Programmering av STM32F411x mikrokontroller
För att programmera STM32F411x mikrokontroller används vanligtvis en integrerad utvecklingsmiljö (IDE) som STM32CubeIDE. Denna IDE erbjuder en enkel och effektiv plattform för att skriva, kompilera och ladda upp kod till mikrokontrollern.
STM32F411x använder HAL (Hardware Abstraction Layer) biblioteket för att erbjuda en hög nivå av abstraktion och förenkla programmeringen av mikrokontrollernas olika funktioner och perifera enheter, såsom UART.

Metod och utförande

Rapport
I projektet för att utveckla en UART-drivrutin för STM32F411x-plattformen inleddes arbetet med att noggrant studera den tekniska dokumentationen och databladet för plattformen och UART-protokollet. Detta var viktigt för att få en bättre förståelse av både plattformens och protokollets egenskaper och begränsningar. Detta gav mig insikt i hur dessa teknologier fungerar och hur de kan kombineras för att skapa en fungerande UART-drivrutin. Jag lade särskild vikt vid att förstå UART-kommunikation, registerkonfiguration och klockhantering. När jag hade fått en djupare förståelse för de teknologier jag arbetade med, påbörjade jag utvecklingen av UART-drivrutinen.

Det första steget var att initiera USART-protokollet och dess komponenter genom funktionen "USART2_Init". Jag aktiverade klocktillgången för USART2 och GPIOA, konfigurerade pins PA2 (TX) och PA3 (RX) för alternativ funktion och ställde in lämpliga värden för att säkerställa korrekt kommunikation. Dessa inställningar involverade att välja rätt klockkälla och förskjutningar, konfigurera pin-namn och inställningar för att matcha det specifika UART-protokollet och säkerställa att allt var korrekt kopplat och initierat.

Därefter konfigurerade jag UART:en genom att ställa in baudraten till 9600 bps, ange att TX och RX skulle arbeta med 8 bitars data, 1 stoppbit och ingen paritet, samt aktiverade UART:en. Dessa inställningar säkerställde att UART:en var konfigurerad för att hantera dataöverföring på både ett effektivt och pålitligt sätt.

Nästa steg var att skapa funktionerna "USART2_write" och "USART2_read" för att hantera överföring och mottagning av data.
"USART2_write" överför data till terminalen och kontrollerar att överföringsstatusen är tom och redo att ta emot nästa byte innan en ny byte kan skickas. Detta säkerställer att vi inte skickar data när överföringsbufferten är full, vilket skulle resultera i förlorade data.
"USART2_read" läser in data som skickas till mikrokontrollerna och kontrollerar om det finns mer data att hämta. Denna funktion hanterar mottagna data på ett säkert och effektivt sätt, och ser till att all data har lästs och behandlats korrekt.

Med UART-drivrutinen på plats gick jag vidare till att utveckla en periferi-drivrutin som skulle fungera tillsammans med UART-drivrutinen. Jag skapade en Led-klass med attribut för färg och tillstånd för LED-lampan samt funktioner för att ställa in och hämta tillståndet för LED-lampan. Detta innebar att jag skapade en klass för att styra en LED-lampa och definiera pins och mode-bitar för varje färg. Dessutom säkerställde jag att klassen var flexibel nog att stödja olika LED-typer och konfigurationer, samt att den kunde hantera flera LED-enheter samtidigt. Denna periferi-drivrutin möjliggjorde styrning av en LED-lampa och kunde användas tillsammans med UART-drivrutinen för att skicka och ta emot data.

Genom att kombinera dessa drivrutiner kunde jag skapa ett exempelprogram i "main.cpp" som demonstrerade hur man kan använda UART och LED-drivrutinerna för att kommunicera med externa enheter och styra LED-lampor baserat på mottagna kommandon.

Slutligen var organiseringen och strukturering av dokumentationen och Github-repot viktigt för att säkerställa att projektet var lättillgängligt och förståeligt för andra, och även för minska risken för att information skulle gå förlorad.

Kod

Github: https://github.com/donsueco/Projekt-UART-Protokoll.git

uart.h:
Filen uart.h definierar en UART-klass för att hantera UART-kommunikation på en mikrokontroller.

Inkluderingsvakter används för att undvika att filen inkluderas mer än en gång, vilket kan leda till kompileringsfel.

```
#ifndef UART_H
#define UART_H
```

<cstdint>-biblioteket inkluderas för att definiera typer med fast bredd, såsom uint8_t, uint16_t, etc., vilket säkerställer att variablerna har samma storlek på alla plattformar.

Klassen UART deklareras och dess medlemsvariabler och funktioner grupperas under offentliga och privata åtkomstmodifierare.
Offentliga medlemmar och funktioner är tillgängliga för användning utanför klassen, medan privata medlemmar och funktioner endast är tillgängliga inom klassen.

