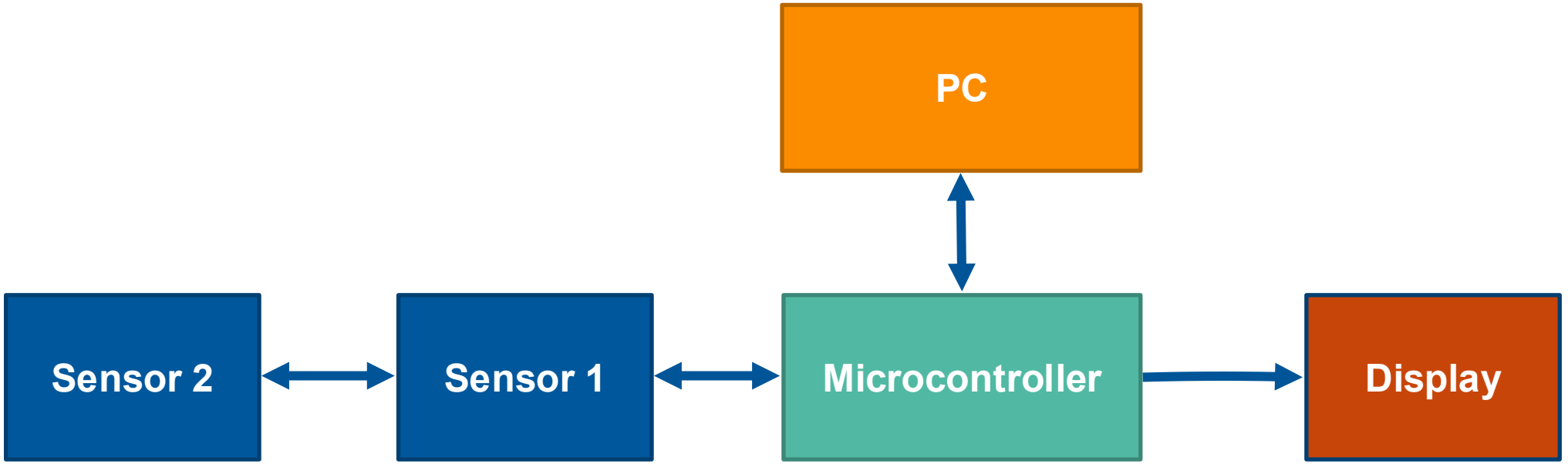
COMMUNICATION

COMMUNICATION

- Most of the time it is not enough to get our information and display it on a screen
- We often want to send it somewhere
- To do this we would need to communicate to the outside world
- We can do this either wirelessly or with wires

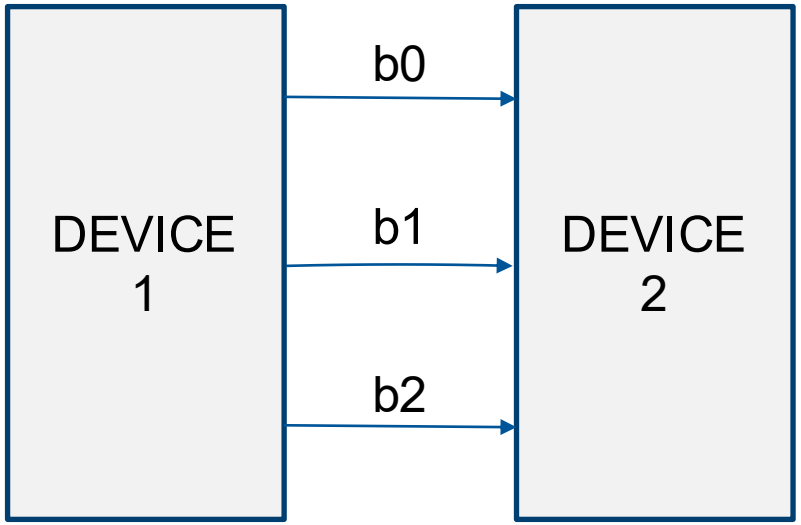
COMMUNICATION

- Microcontrollers do not exist in isolation
- Data must often be transferred between components or peripherals for processing or conversion

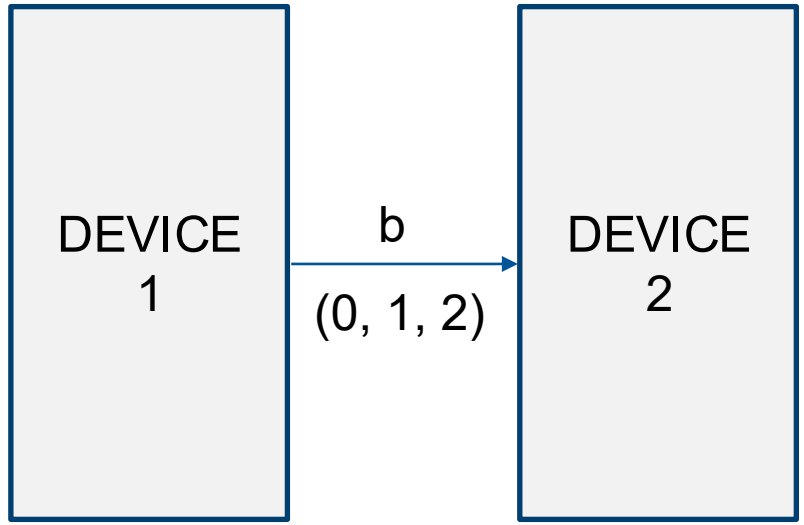


COMMUNICATION

PARALLEL COMMS



SERIAL COMMS



- We will constrain ourselves to serial comms in this course

SERIAL COMMUNICATION

- Some common serial comms protocols

SPI – Serial Peripheral Interface

UART – Universal Asynchronous Receiver Transmitter

I^2C (or IIC or I2C) – Inter-Integrated Circuit

USB – Universal Serial Bus

SERIAL COMMUNICATION

ASYNCHRONOUS COMMS

- Data line only
- Predetermined period per bit transfer
- Devices must be configured to use this timing period/frequency

E.g. UART

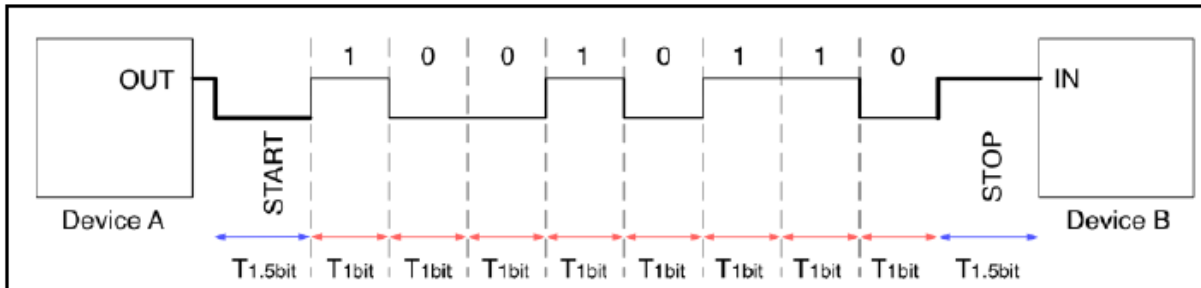


Figure 2: The timing diagram of a serial communication without a dedicated clock line

SYNCHRONOUS COMMS

- Data line and a clock line
- Clock line dictates timing of the data transfer
- Logic levels are sent and received according to clock signal levels/edges.

E.g. I2C, SPI

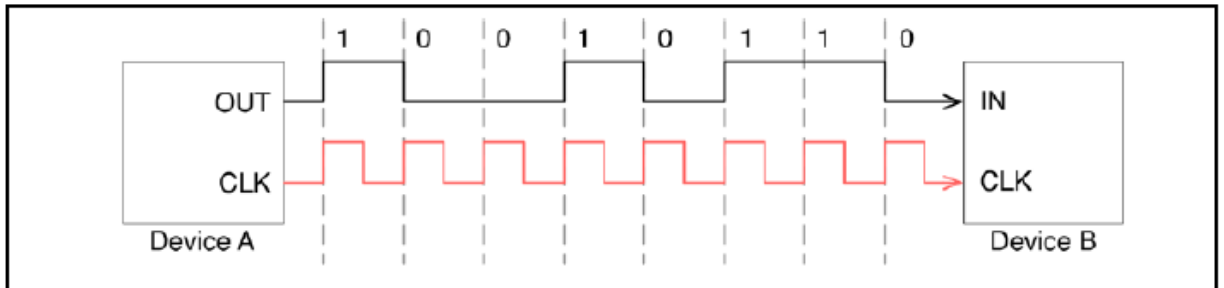
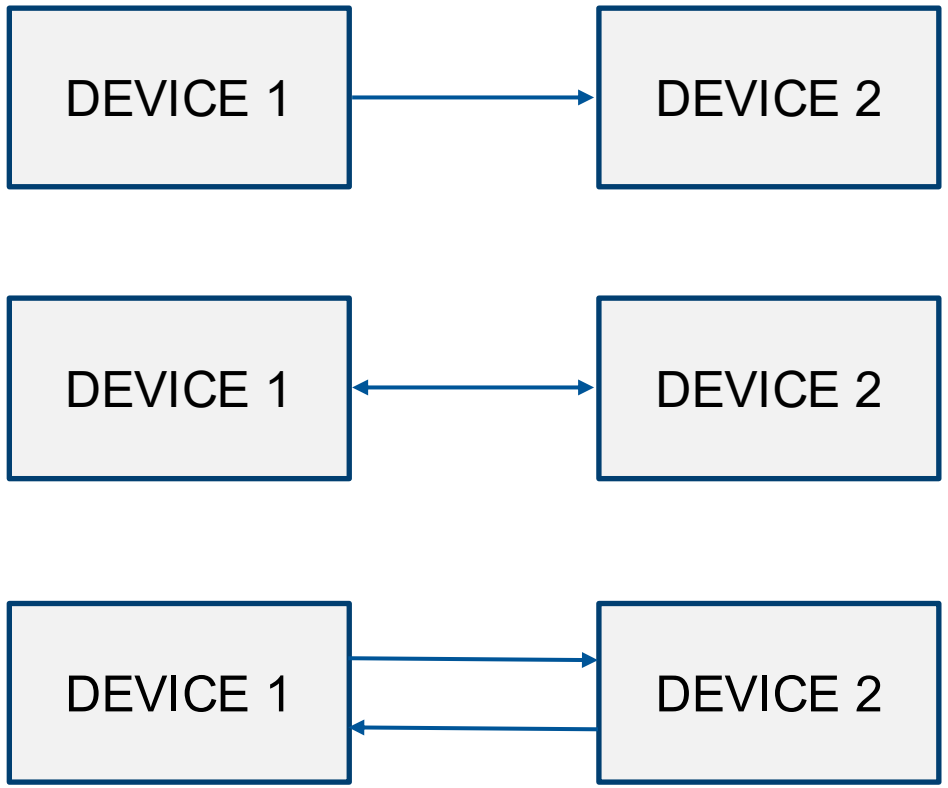


Figure 1: A serial communication between two devices using a shared clock source

Images taken from “Mastering STM32” by Carmine Noviello

SERIAL COMMUNICATION

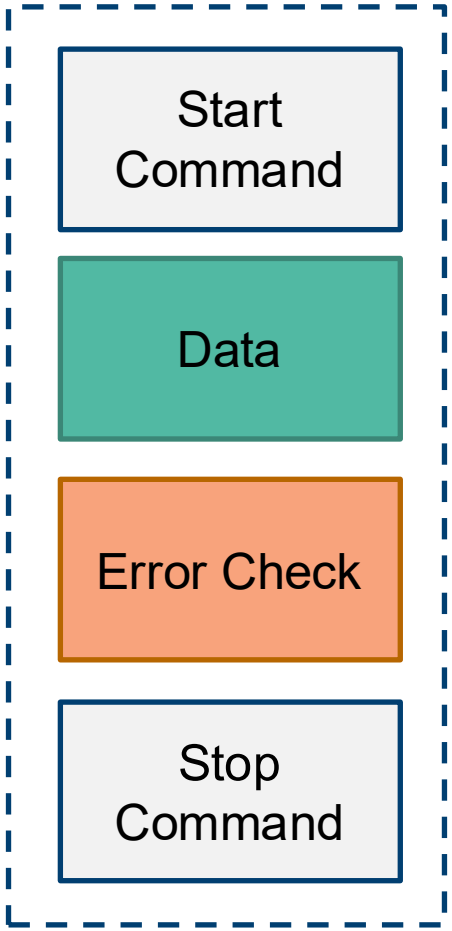


Simplex:
Communications in one direction only

Half-Duplex:
One direction at a time – both possible

Full-Duplex:
Comms in both directions simultaneously

SERIAL COMMUNICATION



SIMPLE PACKET STRUCTURE

Start Command: Defined condition to signal start of transfer

Data: The data to be transferred - Grouped in ‘words’

Error Check: Optional error checking term

Stop Command: Defined condition signalling end of transfer

COMMUNICATIONS

UART

UART

- Stands for Universal Asynchronous Receiver Transmitter
- Is not a protocol or comms standard
- Is a physical piece of hardware which controls the transmission and reception of serial data (bits sent one at a time)
- Can be built in to a microcontroller IC or be a dedicated IC itself
- Asynchronous – No clock line. UART must be configured to a defined transmission rate – Baud rate

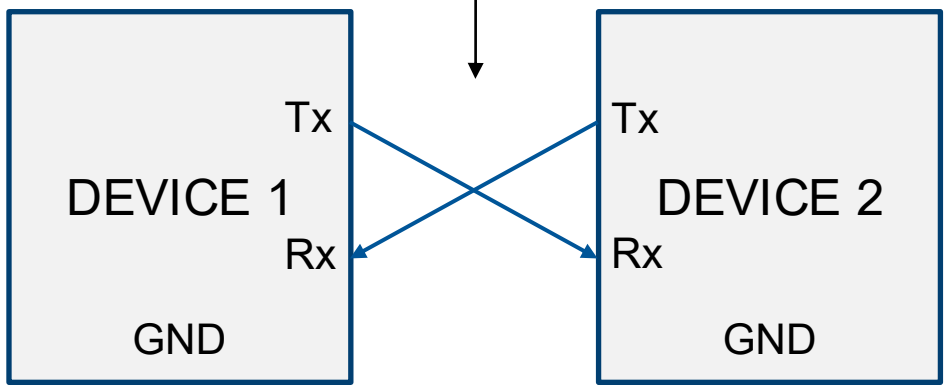
UART

- 2 wire interface – TX (transmit) and RX (receive)

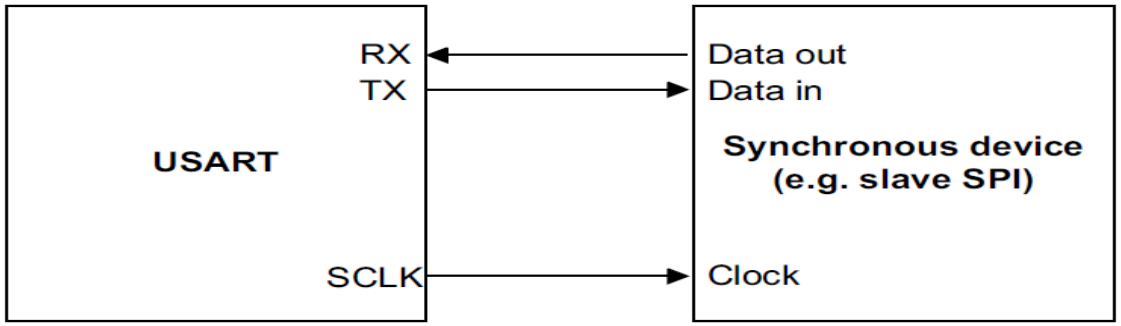
  Devices must share the same GND level. Some refer to this connection as an additional wire making it a 3 wire interface

- Communication between 2 devices only

Full Duplex Connection



USART



Synchronous Comms

Universal Synchronous/Asynchronous Receiver Transmitter

- STM32 implements a USART peripheral which allows configuration in synchronous or asynchronous modes
- The USART can be used to drive synchronous or asynchronous RS-232 or synchronous SPI signals.

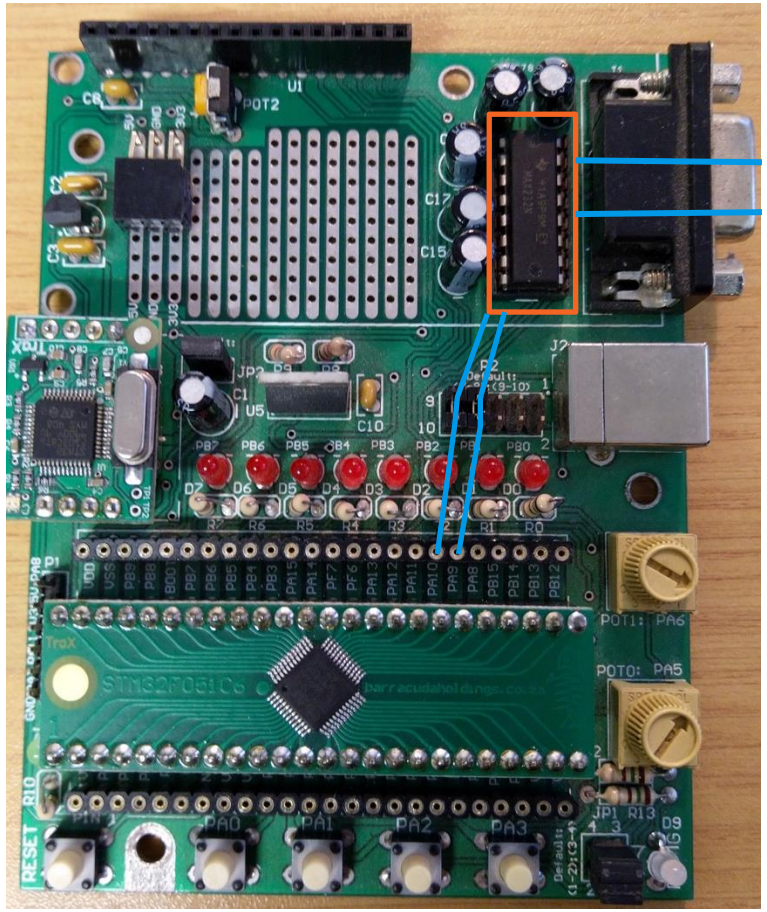
RS-232

- RS-232 is an asynchronous serial communication interface standard that uses the UART hardware interface
- RS-232 uses differential voltages to represent logic levels:

High:	-3 V to -15 V
Low:	3 V to 15 V

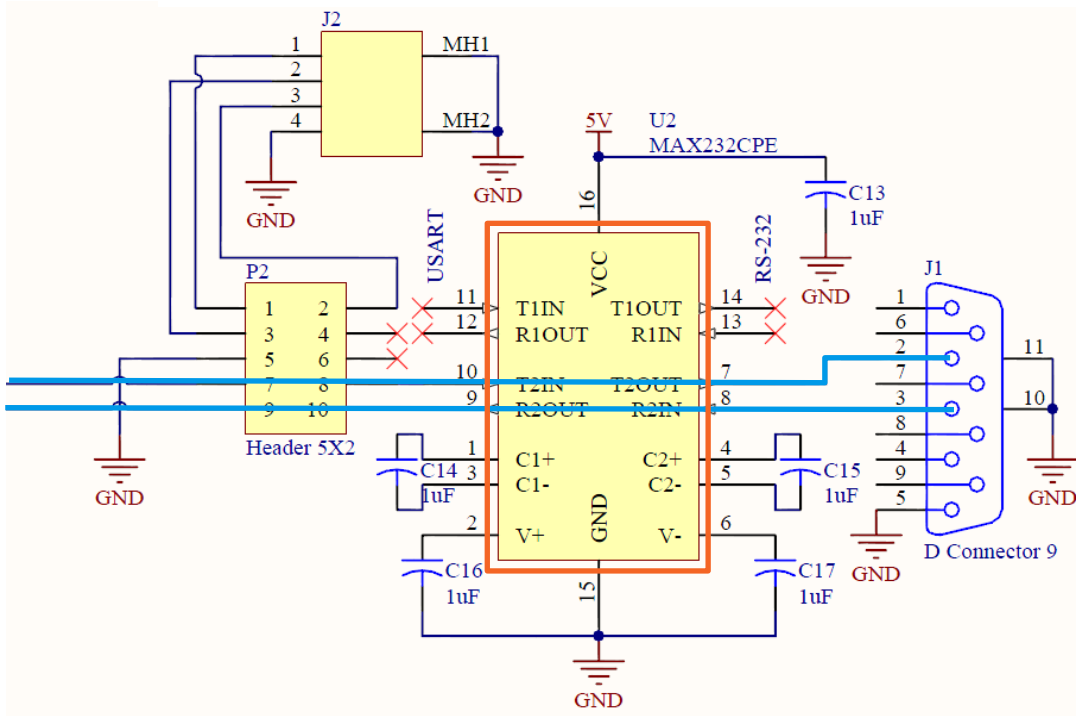
- Clearly need additional circuitry to achieve this

RS-232

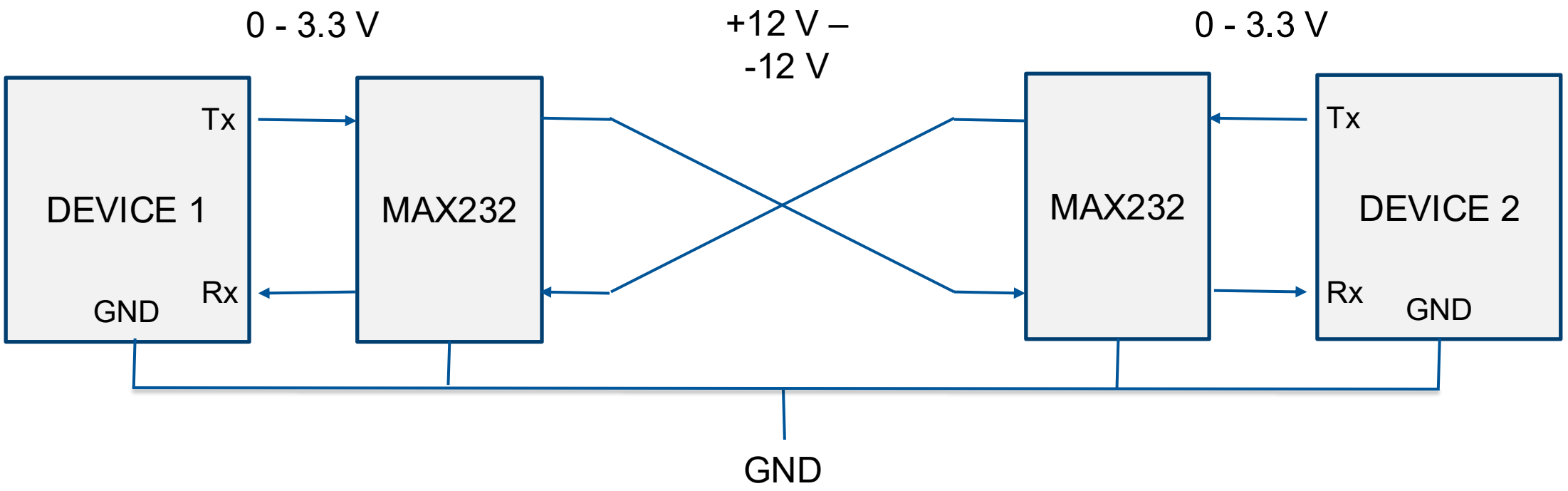


- Done with the MAX232 IC on the dev board. Connects USART to DB9 connector

PA9
PA10



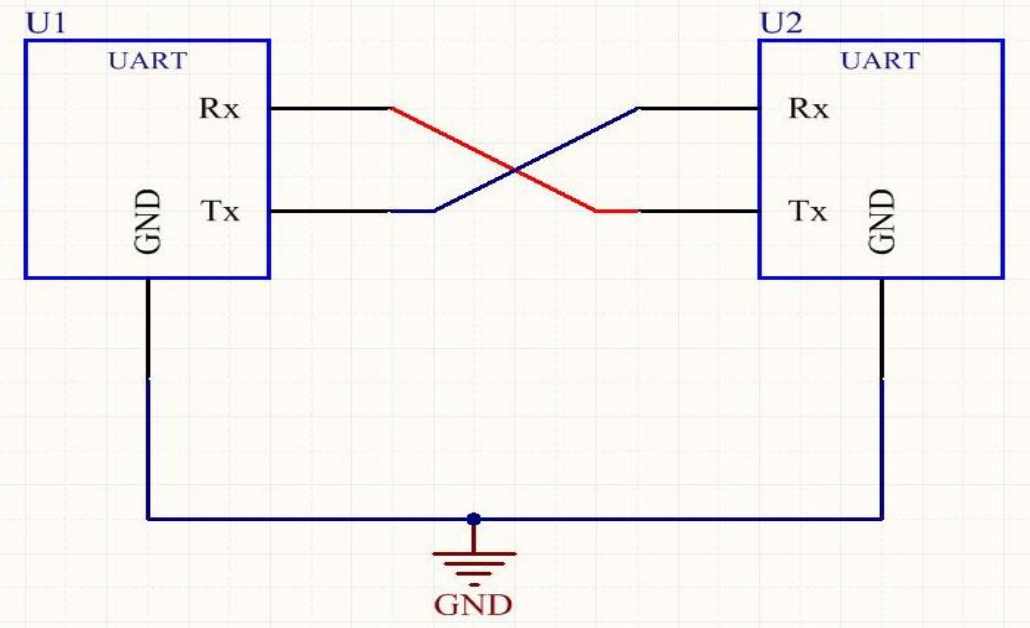
RS-232


0 - 3.3 V
+12 V –
-12 V
0 - 3.3 V
Tx
DEVICE 1
Rx
GND
MAX232
MAX232
Tx
DEVICE 2
Rx
GND
GND

RS-232

- Can work over long distances: > 20 m
  - High voltage swing
  - Matched device impedances
  - Twisted pair cabling
- UART to UART comms possible over short distances without the need for conversion to RS-232 voltage standard.

