

Introduction to I2C

Inter–Integrated Circuit(I2C) Protocol

Pronounced as

“I squared C “ or “ I two C “

What is I2C ?

It is just a protocol to achieve serial data communication between integrated circuits(ICs) which are very close to each other. (but more serious protocol than SPI because companies have come forward to design a specification)

I2C protocol details (how data should sent, how data should received, how hand shaking should happen between sender and receiver, error handling,) are complex than SPI. (In other words SPI is simple protocol compared to I2C)

Difference between SPI and I2C

How different than SPI ?

Specification



I2c is based on dedicated specification. The spec you
can download from here

<https://www.nxp.com/docs/en/user-guide/UM10204.pdf>

For SPI there is no dedicated spec but TI and Motorola have their own
spec

How different than SPI ?

Multi-Master Capability



I2C protocol is multi-master capable ,
whereas SPI has no guidelines to achieve this, but depends on
MCU designers . STM SPI peripherals can be used in multi
master configurations but arbitration should be handled by
software code.

How different than SPI ?

ACK



I2C hardware automatically ACKs every byte received.
SPI does not support any automatic ACKing.

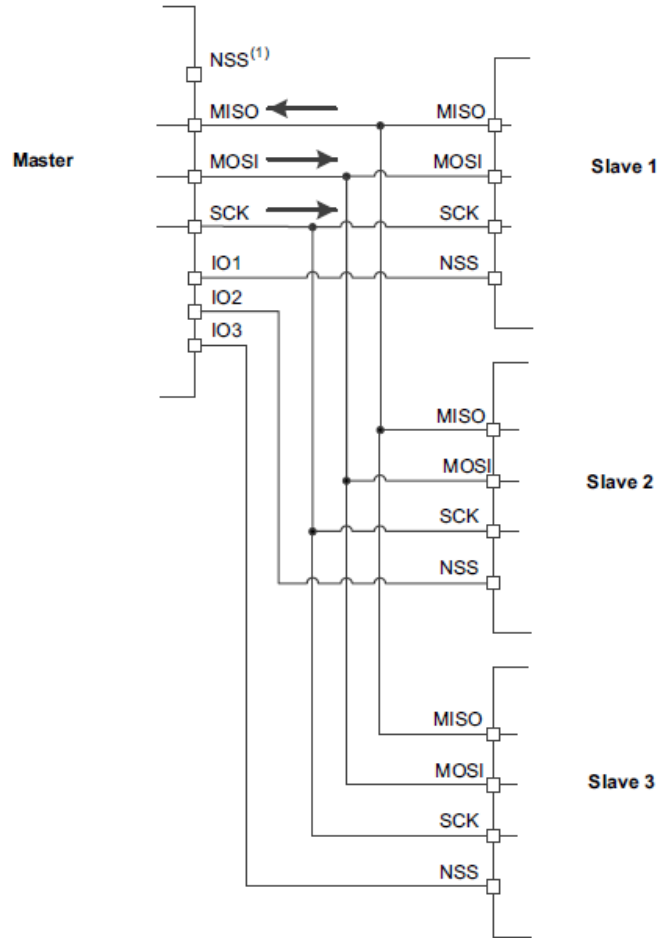
How different than SPI ?

Pins



I2C needs just 2 pins for the communication
whereas SPI may need 4 pins and even more than that
if multiple slaves are involved

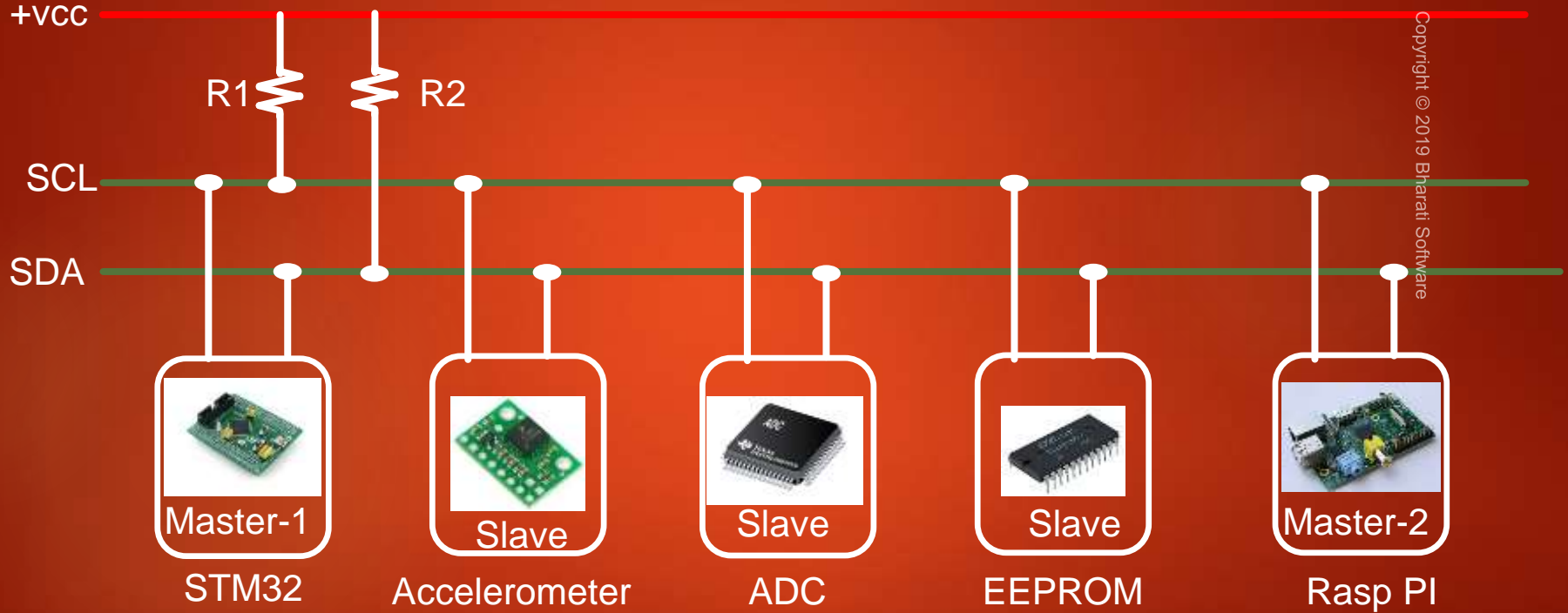
Master and three independent slaves



MSv39626V1

SPI Consumes more pins when more slaves are involved

In I2C, you just need 2 pins to connect all the slaves and masters



How different than SPI ?

Addressing



I 2C master talks to slaves based on slave addresses, whereas in SPI dedicated pin is used to select the slave.

How different than SPI ?

Communication



I2C is half duplex, where is SPI is full duplex

How different than SPI ?

Speed



For I2C the max speed is 4MHz in ultra speed plus .
For some STM microcontrollers the max speed is just 400KHz.
For SPI max speed is its $F_{pclk}/2$. That means if
the peripheral clock is 20MHz, then speed can be 10MHz

How different than SPI ?

Slave's control over serial clock



In I2C slave can make master wait by holding the clock down if it's busy , thanks to clock stretching feature of I2C.
But in SPI, slave has no control over clock, programmers may use their own tricks to overcome this situation .

Data rate

Data rate (number of bits transferred from sender to receiver in 1 sec) is very much lesser in I2C compared to SPI.

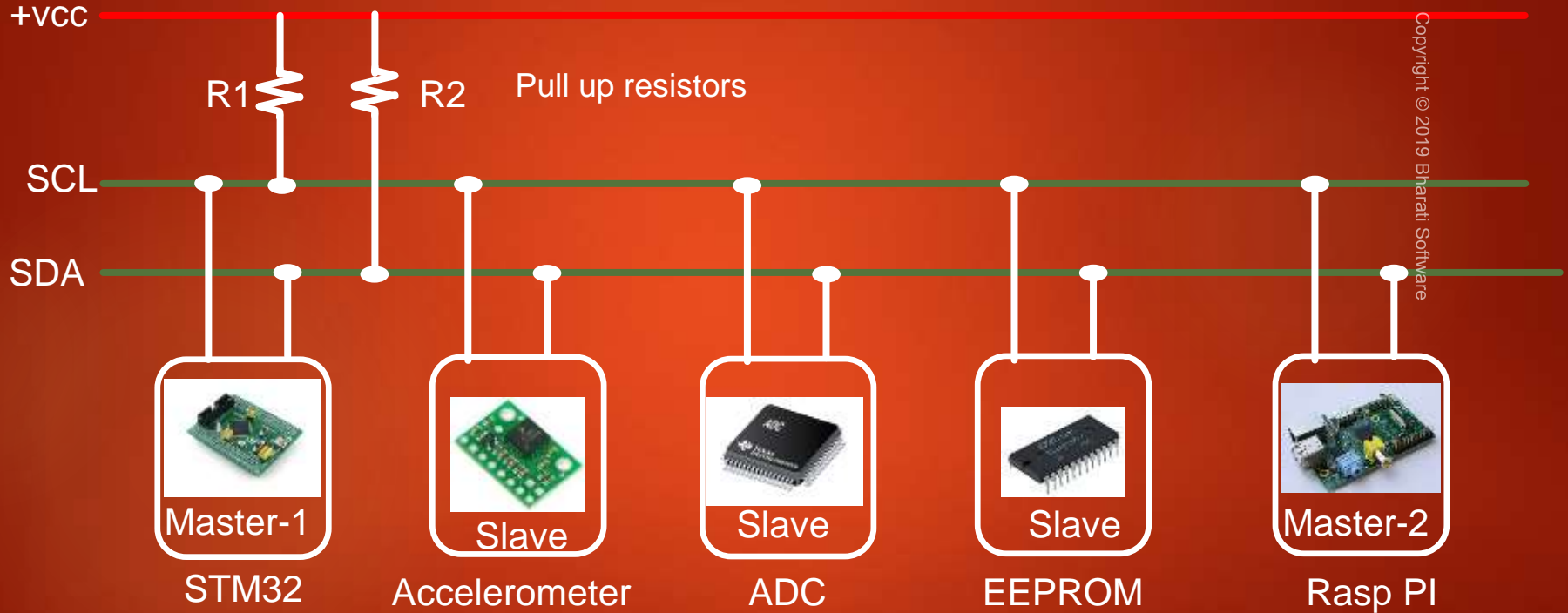
For example in STM32F4X if you have peripheral clock of 40MHz, then in I2C you can achieve data rate of 400Kbps but in SPI it is 20Mbps.

So in the above scenario SPI is 50 times faster than I2C.

Definition of I²C-bus terminology

Term	Description
Transmitter	the device which sends data to the bus
Receiver	the device which receives data from the bus
Master	the device which initiates a transfer, generates clock signals and terminates a transfer
Slave	the device addressed by a master
Multi-master	more than one master can attempt to control the bus at the same time without corrupting the message
Arbitration	procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	procedure to synchronize the clock signals of two or more devices

In I2C, you just need 2 pins to connect all the slaves and masters

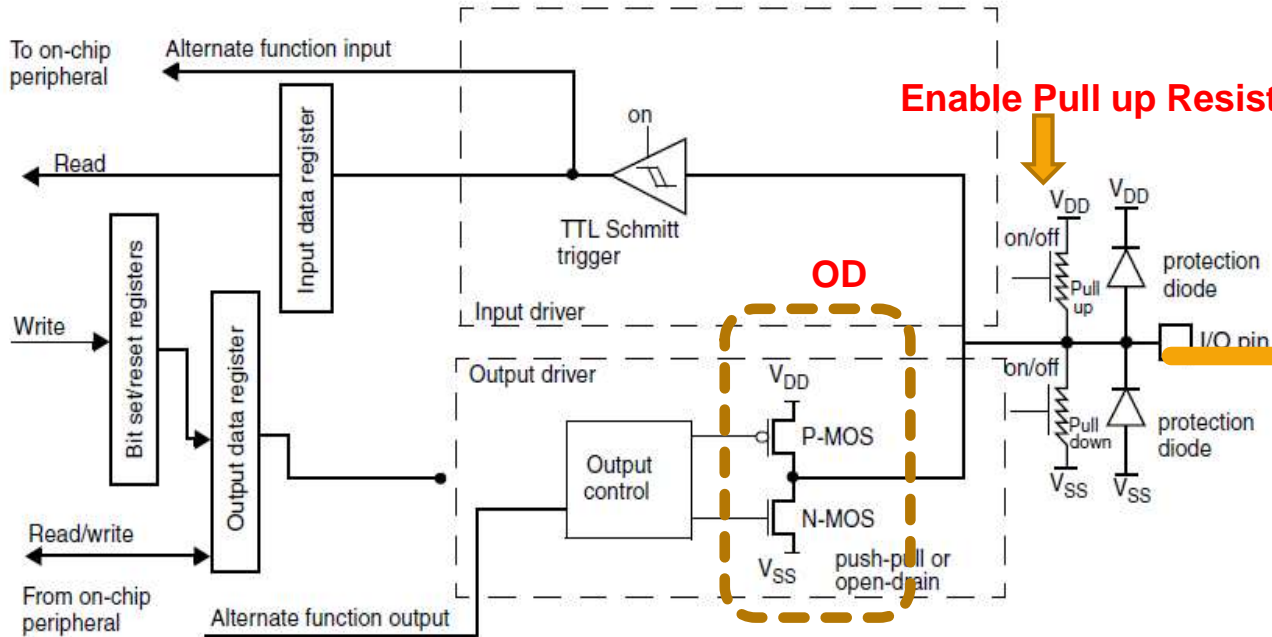


SDA and SCL signals

- ✓ Both SDA and SCL are bidirectional lines connected to a positive supply voltage via pull-up resistors. When the bus is free, both lines are held at HIGH.
- ✓ The output stages of devices connected to the bus must have an open-drain or open-collector configuration
- ✓ The bus capacitance limits the number of interfaces connected to the bus.

