NACKADEMIN

Design och implementation av en LED-drivrutin för STM32-mikrokontroller med stöd för UART-kommunikation

Jimmy kroneld

Jimmy.kroneld@yh.nackademin.se

Innehållsförteckning

Inledning

Denna akademiska rapport beskriver utvecklingen av en drivrutin för UART-kommunikation på STM32F411x-plattformen, med syftet att ge en grundlig beskrivning av designprocessen för en högkvalitativ och pålitlig drivrutin.

Rapporten innehåller en översikt av plattformens egenskaper och begränsningar samt en detaljerad beskrivning av utvecklingsprocessen för drivrutinen. Drivrutinen har utformats och implementerats på plattformen, och har testats och utvärderats för att säkerställa att den möter kraven på tillförlitlighet och effektivitet.

Rapporten innehåller också en beskrivning av de olika delarna i den utvecklade drivrutinen och hur de fungerar tillsammans för att hantera kommunikationen mellan mikrokontrollern och andra enheter som använder UART-protokollet.

Genom att läsa denna rapport kan läsaren förvänta sig att få en grundlig förståelse av utvecklingsprocessen för en drivrutin för UART-kommunikation på STM32F411x-plattformen, inklusive hur den kan användas i praktiska tillämpningar. Rapporten presenterar också resultaten av testerna som utfördes för att säkerställa drivrutinens tillförlitlighet och effektivitet.

Länk till github repo: https://github.com/Shallange/UART_Driver_Led.git

Dagbok

Jag deltog under lektioner och vi bekantade oss med hårdvaran och de teoretiska koncepten som ligger till grund för seriell datakommunikation. Under lektionerna gick vi även kort igenom två välkända protokoll SPI och I2C för att sedan dyka lite djupare in i det tredje ”protokollet” ”UART”. (UART är det protokoll som användes i detta projekt).

Jag tog mig tiden att noggrant analysera varje kodfil individuellt för att få en djupare förståelse för dess funktioner och samband med andra filer. Jag kommenterade varje fil utförligt och samtidigt letade upp beteckningar och studerade de val som gjorts och deras motiveringar.
Ett exempel på detta var MODER-registret, där jag noga analyserade hur konfigurering av dess funktionalitet verkligen gick till.

Medan jag gjorde min analys av kodfilerna stötte jag på vissa svårigheter när det gällde att förstå varför vissa kodrader var skrivna på ett specifikt sätt. Det tog tid att förstå sambanden men till slut föll polletten ner och jag kunde tydligt se vilka bitar som kod raden syftade på.
Jag använde främst referensmanualen som vi hade tillgång till för att leta rätt på de registerna som kod raden i fråga använde sig utav.

Genom att ha kombinerat den teoretiska kunskapen om UART med praktiskt arbete och djupt utforskande av koden så känner jag att jag har skaffat mig en solid grund för att fortsätta lära mig arbeta med UART kommunikation. Har helt enkelt fått en övergripande förståelse för hur URART används i applikationen jag arbetat med.

UART:

UART står för ”Universal Asynchronous Receiver/Transmitter” och används för seriell datakommunikation, där data överförs bit för bit.
UART fungerar som så att data skickas och tas emot via två signalledare, en för dataöverföring(TX - Transmit) och en för datamottagning(RX-Receive).

UART är användbart över olika plattformar och enheter. Det stöds av de flesta mikrokontroller och processorer vilket gör det populärt val för seriell kommunikation. UART kommunikationen kan användas i både något som heter halv-duplex och full-duplex-läge. I halv-duplex så kan data överföras åt en riktning åt gången, vilket innebär att enheten måste växla mellan att ta emot och skicka data. I full-duplex kan data både skickas och tas emot på samma gång, vilket möjliggör en mer effektiv och snabbare kommunikation mellan enheterna. Man kan även använda UART till andra ändamål som tillexempel skicka kommandon och läsa svar eller övervaka och logga dataströmmar.

Kod

Led.cpp

I denna fil finns implementationen av en LED-klass som hanterar styrningen av olika LED-lampor genom att definiera deras färg och status. Varje LED har en unik färg/GPIO-pinne och en Switch-sats avgör vilken GPIO-pinne som ska användas beroende på vilken LED-lampa som ska manipuleras.