UART



Asynchronous Comms

Asynchronous Comms

- Specify clock speed to allow communications:

  Baud Rate (bits/s)

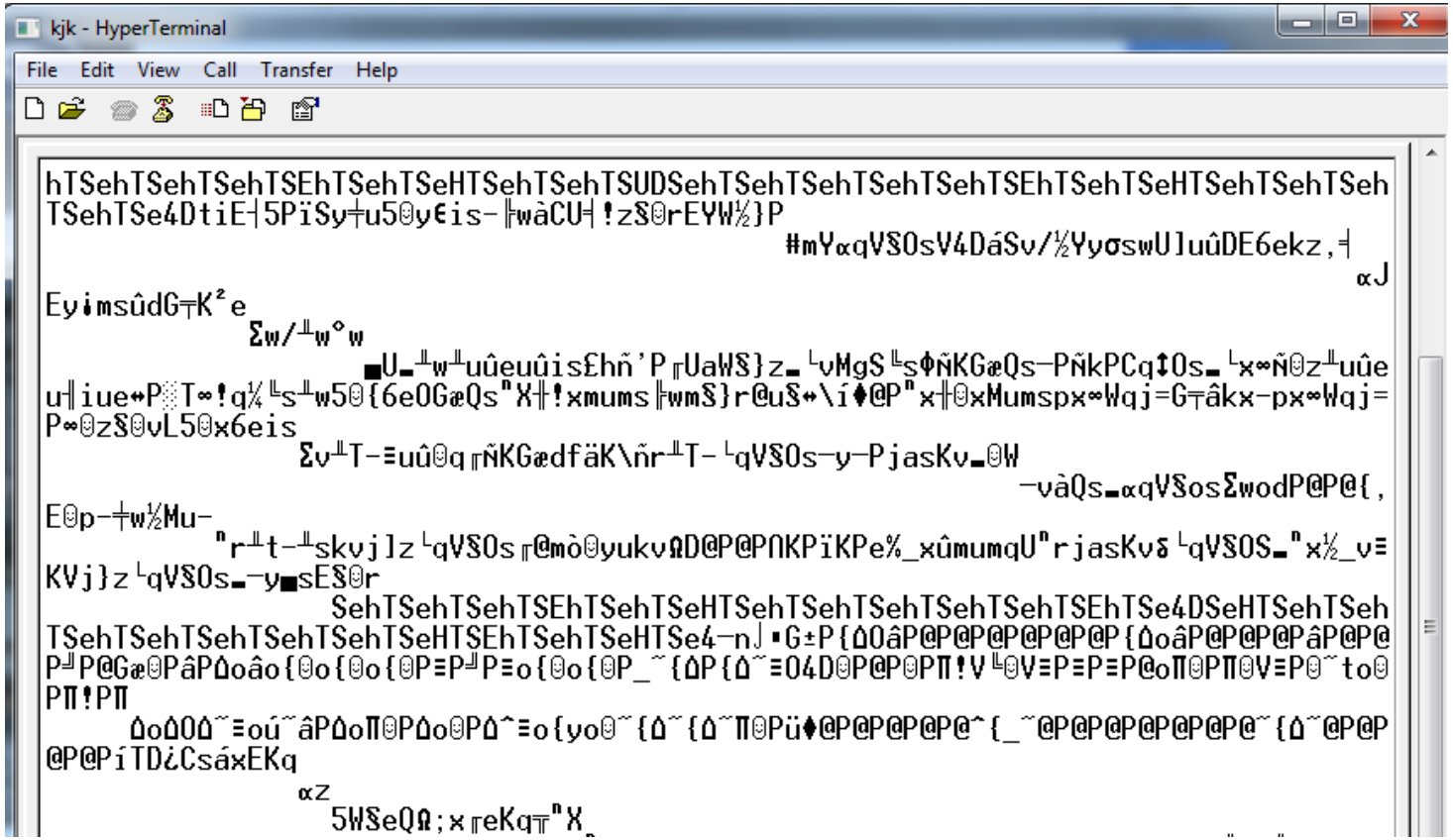
UART

Baud Rate Mismatch

- Various possible baud rates:

E.g.
1.2 kbps, 9.6 kbps, 19.2 kbps
115.2 kbps

- RS-232 designed for rates <20 kbps



UART

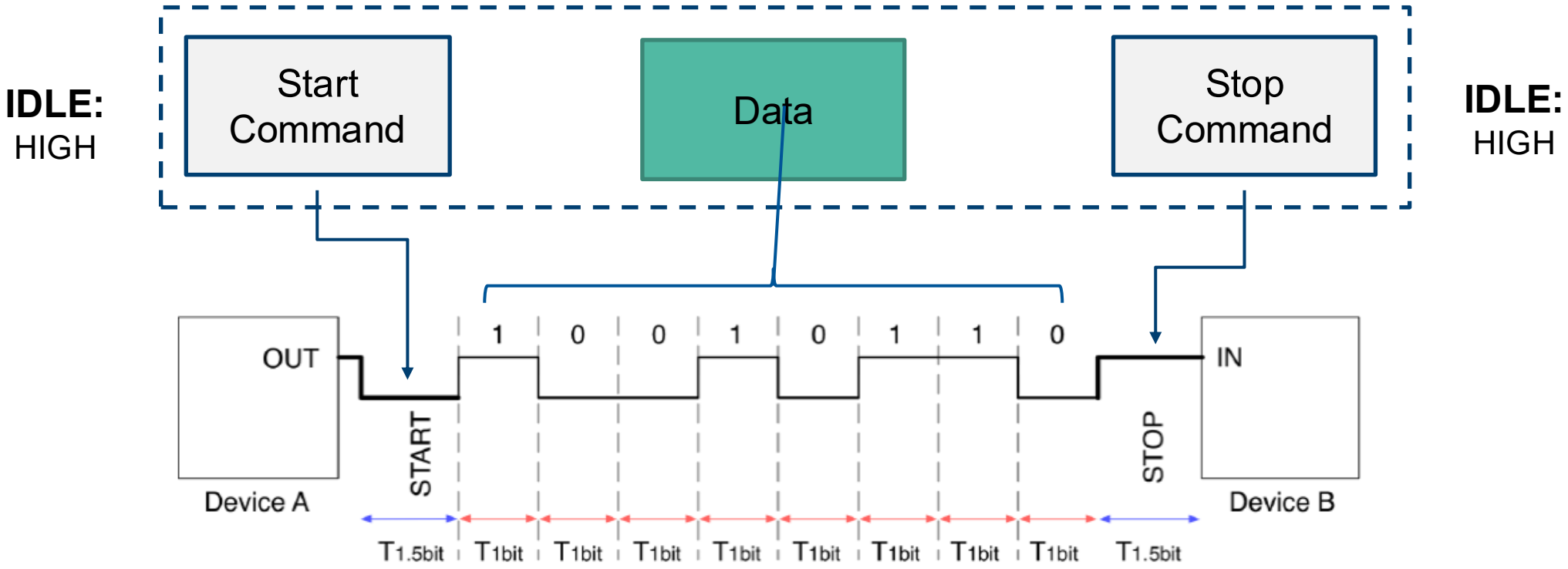
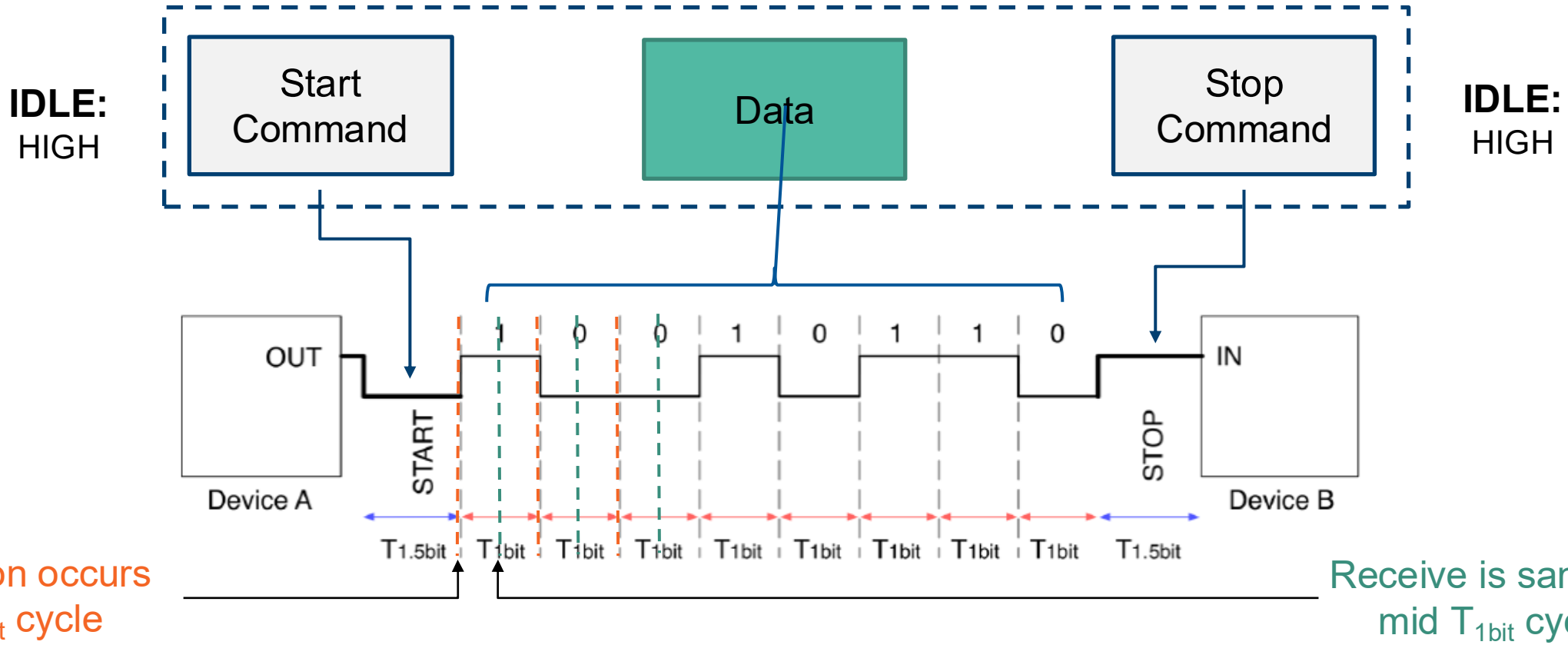


Figure 2: The timing diagram of a serial communication without a dedicated clock line

Image taken from “Mastering STM32” by Carmine Noviello

UART

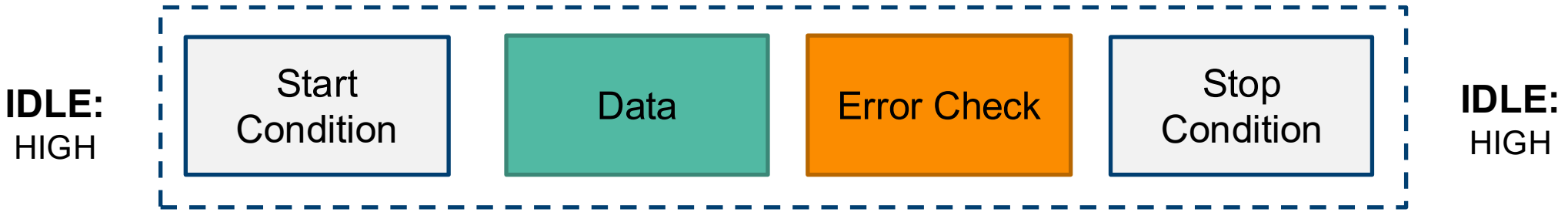


Transmission occurs each T_{1bit} cycle

Receive is sampled mid T_{1bit} cycle

Image taken from “Mastering STM32” by Carmine Noviello

UART



Error Checking: PARITY

- An extra bit is added to the data byte – either 1 or 0 such that:

Even Parity: Sum of 1s in data + parity must be even

or

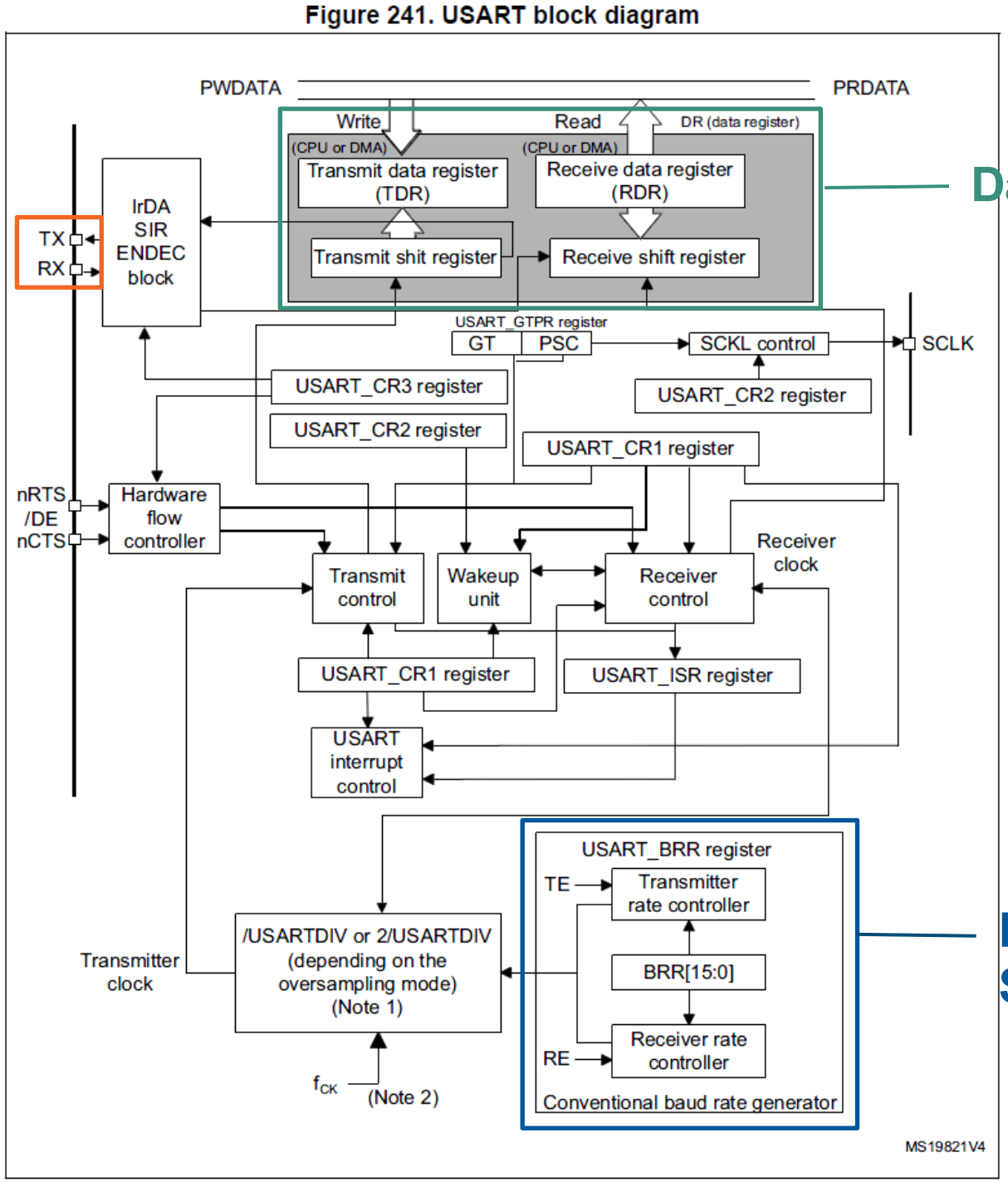
Odd Parity: Sum of 1s in data + parity must be odd

Image taken from “Mastering STM32” by Carmine Noviello

COMMUNICATIONS

UART – STM32F0

UART



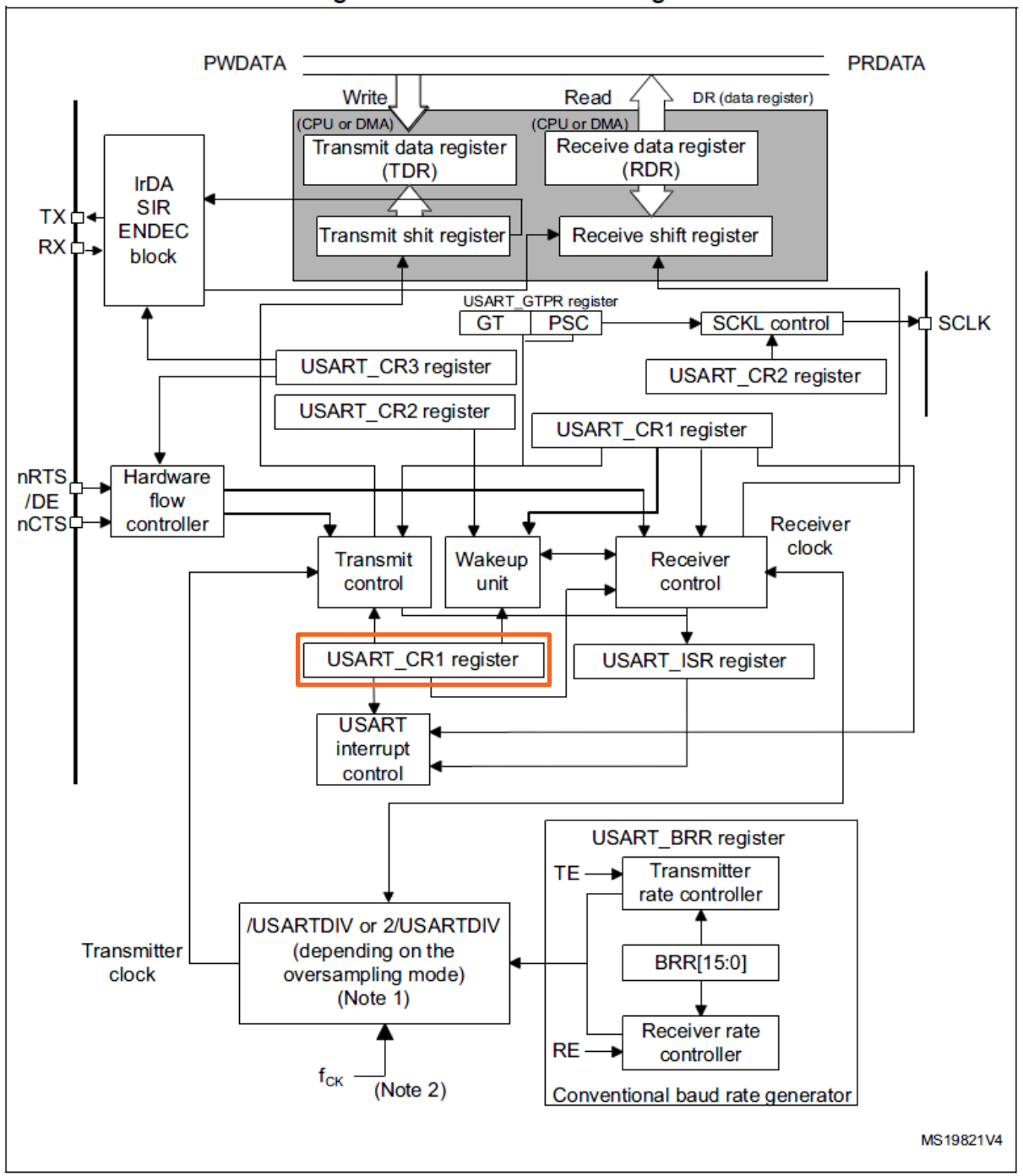
Data Tx & Rx

Baud Rate Settings

- STM32F051 has 2 USARTs: USART1 & USART2
- USART1 is on the APB2 clock, USART2 is on the APB1 clock
- Tx and Rx pins must be set as alternate functions on the GPIO
- USART_BRR register configures baud rate
- USART_TDR and RDR registers are data buffers used for transmit and receive