I2C: Pin Configuration

Alternate function configuration

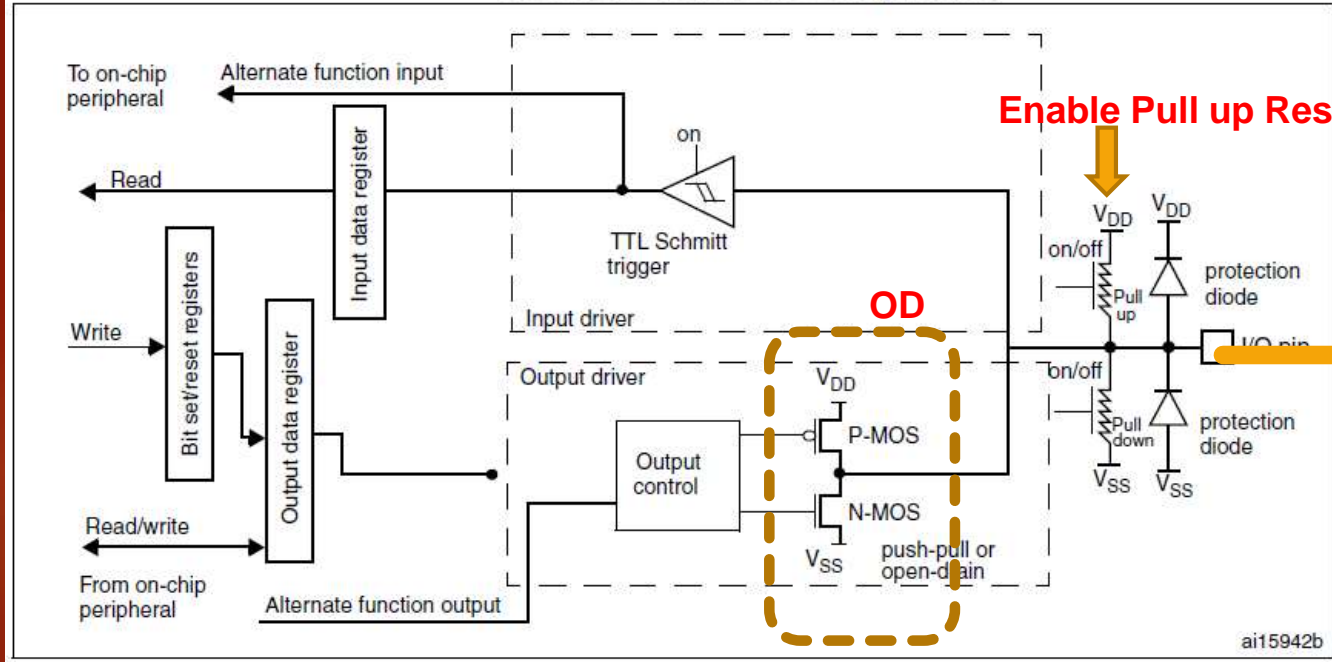


ai15942b

For proper functioning of the I2C bus, pull up resistor value has to be calculated according to the I2C formula (will discuss later)

SCL

Alternate function configuration



For proper functioning of the I2C bus, pull up resistor value has to be calculated according to the I2C formula (will discuss later)

Copyright © 2011 Bharti Software

SDA

Trouble shooting tip : When the bus is idle, both SDA and SCL are pulled to +Vdd

GND

SDA & SCL



Tip

Whenever you face problems in I2C , probe the SDA and SCL line after I2C initialization. It must be held at HIGH (3.3 V or 1.8V depending up on IO voltage levels of your board)

I2C Modes

Mode	Data rate	Notes
Standard Mode	Up to 100 Kbits/sec	Supported by STMf2F4x
Fast Mode	Up to 400 Kbits/sec	Supported by STMf2F4x
Fast Mode +	Up to 1 Mbits/sec	Supported by some STMf2F4x MCUs (refer RM)
High Speed mode	Up to 3.4 Mbits/sec	Not supported by F4x

Fast Mode/Standard Mode

Standard Mode

- ▶ In standard mode communication data transfer rate can reach up to maximum of 100kbits/sec.
- ▶ Standard mode was the very first mode introduced when first i2c spec was released.
- ▶ Standard-mode devices, however, are not upward compatible; They cant communicate with devices of Fast mode or above.

Fast Mode

- ▶ Fast mode is a mode in i2c protocol, where devices can receive and transmit data up to 400 kbits/s.
- ▶ Fast-mode devices are downward-compatible and can communicate with Standard-mode devices in a 0 to 100 kbit/s I2C-bus system
- ▶ Standard-mode devices, however, are not upward compatible; they should not be incorporated in a Fast-mode I2C-bus system as they cannot follow the higher transfer rate and unpredictable states would occur.
- ▶ To achieve data transfer rate up to 400kbits/sec, you must put the i2c device in to fast mode



Basics of I2C Protocol

R/nW = '1' indicates request
for data (READ)

R/nW = '0' indicates a
transmission (WRITE),



- ✓ Every byte put on the SDA line must be eight bits long.
- ✓ Each byte must be followed by an Acknowledge Bit
- ✓ Data is transferred with the Most Significant Bit (MSB) first

MSB

LSB



slave address

mbc608

The first byte after the START procedure



START and STOP conditions

A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition.
A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

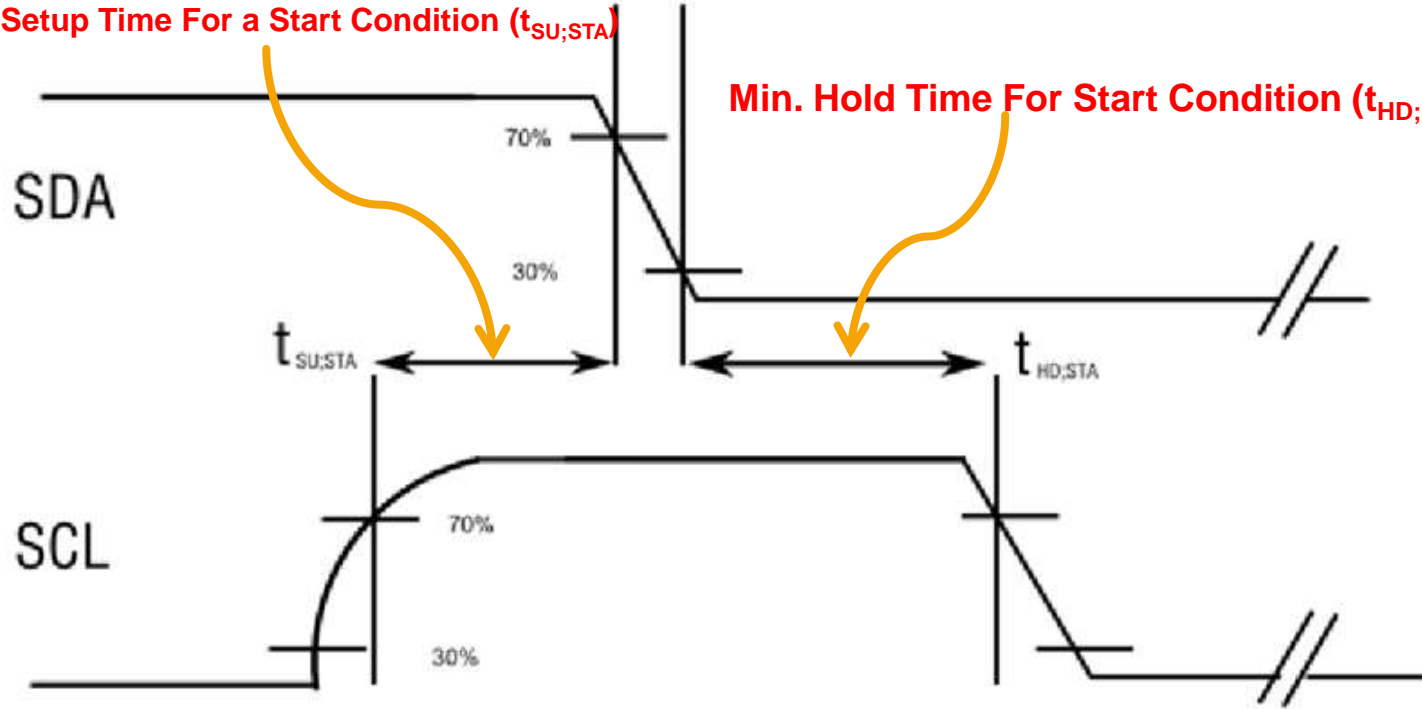
Copyright © 2019 Bharati Software



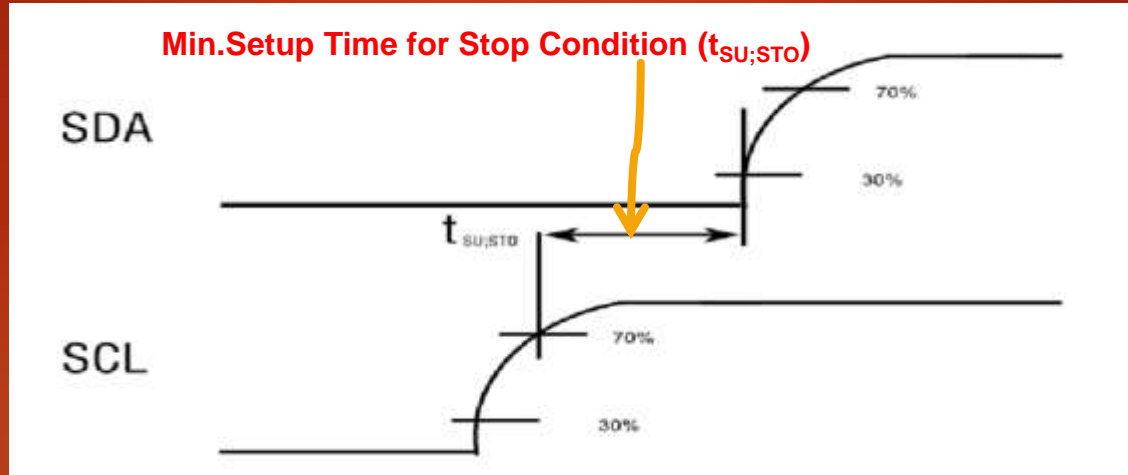
I2C Protocol: Start condition timings

Min. Setup Time For a Start Condition ($t_{SU;STA}$)

Min. Hold Time For Start Condition ($t_{HD;STA}$)



I2C Protocol: stop condition timings



Setup Time for Stop Condition ($t_{SU;STO}$) is measured as the time between 70% amplitude of the rising edge of SCL and 30% amplitude of a rising SDA signal during a stop condition.

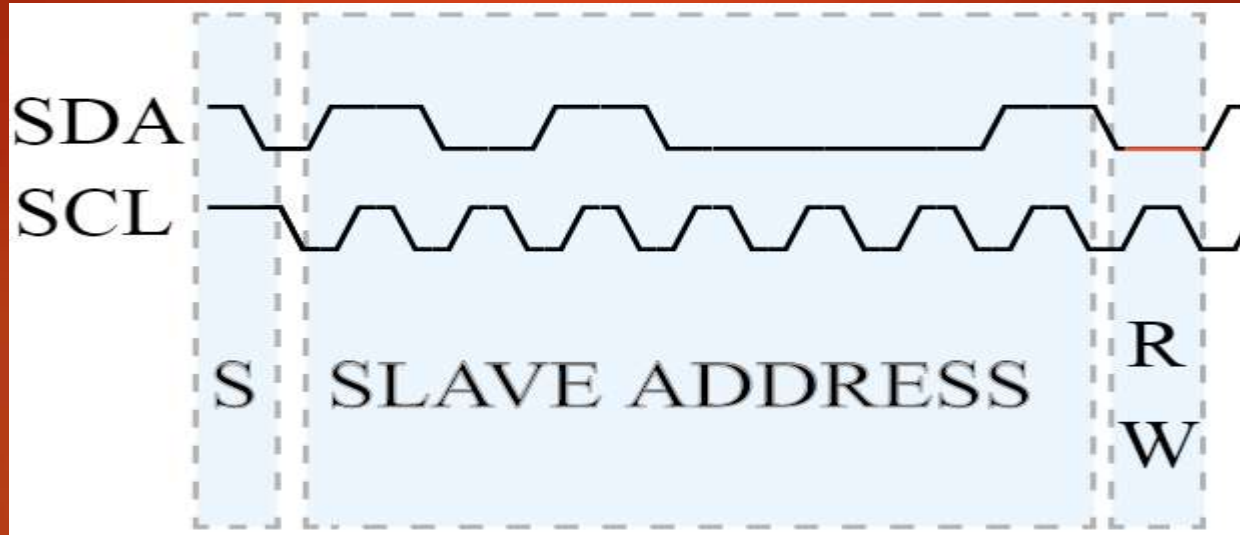
Points to Remember

- ▶ START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition.
- ▶ The bus is considered to be free again a certain time after the STOP condition.
- ▶ When the bus is free another master(if present) can get the chance to claim the bus.
- ▶ The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition.
- ▶ Most of the MCU's I2C peripherals support both master and slave mode. You need not to configure the mode because when the peripheral generates the start condition it automatically becomes the master and when it generates the stop condition it goes back to slave mode.



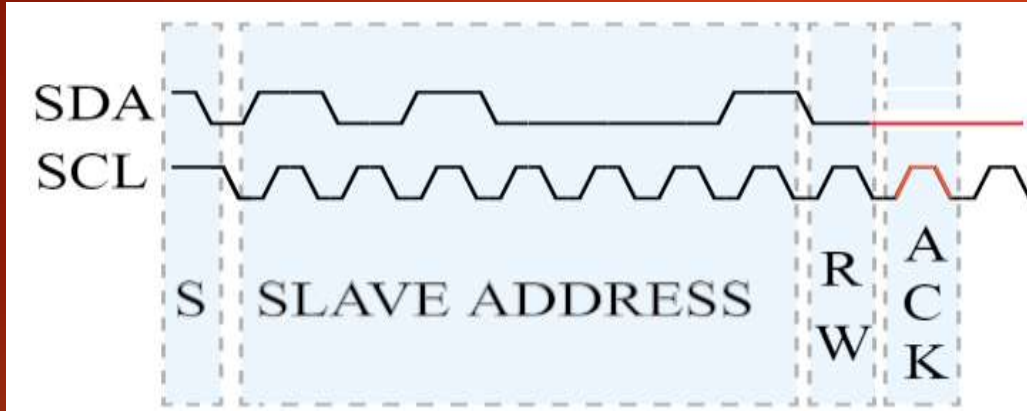
I2C Protocol: Address Phase

I2C Protocol: Address Phase



I2C Protocol: ACK/NACK

I2C Protocol: ACK

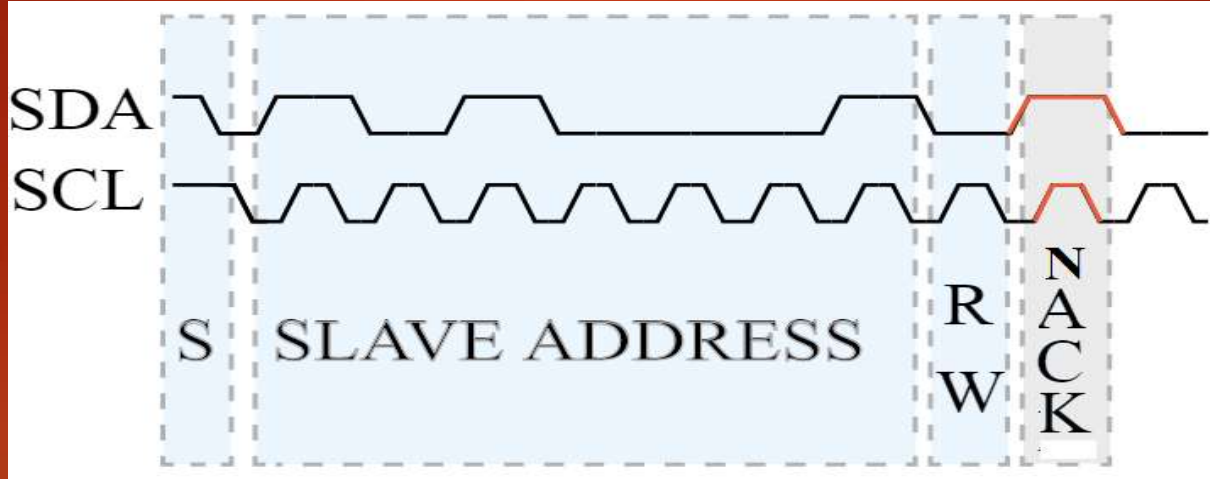


The Acknowledge signal is defined as follows:

The transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse

- ✓The acknowledge takes place after every byte.
- ✓The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent.
- ✓The master generates all clock pulses, including the acknowledge ninth clock pulse

I2C Protocol: NACK

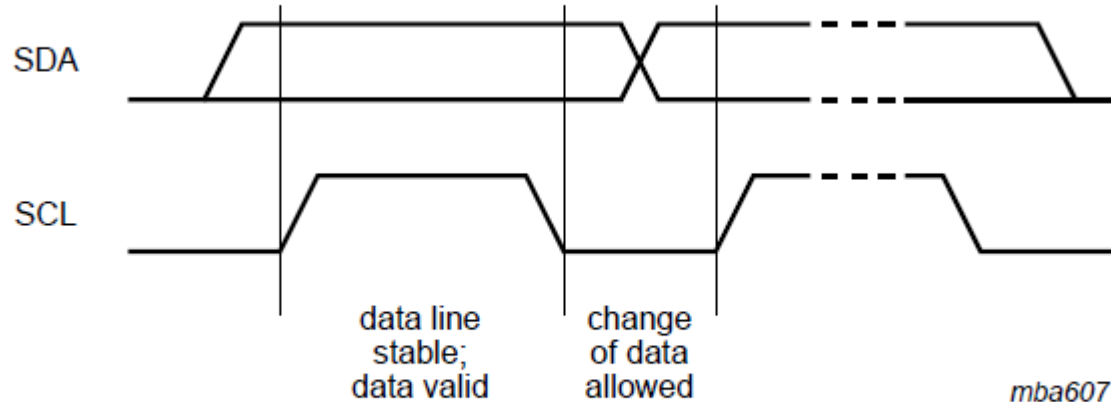


✓When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal.