Vi börjar med att inkludera ”LED.h” biblioteket, i detta bibliotek finns det definitioner för vilka lägen lamporna ska ha och även definitionen av klassen LED som har 2 privata attribut ”color” och ”state”.

Sedan skrivs en konstruktör för klassen LED

En konstruktör är en speciell metod som körs när ett objekt av klassen skapas. I detta fall tar konstruktor två argument, ”färg” och ”status” som definierade LED lampa.

Sedan tas dessa 2 argument och sätter värdet på det privata attribut som fanns i klassen
därefter aktiverar den klockan för LED-porten och konfigurerar LED-pinsen baserat på dess färg och status.

Det två metoderna ”setState” och getState”:

- setState() Har hand om att sätta statusen på vald LED lampa till antingen ON eller OFF.
- getState() hämtar dess nuvarande tillstånd, om vald lampa är ON eller OFF

Koden använder GPIO-pinnarna på mikrokontrollern för att styra LED-lamporna. Varje LED har en unik färg som bestämmer vilken GPIO-pinne som används för att styra dess tillstånd. Koden använder en switch-sats för att avgöra vilken GPIO-pinne som ska användas beroende på vilken LED-lampa som ska manipuleras.

Led.h

I denna fil(header) så definieras konstanter för GPIO porten, klock-signalen, pinsen för olika LED färger och slutligen MODE-bits för varje LED-färg.

Två typedef enums (LedColor_Type och LedState_Type) är definierade och representerar LED färgen och statusen.

Led klassen är deklarerad med privata variabler för färg och status, en konstruktor för led som tar in dessa två argument som tidigare nämnt.
led klassen har även två metods deklarationer, ”setState” och ”getState”.

Porten för LED lamporna är GPIOB
varje GPIO port har 4 ”32-bits” register för konfigurering, i koden används MODER registret

- MODER (GPIO port mode register)

- OTYPER (GPIO port output type register)
- OSPEEDR (GPIO port output speed register)
- PUPDR (GPIO port pull-up/pull-down register)

MODER styr funktionaliteten för en specifik GPIO-pinne man kan genom att sätta ett värde ändra om

Registerna är 16 par av bits, och varje par representerar läget för en specifik pinne, det finns 4 möjliga kombinationer av värden som bestämmer läget(funktionaliteten för vald pinne)
(i Led.cpp så sätts exempelvis Röd lampa till output för att kunna tända/släcka lampan med hjälp av att skicka en låg eller hög signal till pin som lampan är kopplad till)

- 00: Ingångsläge(Standardläge)(Input)
  - Ta emot signaler eller data från annan enhet/krets.
- 01: Allmänt ändamål utgångsläge(Output)
  - Styra eller generera signaler som skickar till andra enheter/kretsar(tända släcka lampor).
- 10: alternativt funktionsläge(Alternate function)
  - UART,SPI eller I2C kommunikation via vald pin.
- 11: Analogt läge(Analog)
  - Mäta(läsa) eller genera analoga signaler.

Man sätter då konstanter på pinsen lamporna är kopplade till och även de MODE_BITS som tillhör dessa pins(varje pin i en GPIO-port har sina motsvarande bits i MODER-registret)

Led lamporna är kopplade till pinsen 12-13-14-15(markerade med grön på bilden)
#define LED_RED_PIN (1U<<14)

#define LED_GREEN_PIN (1U<<12)

#define LED_YELLOW_PIN (1U<<13)

#define LED_BLUE_PIN (1U<<15)

Här sätter man motsvarande bits för pinsen 12-13-14-15 till ”General purpose output mode)
genom att sätta första biten till 1 (markerade med orange på bilden)

#define LED_RED_MODE_BIT (1U<<28)

#define LED_GREEN_MODE_BIT (1U<<24)

#define LED_YELLOW_MODE_BIT (1U<<26)

#define LED_BLUE_MODE_BIT (1U<<30)

Main.cpp

I denna fil inkluderas Led.h filen för att komma åt deklarationer som är kopplade till LED-lamporna.
(I ”Led.h” inkluderas ”UART.h” och på så sätt kan man komma åt USART2 funktionen).
variabler av ledState_Type(enum som håller ON och OFF) deklareras

En instans av klassen led vid namn ”led1” skapas utanför main funktionen och argumenten ”RED” (för LedColor_Type) och ”ON” (för LedState_Type) skickas in.