UART

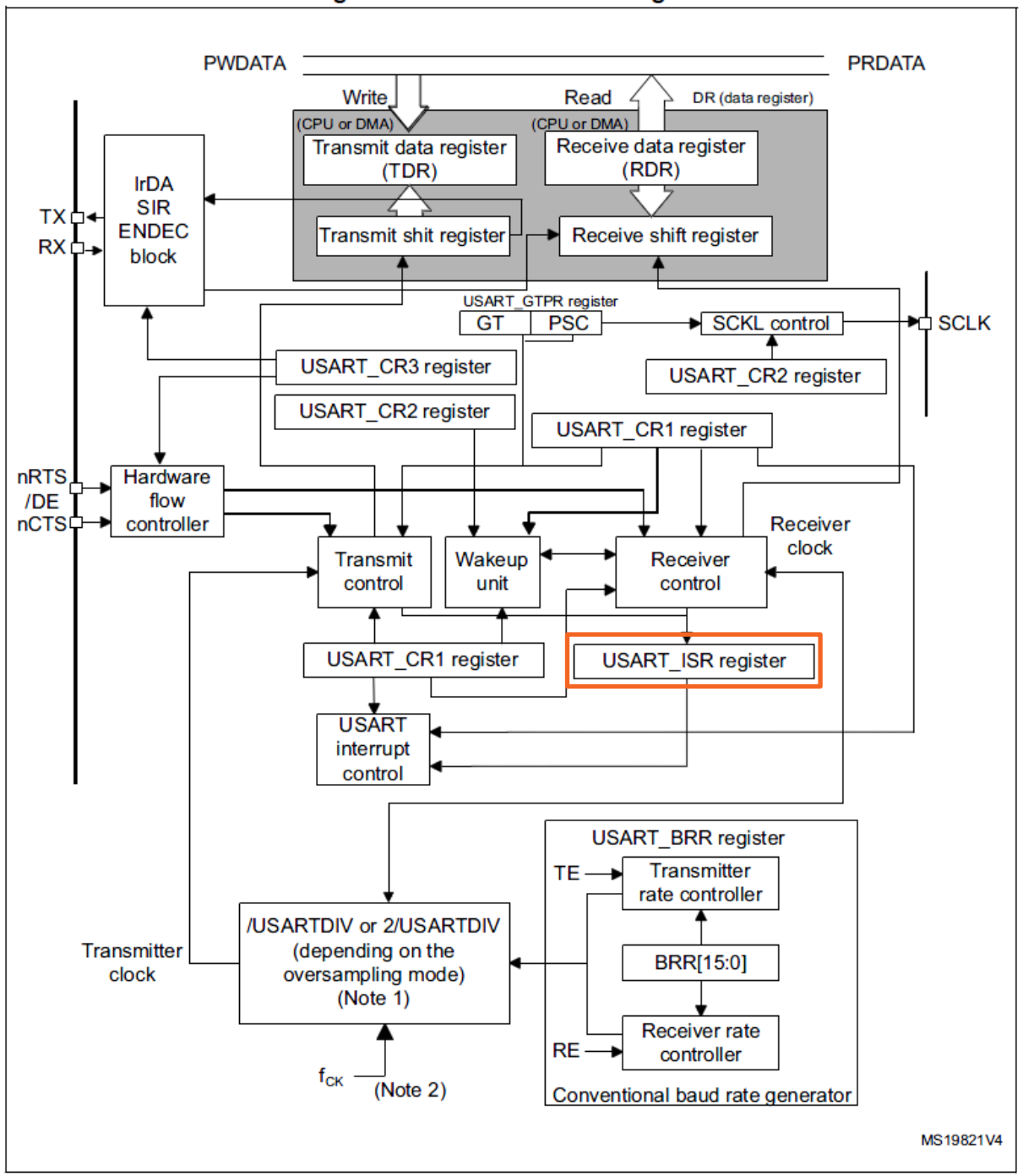
Figure 241. USART block diagram



USART_CR1 - Control Register 1

- Word length to 7, 8 or 9 bits
- Number of stop bits (CR2)
- Transmit and receive mode enables
- Various interrupt enables
- Parity selection
- USART enable

UART

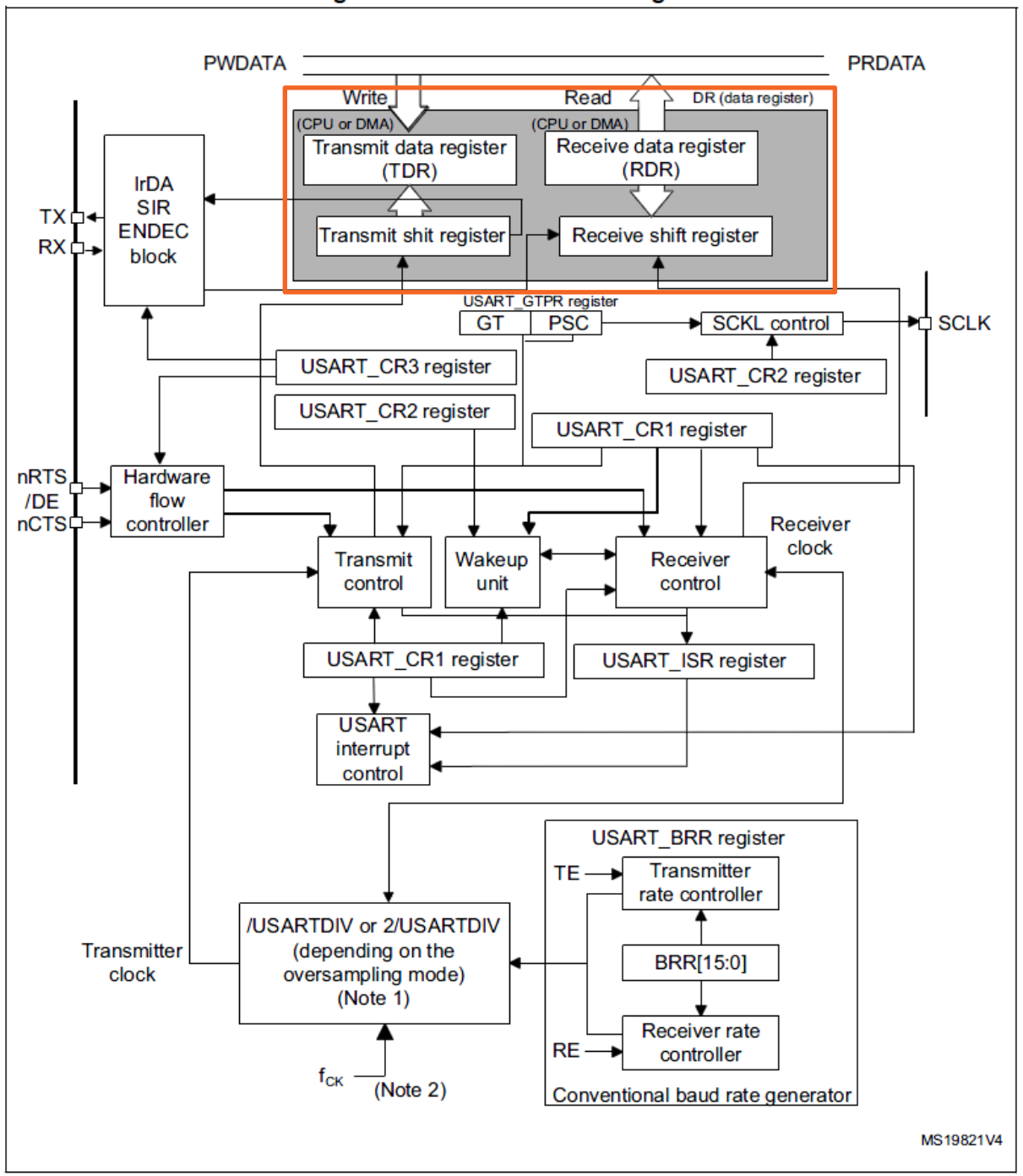


USART_ISR – Interrupt and Status Register

- Contains flags which describe the status of the communication

PE – Parity error
RXNE – Read data register not empty
TC – Transmission complete
TXE – Transmit data register empty

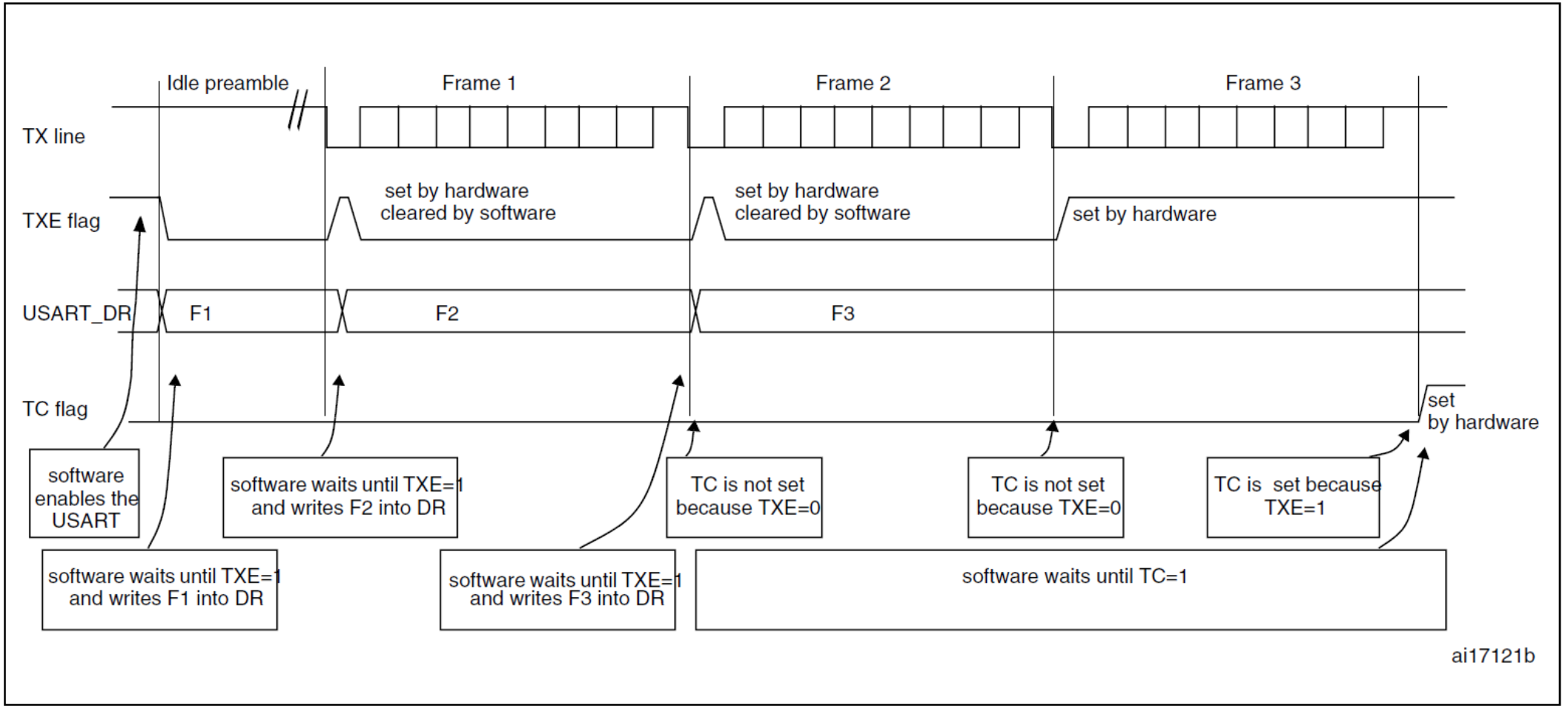
UART



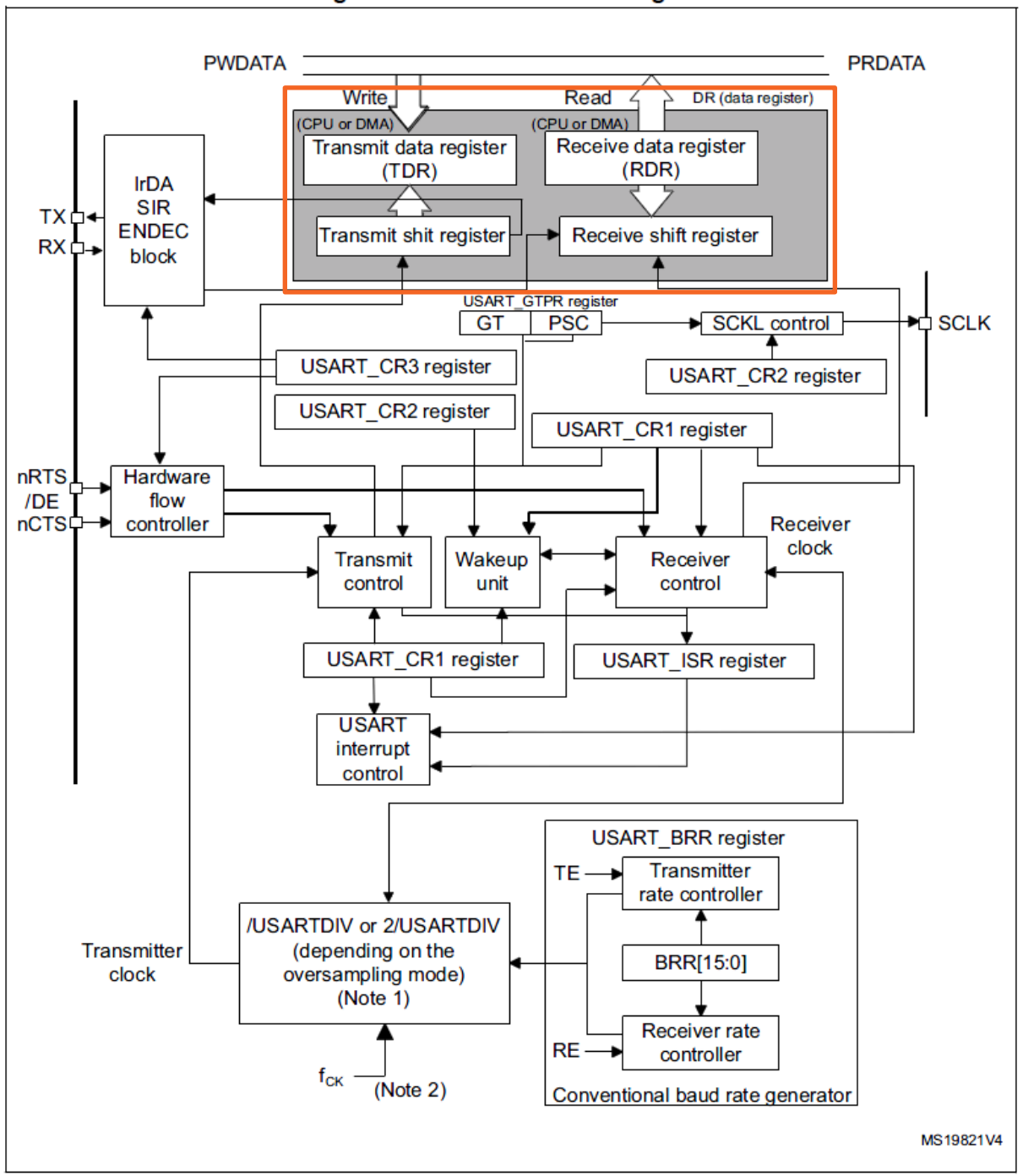
USART_TDR – Transmit Data Register

- Used to write data to be transmitted
- TXE flag raised when data has moved from the TDR register to the shift register. Write new data to the TDR register to lower the TXE flag
- TC flag set once no new data is written to TDR and TXE is high

Figure 244. TC/TXE behavior when transmitting



UART

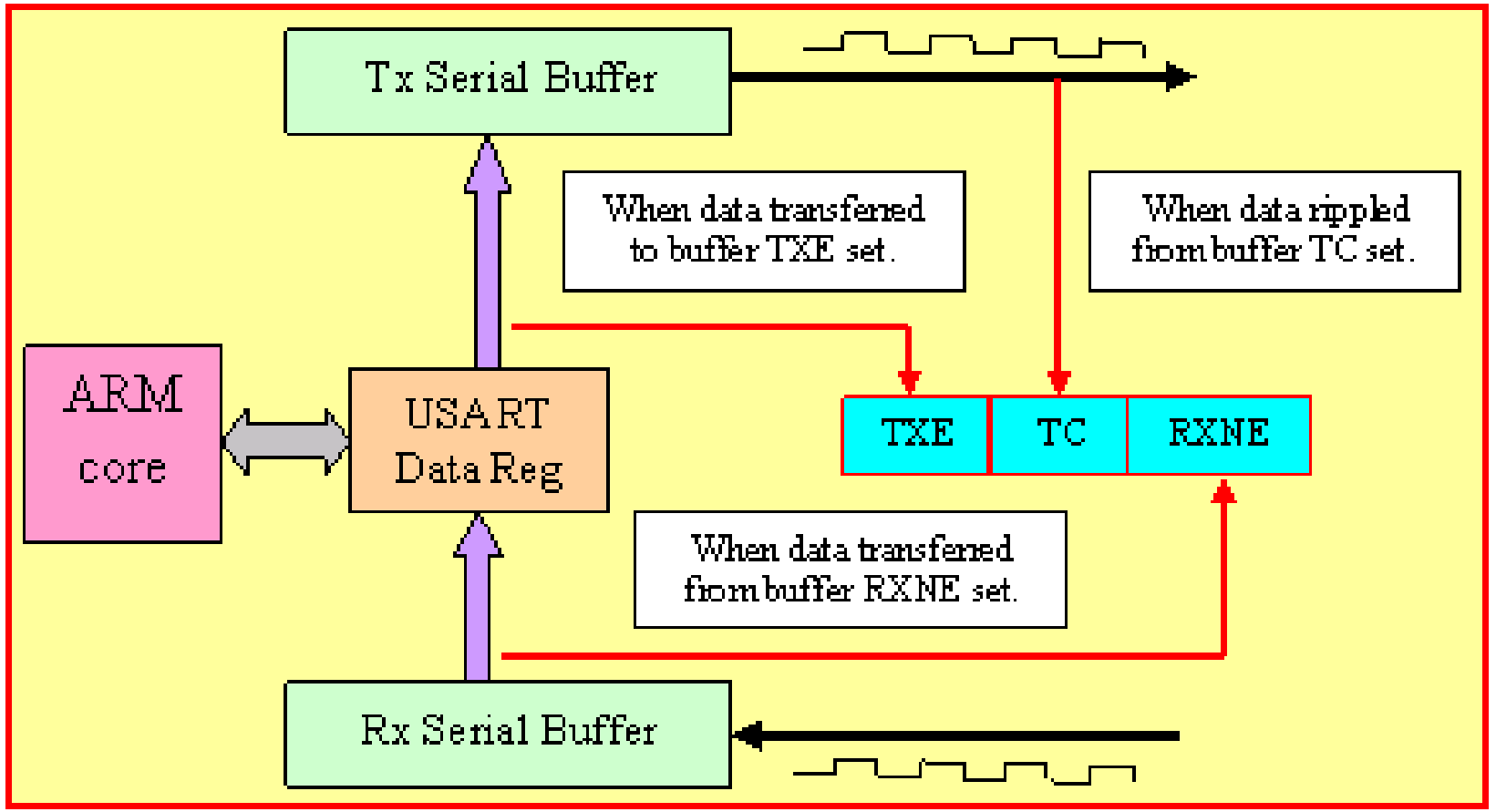


USART_RDR – Receive Data Register

- Used to read data transmitted by another device
- RXNE flag raised when data has arrived in the RDR register and is ready to be read
- RXNE flag is lowered by reading the RDR register

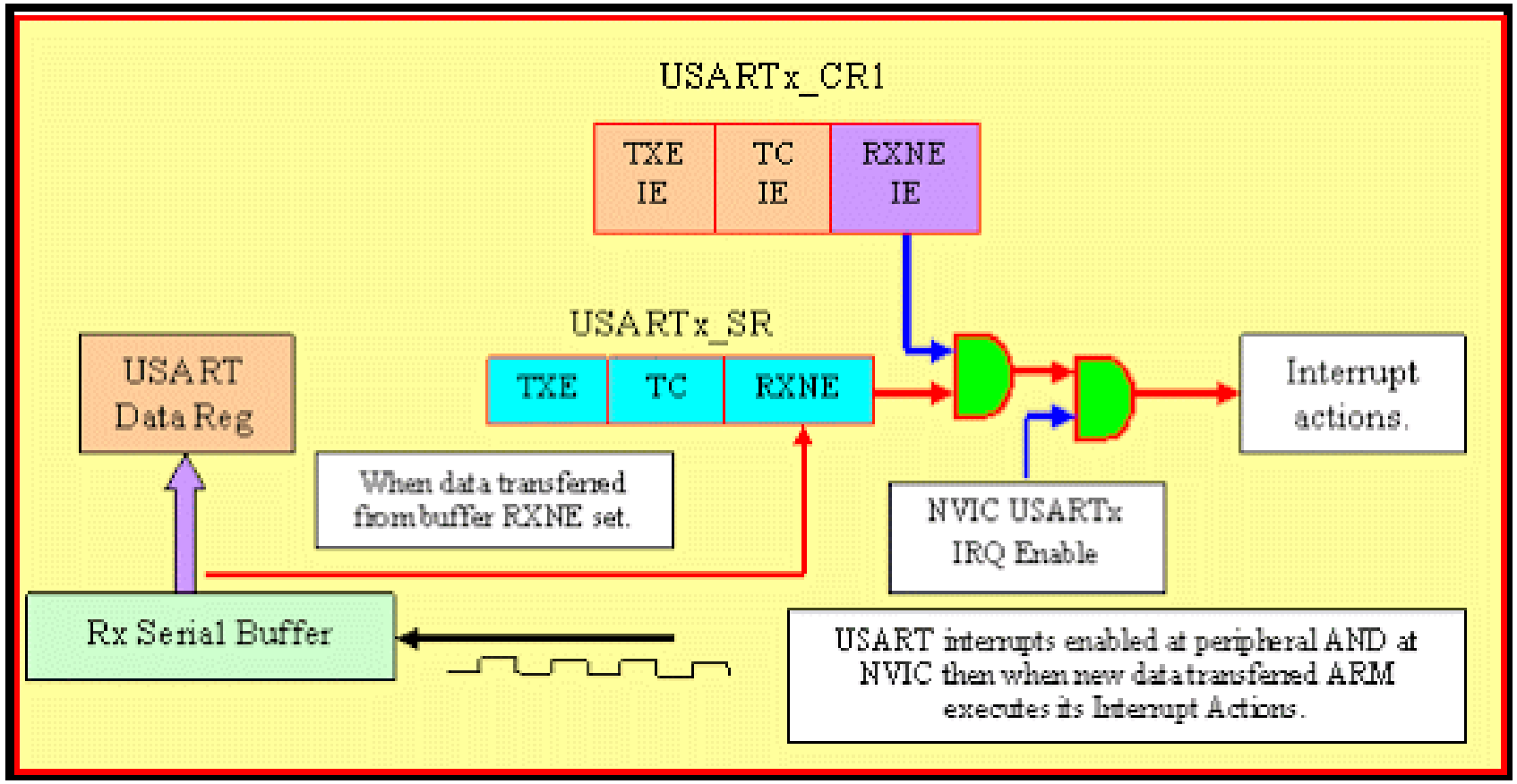
UART

Transmit/Receive Mechanism



UART

Transmit/Receive Interrupts



Step 1: Configure GPIO

UART

Table 14. Alternate functions selected through GPIOA_AFR registers for port A

Pin name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7
PA0		USART2_CTS	TIM2_CH1_ETR	TSC_G1_IO1				COMP1_OUT
PA1	EVENTOUT	USART2_RTS	TIM2_CH2	TSC_G1_IO2				
PA2	TIM15_CH1	USART2_TX	TIM2_CH3	TSC_G1_IO3				COMP2_OUT
PA3	TIM15_CH2	USART2_RX	TIM2_CH4	TSC_G1_IO4				
PA4	SPI1_NSS, I2S1_WS	USART2_CK		TSC_G2_IO1	TIM14_CH1			
PA5	SPI1_SCK, I2S1_CK	CEC	TIM2_CH1_ETR	TSC_G2_IO2				
PA6	SPI1_MISO, I2S1_MCK	TIM3_CH1	TIM1_BKIN	TSC_G2_IO3		TIM16_CH1	EVENTOUT	COMP1_OUT
PA7	SPI1_MOSI, I2S1_SD	TIM3_CH2	TIM1_CH1N	TSC_G2_IO4	TIM14_CH1	TIM17_CH1	EVENTOUT	COMP2_OUT
PA8	MCO	USART1_CK	TIM1_CH1	EVENTOUT				
PA9	TIM15_BKIN	USART1_TX	TIM1_CH2	TSC_G4_IO1				
PA10	TIM17_BKIN	USART1_RX	TIM1_CH3	TSC_G4_IO2				
PA11	EVENTOUT	USART1_CTS	TIM1_CH4	TSC_G4_IO3				COMP1_OUT
PA12	EVENTOUT	USART1_RTS	TIM1_ETR	TSC_G4_IO4				COMP2_OUT
PA13	SWDIO	IR_OUT						
PA14	SWCLK	USART2_TX						
PA15	SPI1_NSS, I2S1_WS	USART2_RX	TIM2_CH1_ETR	EVENTOUT				