✓The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer.

Data validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH Or LOW state of the data line can only change when the clock signal on the SCL line is LOW . One clock pulse is generated for each data bit transferred.



Bit transfer on the I²C-bus



Master Writing data to slave

SCL

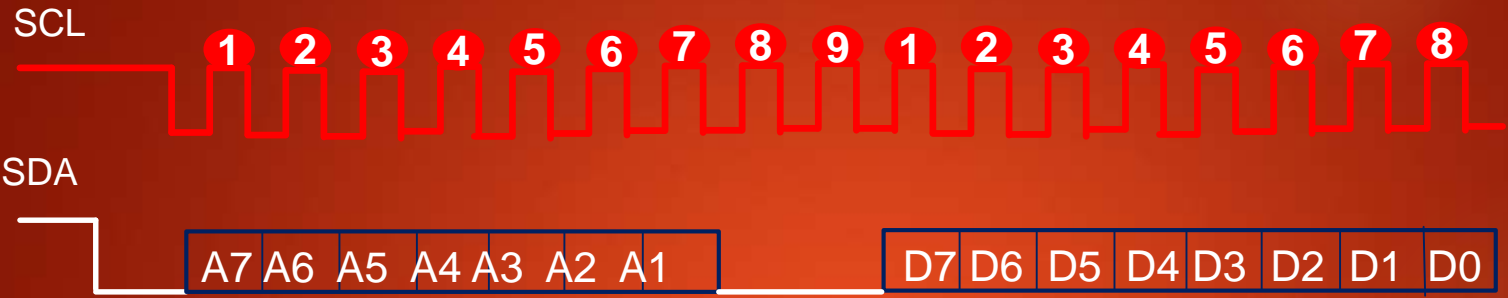


SDA



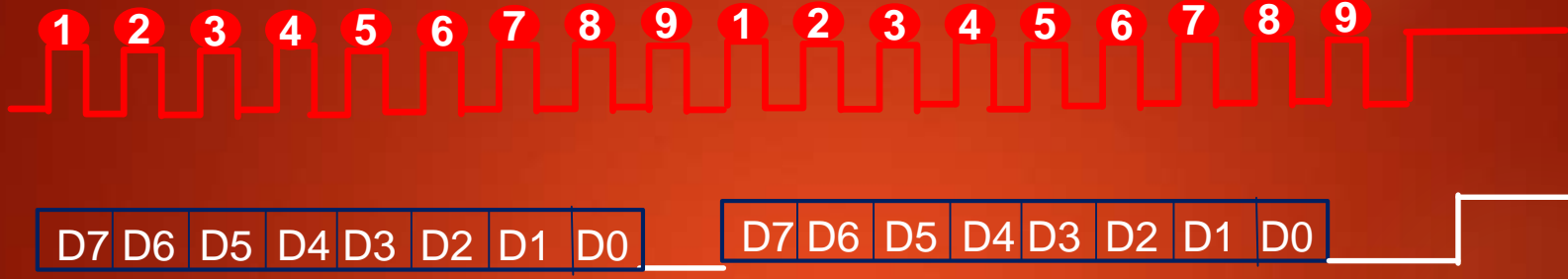














Master Reading data from slave

SCL



SDA















SCL



SDA



SCL

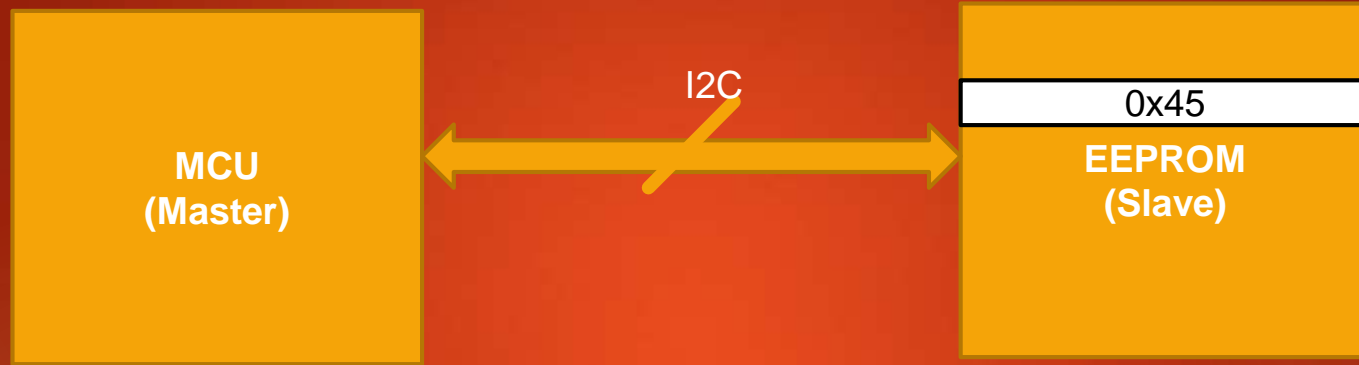
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

SDA



Repeated Start(S_r)

(Start again without Stop)



Master reading the contents of EEPROM at address 0x45

Without repeated start

SCL

1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

SDA

A7 A6 A5 A4 A3 A2 A1

D7 D6 D5 D4 D3 D2 D1 D0

Copyright © 2019 Bharati Software

1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

A7 A6 A5 A4 A3 A2 A1

D7 D6 D5 D4 D3 D2 D1 D0

With repeated start

SCL

1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

SDA

A7 A6 A5 A4 A3 A2 A1

D7 D6 D5 D4 D3 D2 D1 D0

Copyright © 2019 Bharati Software

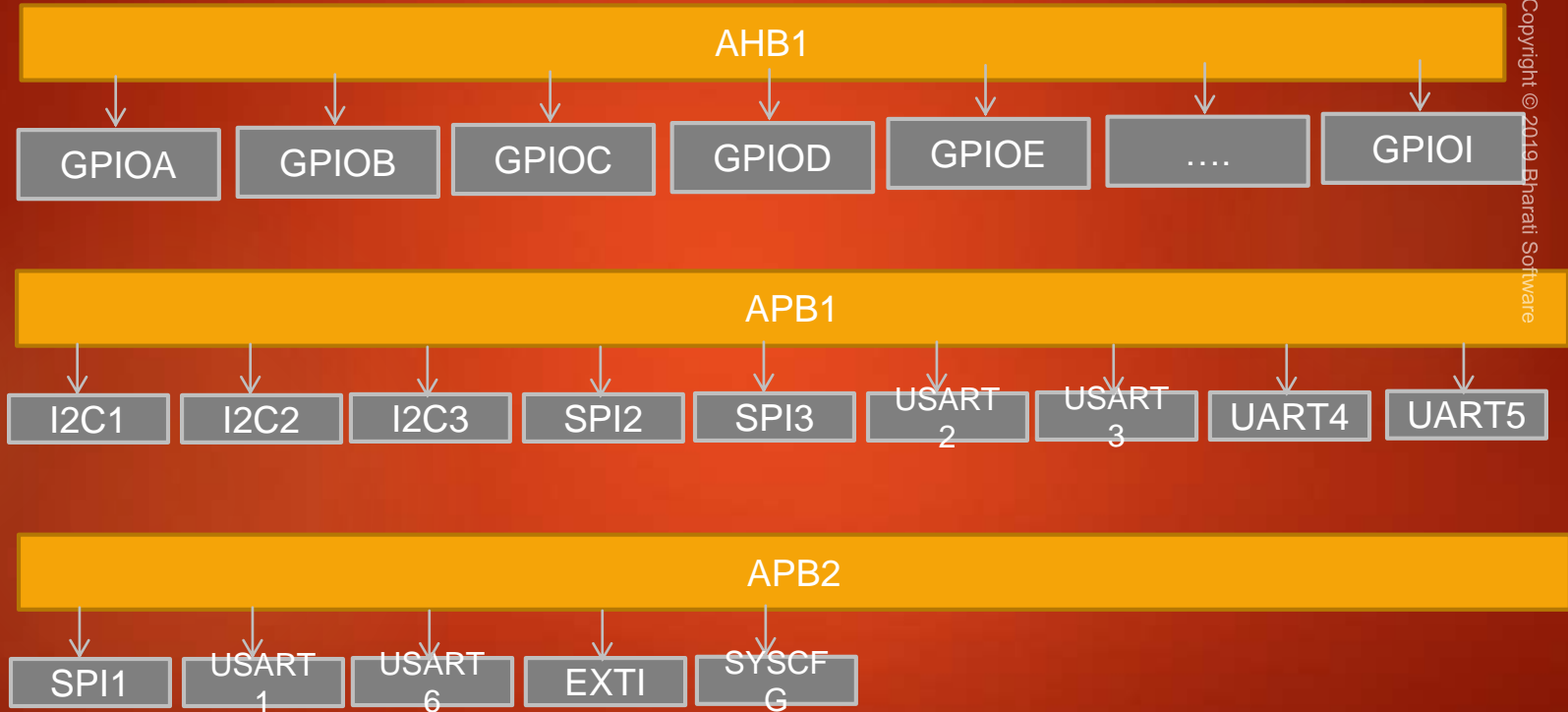
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

A7 A6 A5 A4 A3 A2 A1

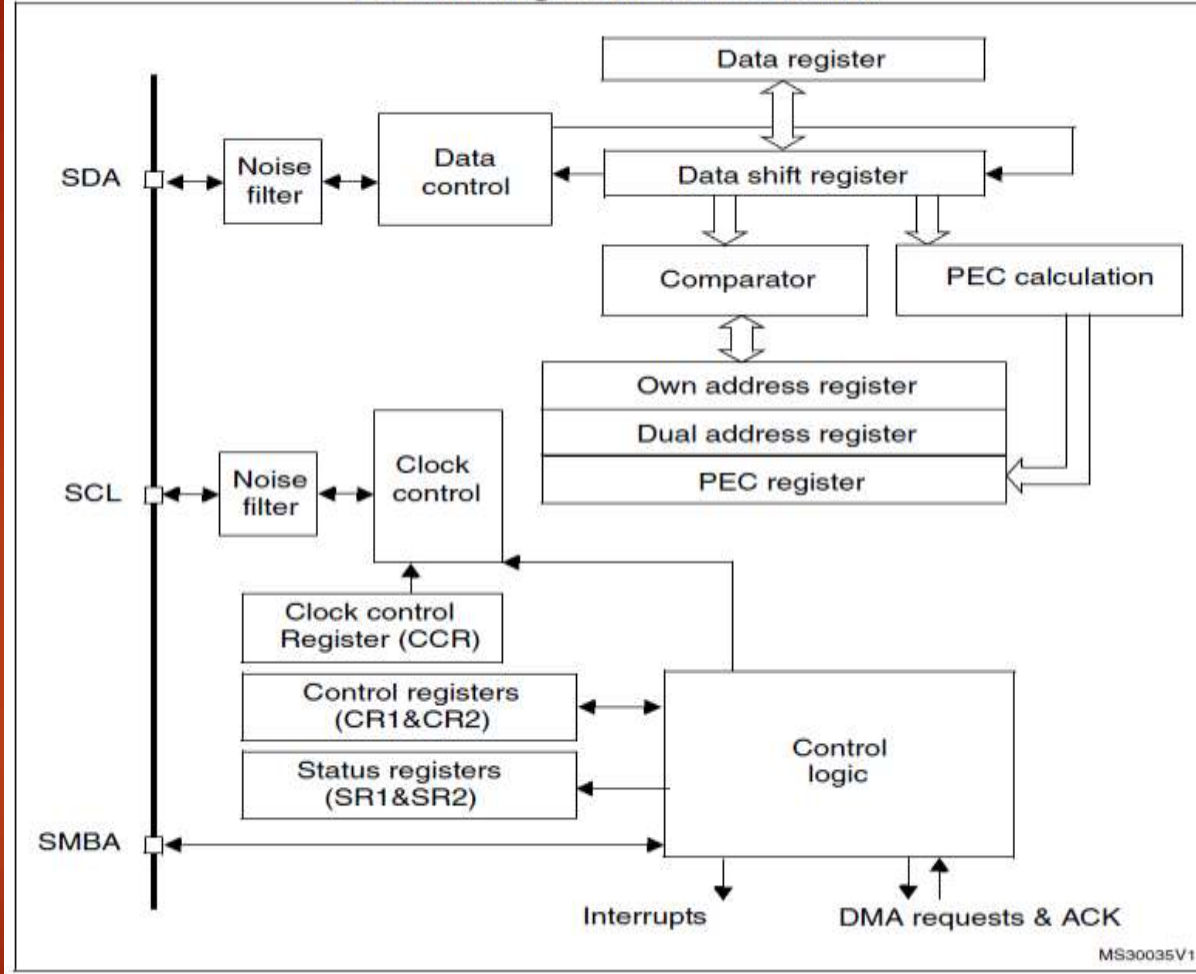
D7 D6 D5 D4 D3 D2 D1 D0

I2C peripherals of your MCU

Copyright © 2019 Bharati Software



I²C block diagram for STM32F40x/41x



I2C Driver Development

Sample Applications



Driver Layer

gpio_driver.c , .h

i2c_driver.c , .h

(Device header)