Inne i main funktionen så ropar man på USART2_Init() för att initiera seriell kommunikation.

Efter det ges ett par exempel på hur man kan skapa objekt och styra dessa lampobjekt med hjälp av olika metoder som skapats i andra filer(kommenterat mer i koden).

Uart.cpp

I den här filen så definieras USAR2_Init funktionen som deklarerades i uart.h filen.

Filen har även 2 andra funktionsdefinitioner:

- USART2_write()
- USART2_read()

USART2_Init:

" Word length may be selected as being either 8 or 9 bits by programming the M bit in the USART_CR1 register" [1]

Bit 17 i APB1ENR sätts till 1, för att aktivera klockåtkomst för UART2 och
Bit 0 i AHB1ENR sätts till 1, för att aktivera klockåtkomst till PORT-A(GPIOA).
Detta gör det möjligt att aktivera pins som är relaterade till valda porten och sätta alternativ funktionalitet till dom. Konfiguration av standard baudrate och dataformat för UART blir också tillgänglig.
Som i Led.h filen så ändrar man pin funktionaliteten med hjälp av MODER.

- *00: Ingångsläge(Standardläge)(Input)*
  - *Ta emot signaler eller data från annan enhet/krets.*
- *01: Allmänt ändamål utgångsläge(Output)*
  - *Styra eller generera signaler som skickar till andra enheter/kretsar(tända släcka lampor).*
- *10: alternativt funktionsläge(Alternate function)*
  - *UART,SPI eller I2C kommunikation via vald pin.*
- *11: Analogt läge(Analog)*
  - *Mäta(läsa) eller genera analoga signaler.*

För pins 2 och 3 i PORT-A så väljs "alternativt funktionsläge" eftersom dessa kommer att vara kommunikations portat med UART(TX och RX), bitarna 4-7(orange markerade) rensas först och sedan sätts bit 5 och 7 till 1(både pin 2 och 3 får "10" som är alternativt funktions läge).
varje pin i MODER har som sagt 2 bits var med totalt 4 möjliga kombinationer, vi sätter pin 2 och 3 till "10"(alternativ funktionsläge).

Efter att valt alternativ läge på pinsen måste man ytterligare specificera vilken kommunikations protokoll vi kommer att använda, på bilden (AFRL registret) kan man se att varje pin har 4 bit i stället och har 15 möjliga kombinationer(AF0 - AF15) totalt 16 pins är fördelade på ett Low register(0–7) och ett High register(8–15). Vi tittar endast på Low register då vi har aktiverat pin 2 och 3(bit 8–15).
bitarna 8–11(pin 2) sätts till 0111(AF7) och bitarna 12–15(pin 3) sätts till 0111(AF7).
Varför just AF7 kan man tänka, på bilden kan man se det som är orange markerat
att PA2 och PA3 på AF7 specificerar PA2 till USART2_tx och PA3 till USART2_rx.

Sedan konfigureras USART2 (USART register map) man börjar med att sätta vilken baud-rate man ska ha, Baudrate är viktigt eftersom den styr överföringshastigheten mellan enheten och den anslutna

[1] STM32F411x Reference Manual (2018), s. 509.

enheten som datan kommer överföras till. I koden sätts den till(9600bps) och varför man sätter den just till 9600 är för att de är en vanligt förekommande standardhastighet som ändvänts förr och fortsatt att vara standard än idag. I denna kod har vi inte heller behovet av att ha en högre baudrate eftersom vi skickar 8 bitar i taget(så 9600 är mer än tillräckligt, plus att eftersom 9600 bps är standard minskar det risken för inkompatibilitet med andra enheter) .
0x000C(0000 0000 0000 1100) sätter CR1 bit 2(RE) och bit 3 (TE) till 1 och har bit 12 (M) fortfarande på 0 vilket gör att den arbetar i 8bits läget.

CR2 och CR3 rensas för säkerhetskull, så att inte ytterligare inställningar är satta som skulle kunna skapa oönskade konflikter(Dessa används inte).

Slutligen sätts bit 13(UE som står för UART Enable) till 1.