Konstruktorn för UART-klassen deklareras och används för att skapa en instans av klassen samt initiera medlemsvariabler.

En offentlig medlemsfunktion init deklareras för att initiera UART:en, såsom att aktivera klockor, konfigurera pinnar och ställa in lämpliga register.
En annan offentlig medlemsfunktion write deklareras, som tar en parameter av typen uint8_t och används för att överföra data via UART-kommunikation.
En tredje offentlig medlemsfunktion read deklareras som returnerar en uint8_t, och används för att läsa inkommande data via UART-kommunikation.

Privata medlemsfunktioner deklareras för att hantera interna konfigurationsuppgifter för UART-klassen.

Funktionen enableClocks aktiverar klockor som krävs för att UART:en ska fungera.
Funktionen configurePins konfigurerar de pinnar som används för UART-kommunikation.
Slutligen används funktionen configureRegisters för att konfigurera UART:ens register för att säkerställa korrekt funktion.

uart.cpp

Filen uart.cpp innehåller implementationen av UART-klassen som är deklarerad i uart.h.

Filen börjar med att inkludera nödvändiga headerfiler: uart.h för att inkludera UART-klassdeklarationen och <libopencm3/stm32/rcc.h> samt <libopencm3/stm32/gpio.h> för att använda libopencm3-biblioteket för att konfigurera och kontrollera klockor, GPIO och andra mikrokontrollerresurser.

Konstruktorn för UART-klassen implementeras och tar två parametrar: USART_TypeDef *usart och uint32_t baudrate.
Dessa parametrar används för att initiera medlemsvariabler _usart och _baudrate.

Medlemsfunktionen init implementeras och börjar med att anropa privata metoder enableClocks, configurePins och configureRegisters för att konfigurera nödvändiga inställningar för UART-kommunikation. Därefter aktiveras UART-enheten genom att sätta USART_CR1_UE-biten i kontrollregistret.

Medlemsfunktionen write implementeras och tar en uint8_t-parameter. Funktionen väntar på att USART_SR_TXE-biten i statusregistret ska vara satt, vilket indikerar att överföringsbufferten är tom och redo att ta emot nästa byte.
Därefter skickas bytet genom att skriva det till dataregistret.

Medlemsfunktionen read implementeras och returnerar en uint8_t. Funktionen väntar på att USART_SR_RXNE-biten i statusregistret ska vara satt, vilket indikerar att mottagningsbufferten innehåller data. Därefter returneras det mottagna bytet genom att läsa från dataregistret.

De privata medlemsfunktionerna enableClocks, configurePins och configureRegisters implementeras för att konfigurera UART:ens interna inställningar. enableClocks aktiverar

klockor för USART2 och GPIOA. configurePins ställer in pinnarna PA2 och PA3 för att användas för USART2 TX och RX, respektive. configureRegisters konfigurerar USART2-register för att ställa in baudraten, använda 8 bitars data, 1 stoppbit och ingen paritet, samt aktiverar sändning och mottagning.

Led.h
Filen Led.h innehåller definitionen av Led-klassen som används för att styra en LED-lampa. Filen börjar med att inkludera nödvändiga headerfiler: <cstdint> för att använda standardtyper som uint8_t, samt <libopencm3/stm32/gpio.h> för att använda libopencm3-biblioteket för att kontrollera GPIO-pinnar.

För att förhindra dubbelinkludering av headern, används #pragma once, som säkerställer att Led.h endast inkluderas en gång under kompileringen.

Klassen Led definieras. Den innehåller en privat enum-klass State med två värden: ON och OFF, som representerar LED-lampans tillstånd.

För att lagra och hantera LED-lampans tillstånd, definieras privata medlemsvariabler _pin, _port och _state. _pin och _port används för att lagra GPIO-pinnens och portens nummer, medan _state lagrar LED-lampans nuvarande tillstånd.

Konstruktorn för Led-klassen definieras med två parametrar: uint32_t pin och uint32_t port, som används för att initiera medlemsvariablerna _pin och _port. Konstruktorn ställer också in _state till State::OFF som standard.

Följande offentliga medlemsfunktioner deklareras:

void on():
Sätter LED-lampans tillstånd till ON och uppdaterar GPIO-pinnen för att tända lampan.
void off():
Sätter LED-lampans tillstånd till OFF och uppdaterar GPIO-pinnen för att släcka lampan.
void toggle():
Byter LED-lampans tillstånd mellan ON och OFF och uppdaterar GPIO-pinnen därefter.
State getState() const:
Returnerar LED-lampans nuvarande tillstånd som en State-enum.

Led.cpp
Filen Led.cpp innehåller implementationen av Led-klassen som används för att styra en LED-lampa. Filen börjar med att inkludera nödvändiga headerfiler: <libopencm3/stm32/rcc.h> och <libopencm3/stm32/gpio.h> för att använda libopencm3-biblioteket för att kontrollera GPIO-pinnar och klockor, samt "Led.h" som innehåller definitionen av Led-klassen.