Stm3f407xx.h

spi_driver.c , .h

uart_driver.c , .h

Copyright © 2019 Bharati Software



GPIO

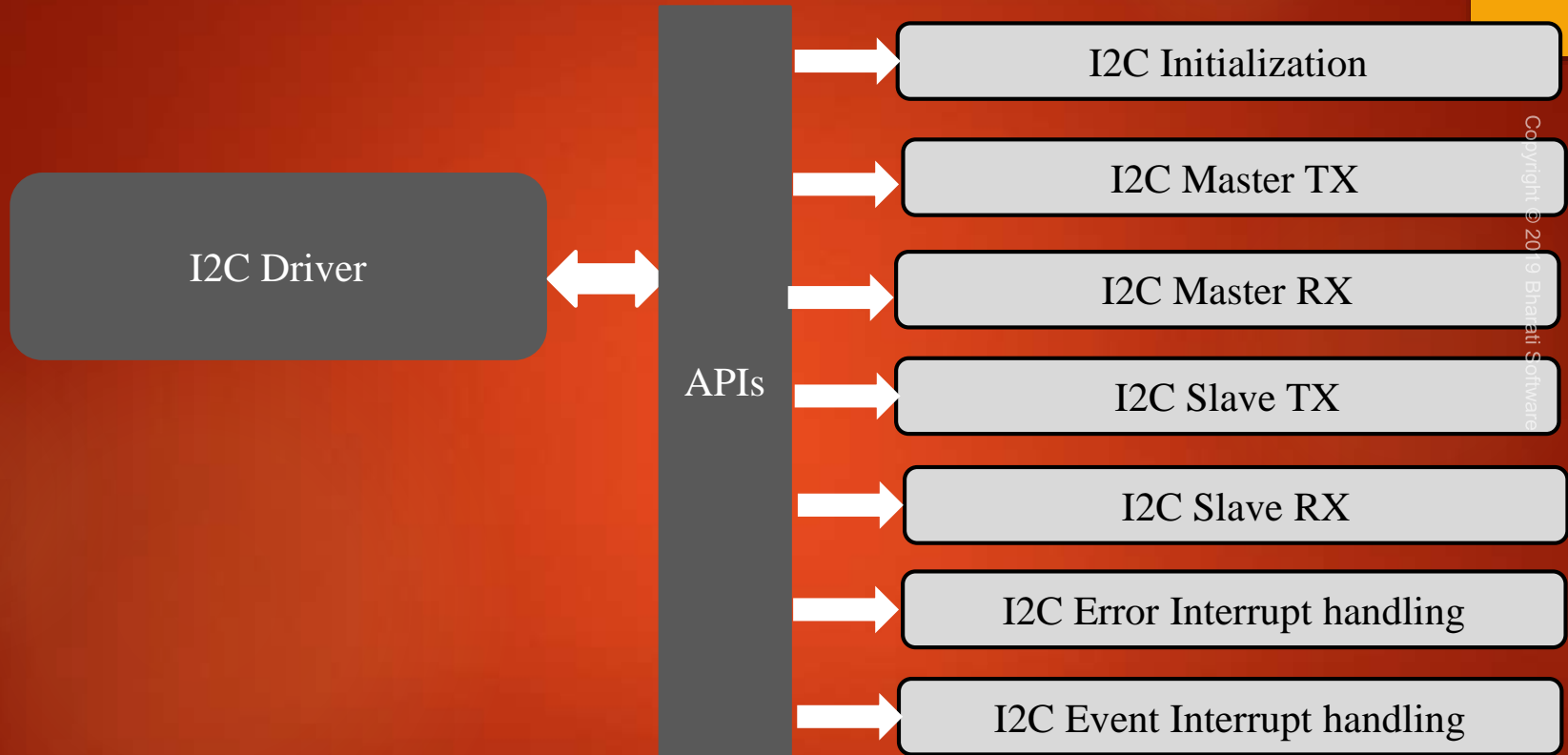
SPI

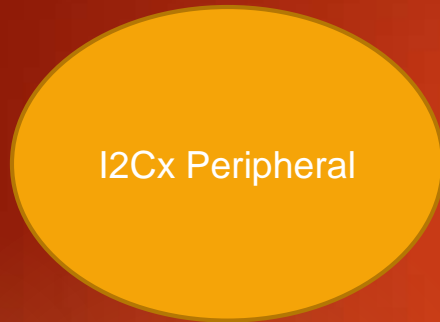
I2C

UART

STM3F407x MCU

Driver API requirements and user configurable items





I2C_SCLSpeed

I2C_DeviceAddress

I2C_ACKControl

I2C_FMDutyCycle



Configurable items
For user application

Exercise:

1. Create `stm32f407xx_i2c_driver.c` and `stm32f407xx_i2c_driver.h`
2. Add I2Cx related details to MCU specific header file
 - I. I2C peripheral register definition structure
 - II. I2Cx base address macros
 - III. I2Cx peripheral definition macros
 - IV. Macros to enable and disable I2Cx peripheral clock
 - V. Bit position definitions of I2C peripheral

I2C handle structure and configuration structure

Create Configuration and Handle structure

```
/*
 * Configuration structure for I2Cx peripheral
 */
typedef struct
{
    uint32_t I2C_SCLSpeed;
    uint8_t I2C_DeviceAddress;
    uint8_t I2C_ACKControl;
    uint16_t I2C_FMDutyCycle;
}I2C_Config_t;

/*
 *Handle structure for I2Cx peripheral
 */
typedef struct
{
    I2C_RegDef_t *pI2Cx;
    I2C_Config_t I2C_Config;
}I2C_Handle_t;
```

▶ check comments

Implementation

I2C_Init() API

Steps for I2C init (Generic)

1. Configure the Mode (standard or fast)
2. Configure the speed of the serial clock (SCL)
3. Configure the device address (Applicable when device is slave)
4. Enable the Acking
5. Configure the rise time for I2C pins (will discuss later)

All the above configuration must be done when the peripheral is disabled in the control register

100

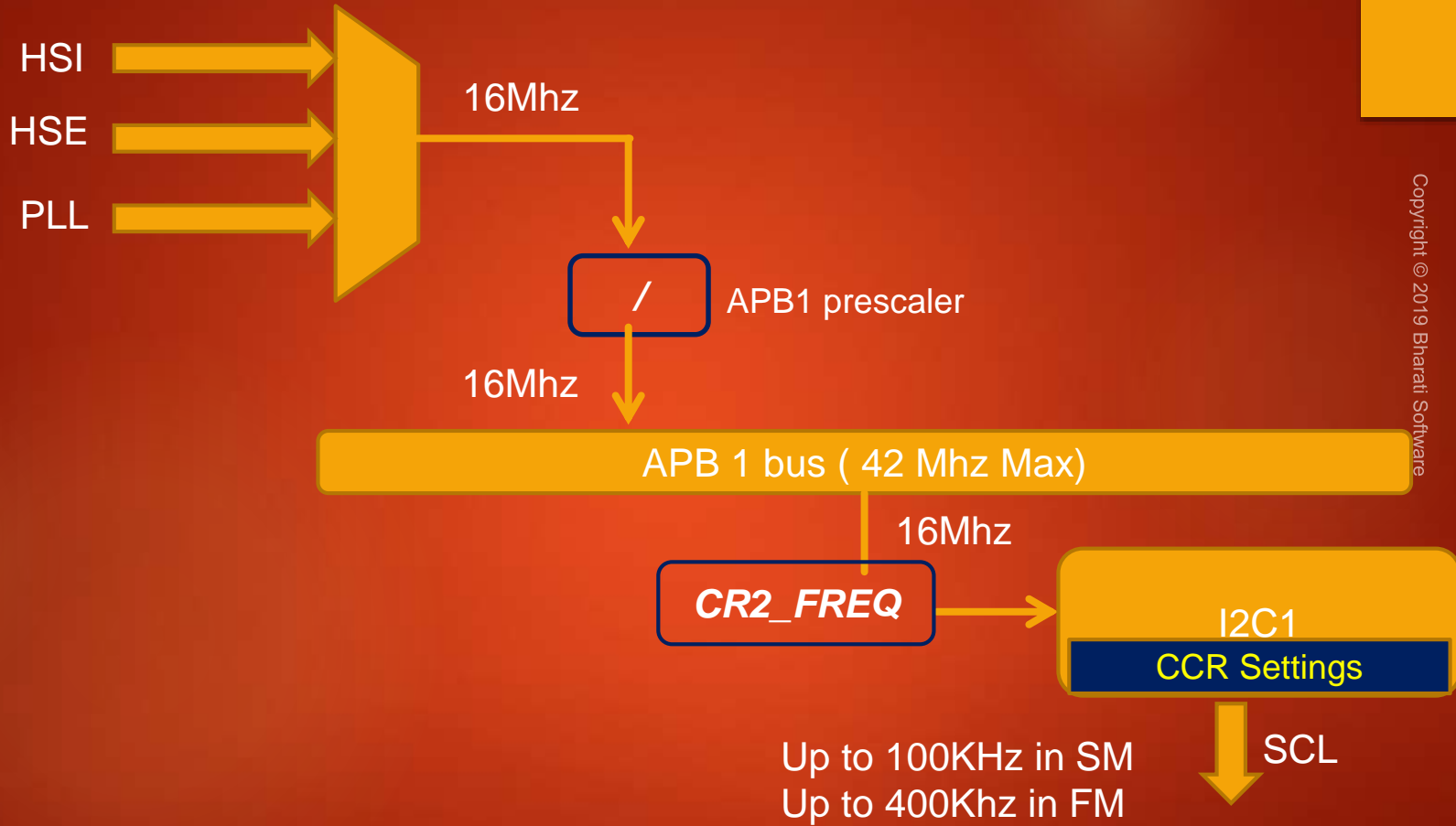
Copyright ©

~~3~~

6	5	4	3	2	1	0
es.	FREQ[5:0]					
	rw	rw	rw	rw	rw	rw

Register (I2C_CCR)

12	11	10	9	8	7	6	5	4	3	2	1	0
CCR[11:0]												
rw												



Example :

In SM Mode , generate a 100 kHz SCL frequency
APB1 Clock (PCLK1) = 16MHz

$$\begin{aligned} \text{CCR} &= \text{PCLK1} / (2 * 100\,000) \\ \text{CCR} &= 80 \end{aligned}$$

- 1) Configure the mode in CCR register (15th bit)
- 2) Program FREQ field of CR2 with the value of PCLK1
- 3) Calculate and Program CCR value in CCR field of CCR register

$$\begin{aligned} T_{\text{high(scl)}} &= \text{CCR} * T_{\text{PCLK1}} \\ T_{\text{low(scl)}} &= \text{CCR} * T_{\text{PCLK1}} \end{aligned}$$

Example :

In FM Mode , generate a 200 kHz SCL frequency
APB1 Clock (PCLK1) = 16MHz

$$\text{CCR} = \text{PCLK1} / (3 * 100\,000)$$
$$\text{CCR} = 80$$

- 1) Configure the mode in CCR register (15th bit)
- 2) Select the duty cycle of Fast mode SCL in CCR register (14th bit)
- 3) Program FREQ field of CR2 with the value of PCLK1
- 4) Calculate and Program CCR value in CCR field of CCR register

If DUTY = 0:

$$T_{\text{high}} = \text{CCR} * \text{TPCLK1}$$

$$T_{\text{low}} = 2 * \text{CCR} * \text{TPCLK1}$$

If DUTY = 1: (to reach 400 kHz)

$$T_{\text{high}} = 9 * \text{CCR} * \text{TPCLK1}$$

$$T_{\text{low}} = 16 * \text{CCR} * \text{TPCLK1}$$

I2C Duty Cycle

SM



FM



In STM32f4x I2C

For SM

(T_{low} may be equal to T_{high})

For FM

$$T_{\text{low}} = 2T_{\text{high}}$$

Or

$$T_{\text{low}} = 1.8 T_{\text{high}}$$

I2C Duty Cycle

$$T_{\text{high(scl)}} = \text{CCR} * T_{\text{PCLK1}}$$
$$T_{\text{low(scl)}} = \text{CCR} * T_{\text{PCLK1}}$$



I2C Duty Cycle

If DUTY = 0:

$$T_{\text{high}} = \text{CCR} * \text{TPCLK1}$$

$$T_{\text{low}} = 2 * \text{CCR} * \text{TPCLK1}$$

If DUTY = 1: (to reach 400 kHz)

$$T_{\text{high}} = 9 * \text{CCR} * \text{TPCLK1}$$

$$T_{\text{low}} = 16 * \text{CCR} * \text{TPCLK1}$$





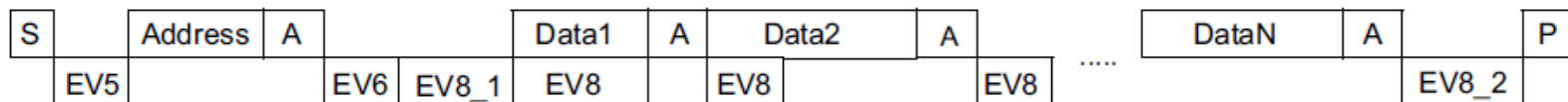
Implementation

I2C_MasterSendData API

Master sending data to slave

Transfer sequence diagram for master transmitter

7-bit master transmitter



Legend: S = Start, SR = Repeated start, P = stop, A = Acknowledge

EVx = Event (with interrupt if ITEVFEN = 1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register with address.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2.

EV8_1: TxE=1, shift register empty, data register empty, write Data1 in DR.

EV8: TxE=1, shift register not empty, data register empty, cleared by writing DR register.

EV_2: TxE=1, BTF=1, Program stop request, TxE and BTF are cleared by hardware by the stop condition.

1. The EV5, EV6, EV9, EV8_1 and EV8_2 events stretch SCL low until the end of the corresponding software sequence.
2. The EV8 event stretches SCL low if the software sequence is not complete before the end of the next byte transmission.

```
void I2C_MasterSendData(I2C_Handle_t *pI2CHandle, uint8_t *pTxBuffer, uint8_t Len, uint8_t SlaveAddr, uint8_t Sr)
{

    //1. Generate the START condition

    //2. confirm that start generation is completed by checking the SB flag in the SR1
    //    Note: Until SB is cleared SCL will be stretched (pulled to LOW)

    //3. Send the address of the slave with r/nw bit set to w(0) (total 8 bits )

    //4. Confirm that address phase is completed by checking the ADDR flag in teh SR1

    //5. clear the ADDR flag according to its software sequence
    //    Note: Until ADDR is cleared SCL will be stretched (pulled to LOW)

    //6. send the data until Len becomes 0

    //7. when Len becomes zero wait for TXE=1 and BTF=1 before generating the STOP condition
    //    Note: TXE=1 , BTF=1 , means that both SR and DR are empty and next transmission should begin
    //    when BTF=1 SCL will be stretched (pulled to LOW)

    //8. Generate STOP condition and master need not to wait for the completion of stop condition.
    //    Note: generating STOP, automatically clears the BTF

}
```

Exercise :

I2C Master(STM) and I2C Slave(Arduino) communication .

When button on the master is pressed , master should send data to the Arduino slave connected. The data received by the Arduino will be displayed on the Arduino serial port.