Step 1: Configure GPIO


```
// Clock to GPIOA
RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
// PA9 and PA10 to AF
GPIOA->MODER |= GPIO_MODER_MODER9_1 | GPIO_MODER_MODER10_1;
// Remap to correct AF: PA9 to AF1, PA10 to AF1
GPIOA->AFR[1] |= (1 << 4) | (1 << 8);
```

Step 2: Enable USART Clock

7.4.7 APB peripheral clock enable register 2 (RCC_APB2ENR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DBG MCUEN	Res.	Res.	Res.	TIM17 EN	TIM16 EN	TIM15EN
									rw				rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	USART1 EN	Res.	SPI1EN	TIM1EN	Res.	ADCEN	Res.	USART8 EN	USART7 EN	USART6 EN	Res.	Res.	Res.	Res.	SYSCFG COMPEN
	rw		rw	rw		rw		rw	rw	rw					rw

```
// Clock to USART1
RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
```

Step 3: Configure Baud Rate

26.8.4 Baud rate register (USARTx_BRR)

This register can only be written when the USART is disabled (UE=0). It may be automatically updated by hardware in auto baud rate detection mode.

Address offset: 0x0C

Reset value: 0x0000

48 MHz

$$\text{Tx/Rx baud} = \frac{f_{CK}}{\text{USARTDIV}}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BRR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:4 **BRR[15:4]**

BRR[15:4] = USARTDIV[15:4]

Bits 3:0 **BRR[3:0]**

When OVER8 = 0, BRR[3:0] = USARTDIV[3:0].
When OVER8 = 1:
BRR[2:0] = USARTDIV[3:0] shifted 1 bit to the right.
BRR[3] must be kept cleared.

```
// Set baud rate to 115200
// USART1->BRR = 48000000/115200
USART1->BRR = 0x1A1;
```

Step 4: Choose Word Length

26.8.1 Control register 1 (USARTx_CR1)

Address offset: 0x00

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			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
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 28 **M1**: Word length

This bit, with bit 12 (M0), determines the word length. It is set or cleared by software.
M[1:0] = 00: 1 Start bit, 8 data bits, n stop bits
M[1:0] = 01: 1 Start bit, 9 data bits, n stop bits
M[1:0] = 10: 1 Start bit, 7 data bits, n stop bits
This bit can only be written when the USART is disabled (UE=0).

```
// Set word length to 8 bits
USART1->CR1 &= ~USART_CR1_M;
```

Step 5: Choose Parity

26.8.1 Control register 1 (USARTx_CR1)

Address offset: 0x00

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			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
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 10 **PCE**: Parity control enable
0: Parity control disabled
1: Parity control enabled
Must be configured while USART disabled (UE ==0).

Bit 9 **PS**: Parity selection
0: Even parity
1: Odd parity
Must be configured while USART disabled (UE ==0).

```
// Set to even parity
USART1->CR1 |= USART_CR1_PCE;
USART1->CR1 &= ~USART_CR1_PS;
```

Step 6: Configure Interrupts- OPTIONAL

26.8.1 Control register 1 (USARTx_CR1)

Address offset: 0x00

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			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
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 5 **RXNEIE**: RXNE interrupt enable

This bit is set and cleared by software.

0: Interrupt is inhibited

1: A USART interrupt is generated whenever ORE=1 or RXNE=1 in the USARTx_ISR register

```
// Set to RXNE interrupt generation
USART1->CR1 |= USART_CR1_RXNEIE;
```

Step 6: Enable Transmit/Receive Mode

26.8.1 Control register 1 (USARTx_CR1)

Address offset: 0x00

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			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
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 3 **TE**: Transmitter enable
Set and cleared by software
0: Transmitter is disabled
1: Transmitter is enabled

Bit 2 **RE**: Receiver enable
Set and cleared by software
0: Receiver is disabled
1: Receiver is enabled

```
// Set to transmit and receive
USART1->CR1 |= USART_CR1_TE | USART_CR1_RE;
```

Step 6: Enable USART

26.8.1 Control register 1 (USARTx_CR1)

Address offset: 0x00

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			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
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **UE**: USART enable

When this bit is cleared, the USART prescalers and outputs are stopped immediately, and current operations are discarded. The configuration of the USART is kept, but all the status flags, in the USARTx_ISR are set to their default values. This bit is set and cleared by software.

0: USART prescaler and outputs disabled, low-power mode

1: USART enabled

```
// Enable USART
USART1->CR1 |= USART_CR1_UE;
```

UART INTERRUPTS

- Table 101
- Event flags contained by ISR register – Check Reference Manual for reset mechanism
- If multiple interrupt sources are configured the ISR should be polled to determine the interrupt cause

Interrupt event	Event flag	Enable Control bit
Transmit data register empty	TXE	TXEIE
CTS interrupt	CTSIF	CTSIE
Transmission Complete	TC	TCIE
Receive data register not empty (data ready to be read)	RXNE	RXNEIE
Overrun error detected	ORE	
Idle line detected	IDLE	IDLEIE
Parity error	PE	PEIE
LIN break	LBDF	LBDIE
Noise Flag, Overrun error and Framing Error in multibuffer communication.	NF or ORE or FE	EIE
Character match	CMF	CMIE
Receiver timeout error	RTOF	RTOIE
End of Block	EOBF	EOBIE
Wakeup from Stop mode	WUF(1)	WUFIE

UART

Setup USART for 8 bit word length,
Receiver and transmitter mode on PA9&10,
Odd parity,
Receive interrupt

Read two incoming bytes, increment their values and transmit them back.

```c
#include <stdio.h>
#include "stm32f0xx.h"

void init_USART1();

uint8_t CommData[] = {0, 0};
uint8_t Counter = 0;
__Bool DataReceived = 0;

// Main ------------------------------------------------|
void main()
{
  init_USART1();
  while(1)
  {
    if (DataReceived == 1)
    {
      USART1->TDR = CommData[0] + 1;
      while ((USART1->ISR & USART_ISR_TXE) == 0);
      USART1->TDR = CommData[1] + 1;
      while ((USART1->ISR & USART_ISR_TC) == 0);
      Counter = 0;
      DataReceived = 0;
    }
  }
}
```

UART

```
void init_USART1()
{
  // Clock to GPIOA
  RCC->AHBENR |= RCC_AHBENR_GPIOAEN;
  // PA9 and PA10 to AF
  GPIOA->MODER |= GPIO_MODER_MODER9_1 | GPIO_MODER_MODER10_1;
  // Remap to correct AF: PA9 to AF1, PA10 to AF1
  GPIOA->AFR[1] |= (1 << 4) | (1 << 8);
  // Clock to USART1
  RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
  // Set baud rate to 9600
  USART1->BRR = 48000000 / 9600;
  // USART receive interrupt enable
  USART1->CR1 |= USART_CR1_RXNEIE;
  // Transmit and Receive Enable
  USART1->CR1 |= USART_CR1_TE | USART_CR1_RE;
  // USART Enable
  USART1->CR1 |= USART_CR1_UE;
  NVIC_EnableIRQ(USART1_IRQn);
}

void USART1_IRQHandler(void)
{
  CommData[Counter] = USART1->RDR;
  if (Counter > 0)
  {
    DataReceived = 1;
  }
  Counter++;
}
```

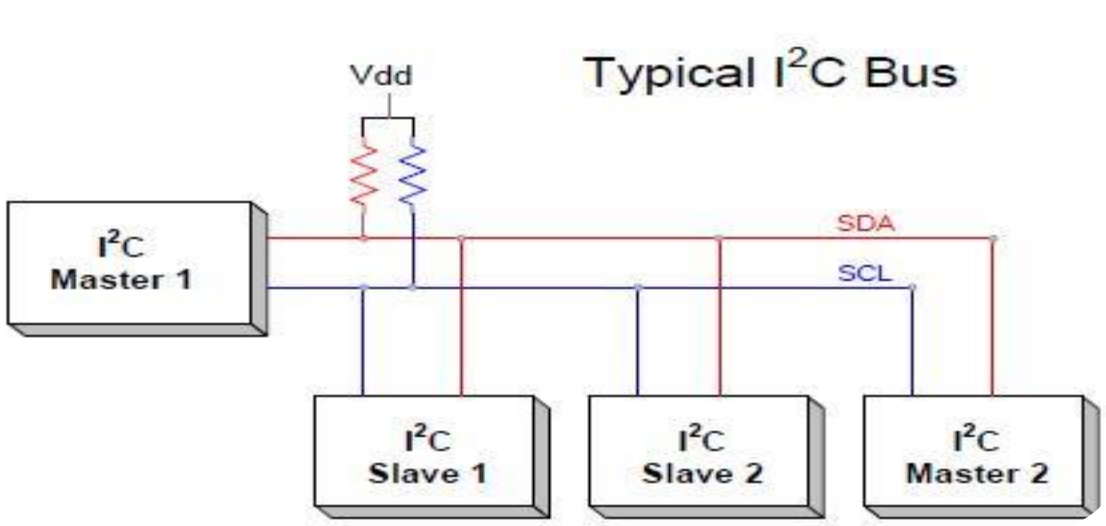
COMMUNICATIONS

I2C

I2C

- Inter-integrated circuit – I^2C, I2C, IIC
- Master slave system – one master, many slaves
- Two wire bus (SDA and SCL)
- Clock and data lines

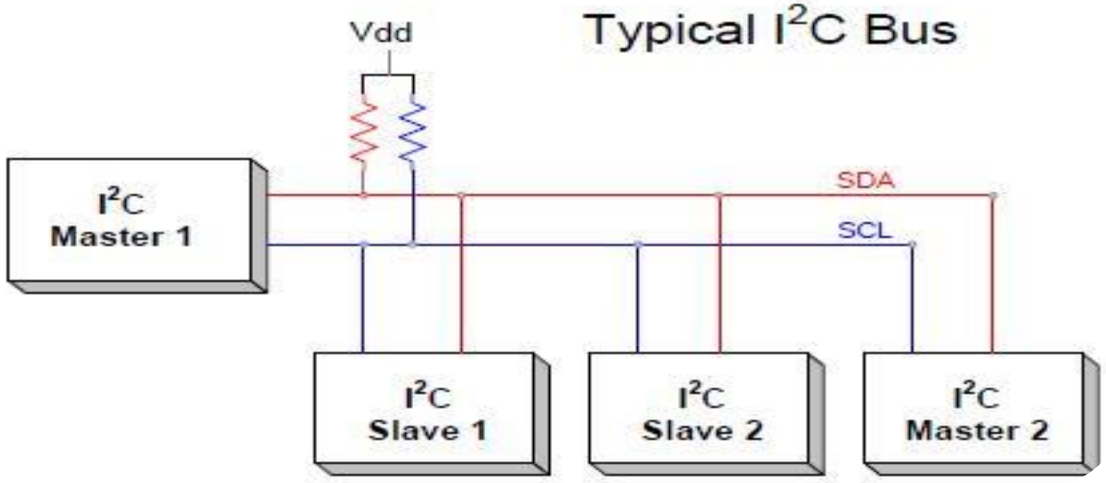
Synchronous protocol



I2C

- Lines pulled low by any device
- Devices have addresses – unique identifiers
- Master always initiates comms

Pin connections open drain



I2C

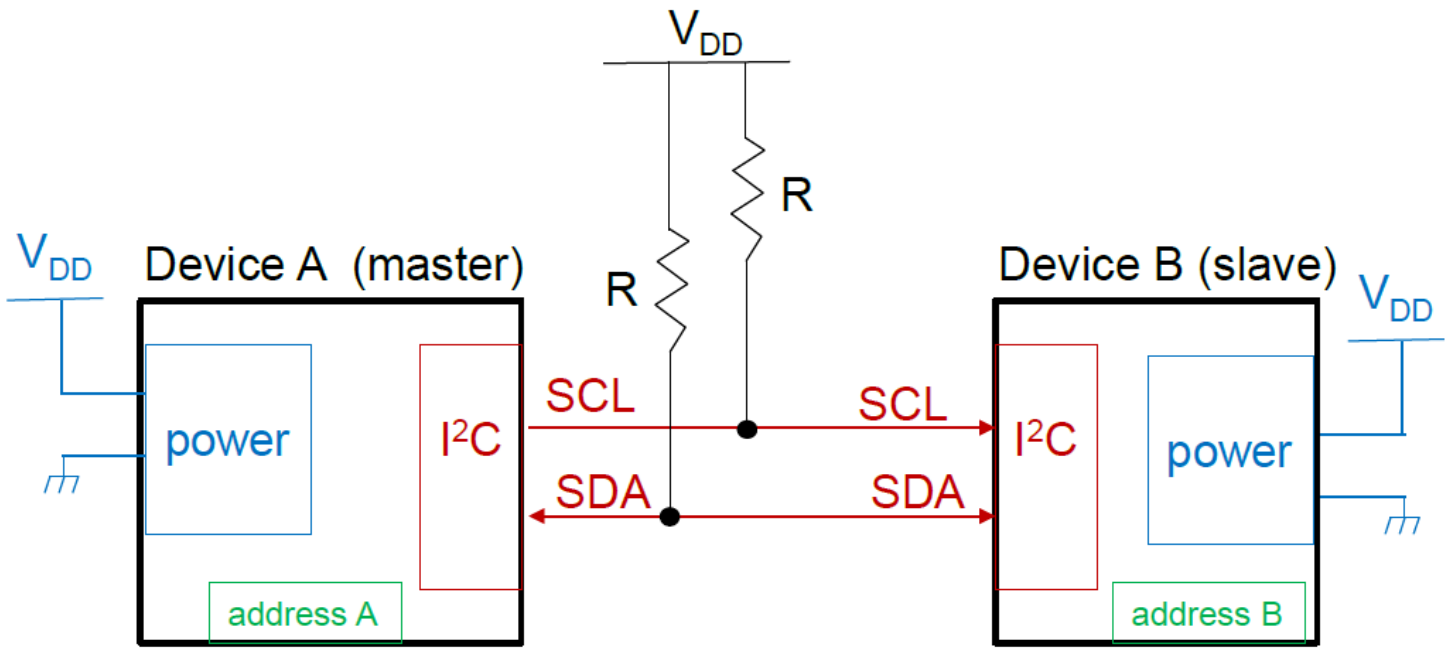
- Small number of wires
- Fast speeds (3.4 Mbps)
- Limited speed options. Common speeds:

  Low Speed - 10 kbps

  Standard Mode - 100 kbps

  Fast Mode - 400 kbps
- Shared bus – anyone holds the line low it is all broken
- Can support multi-master configurations

I2C



Hardware Connection

- SCL – Serial Clock Line
- SDA – Serial Data Line
- Each device has a unique address – Usually 7 bits
- R <= 4.7 kΩ
- Designed for comms on a single PCB but up to 3 m possible

I2C

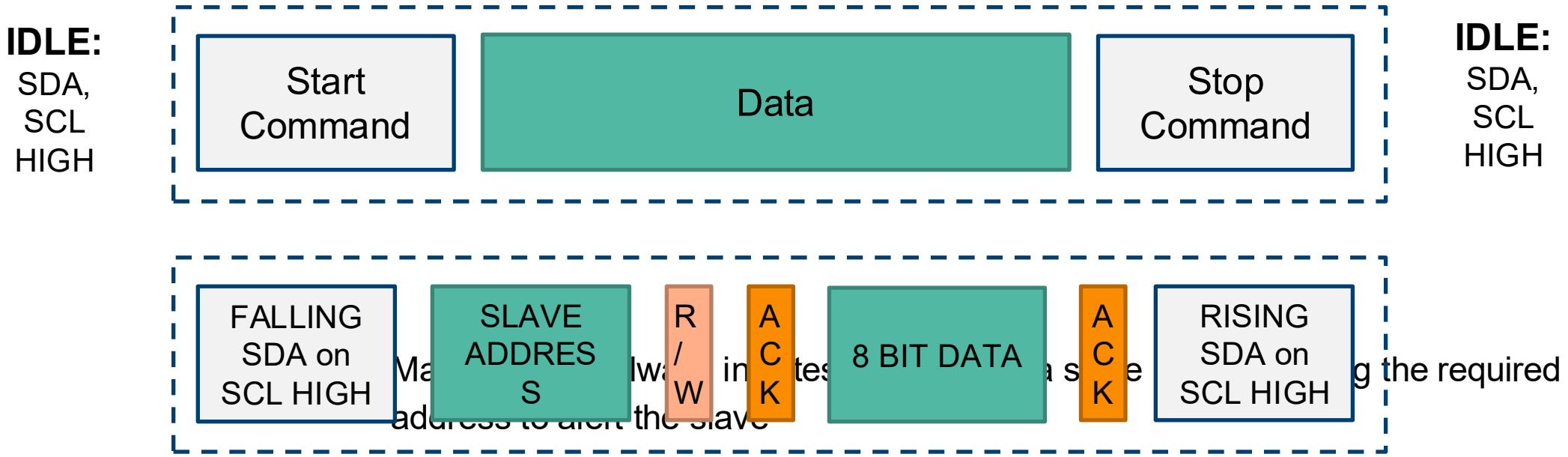
Multi slave example:

- Slaves = same sensors with similar addresses
- Carefully place order to ensure that each slave has a unique address.
- Address is 7 bits long because the 8th bit is to indicate whether it is a read or write command.