Följande medlemsfunktioner implementeras i filen:

Led::Led(uint32_t pin, uint32_t port):
Konstruktorn för Led-klassen tar emot pin och port som parametrar och initierar medlemsvariablerna _pin och _port med dessa värden.
Den ställer också in _state till State::OFF som standard. Funktionen aktiverar också klockan för den angivna porten och konfigurerar GPIO-pinnen som en utgång.

void Led::on():
Funktionen sätter LED-lampans tillstånd till ON, uppdaterar GPIO-pinnen för att tända lampan och uppdaterar _state-variabeln.

void Led::off():
Funktionen sätter LED-lampans tillstånd till OFF, uppdaterar GPIO-pinnen för att släcka lampan och uppdaterar _state-variabeln.

void Led::toggle():
Funktionen byter LED-lampans tillstånd mellan ON och OFF genom att först kontrollera _state-variabeln. Om den är ON, anropas off()-funktionen, och om den är OFF, anropas on()-funktionen.

Led::State Led::getState() const:
Funktionen returnerar LED-lampans nuvarande tillstånd som en State-enum.

Denna implementation av Led-klassen möjliggör kontroll av en LED-lampa genom att hantera dess tillstånd och interagera med GPIO-pinnar på mikrokontrollern.

main.cpp
Filen main.cpp innehåller huvudprogrammet som utnyttjar Led- och Uart-klasserna för att styra en LED-lampa och kommunicera över UART. Filen inkluderar nödvändiga headerfiler: "uart.h" för att använda Uart-klassen, "Led.h" för att använda Led-klassen, och <libopencm3/cm3/nvic.h> för att använda NVIC-funktioner (Nested Vectored Interrupt Controller) i libopencm3-biblioteket.

Huvudprogrammet följer följande steg:

1- Variabler och objekt deklareras:
   Uart uart(USART2) skapar en instans av Uart-klassen och konfigurerar den för att använda USART2-periferin.
   Led led(GPIO_PIN_5, GPIOA) skapar en instans av Led-klassen för att kontrollera en LED-lampa kopplad till pinne 5 på port A.
2- main()-funktionen börjar med att initiera uart-objektet genom att anropa dess init()-metod. Detta konfigurerar UART-kommunikationen med de inställningar som har angetts tidigare i Uart-klassen.
3- Programmet går sedan in i en oändlig while-loop där följande sker:
   A - uint8_t data = uart.read(); läser inkommande data från UART och lagrar det i data-variabeln.
   B - Ett switch-uttryck används för att utvärdera data-variabeln och utföra olika åtgärder baserat på dess värde:

i. Om data är '1', sätts LED-lampan till på genom att anropa led.on().
ii. Om data är '0', stängs LED-lampan av genom att anropa led.off().
iii. Om data är något annat värde, ignoreras det och ingenting händer.

Programmet fortsätter att loopa och väntar på ytterligare data över UART för att styra LED-lampan.

Denna implementation demonstrerar hur Led- och Uart-klasserna kan användas tillsammans för att skapa ett enkelt system som styrs över UART-kommunikation. LED-lampan kan slås på och av beroende på de inkommande datavärdena som skickas till mikrokontrollern via UART.

Förbättringar och vidareutveckling

Projektet syftade till att utveckla en UART-drivrutin för STM32F411x-plattformen och en LED-periferi-drivrutin. Genom grundlig forskning och utveckling skapades en effektiv och pålitlig UART-drivrutin samt en LED-drivrutin för att styra LED-lampans tillstånd. Felsökning och konfiguration var utmanande men övervanns genom noggranna tester och justeringar.

Framtida förbättringar kan innefatta kodoptimering och prestanda, avancerade funktioner som avbrottshantering, förbättrad felhantering, utveckling av en mer användarvänlig och skalbar LED-drivrutin, integration med andra kommunikationsprotokoll och enheter, ytterligare testning och validering, samt utökad dokumentation och användarhandledning. Dessa förbättringar kan öka projektets användbarhet, flexibilitet och potential för framtida tillämpningar och utveckling, samt anpassa det för att möta föränderliga krav och behov.

Referenslista

1. Getting started with STM32: STM32 step-by-step - stm32mcu
2. STM32F411xC/E advanced Arm®-based 32-bit MCUs - Reference manual
3. Arm® Cortex®-M4 32b MCU+FPU, 125 DMIPS, 512KB Flash, 128KB RAM, USB OTG FS, 11 TIMs, 1 ADC, 13 comm. interfaces (st.com)
4. STM32 Nucleo-64 boards (MB1136) - UM1724
5. https://youtu.be/dnfuNT1dPiM