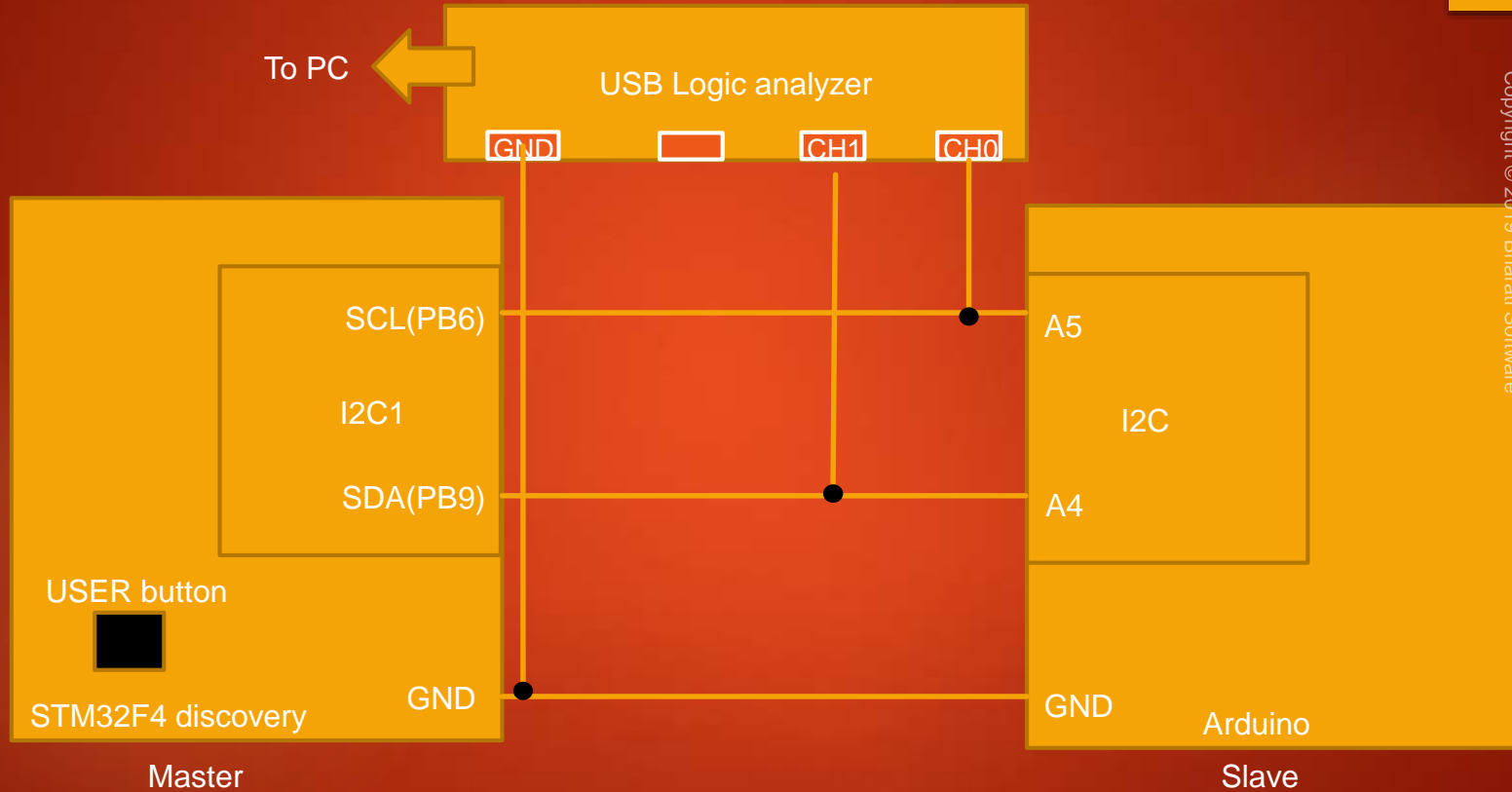
- 1 . Use I2C SCL = 100KHz(Standard mode)
2. Use internal pull resistors for SDA and SCL lines

Things you need

1. Arduino board
2. ST board
3. Some jumper wires
4. Bread board
5. 2 Pull up resistors of value $4.4K\ \Omega$ (only if your pin doesn't support internal pull up resistors)



STEP-1 Connect Arduino and ST board SPI pins as shown



STEP-2

Power your Arduino board and download I2C Slave sketch to Arduino

Sketch name : 001I2CSlaveRxString.ino

Find out the GPIO pins over which I2C1 can communicate

I2C pull up resistance , Rise time and Bus capacitance discussion

Pull-up Resistor(R_p) Calculation

R_p (min) is a function of V_{CC} , V_{OL} (max), and I_{OL} :

$$R_p(\text{min}) = \frac{(V_{CC} - V_{OL}(\text{max}))}{I_{OL}}$$

V_{OL} = LOW-level output voltage

I_{OL} = LOW-level output current

t_r = rise time of both SDA and SCL signals

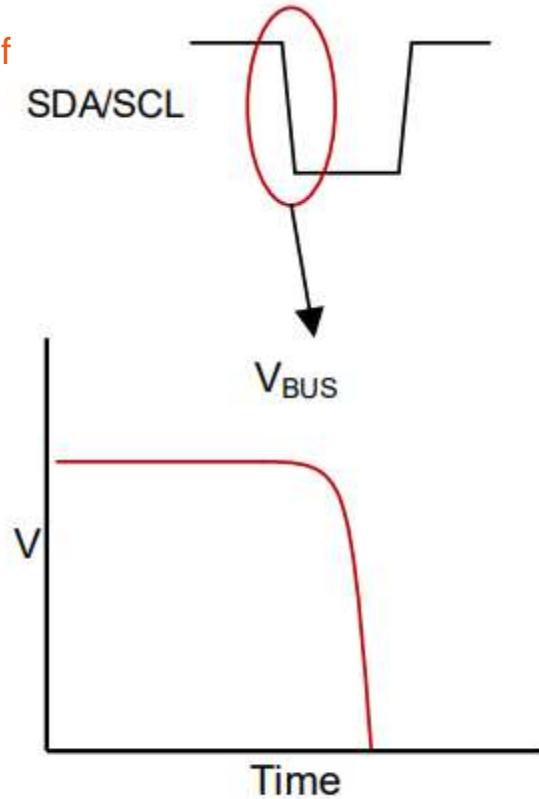
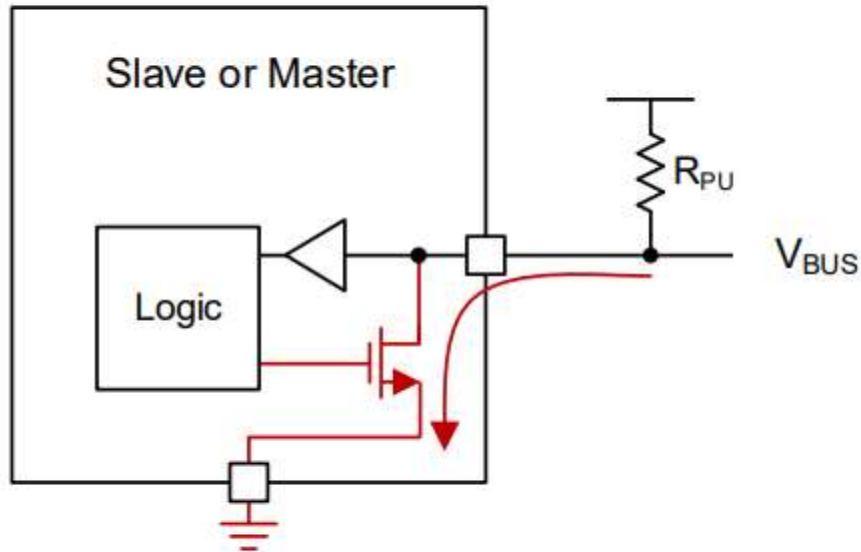
C_b = capacitive load for each bus line

Copyright © 2019 Bharati Software

The maximum pullup resistance is a function of the maximum rise time (t_r):

$$R_p(\text{max}) = \frac{t_r}{(0.8473 \times C_b)}$$

Image taken from : <http://www.ti.com/lit/an/slva704/slva704.pdf>



Pulling the Bus Low With An Open-Drain Interface

Rise (t_r) and Fall (t_f) Times



t_r is defined as the amount of time taken by the rising edge to reach 70% amplitude from 30% amplitude for either SDA and SCL

Copyright © 2019 Bharati Software

I2C spec cares about t_r value and you have to respect it while calculating the pull up resistor value.

Higher value of pull up resistors(weak pull-ups) increases t_r value. (not acceptable if t_r crosses max. limit mentioned in the spec)

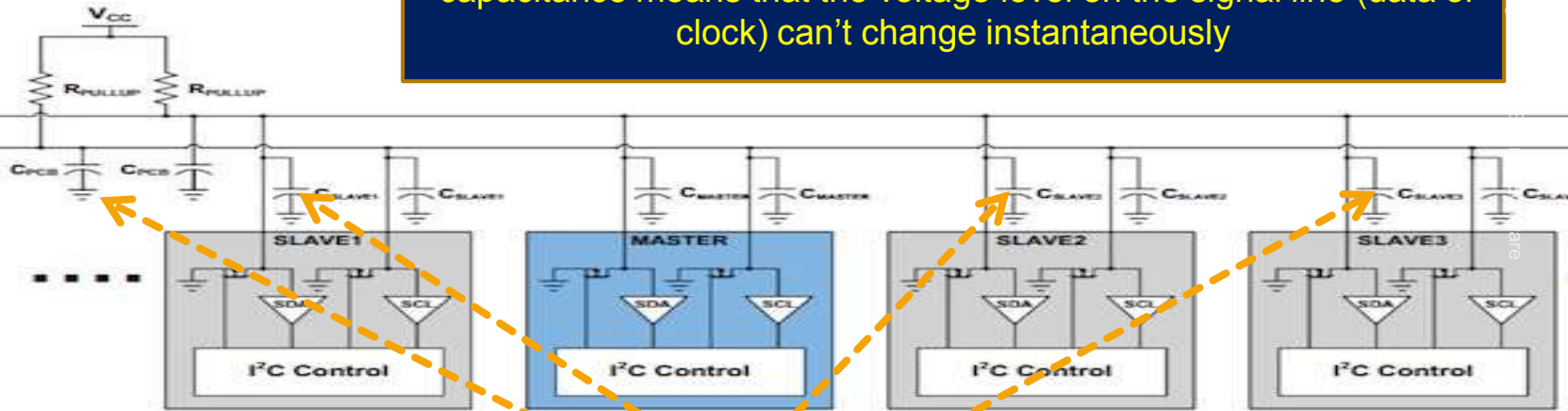
Lower value of pull up resistors (strong pull-ups) decreases t_r value (good) but they also lead higher current consumption (bad)

Using very high value of pull up resistors may cause issues like this where the pin may not able to cross the V_{IH} limit (Input Level High Voltage)



I2C Bus Capacitance(C_b)

capacitance means that the voltage level on the signal line (data or clock) can't change instantaneously



accidental capacitors

I2C Bus Capacitance(C_b)

- ✓ Bus capacitance is a collection of individual pin capacitance wrt gnd , capacitance between the sda and scl , parasitic capacitance, capacitance added by the devices hanging on the bus , bus length (wire) , dielectric material etc.
- ✓ Bus capacitance limits how long your i2c wiring can be and how many devices you can connect on the bus.
- ✓ For maximum allowed bus capacitance check the spec.

Exercise

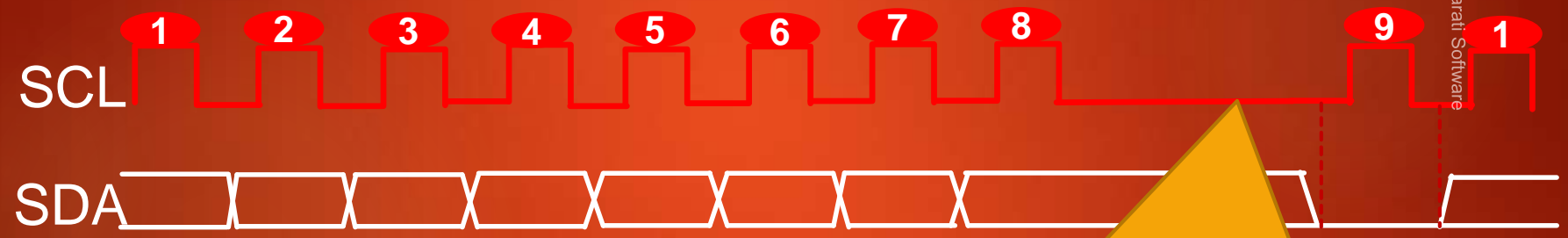
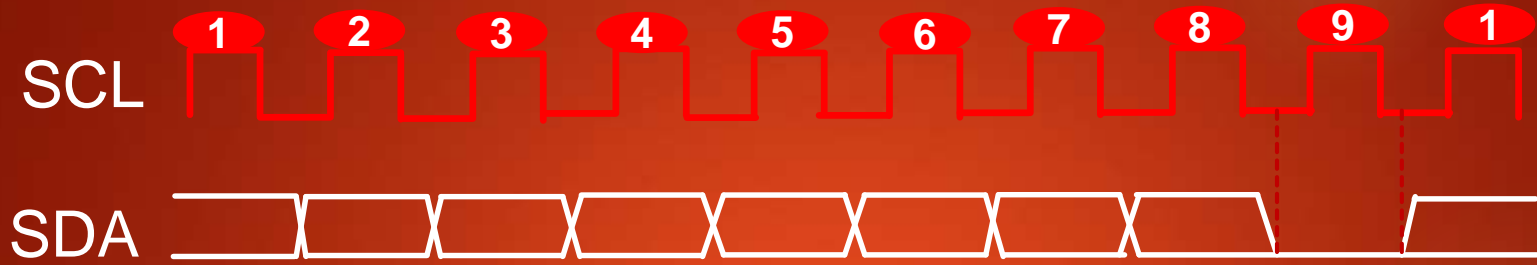
For Fast-mode I2C communication with the following parameters, calculate the pullup resistor value. $C_b = 150 \text{ pF}$, $V_{CC} = 3.3 \text{ V}$

TRISE calculation

Clock Stretching

Clock Stretching

- ▶ Clock stretching pauses a transaction by holding the SCL line LOW.
- ▶ The transaction cannot continue until the line is released HIGH again. Clock stretching is optional and in fact, most slave devices do not include an SCL driver so they are unable to stretch the clock.
- ▶ Slaves can hold the SCL line LOW after reception and acknowledgment of a byte to force the master into a wait state until the slave is ready for the next byte transfer in a type of handshake procedure
- ▶ If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.



The slave is not ready for more data, so it buys time by holding the clock low.
The master will wait for the clock line to be released before proceeding to the next frame

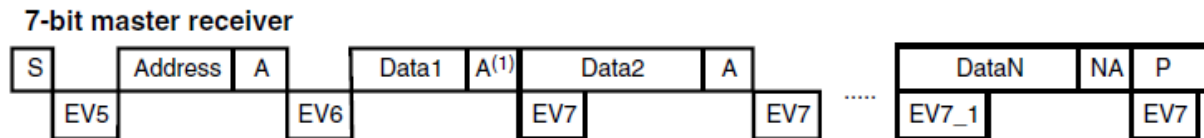


Implementation

I2C_MasterReceiveData API

Master Receiving data from slave

Transfer sequence diagram for master receiver



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.