SOT-23 Package Marking Codes

SOT-23 (V)	Address	Code	SOT-23 (V)	Address	Code
TC74A0-3.3VCT	1001 000	V0	TC74A0-5.0VCT	1001 000	U0
TC74A1-3.3VCT	1001 001	V1	TC74A1-5.0VCT	1001 001	U1
TC74A2-3.3VCT	1001 010	V2	TC74A2-5.0VCT	1001 010	U2
TC74A3-3.3VCT	1001 011	V3	TC74A3-5.0VCT	1001 011	U3
TC74A4-3.3VCT	1001 100	V4	TC74A4-5.0VCT	1001 100	U4
TC74A5-3.3VCT	1001 101*	V5	TC74A5-5.0VCT	1001 101*	U5
TC74A6-3.3VCT	1001 110	V6	TC74A6-5.0VCT	1001 110	U6
TC74A7-3.3VCT	1001 111	V7	TC74A7-5.0VCT	1001 111	U7

I2C


IDLE:
SDA,
SCL
HIGH
Start Command
Data
Stop Command
IDLE:
SDA,
SCL
HIGH
FALLING SDA on SCL HIGH
SLAVE ADDRESS
R / W
ACK
8 BIT DATA
ACK
RISING SDA on SCL HIGH

I2C

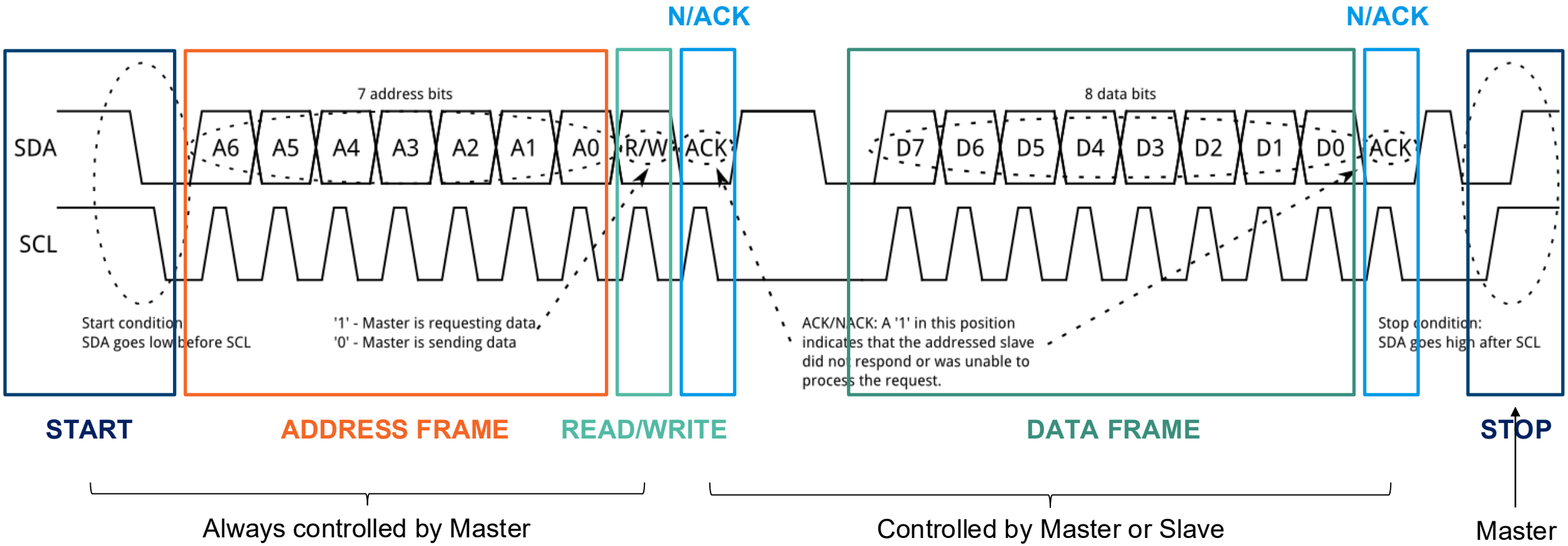
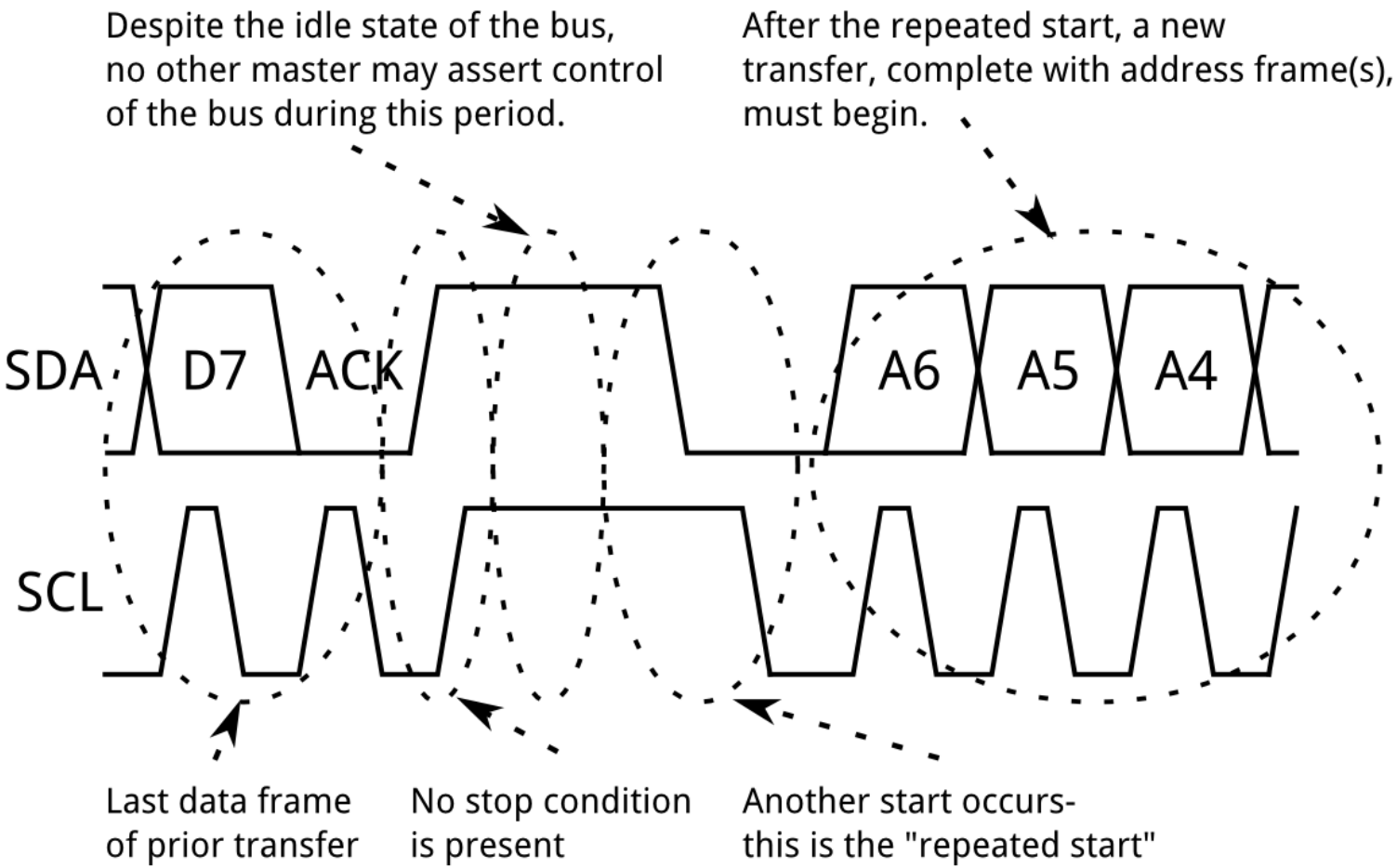


Image taken from https://learn.sparkfun.com/tutorials/i2c/all

I2C

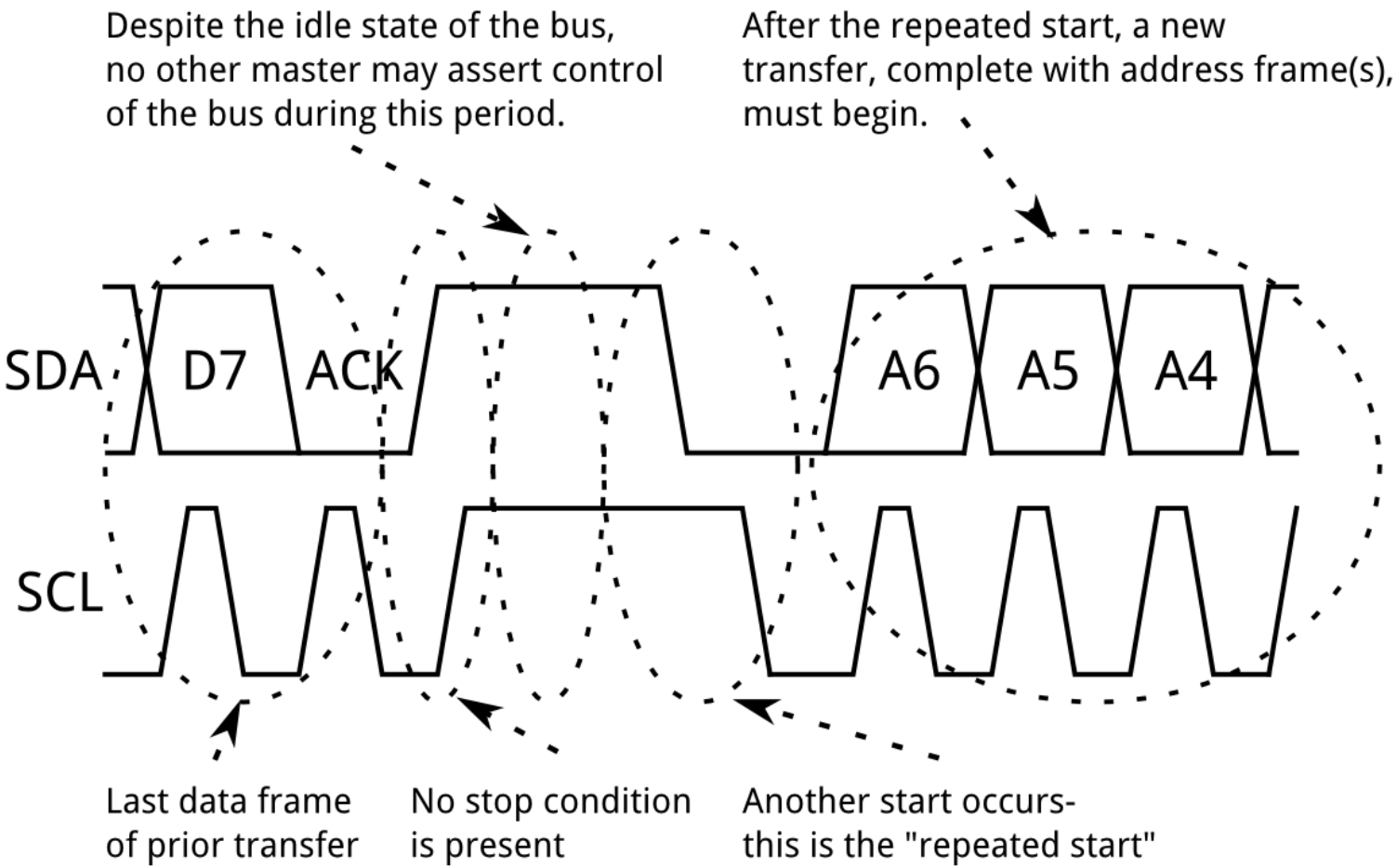


Repeated Start

- Sometimes more than one message must be exchanged between and master and its slaves
- Repeated start occurs to facilitate this – No Stop condition created

Image taken from https://learn.sparkfun.com/tutorials/i2c/all

I2C

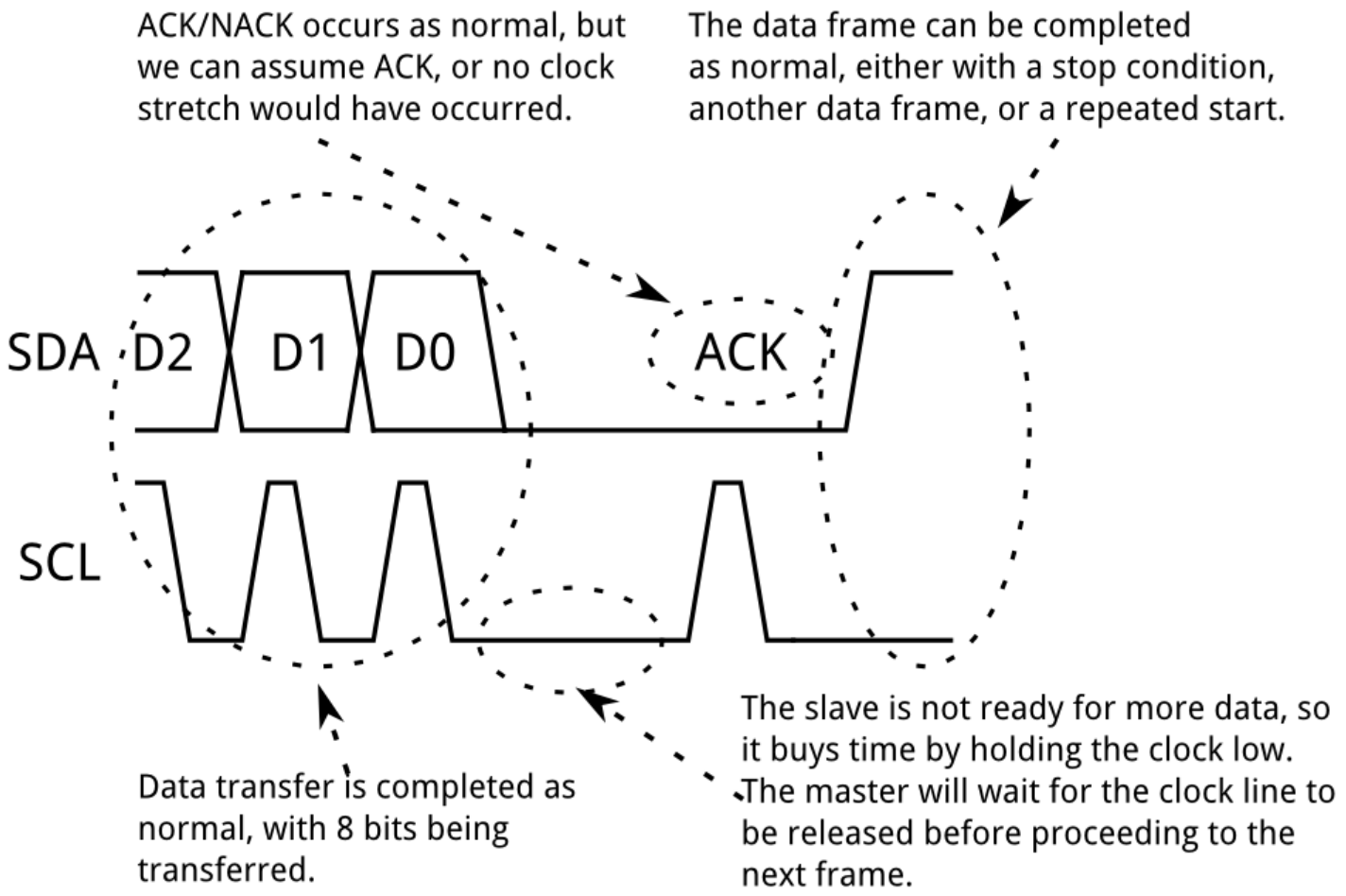


Repeated Start

- SDA goes HIGH while SCL is LOW
- SCL then goes HIGH
- SDA then goes LOW while SCL is HIGH creating another START condition

Image taken from https://learn.sparkfun.com/tutorials/i2c/all

I2C



Clock Stretching

- Allows slave to hold SCL LOW until it is ready for more data from the master
- Occurs when a slave is not ready to provide the master with data.

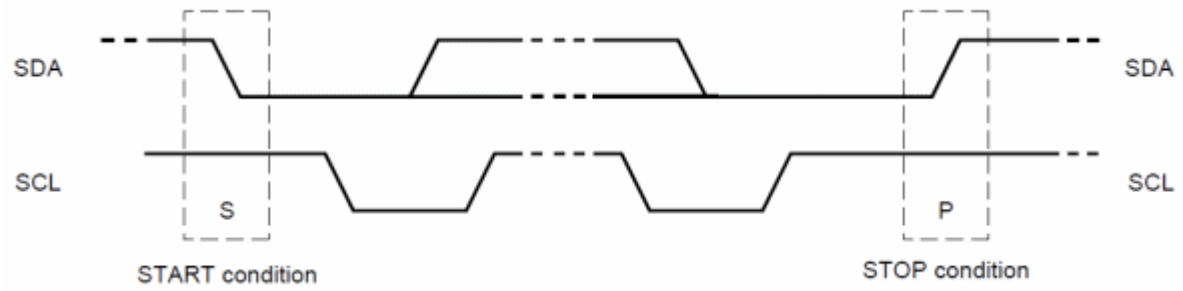
E.g. Waiting for an internal ADC conversion to complete

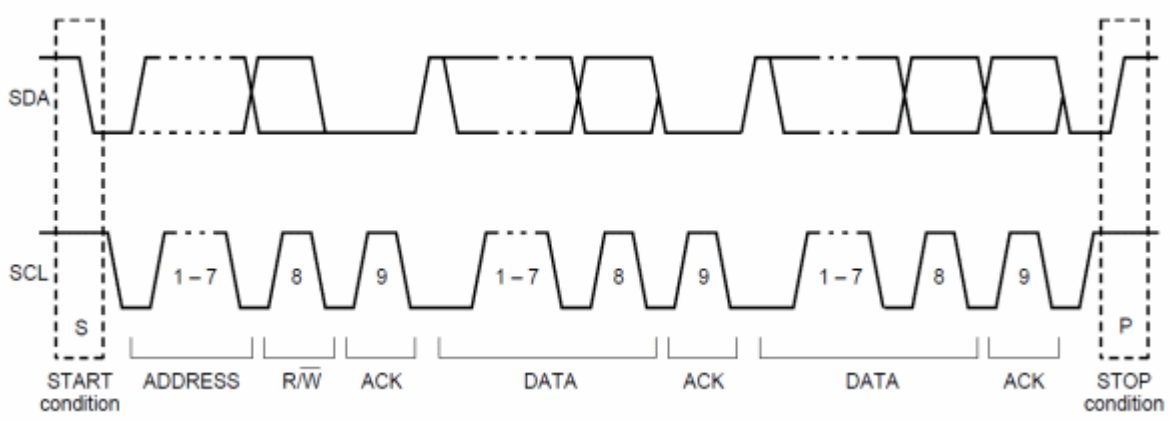
Image taken from https://learn.sparkfun.com/tutorials/i2c/all

I2C

Data sent in a package:

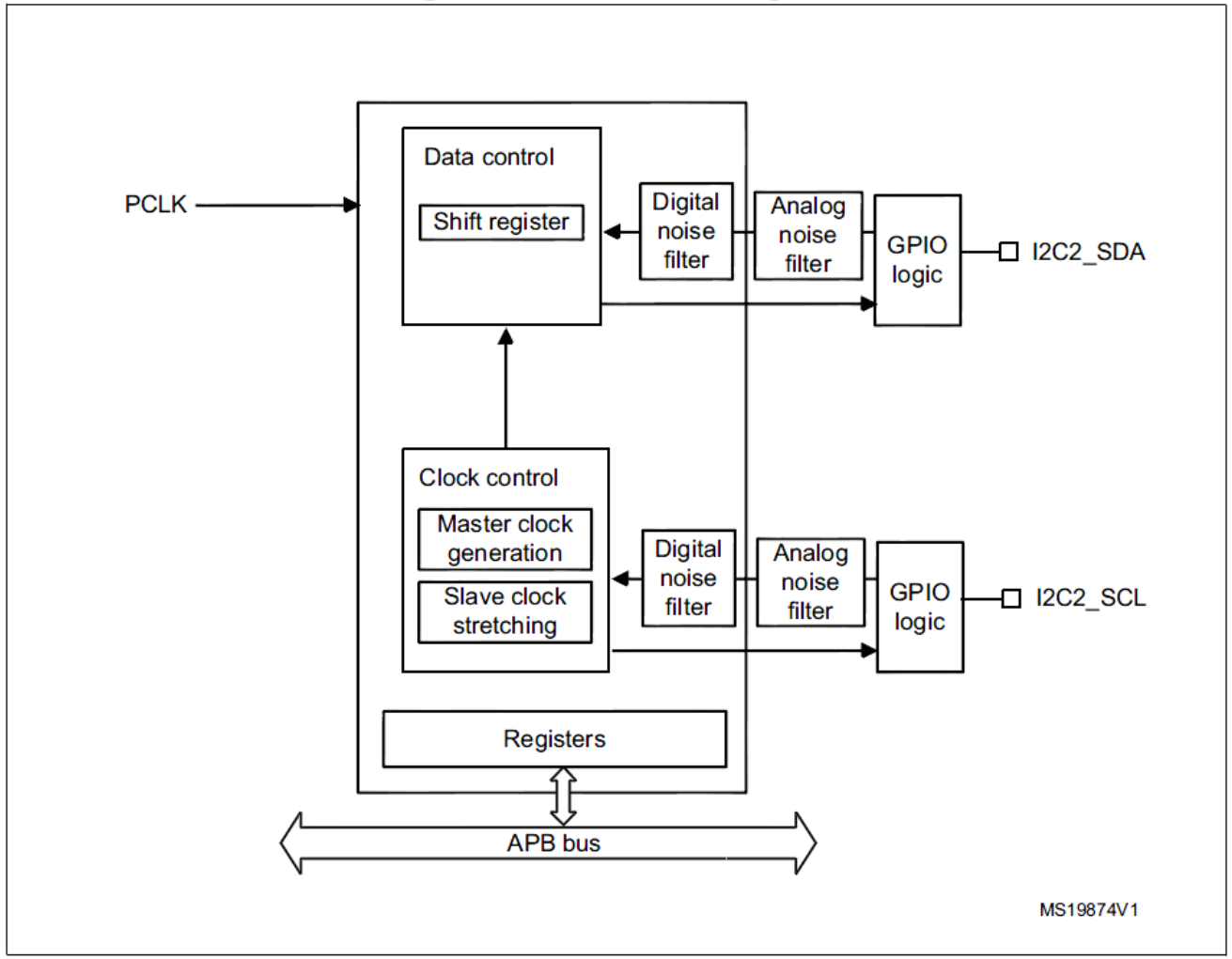
1. Start bit
2. Slave address
3. Read/Write Bit
4. N/ACK
5. Data (8 bits)
6. N/ACK
7. Stop





I2C

Figure 210. I2C2 block diagram



- I2C2 on STM32F0 has reduced functionality but still supports 7 or 10 bit addressing, Standard mode and Fast mode comms
- I2C1 includes further clock signal selections, Wakeup from Stop mode, and additional extended functions

I2C

Advantages

2 wires - many devices

Software addressing = extra devices readily added

Protocol requires acknowledge from slave so corrective action possible if acknowledgement not received

Disadvantages

Short distances

Relatively slow speeds/bandwidth

Clock speed relatively slow

Flexible addressing = significant overhead

I2C

Table 13. Pin definitions (continued)

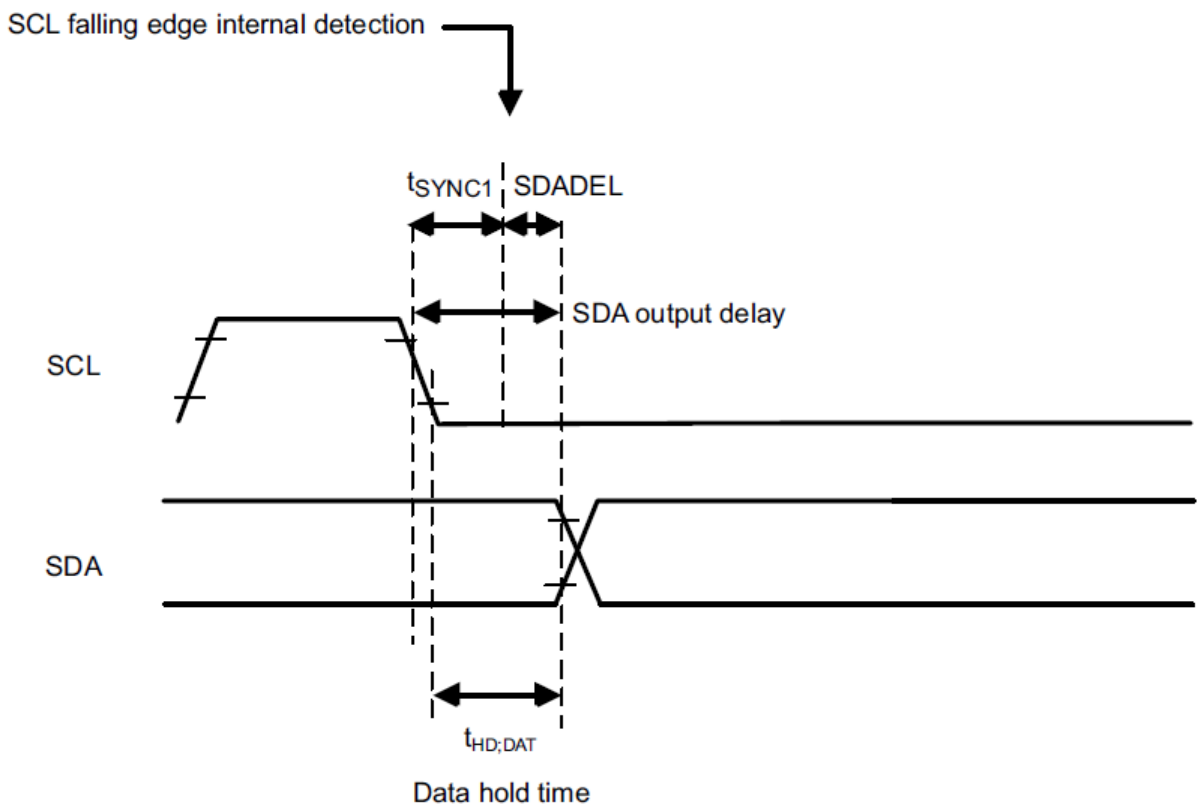
Pin number					Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
LQFP64	UFBGA64	LQFP48/UFQFPN48	LQFP32	UFQFPN32					Alternate functions	Additional functions
45	B8	33	22	22	PA12	I/O	FT		USART1_RTS, TIM1_ETR, COMP2_OUT, TSC_G4_IO4, EVENTOUT	-
46	A8	34	23	23	PA13 (SWDIO)	I/O	FT	(4)	IR_OUT, SWDIO	-
47	D6	35	-	-	PF6	I/O	FTf		I2C2_SCL	-
48	E6	36	-	-	PF7	I/O	FTf		I2C2_SDA	-