SR(Status register)

Ger information om vad för status data överföringen har med bits som blir aktiva vid specifik status. Några av Status flaggorna:

- TC(bit 6) Transmission complete
- TXE(bit 7) Transmit data register empty
- RXNE(bit 5) Receiver not empty
- PE(bit 0) Parity error
- FE(bit 1) Framing error

Både TXE och TC är som standard satt till 1, eftersom ingen data skickas så räknas både överföringen som klar(Transmission complete) och att ingen data skickas just nu (Transmit data register empty)och är redo att acceptera data för överföring.

DR(Data register)

Är ett register som används både till att lagra data som ska skickas/överföras och data som tagits emot. Ska något skickas så skrivs den datan till registret och UASART tar datan och skickar den bit för bit. Vid mottagning av data läser i stället UART från registret bit för bit och återskapar den ursprungliga byten.

USART2_write()

Funktionen USART2_write() används för att överföra data och gör det så länge bit 7 (0x0080 = 0000 0000 1000 0000) inte är 1(när bit 7 är 1 så betyder det att ”transmit data registret” är tomt dvs att de finns ingen mer data som är köad för överföring).
Därefter så skrivs värdet av variabeln ”ch” till DR-registret.
”ch &0xFF” denna bitvis AND-operation med värdet 0xFF maskar alla bitar i ”ch” förutom de Least signifikant 8 bitarna. Eftersom vi konfigurerat att använda 8-bitars dataöverföring så kommer hela byten skickas till DR.

USART2_read()

Funktionen USART2_read() används för att ta emot data från DR och gör det så länge bit 5(0x0020 = 0000 0000 0010 0000) inte är 0(när bit 5 är 0 så betyder det att Receive data registret är tomt dvs att de finns ingen mer data att läsa).
Därefter så returneras värdet från DR(hela Byten 8 bitar).

Uart.h

Denna fil är en header-fil för UART som har två funktionsdeklarationer:

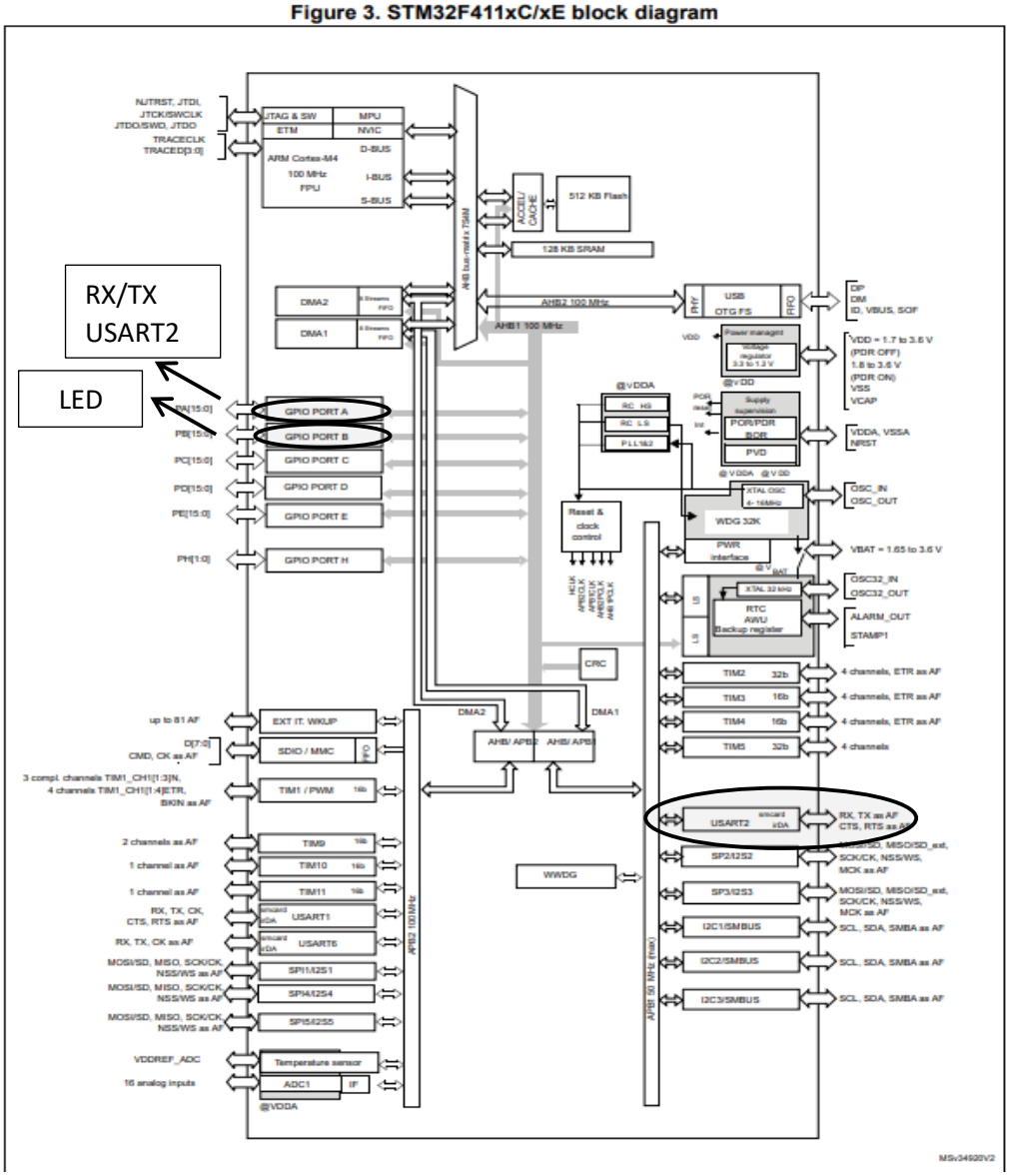
- USART2_Init()
- test_setup()
    - Denna funktion användes i början för att testa att kommunikationen fungerade
      I koden är den utkommenterad då den endast används vid test och den definieras inte någonstans i koden.