In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.

EV7: RxNE=1 cleared by reading DR register.

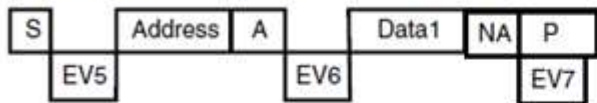
EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

1. If a single byte is received, it is NA.
2. The EV5, EV6 and EV9 events stretch SCL low until the end of the corresponding software sequence.
3. The EV7 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
4. The EV7_1 software sequence must be completed before the ACK pulse of the current byte transfer.

Master Receiving 1 byte from slave

Transfer sequence diagram for master receiver

7-bit master receiver



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.

In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.

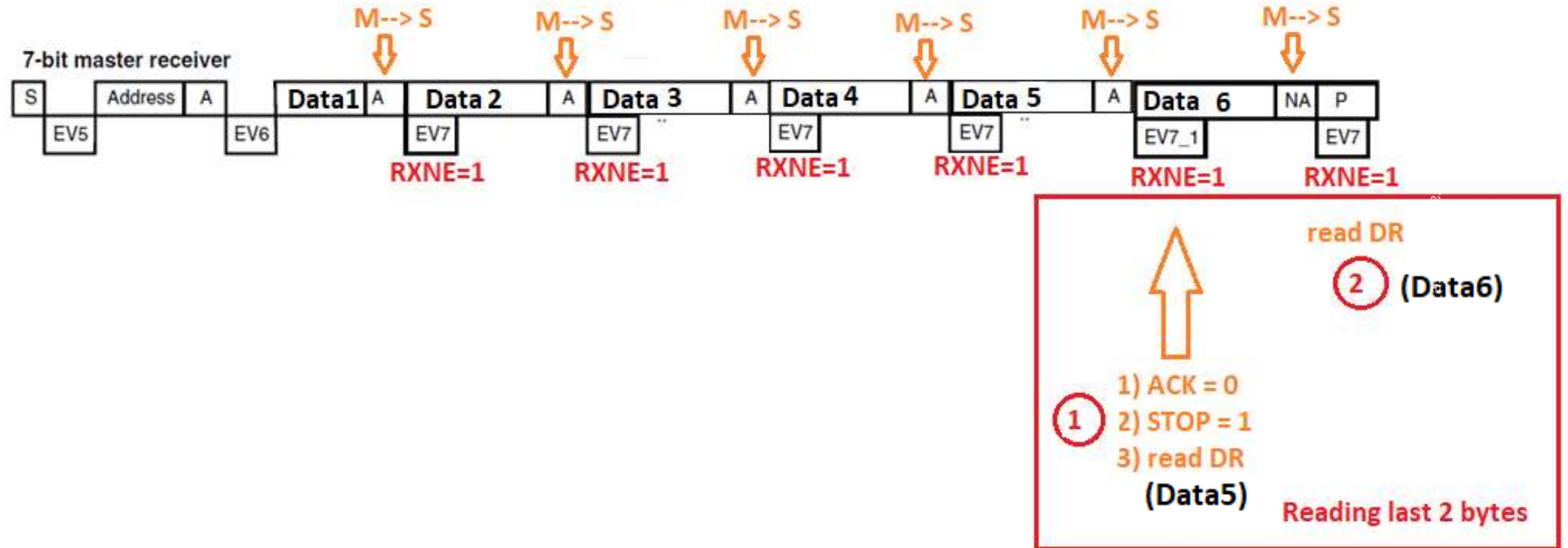
EV7: RxNE=1 cleared by reading DR register.

EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

1. If a single byte is received, it is NA.
2. The EV5, EV6 and EV9 events stretch SCL low until the end of the corresponding software sequence.
3. The EV7 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
4. The EV7_1 software sequence must be completed before the ACK pulse of the current byte transfer.

Master Receiving more than 1 byte

Transfer sequence diagram for master receiver when Len > 1



Exercise :

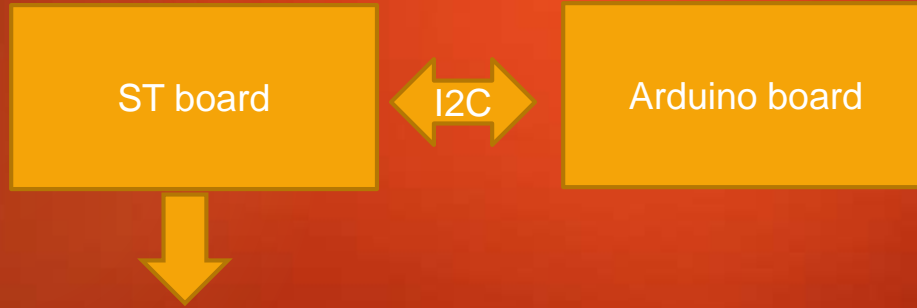
I2C Master(STM) and I2C Slave(Arduino) communication .

When button on the master is pressed , master should read and display data from Arduino Slave connected. First master has to get the length of the data from the slave to read subsequent data from the slave.

- 1 . Use I2C SCL = 100KHz(Standard mode)
2. Use internal pull resistors for SDA and SCL lines

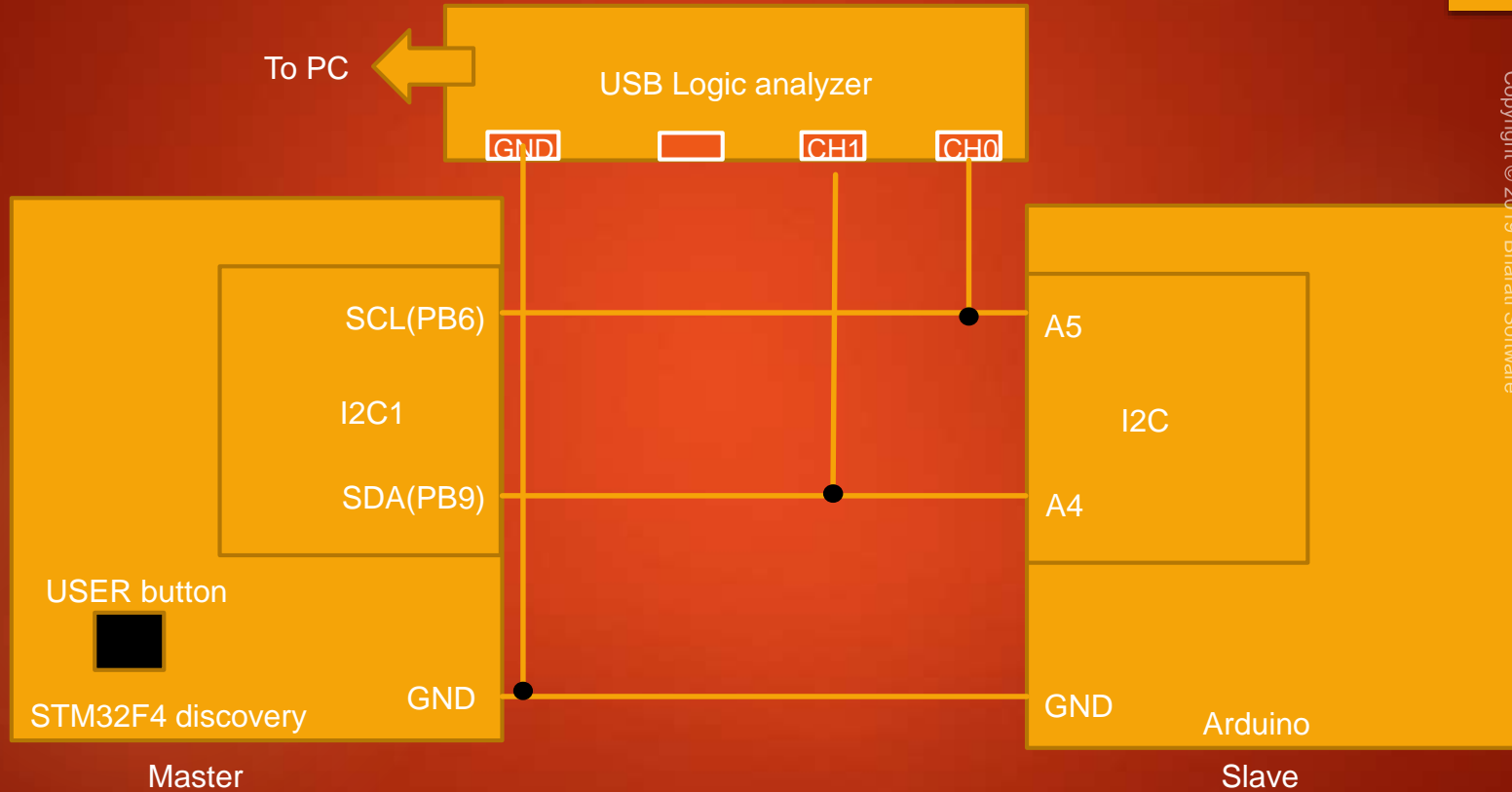
Things you need

1. Arduino board
2. ST board
3. Some jumper wires
4. Bread board
5. 2 Pull up resistors of value $4.7K \Omega$ (only if your pin doesn't support internal pull up resistors)



Print received data using semihosting

STEP-1 Connect Arduino and ST board SPI pins as shown



STEP-2

Power your Arduino board and download I2C Slave sketch to Arduino

Sketch name : 002I2CSlaveTxString.ino

Procedure to read the data from Arduino Slave

1

Master sends command code 0x51 to read the length(1 byte) of the data from the slave

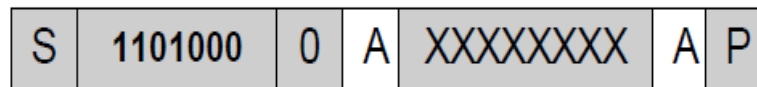
2

Master sends command code 0x52 to read the complete data from the slave

I2C transactions to read the 1 byte
length information from slave

Data Write— Master sending command to slave

<Slave Address> $\overline{\text{RW}}$ < command code >



S - Start

A - Acknowledge (ACK)

P - Stop

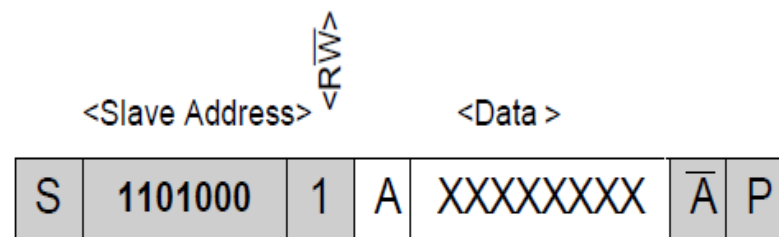


Master to slave



Slave to master

Data Read— Master reading response from the slave





S - Start

A - Acknowledge (ACK)

P - Stop

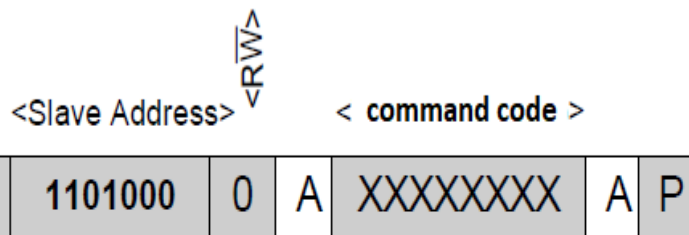
\overline{A} - Not Acknowledge (NACK)

 Master to slave

 Slave to master

I2C transactions to read “length” bytes
of data from the slave

Data Write— Master sending command to slave



S - Start

A - Acknowledge (ACK)

P - Stop

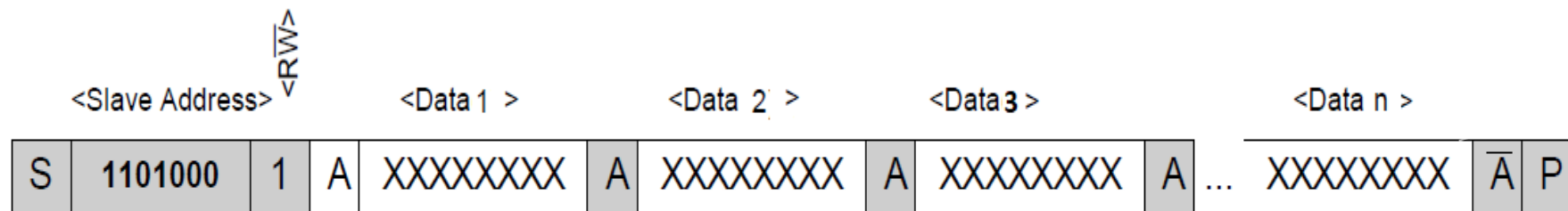


Master to slave



Slave to master

Data Read— Master reading response from the slave



S - Start

A - Acknowledge (ACK)

P - Stop

\bar{A} - Not Acknowledge (NACK)



Master to slave



Slave to master

START

Reset both the boards

Wait for button press on the master


Master first sends command code 0x51 to
slave (to read length)

Master reads "length" from slave
(1 byte)

Master reads "length" number of bytes from
the slave

Master display the data received using printf
(semihosting)

END



I2C Functional block and Peripheral Clock

f_{pclk} (Peripheral Clock Frequency) must be at least 2MHz to achieve standard mode I2c frequencies that are up to 100khz

f_{pclk} must be at least 4Mhz to achieve FM mode i2c frequencies. that is above 100khz but below 400Khz

f_{pclk} must be a multiple of 10Mhz to reach the 400khz max i2c fm mode clock frequency