- 2 I2C peripherals
- I2C2 on PB10,11 and PF6,7
- I2C1 on PB8,9

I2C

Data Hold Time

Refer to Section 25.4.5 of the Reference Manual

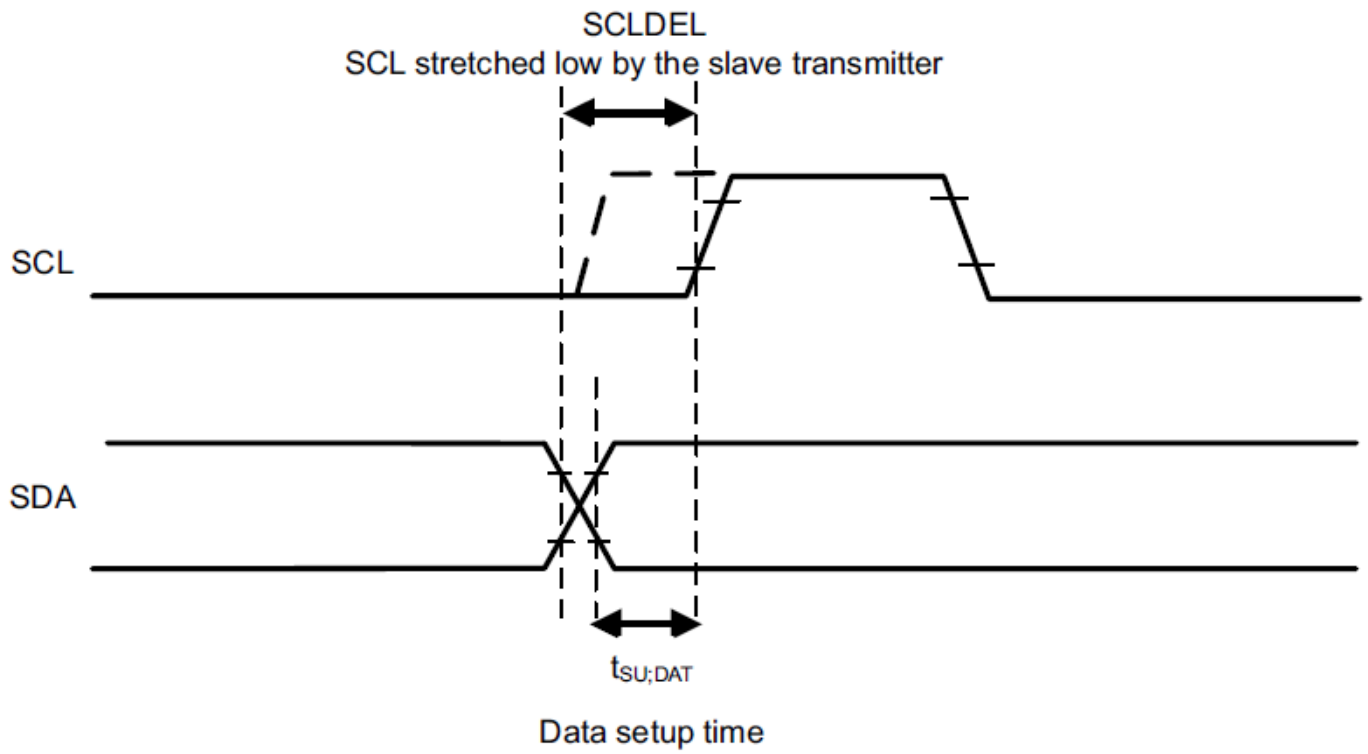


Total SDA output delay = t_{SYNC1} + {[SDADEL x (PRESC+1) + 1] x t_{I2CCLK}}

I2C

Data Setup Time

Refer to Section 25.4.5 of the Reference Manual



$$t_{SCLDEL} = (SCLDEL+1) \times t_{PRESC}$$

I2C

Table 85. Examples of timings settings for f_{I2CCLK} = 48 MHz

Parameter	Standard-mode (Sm)		Fast-mode (Fm)	Fast-mode Plus (Fm+)
	10 kHz	100 kHz	400 kHz	1000 kHz
PRESC	0xB	0xB	5	5
SCLL	0xC7	0x13	0x9	0x3
t_{SCLL}	200 x 250 ns = 50 µs	20 x 250 ns = 5.0 µs	10 x 125 ns = 1250 ns	4 x 125 ns = 500 ns
SCLH	0xC3	0xF	0x3	0x1
t_{SCLH}	196 x 250 ns = 49 µs	16 x 250 ns = 4.0 µs	4 x 125 ns = 500 ns	2 x 125 ns = 250 ns
t_{SCL} (1)	~100 µs(2)	~10 µs(2)	~2500 ns(3)	~875 ns(4)
SDADEL	0x2	0x2	0x3	0x0
t_{SDADEL}	2 x 250 ns = 500 ns	2 x 250 ns = 500 ns	3 x 125 ns = 375 ns	0 ns
SCLDEL	0x4	0x4	0x3	0x1
t_{SCLDEL}	5 x 250 ns = 1250 ns	5 x 250 ns = 1250 ns	4 x 125 ns = 500 ns	2 x 125 ns = 250 ns

1. The SCL period t_{SCL} is greater than t_{SCLL} + t_{SCLH} due to the SCL internal detection delay. Values provided for t_{SCL} are only examples.
2. t_{SYNC1} + t_{SYNC2} minimum value is 4x t_{I2CCLK} = 83.3 ns. Example with t_{SYNC1} + t_{SYNC2} = 1000 ns
3. t_{SYNC1} + t_{SYNC2} minimum value is 4x t_{I2CCLK} = 83.3 ns. Example with t_{SYNC1} + t_{SYNC2} = 750 ns
4. t_{SYNC1} + t_{SYNC2} minimum value is 4x t_{I2CCLK} = 83.3 ns. Example with t_{SYNC1} + t_{SYNC2} = 250 ns

I2C

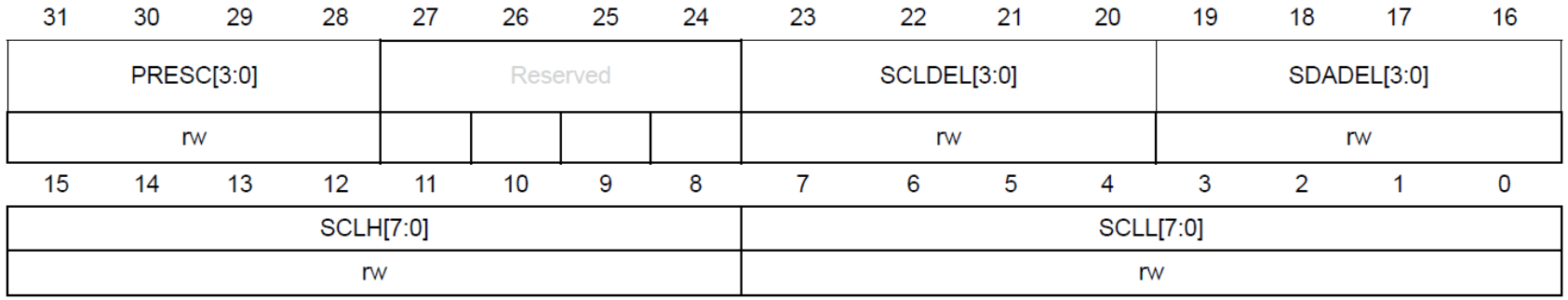
Must be configured whilst peripheral is disabled

25.7.5 Timing register (I2C_TIMINGR)

Address offset: 0x10

Reset value: 0x0000 0000

Access: No wait states



- Use values from Table 85 of the Reference Manual or calculate your own if slave device supports alternative speeds

I2C

25.7.1 Control register 1 (I2C_CR1)

Address offset: 0x00

Reset value: 0x0000 0000

Access: No wait states, except if a write access occurs while a write access to this register is ongoing. In this case, wait states are inserted in the second write access until the previous one is completed. The latency of the second write access can be up to 2 x PCLK1 + 6 x I2CCLK.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PECEN	ALERT EN	SMBD EN	SMBH EN	GCEN	WUP EN	NOSTR ETCH	SBC
								rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXDMA EN	TXDMA EN	Res.	ANF OFF	DNF				ERRIE	TCIE	STOP IE	NACK IE	ADDR IE	RXIE	TXIE	PE
rw	rw		rw	rw				rw	rw	rw	rw	rw	rw	rw	rw

I2C_CR1

- Peripheral Enable
- Interrupt enables for various conditions

I2C

25.7.2 Control register 2 (I2C_CR2)

Address offset: 0x04

Reset value: 0x0000 0000

Access: No wait states, except if a write access occurs while a write access to this register is ongoing. In this case, wait states are inserted in the second write access until the previous one is completed. The latency of the second write access can be up to 2 x PCLK1 + 6 x I2CCLK.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	PEC BYTE	AUTO END	RE LOAD	NBYTES[7:0]							
					rs	rw	rw	rw							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NACK	STOP	START	HEAD 10R	ADD10	RD_WRN	SADD[9:0]									
rs	rs	rs	rw	rw	rw	rw									

I2C_CR2

- Slave Address
- Start, Stop
- Number of bytes to be transferred
- Transfer direction
- Set RELOAD bit if transferring > 255 bytes

I2C

- It is essential to use the Reference Manual to find out the details of comms protocols
- Typically the STM32F0 will be used as a Master device – Section 25.4.9 is extremely useful
- It contains operational descriptions, flowcharts, timing formulae and specifications needed for the use of the I2C peripheral
- Figure 213 details the initialisation steps for I2C
- Section 25.6 details the use of interrupts with I2C

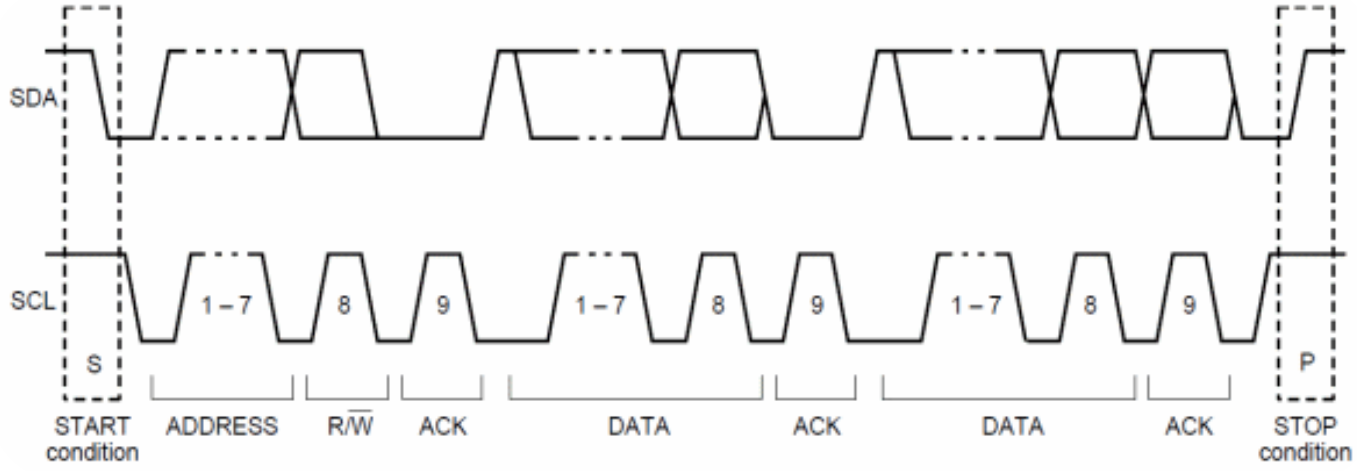
Read from device to STM using I2C & write Rx to LED's using correct data format for communications

I2C

Master Receiver Mode, Read 1 Byte

Read temperature from TC74 sensor on PF6 and PF7

```
#include "stm32f0xx.h"
#define TC74ADDR 0b1001000

void main(void);
void init_iic(void);
void init_leds(void);

void main(void)
{
 init_leds();
 init_iic();

 while(1)
 {
  I2C2->CR2 |= (TC74ADDR << 1); //set slave address in SADD
  I2C2->CR2 |= (1 << 16); // set NBYTES to 1

  // send start and address with read byte.
  // indicate we are going to be doing a read
  I2C2->CR2 |= I2C_CR2_RD_WRN;
  I2C2->CR2 |= I2C_CR2_START;
  // wait for ACK (start bit reset)
  // wait for RX flag
  while ((I2C2->ISR & I2C_ISR_RXNE) == 0);
  // clock in a byte and write to LEDs
  GPIOB->ODR = I2C2->RXDR;
  // STOP condition
  I2C2->CR2 |= I2C_CR2_STOP;
 }
}
```



I2C

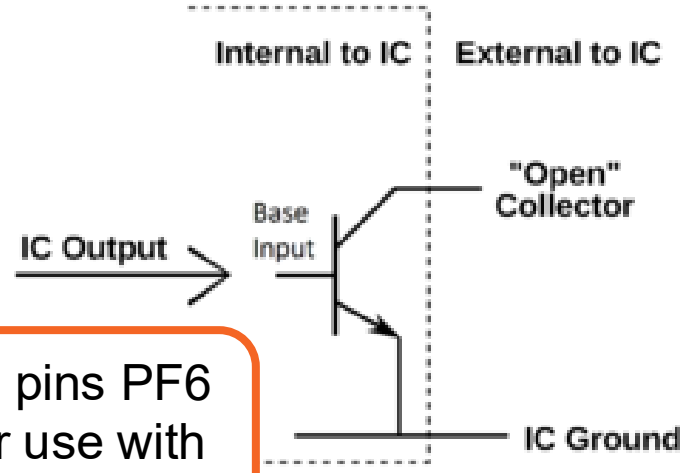
Master Receiver Mode, Read 1 Byte

Read temperature from TC74 sensor on PF6 and PF7

```
void init_iic(void)
{
 // enable clock to port F
 RCC->AHBENR |= RCC_AHBENR_GPIOFEN;
 // there is a risk that the slave is sitting in the middle
 // of a transfer when we reset the master.
 // the following block of code simply toggles the clock
 // line 20 times to ensure that the slave gets a chance to
 // clock out its data.
 // set clock line to open drain, output
 GPIOF->OTYPER |= GPIO_OTYPER_OT_6;   // open drain
 GPIOF->MODER |= GPIO_MODER_MODER6_0; // GP output

 for (uint32_t loop_counter = 0; loop_counter < 20; loop_counter++)
 {
   for (volatile uint32_t delay = 0; delay < 10; delay++);
   GPIOF->BSRR |= GPIO_BSRR_BR_6; // set clock low
   for (volatile uint32_t delay = 0; delay < 10; delay++);
   GPIOF->BSRR |= GPIO_BSRR_BS_6; // set clock high
 }

 // set SCLK (PF6) to alternate function, open drain
 GPIOF->MODER &= ~GPIO_MODER_MODER6; // reset the MODER bits
 GPIOF->MODER |= GPIO_MODER_MODER6_1;
 GPIOF->OTYPER |= GPIO_OTYPER_OT_6;
 // set SDA (PF7) to alternate function, open drain
 GPIOF->MODER |= GPIO_MODER_MODER7_1;
 GPIOF->OTYPER |= GPIO_OTYPER_OT_7;
 // PF6 and PF7 only have 1 alternate function, so it's not
 // necessary to map them
```

PF6 & 7 = SCL & SDA

Toggle line to clear any remaining data before resetting the master



Set up pins PF6 & 7 for use with I2C

I2C

Master Receiver Mode, Read 1 Byte

Read temperature from TC74 sensor
on PF6 and PF7

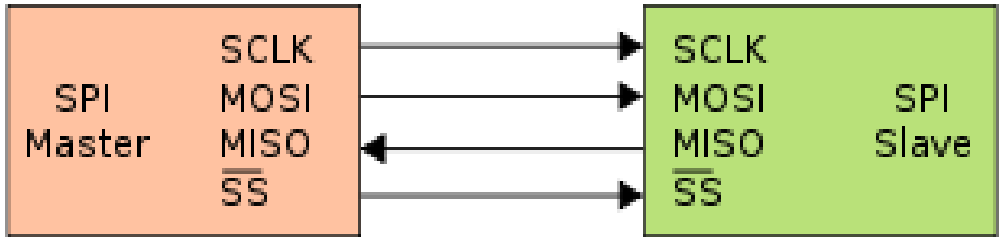
```
void init_iic(void)
{

  ...

  ...

  // enable clock to I2C2
  RCC->APB1ENR |= RCC_APB1ENR_I2C2EN;
  // disable the peripheral
  I2C2->CR1 &= ~I2C_CR1_PE;
  // configure timing in PRESC, SCLDEL, SDADEL in TIMINGR
  I2C2->TIMINGR |= (0xC7 << 0); // SCLL
  I2C2->TIMINGR |= (0xC3 << 8); // SCLH
  I2C2->TIMINGR |= (0x02 << 16); // SDADEL
  I2C2->TIMINGR |= (0x04 << 20); // SCLDEL
  I2C2->TIMINGR |= (0x0B << 28);// PRESC
  // enable I2C
  I2C2->CR1 |= I2C_CR1_PE;
}
```

Set up I2C Timing

COMMUNICATIONS

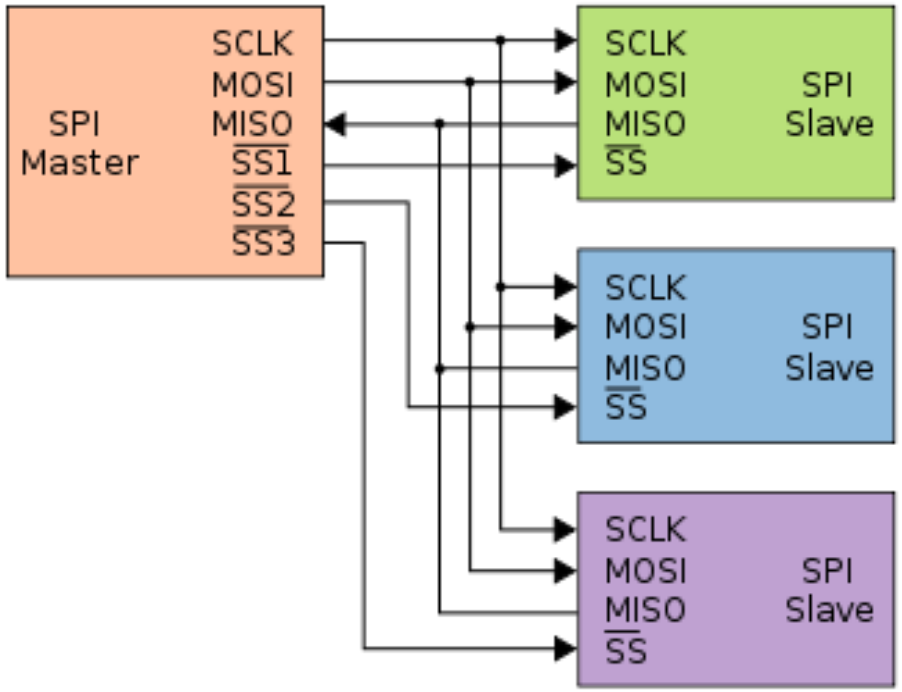
SPI

SPI



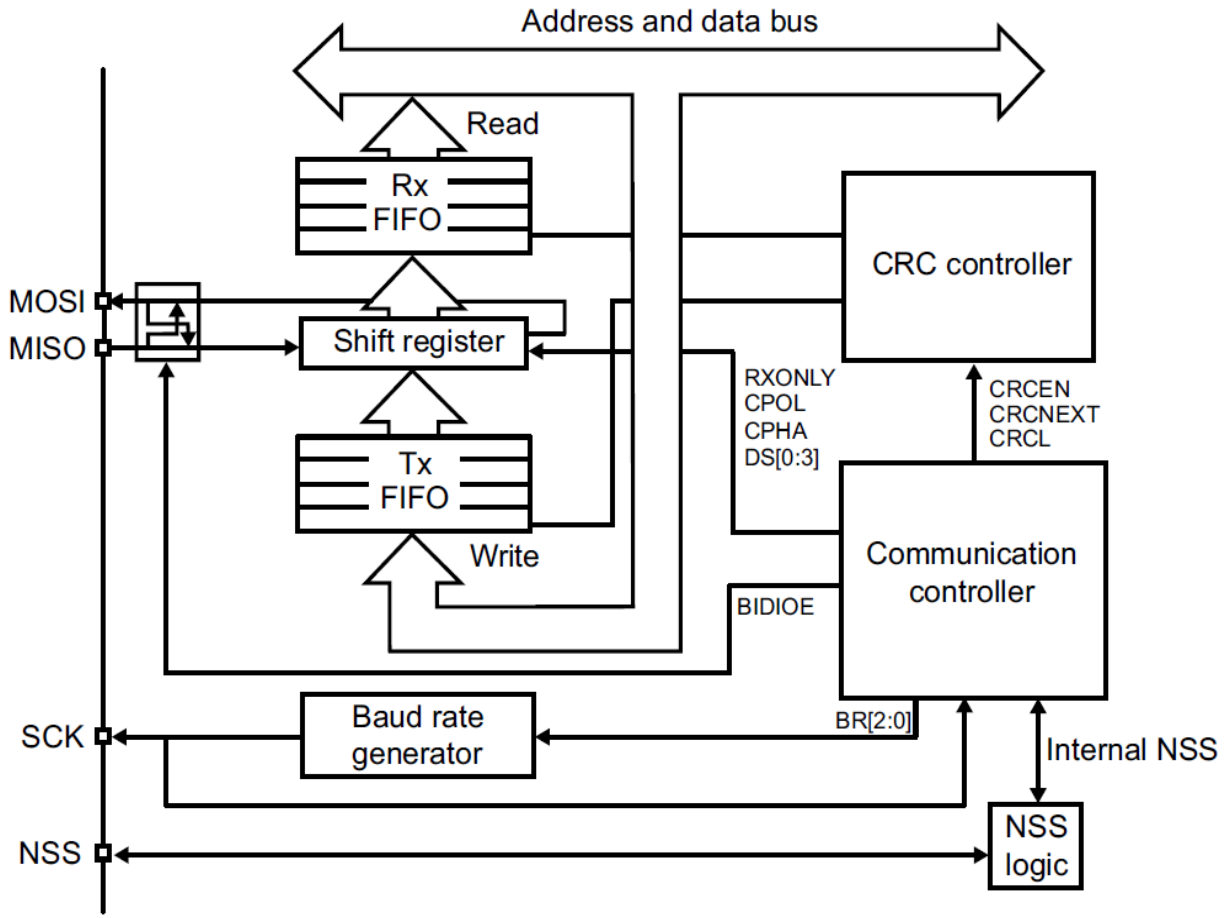
- Serial Peripheral Interface
- Master and slave protocol
- The master device is controls the SCLK signal
- MOSI line – Master Out/Slave In
- MISO line – Master In/Slave Out
- Slave Select line
- Each slave gets its own SS line – rest are shared
- Master pulls the slave line low to initiate comms