USART2_Init() funktionen definieras sedan i [uart.cpp](#) filen initierar UART-modulen, sätter upp klockan och porten, och konfigurerar UART med standard baudrate och dataformat.

Referenser och bilagor

STM32f411xC/xE Block Diagram

Figure 3. STM32F411xC/xE block diagram



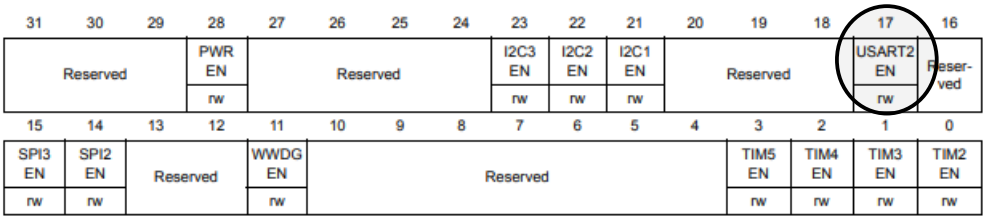
RCC_APB1ENR

6.3.11 RCC APB1 peripheral clock enable register (RCC_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.



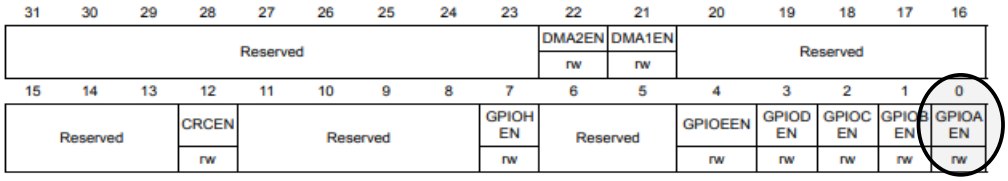
AHB1ENR

6.3.9 RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.



GPIO register map

8.4.11 GPIO register map

The following table gives the GPIO register map and the reset values.