I2C Interrupts

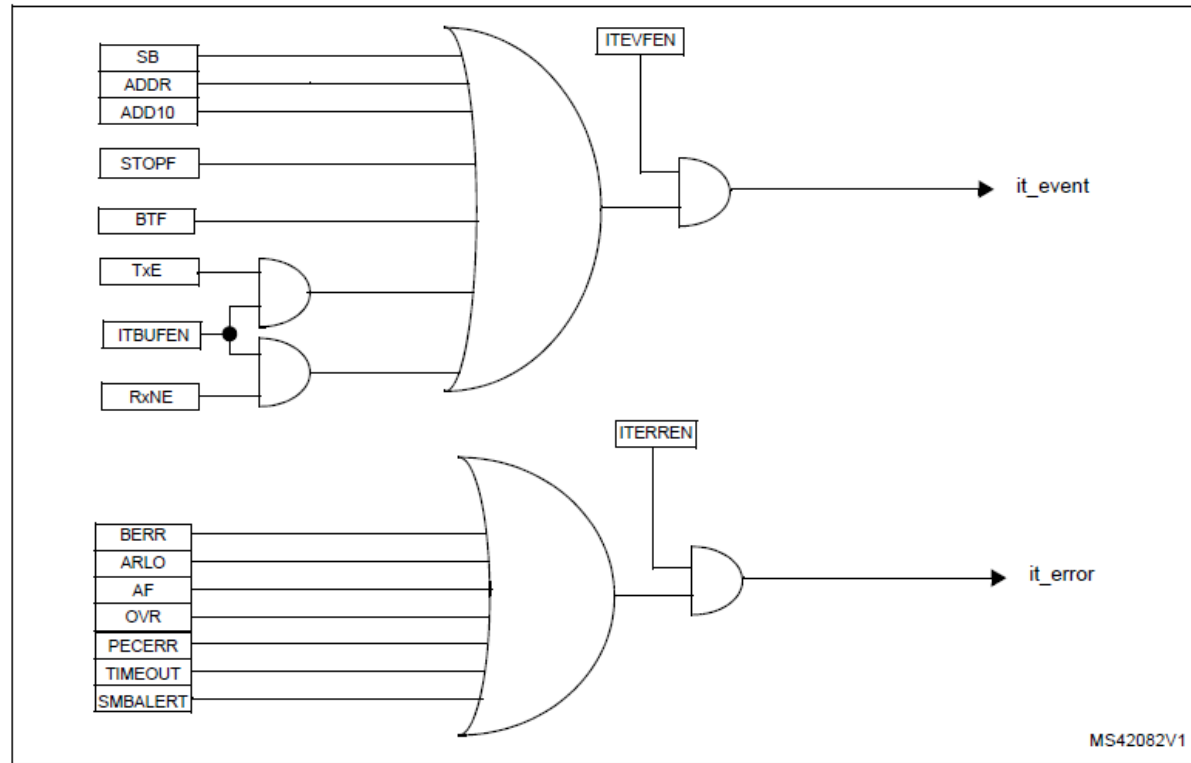


I2C IRQs and Interrupt mapping

I²C Interrupt requests

Interrupt event	Event flag	Enable control bit
Start bit sent (Master)	SB	ITEVFEN
Address sent (Master) or Address matched (Slave)	ADDR	
10-bit header sent (Master)	ADD10	
Stop received (Slave)	STOPF	
Data byte transfer finished	BTF	
Receive buffer not empty	RxNE	ITEVFEN and ITBUFEN
Transmit buffer empty	TxE	
Bus error	BERR	ITERREN
Arbitration loss (Master)	ARLO	
Acknowledge failure	AF	
Overrun/Underrun	OVR	
PEC error	PECERR	
Timeout/Tlow error	TIMEOUT	
SMBus Alert	SMBALERT	

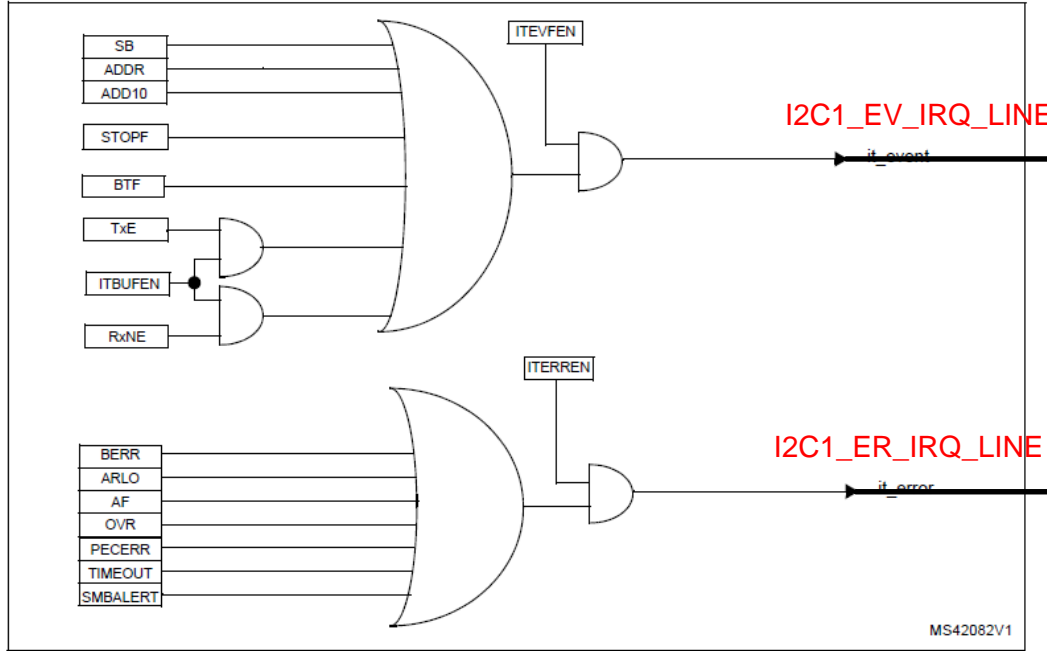
I²C interrupt mapping diagram



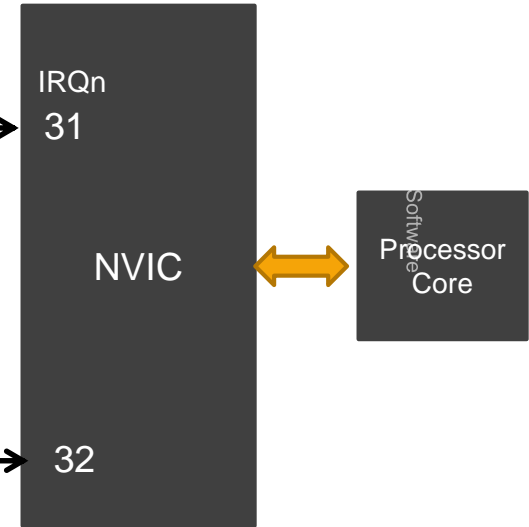
I2C1 interrupts

I²C interrupt mapping diagram

Peripheral

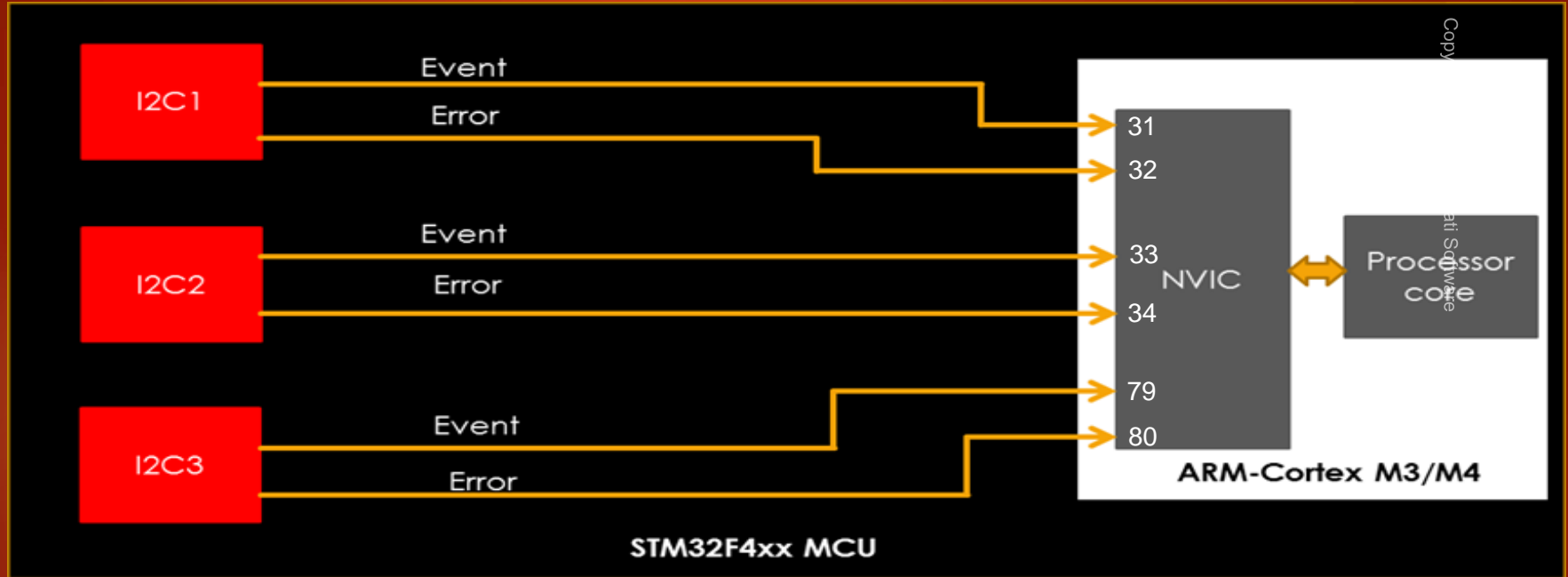


Processor



ARM Cortex M4

I2C Interrupting the Processor



IRQ numbers are specific to MCUs. Please check the vector table of your RM

Bus Error

This error happens when the interface detects an SDA rising or falling edge while SCL is high, occurring in a non-valid position during a byte transfer

Arbitration Loss Error

This error can happen when the interface loses the arbitration of the bus to another master

ACK Failure Error

This error happens when no ACK is returned for the byte sent

Overflow Error

Happens during reception, when a new byte is received and the data register has not been read yet and the New received byte is lost.

Under-run Error

Happens when In transmission when a new byte should be sent and the data register has not been written yet and the same byte is sent twice

PEC Error

Happens when there is CRC mismatch, if you have enabled the CRC feature

Time-Out Error

Happens when master or slave stretches the clock , by holding it low more than recommended amount of time.

BTF flag in TX and preventing underrun

During Txing of a data byte, if TXE=1, then that means data register is empty.

And if the firmware has not written any byte to data register before shift register becomes empty(previous byte transmission), then *BTF flag will be set and clock will be stretched to prevent the under run.*

BTF flag in RX and preventing overrun

If RXNE =1, Then it means new data is waiting in the data register, and if the firmware has not read the data byte yet before Shift register is filled with another new data, then also the BTF flag will be set and clock will be stretched to prevent the overrun.

I2C sending and receiving data in interrupt mode

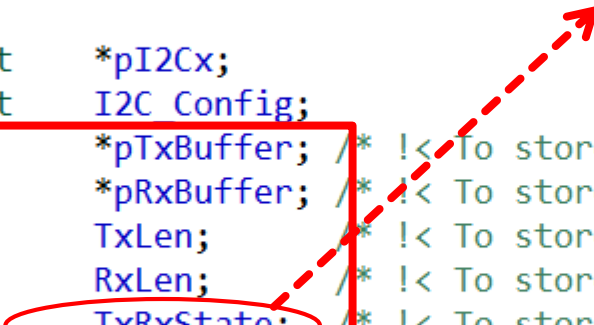


I2C_MasterSendDataIT API

I2C_MasterReceiveDataIT API

Modifying handle structure to store place holder variables

```
/*
 *Handle structure for I2Cx peripheral
 */
typedef struct
{
    I2C_RegDef_t      *pI2Cx;
    I2C_Config_t      I2C_Config;
    uint8_t            *pTxBuffer; /* !< To store the app. Tx buffer address > */
    uint8_t            *pRxBuffer; /* !< To store the app. Rx buffer address > */
    uint32_t            TxLen;      /* !< To store Tx len > */
    uint32_t            RxLen;      /* !< To store Rx len > */
    uint8_t            TxRxState;  /* !< To store Communication state > */
    uint8_t            DevAddr;    /* !< To store slave/device address > */
    uint32_t            RxSize;     /* !< To store Rx size > */
    uint8_t            Sr;         /* !< To store repeated start value > */
}I2C_Handle_t;
```



```
/*
 * I2C application states
 */
#define I2C_READY                0
#define I2C_BUSY_IN_RX          1
#define I2C_BUSY_IN_TX          2
```



Implementing I2C_MasterSendDataIT API



Implementing I2C_MasterReceiveDataIT API

Adding I2C IRQ number macros



Implementing
I2C_IRQInterruptConfig();
I2C_IRQPriorityConfig();