SPI



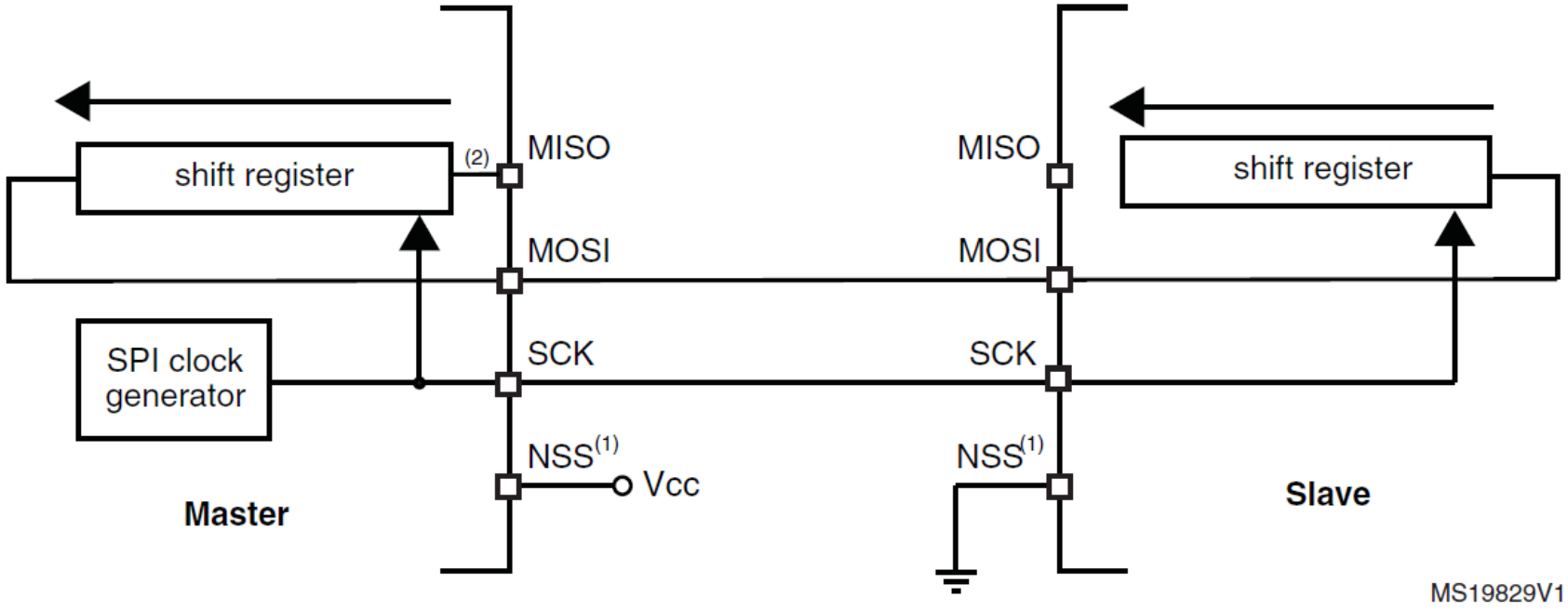
- Multiple devices
- Transmit and receive at the same time (Device Dependant)
  - Full Duplex default on STM32
- Synchronous so you can change the clock speed
- Requires more pins
- Short distances

SPI


Address and data bus
Read
Rx
FIFO
MOSI
MISO
Shift register
Tx
FIFO
Write
CRC controller
RXONLY
CPOL
CPHA
DS[0:3]
CRCEN
CRCNEXT
CRCL
Communication
controller
BIDIOE
SCK
Baud rate
generator
BR[2:0]
Internal NSS
NSS
NSS
logic

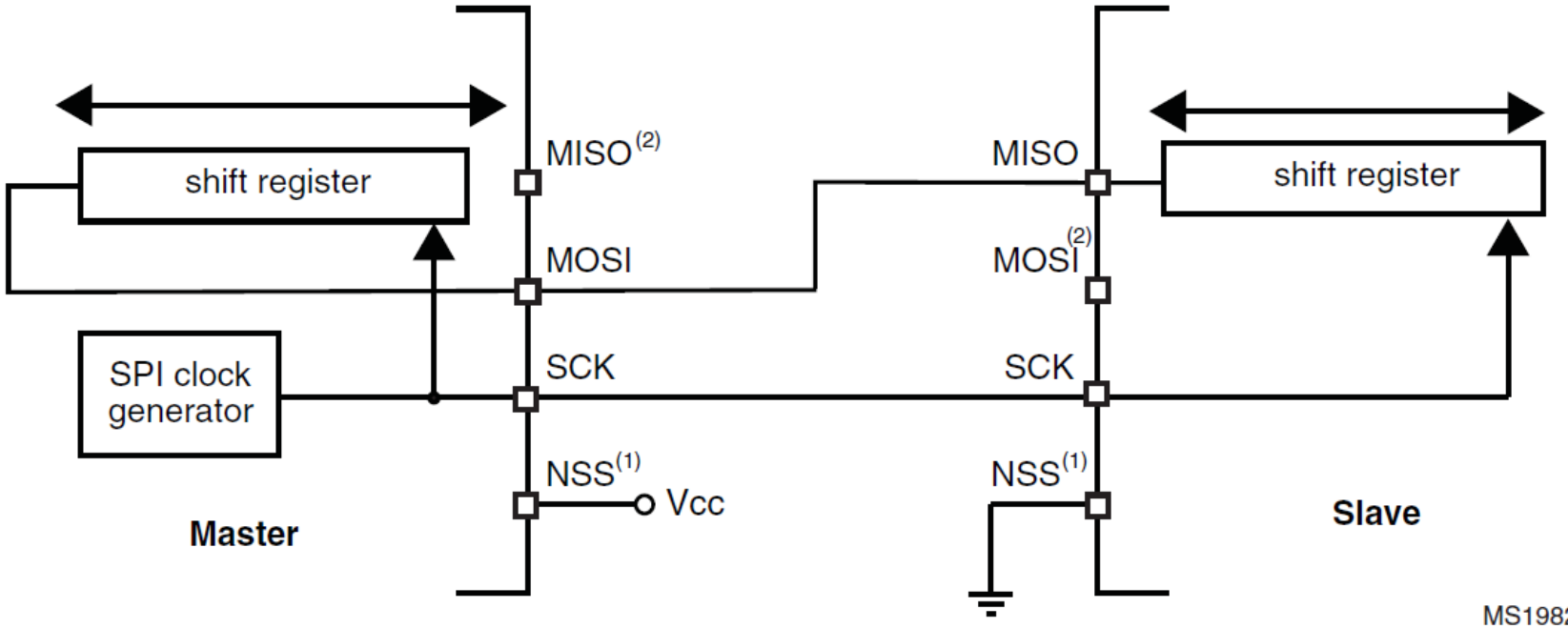
SPI

Simplex single master/single slave application (master in transmit-only/ slave in receive-only mode)



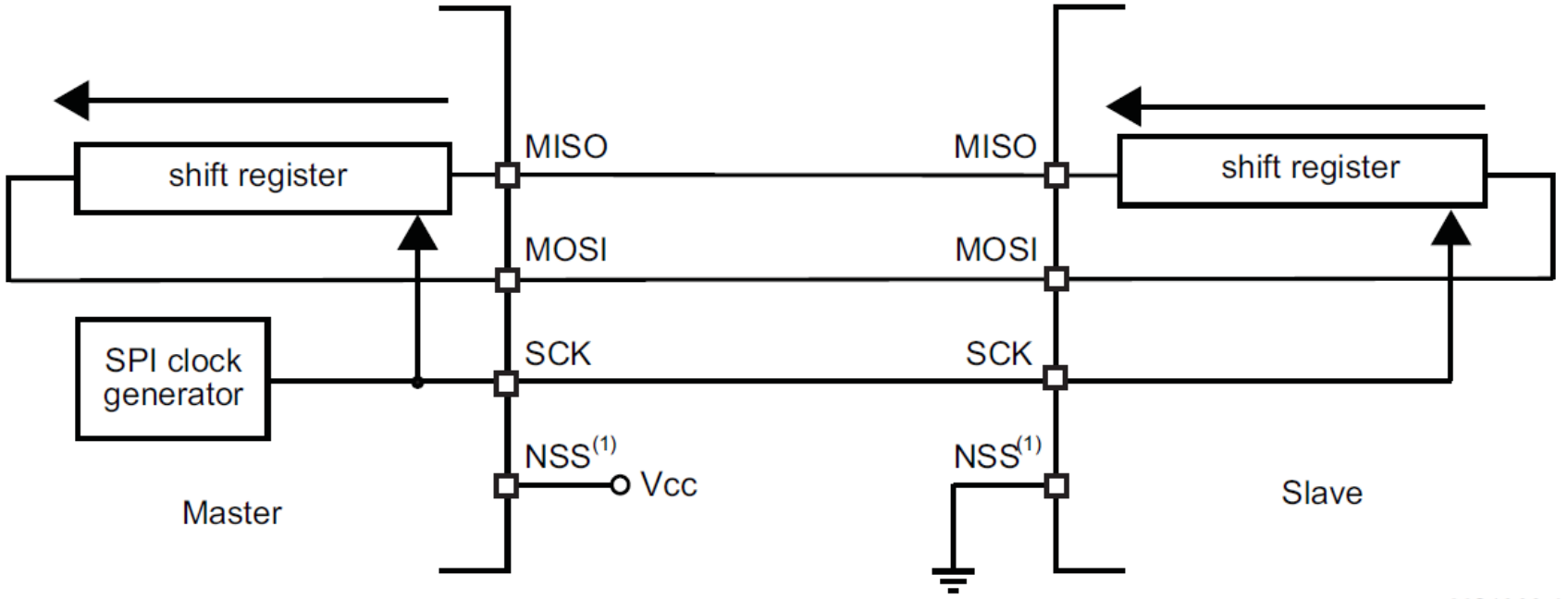
SPI

Half-duplex single master/ single slave application

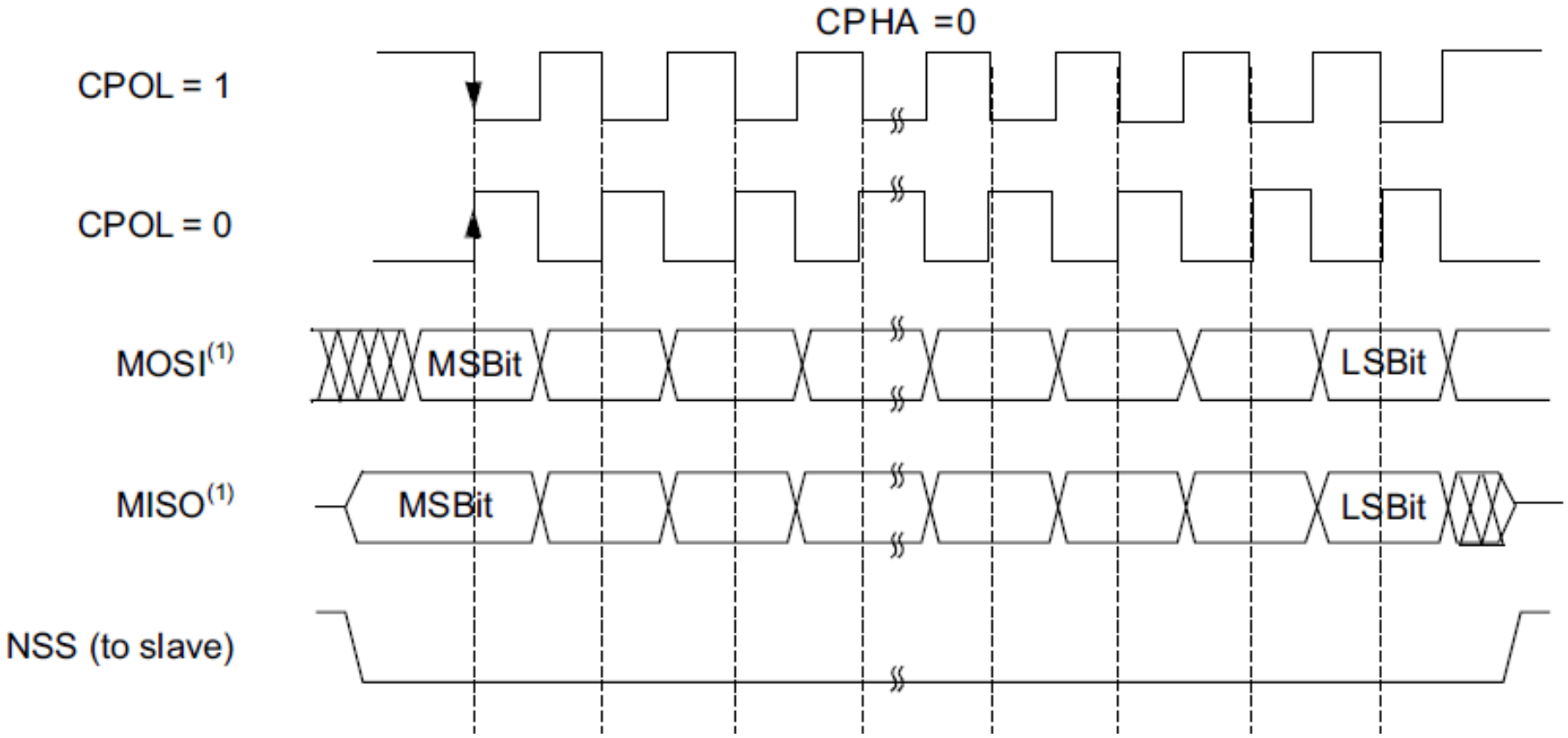


SPI

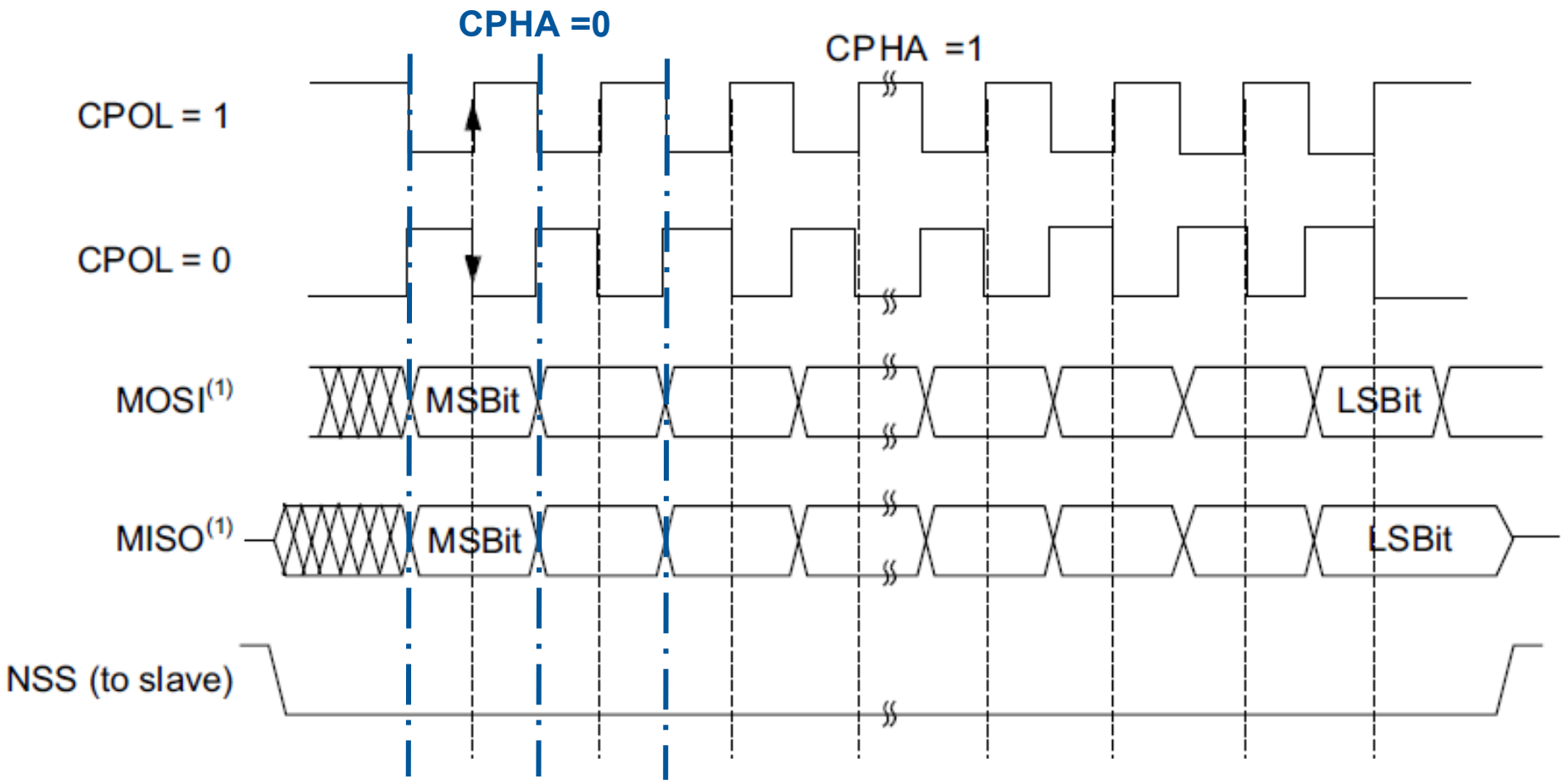
Full-duplex single master/ single slave application



SPI


CPHA =0
CPOL = 1
CPOL = 0
MOSI(1)
MSBit
LSBit
MISO(1)
MSBit
LSBit
NSS (to slave)

SPI


CPHA =0
CPHA =1
CPOL = 1
CPOL = 0
MOSI(1)
MSBit
LSBit
MISO(1)
MSBit
LSBit
NSS (to slave)

SPI

Operation in master slave full-duplex mode:

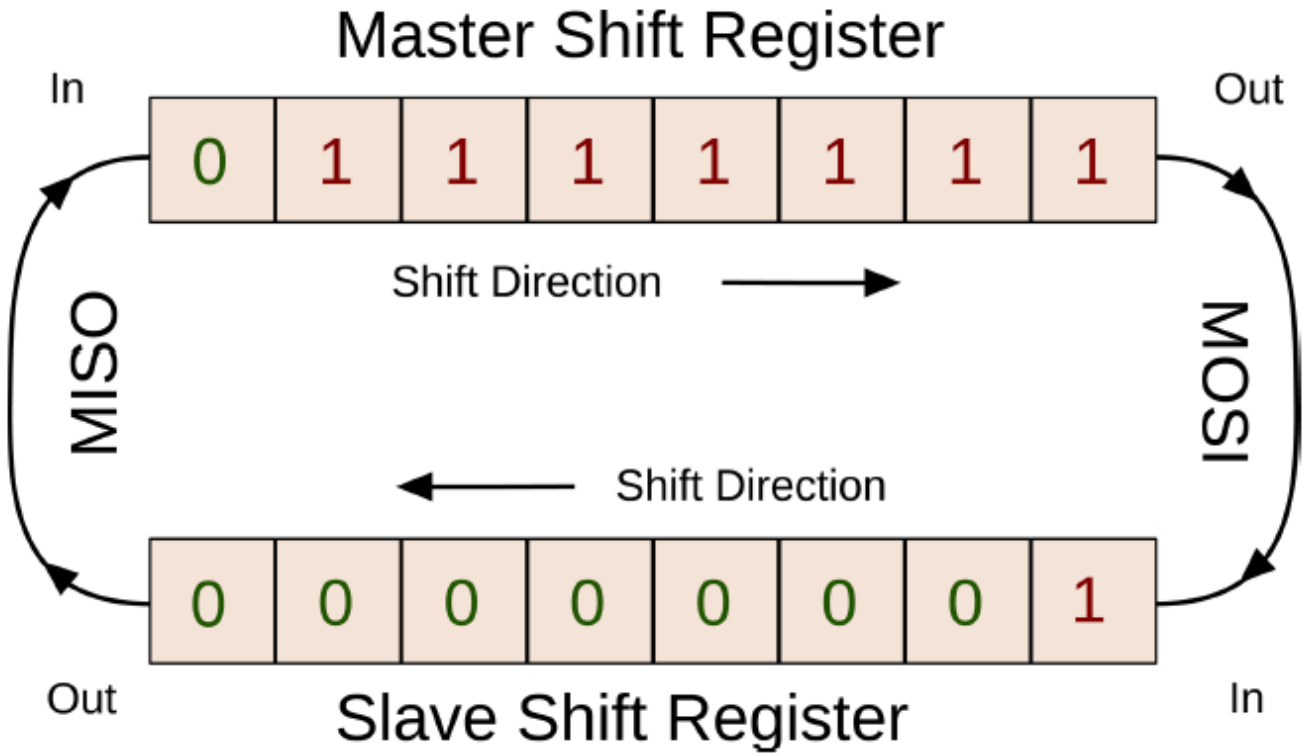
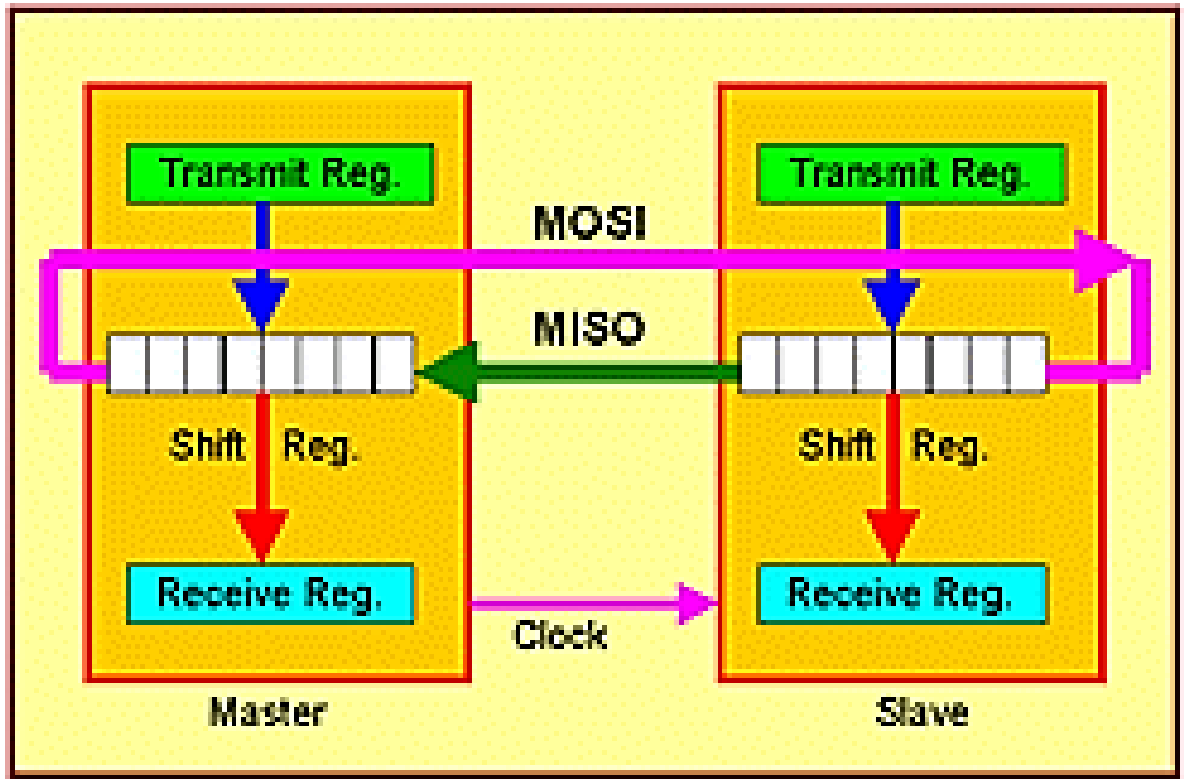