Table 26. GPIO register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOA_ MODER	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIO port mode register

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..E and H)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

GPIO alternate function low register(AFR)

8.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..E and H)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFRLy:** Alternate function selection for port x bit y (y = 0..7)

These bits are written by software to configure alternate function I/Os

AFRLy selection:

0000: AF0
0001: AF1
0010: AF2
0011: AF3
0100: AF4
0101: AF5
0110: AF6
0111: AF7
1000: AF8
1001: AF9
1010: AF10
1011: AF11
1100: AF12
1101: AF13
1110: AF14
1111: AF15

AFR(Alternate function mapping)

Table 9. Alternate function mapping

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	TIM1/2	TIM3/4/5	TIM8/9/10 /11	I2C1/2/3	SPI1/SPI2/ I2S2/I2S2e xt	SPI3/I2Sext /I2S3	USART1/2/3/ I2S3ext	UART4/5/ USART6	CAN1/2 TIM12/13/ 14	OTG_FS/ OTG_HS	ETH	FSMC/SDIO /OTG_FS	DCMI		
Port A	PA0	-	TIM2_CH1_ ETR	TIM 5_CH1	TIM8_ETR	-	-	-	USART2_CTS	UART4_TX	-	-	ETH_MII_CRS	-	-	-	EVENTOUT
	PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-	USART2_RTS	UART4_RX	-	-	ETH_MII _RX_CLK ETH_RMII_ _REF _CLK	-	-	-	EVENTOUT
	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-	USART2_TX	-	-	-	ETH_MDIO	-	-	-	EVENTOUT
	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	-	-	USART2_RX	-	-	OTG_HS_ULPI_ D0	ETH _MII_COL	-	-	-	EVENTOUT
	PA4	-	-	-	-	-	SPI1_NSS	SPI3_NSS I2S3_WS	USART2_CK	-	-	-	-	OTG_HS_SOF	DCMI_ HSYNC	-	EVENTOUT
	PA5	-	TIM2_CH1_ ETR	-	TIM8_CH1N	-	SPI1_SCK	-	-	-	-	OTG_HS_ULPI_ CK	-	-	-	-	EVENTOUT
	PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO	-	-	-	TIM13_CH1	-	-	-	DCMI_ PIXCLK	-	EVENTOUT
	PA7	-	TIM1_CH1N	TIM3_CH2	TIM8_CH1N	-	SPI1_MOSI	-	-	-	TIM14_CH1	-	ETH_MII_RX_DV ETH_RMII _CRS_DV	-	-	-	EVENTOUT
	PA8	MCO1	TIM1_CH1	-	-	I2C3_SCL	-	-	USART1_CK	-	-	OTG_FS_SOF	-	-	-	-	EVENTOUT
	PA9	-	TIM1_CH2	-	-	I2C3_ SMBA	-	-	USART1_TX	-	-	-	-	-	DCMI_D0	-	EVENTOUT
	PA10	-	TIM1_CH3	-	-	-	-	-	USART1_RX	-	-	OTG_FS_ID	-	-	DCMI_D1	-	EVENTOUT
	PA11	-	TIM1_CH4	-	-	-	-	-	USART1_CTS	-	CAN1_RX	OTG_FS_DM	-	-	-	-	EVENTOUT
	PA12	-	TIM1_ETR	-	-	-	-	-	USART1_RTS	-	CAN1_TX	OTG_FS_DP	-	-	-	-	EVENTOUT
	PA13	JTMS-SWDIO	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA14	JTCK-SWCLK	-	-	-	-	-	-	-	-	-	-	-	-	-	-	EVENTOUT
	PA15	JTDI	TIM 2_CH1 TIM 2_ETR	-	-	-	SPI1_NSS	SPI3_NSS/ I2S3_WS	-	-	-	-	-	-	-	-	EVENTOUT

USART register map

19.6.8 USART register map

The table below gives the USART register map and reset values.

Table 88. USART register map and reset values

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	USART_SR	Reserved																						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
	Reset value																							0	0	1	1	0	0	0	0	0	0
0x04	USART_DR	Reserved																						DR[8:0]									
	Reset value																							0	0	0	0	0	0	0	0	0	0
0x08	USART_BRR	Reserved																DIV_Mantissa[15:4]												DIV_Fraction [3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	USART_CR1	Reserved																OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	IRE	RWU	SBK
	Reset value																	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	USART_CR2	Reserved																	LINEN	STOP [1:0]		CLKEN	CPOL	CPHA	LBCL	Reserved	LBDIE	LBDL	Reserved	ADD[3:0]			
	Reset value																		0	0	0	0	0	0	0	Reserved	0	0		0	0	0	0
0x14	USART_CR3	Reserved																				ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
	Reset value																					0	0	0	0	0	0	0	0	0	0	0	0
0x18	USART_GTPR	Reserved																GT[7:0]								PSC[7:0]							
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0