Image taken from http://www.handsonembedded.com/stm32f103-spl-tutorial-5/

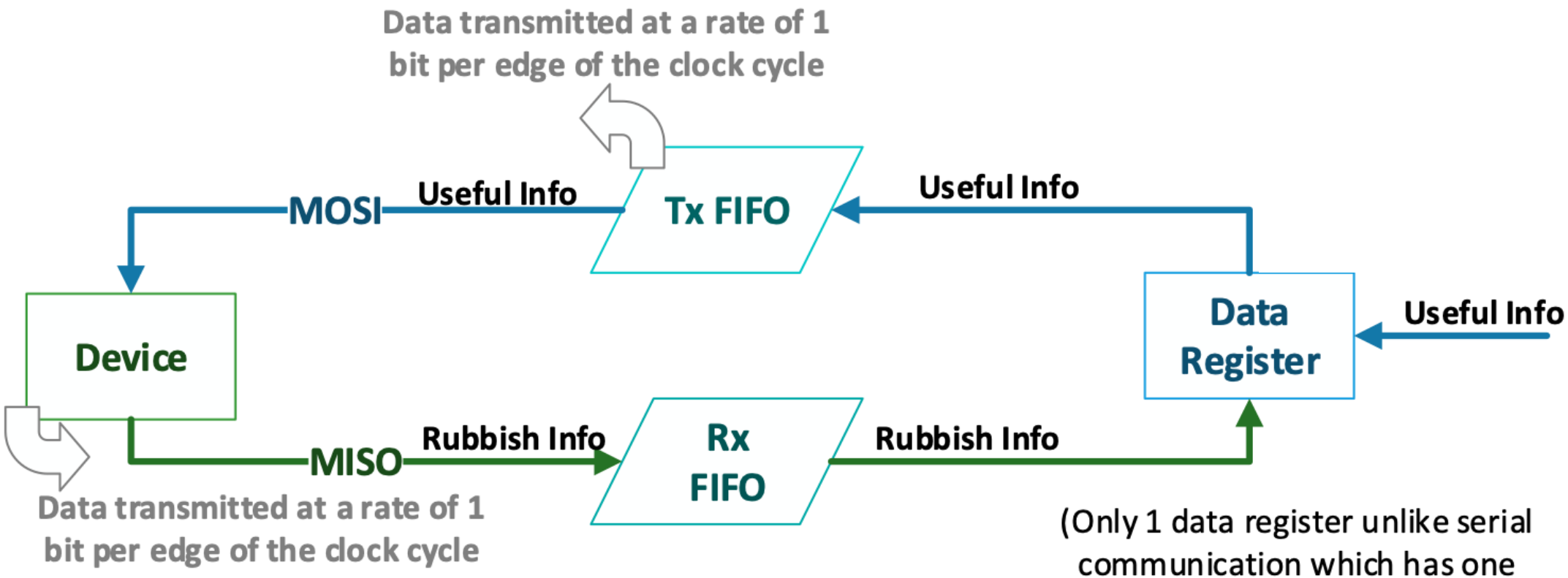
SPI

Operation in master slave full-duplex mode:



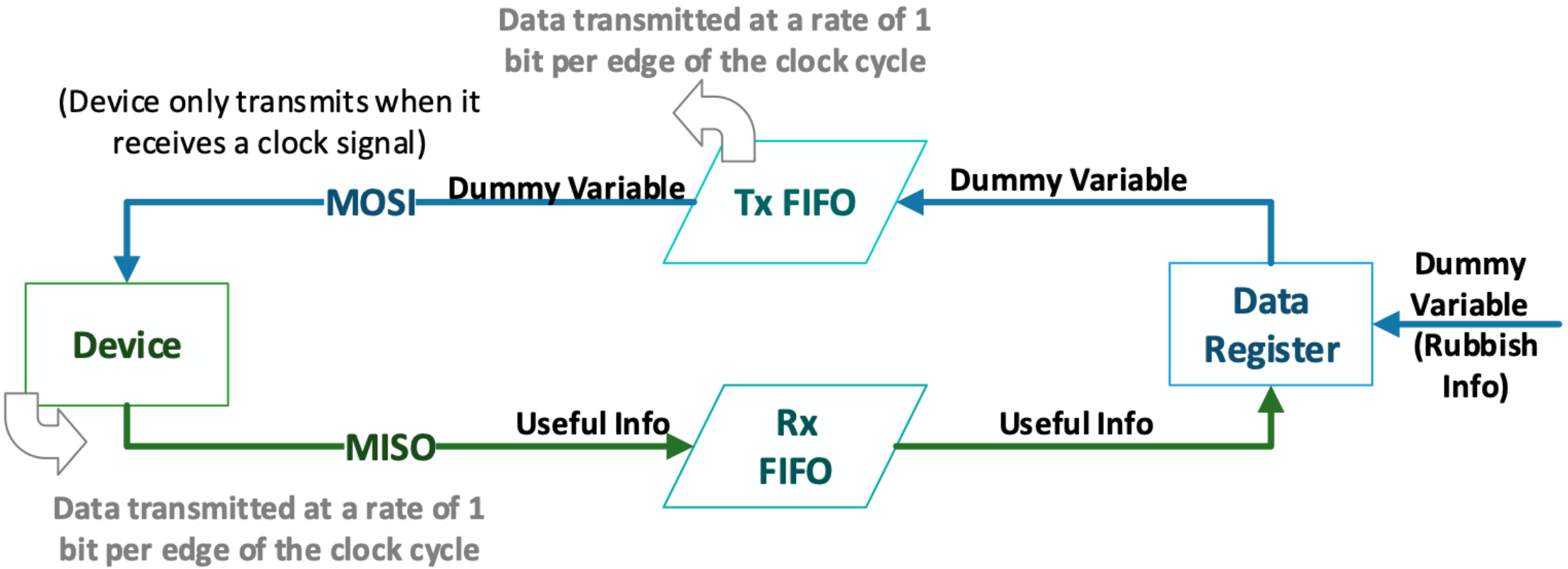
SPI

Transmit data to device from microprocessor:

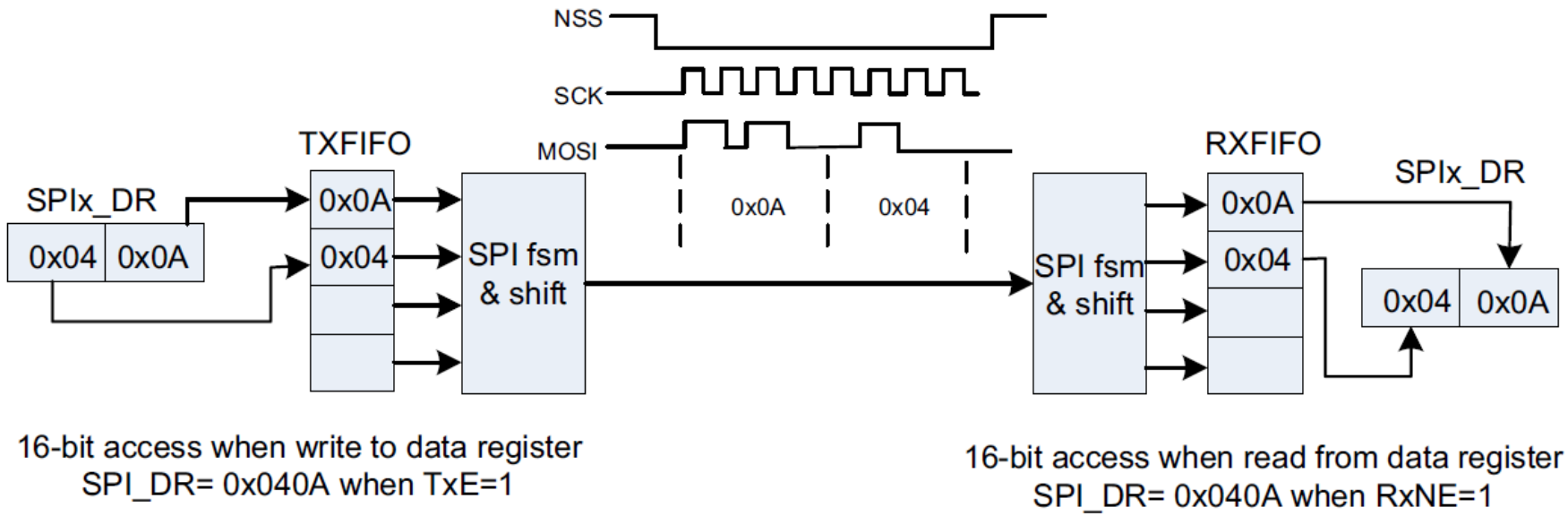


SPI

Read data from device to microprocessor:



SPI



16-bit access when write to data register
SPI_DR= 0x040A when TxE=1

16-bit access when read from data register
SPI_DR= 0x040A when RxNE=1

SPI

SPI interrupt requests

Interrupt event	Event flag	Enable Control bit
Transmit TXFIFO ready to be loaded	TXE	TXEIE
Data received in RXFIFO	RXNE	RXNEIE
Master Mode fault event	MODF	ERRIE
Overrun error	OVR	
TI frame format error	FRE	
CRC protocol error	CRCERR	

SPI

Advantages

- It's faster than asynchronous serial
- The receive hardware can be a simple shift register - Cheaper
- It supports multiple slaves
- No overheads (Start/Stop bits)

Disadvantages

- Numerous signal lines
- Amount of data must be defined before transfer
- It usually requires separate SS lines to each slave, which can be problematic if numerous slaves are needed

SPI

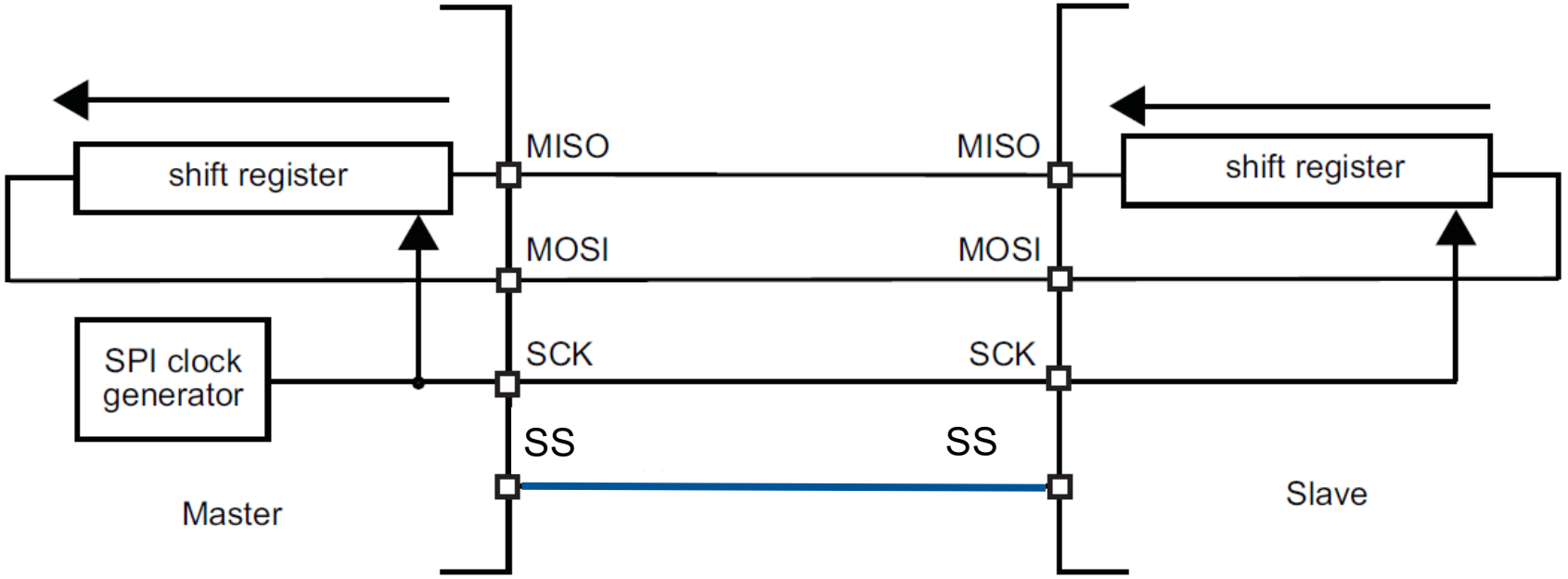
SPI set up steps:

1. Set up GPIO pins for MOSI, MISO and SCK
2. SPI_CR1 register:
   - Set serial clock Baud Rate
   - Set up relationship between data tx & serial clock (CPOL & CPHA)
   - Simplex or half duplex or duplex mode mode (RXONLY/BIDIMODE & BIDIOE)
   - Define frame format (LSBFIRST)
   - Configure SSM & SSI (If using NSS pin)
   - Configure MSTR
3. SPI_CR2 register:
   - Select the data length for the transfer (DS)
   - Configure FRXTH
   - Configure SSOE (If using NSS pin)
4. SPI_CR1 register
   - Set SPE

SPI

Use SPI2 in full duplex mode to read and write a byte from a slave device.

SS = PB12, SPI2 on PB13-15



SPI

This code transmits & receives data from the STM to a slave device

```c
#include "stm32f0xx.h"

void main(void);
void init_spi(void);
void spi_write_byte(uint8_t data);
uint8_t spi_read_byte(void)

void main(void)
{
 init_spi();

 spi_write_byte(5);
 uint8_t data = spi_read_byte();

 while(1)
 {}
}

void init_spi(void)
{
 // enable clock to port B
 RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
 // set pins PB13-15 to alternate function, PB12 to output
 GPIOB->MODER |= GPIO_MODER_MODER12_0 | GPIO_MODER_MODER13_1
                 | GPIO_MODER_MODER14_1 | GPIO_MODER_MODER15_1;
 // disable slave: PB12 High (SS High)
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // enable SPI2
 RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
```

Default State

AF0

PB13	SPI2_SCK
PB14	SPI2_MISO
PB15	SPI2_MOSI

SPI

```
void init_spi(void)
{
 // enable clock to port B
 RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
 // set pins PB13-15 to alternate function, PB12 to output
 GPIOB->MODER |= GPIO_MODER_MODER12_0 | GPIO_MODER_MODER13_1
                 | GPIO_MODER_MODER14_1 | GPIO_MODER_MODER15_1;
 // disable slave: PB12 High (SS High)
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // clock SPI2
 RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
 // set MOSI to transmit only
 SPI2->CR1 |= SPI_CR1_BIDIOE;
```

Bit 14 **BIDIOE:** Output enable in bidirectional mode

This bit combined with the BIDIMODE bit selects the direction of transfer in bidirectional mode

0: Output disabled (receive-only mode)

1: Output enabled (transmit-only mode)

27.9.1 SPI control register 1 (SPIx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	CRCL	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SPI

```
void init_spi(void)
{
 // enable clock to port B
 RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
 // set pins PB13-15 to alternate function, PB12 to output
 GPIOB->MODER |= GPIO_MODER_MODER12_0 | GPIO_MODER_MODER13_1
                 | GPIO_MODER_MODER14_1 | GPIO_MODER_MODER15_1;
 // disable slave: PB12 High (SS High)
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // clock SPI2
 RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
 // set MOSI to transmit only
 SPI2->CR1 |= SPI_CR1_BIDIOE;
 // set clock to fPCLK/16 = 3 MHz (fPCLK = 48 MHz)
 SPI2->CR1 |= SPI_CR1_BR_1 | SPI_CR1_BR0;
```

Bits 5:3 **BR[2:0]:** Baud rate control

000: $f_{PCLK}/2$
001: $f_{PCLK}/4$
010: $f_{PCLK}/8$
011: $f_{PCLK}/16$
100: $f_{PCLK}/32$
101: $f_{PCLK}/64$
110: $f_{PCLK}/128$
111: $f_{PCLK}/256$

27.9.1 SPI control register 1 (SPIx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	CRCL	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SPI

```
void init_spi(void)
{
 // enable clock to port B
 RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
 // set pins PB13-15 to alternate function, PB12 to output
 GPIOB->MODER |= GPIO_MODER_MODER12_0 | GPIO_MODER_MODER13_1
                 | GPIO_MODER_MODER14_1 | GPIO_MODER_MODER15_1;
 // disable slave: PB12 High (SS High)
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // clock SPI2
 RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
 // set MOSI to transmit only
 SPI2->CR1 |= SPI_CR1_BIDIOE;
 // set clock to fPCLK/16 = 3 MHz (fPCLK = 48 MHz)
 SPI2->CR1 |= SPI_CR1_BR_1 | SPI_CR1_BR0;
 // set STM to master
 SPI2->CR1 |= SPI_CR1_MSTR;
```

Bit 2 **MSTR:** Master selection

0: Slave configuration

1: Master configuration

27.9.1 SPI control register 1 (SPIx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	CRCL	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SPI

```
void init_spi(void)
{
 // enable clock to port B
 RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
 // set pins PB13-15 to alternate function, PB12 to output
 GPIOB->MODER |= GPIO_MODER_MODER12_0 | GPIO_MODER_MODER13_1
                 | GPIO_MODER_MODER14_1 | GPIO_MODER_MODER15_1;
 // disable slave: PB12 High (SS High)
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // clock SPI2
 RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
 // set MOSI to transmit only
 SPI2->CR1 |= SPI_CR1_BIDIOE;
 // set clock to fPCLK/16 = 3 MHz (fPCLK = 48 MHz)
 SPI2->CR1 |= SPI_CR1_BR_1 | SPI_CR1_BR0;
 // set STM to master
 SPI2->CR1 |= SPI_CR1_MSTR;
 // set data length to 8 bits
 SPI2->CR2 |= SPI_CR2_FRXTH | SPI_CR2_DS2
              | SPI_CR2_DS1 | SPI_CR2_DS0;
```

Bits 11:8 **DS [3:0]**: Data size

These bits configure the data length for SPI transfers:

0000: Not used
0001: Not used
0010: Not used
0011: 4-bit
0100: 5-bit
0101: 6-bit
0110: 7-bit
0111: 8-bit

Bit 12 **FRXTH**: FIFO reception threshold

This bit is used to set the threshold of the RXFIFO that triggers an RXNE event

0: RXNE event is generated if the FIFO level is greater than or equal to 1/2 (16-bit)
1: RXNE event is generated if the FIFO level is greater than or equal to 1/4 (8-bit)

27.9.2 SPI control register 2 (SPIx_CR2)

Address offset: 0x04

Reset value: 0x0700

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	LDMA_TX	LDMA_RX	FRXTH	DS [3:0]				TXEIE	RXNEIE	ERRIE	FRF	NSSP	SSOE	TXDMAEN	RXDMAEN
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SPI

```
void init_spi(void)
{
 // enable clock to port B
 RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
 // set pins PB13-15 to alternate function, PB12 to output
 GPIOB->MODER |= GPIO_MODER_MODER12_0 | GPIO_MODER_MODER13_1
                 | GPIO_MODER_MODER14_1 | GPIO_MODER_MODER15_1;
 // disable slave: PB12 High (SS High)
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // clock SPI2
 RCC->APB1ENR |= RCC_APB1ENR_SPI2EN;
 // set MOSI to transmit only
 SPI2->CR1 |= SPI_CR1_BIDIOE;
 // set clock to fPCLK/16 = 3 MHz (fPCLK = 48 MHz)
 SPI2->CR1 |= SPI_CR1_BR_1 | SPI_CR1_BR0;
 // set STM to master
 SPI2->CR1 |= SPI_CR1_MSTR;
 // set data length to 8 bits
 SPI2->CR2 |= SPI_CR2_FRXTH | SPI_CR2_DS2
              | SPI_CR2_DS1 | SPI_CR2_DS0;
 // enable SPI2
 SPI2->CR1 |= SPI_CR2_SPE;
}
```

27.9.1 SPI control register 1 (SPIx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	CRCL	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

SPI

```
void spi_write_byte(uint8_t data)
{
 // enable slave: PB12 LOW
 GPIOB->BSRR |= GPIO_BSRR_BR_12;
 // write data
 SPI2->DR = data;
 // wait for 8 bits to be placed in rx-fifo
 while((SPI2->SR & SPI_SR_RXNE) == 0);
 // read dummy data transferred to shift register
 uint8_t DummyData = SPI2->DR;
 // disable slave: PB12 HIGH
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
}

uint8_t spi_read_byte(void)
{
 uint8_t DummyData = 0x11;

 // enable slave: PB12 LOW
 GPIOB->BSRR |= GPIO_BSRR_BR_12;
 // transmit dummy data to shift data from slave
 SPI2->DR = DummyData;
 // wait for 8 bits to be placed in rx-fifo
 while((SPI2->SR & SPI_SR_RXNE) == 0);
 // read data transferred to shift register
 uint8_t ReceivedData = SPI2->DR;
 // disable slave: PB12 HIGH
 GPIOB->BSRR |= GPIO_BSRR_BS_12;
 // return data from function
 return ReceivedData;
}
```