I2C ISR handling

ISR1

Interrupt handling for interrupts
generated by I2C events

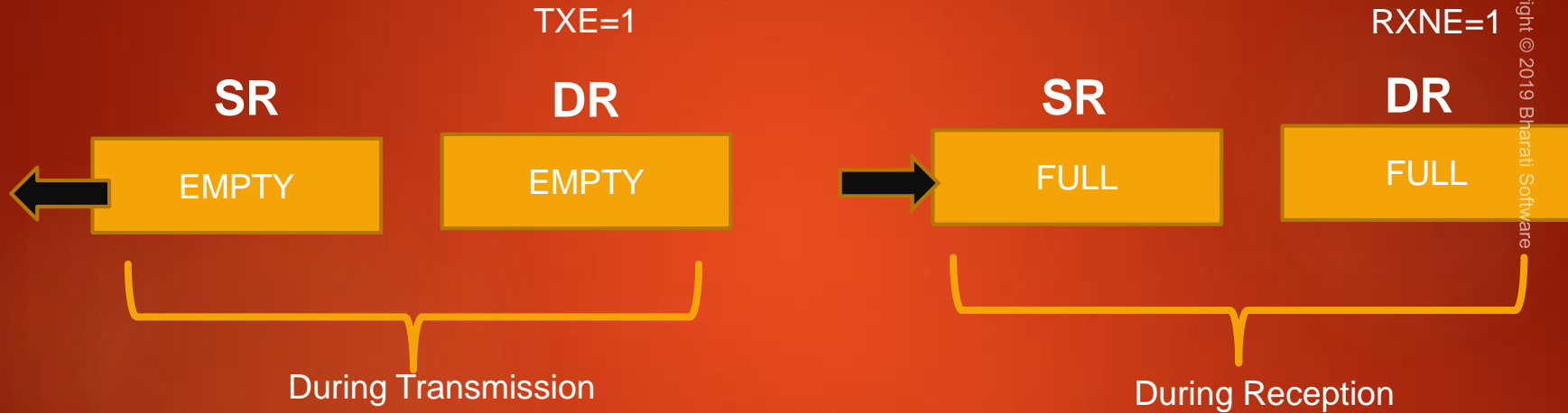
I2C_EV_IRQHandling

ISR2

Interrupt handling for interrupts
generated by I2C errors

I2C_ER_IRQHandling

When BTF flag is set



I2C Slave Programming







I2C Master

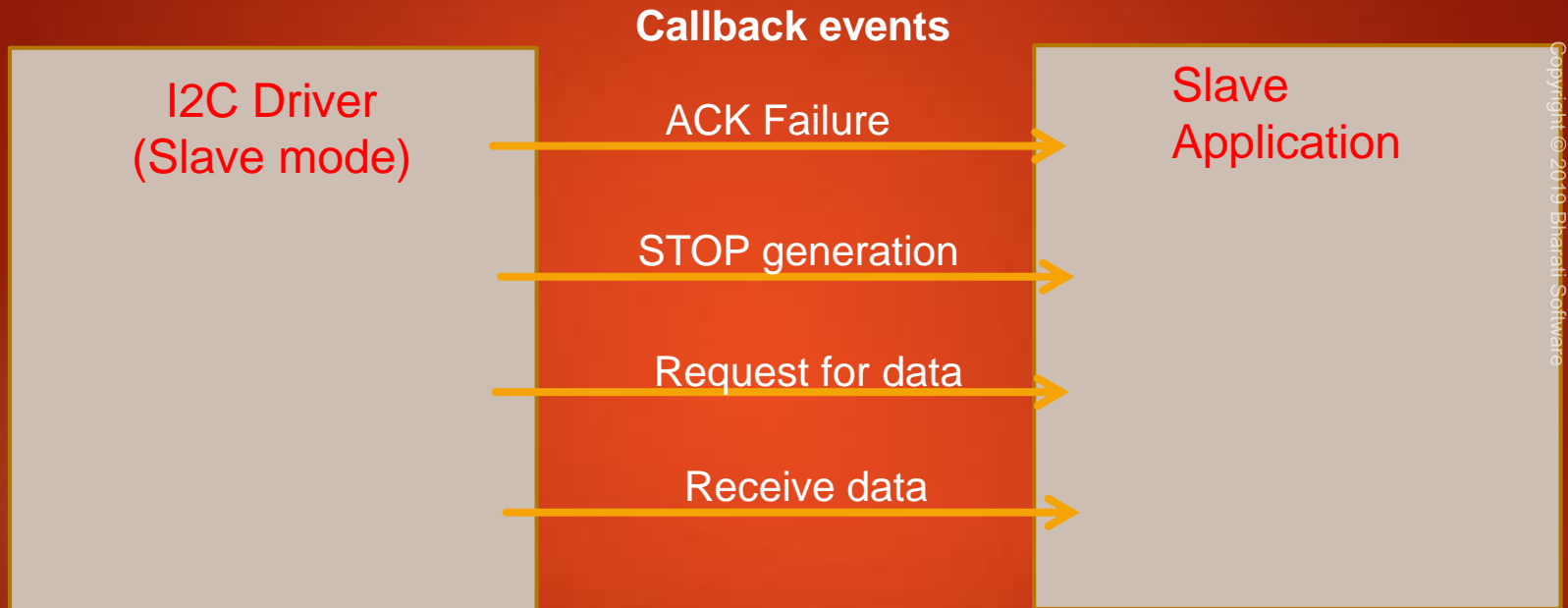
I2C

I2C Slave

STAR
T
STOP
Read
Write

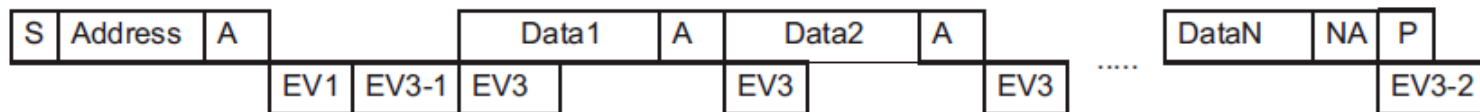
Request for data

Receive data



Transfer sequence diagram for slave transmitter

7-bit slave transmitter



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV1: ADDR=1, cleared by reading SR1 followed by reading SR2

EV3-1: TxE=1, shift register empty, data register empty, write Data1 in DR.

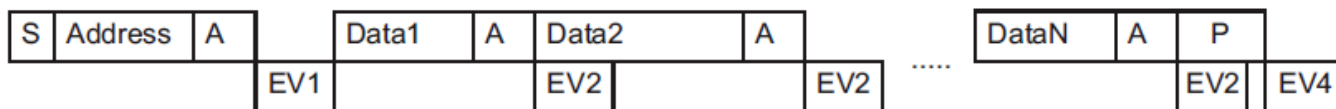
EV3: TxE=1, shift register not empty, data register empty, cleared by writing DR

EV3-2: AF=1; AF is cleared by writing '0' in AF bit of SR1 register.

1. The EV1 and EV3_1 events stretch SCL low until the end of the corresponding software sequence.
2. The EV3 event stretches SCL low if the software sequence is not completed before the end of the next byte transmission.

Transfer sequence diagram for slave receiver

7-bit slave receiver



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge,
EVx= Event (with interrupt if ITEVFEN=1)

EV1: ADDR=1, cleared by reading SR1 followed by reading SR2

EV2: RxNE=1 cleared by reading DR register.

EV4: STOPF=1, cleared by reading SR1 register followed by writing to the CR1 register

1. The EV1 event stretches SCL low until the end of the corresponding software sequence.
2. The EV2 event stretches SCL low if the software sequence is not completed before the end of the next byte reception.
3. After checking the SR1 register content, the user should perform the complete clearing sequence for each flag found set.
Thus, for ADDR and STOPF flags, the following sequence is required inside the I2C interrupt routine:
READ SR1
if (ADDR == 1) {READ SR1; READ SR2}
if (STOPF == 1) {READ SR1; WRITE CR1}
The purpose is to make sure that both ADDR and STOPF flags are cleared if both are found set.

Exercise :

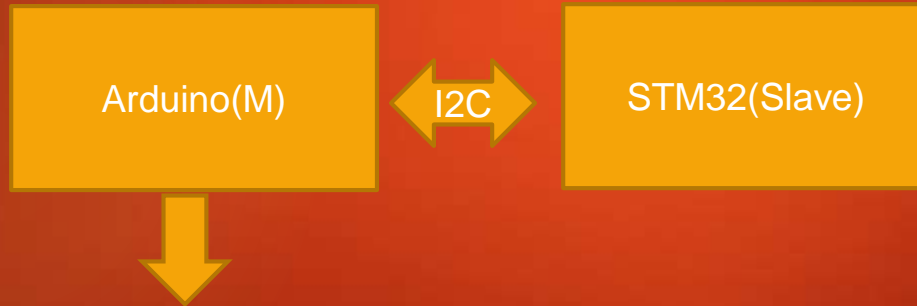
I2C Master(Arduino) and I2C Slave(STM32) communication .

Master should read and display data from STM32 Slave connected. First master has to get the length of the data from the slave to read subsequent data from the slave.

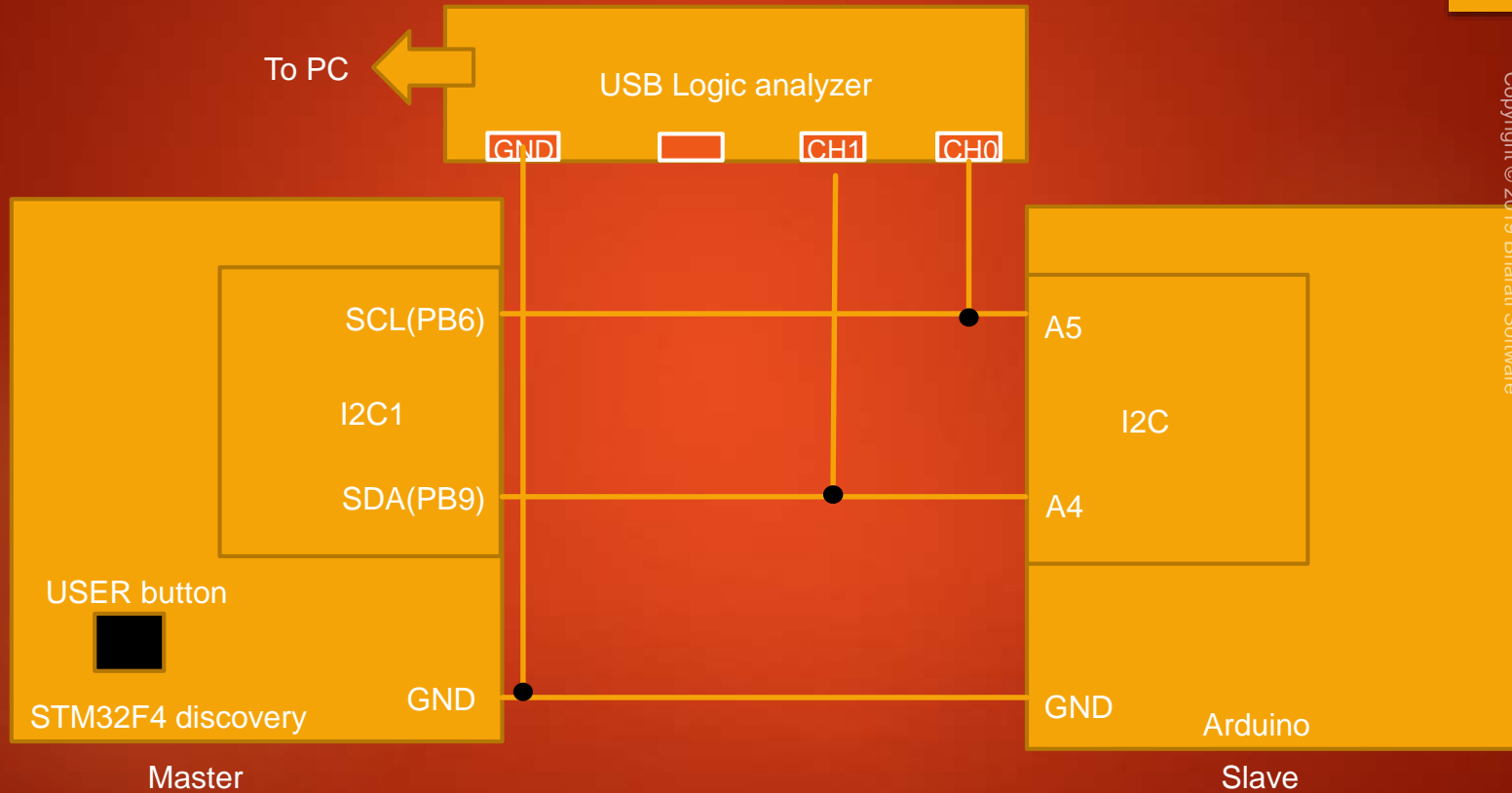
- 1 . Use I2C SCL = 100KHz(Standard mode)
2. Use internal pull resistors for SDA and SCL lines

Things you need

1. Arduino board
2. ST board
3. Some jumper wires
4. Bread board
5. 2 Pull up resistors of value $4.7K \Omega$ (only if your pin doesn't support internal pull up resistors)



STEP-1 Connect Arduino and ST board SPI pins as shown



STEP-2

Power your Arduino board and download I2C Slave sketch to Arduino

Sketch name: 003I2CMasterRxString.ino

Procedure to read the data from STM32 Slave

1

Master sends command code 0x51 to read the length(1 byte) of the data from the slave

2

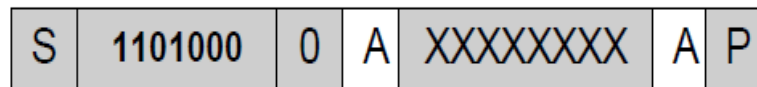
Master sends command code 0x52 to read the complete data from the slave

I2C transactions to read the 1 byte
length information from slave

Data Write— Master sending command to slave

<Slave Address> \overline{RW} < command code >

Master : Arduino



S - Start

A - Acknowledge (ACK)

P - Stop



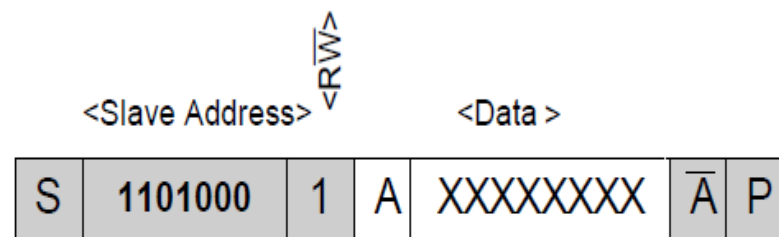
Master to slave



Slave to master

Data Read— Master reading response from the slave

Master : Arduino



S - Start

A - Acknowledge (ACK)

P - Stop

\bar{A} - Not Acknowledge (NACK)



Master to slave



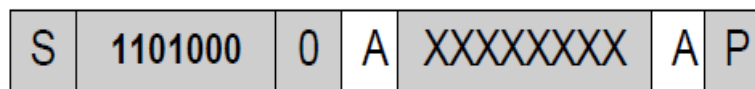
Slave to master

I2C transactions to read “length” bytes
of data from the slave

Data Write— Master sending command to slave

Master : Arduino

<Slave Address> \overleftarrow{RW} < command code >



S - Start

A - Acknowledge (ACK)

P - Stop



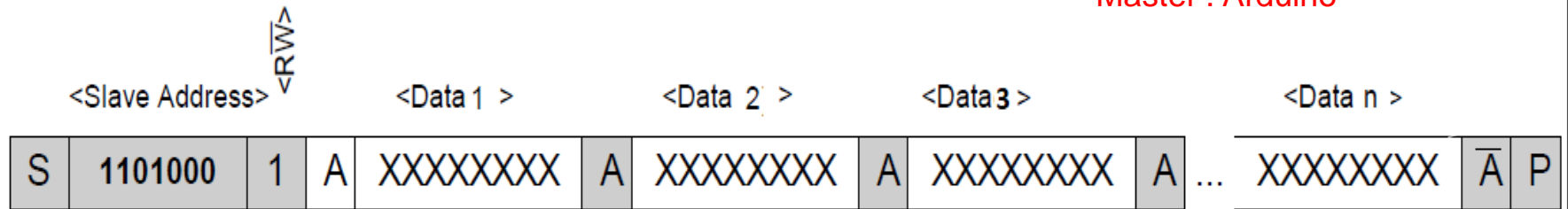
Master to slave



Slave to master

Data Read— Master reading response from the slave

Master : Arduino



S - Start

A - Acknowledge (ACK)

P - Stop

Ā - Not Acknowledge (NACK)



Master to slave

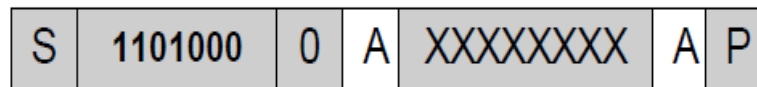


Slave to master

I2C transactions to read the 4 bytes of length information from the slave

Data Write— Master sending command to slave

<Slave Address> $\overline{\text{RW}}$ < command code >



S - Start

A - Acknowledge (ACK)

P - Stop



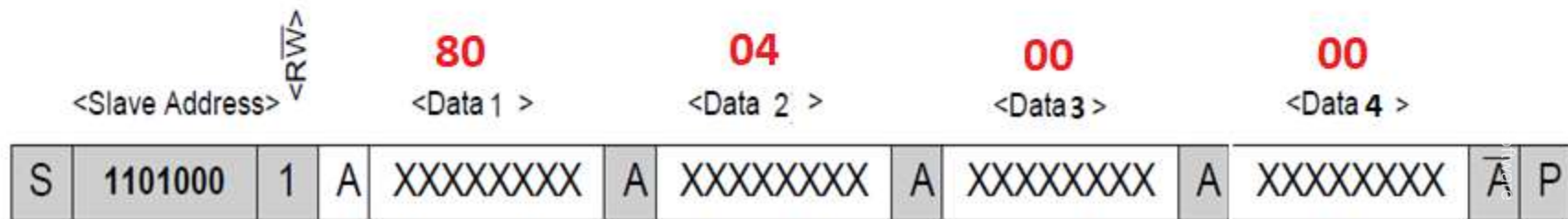
Master to slave



Slave to master

Data Read— Master reading response from the slave

0x00000480



S - Start

A - Acknowledge (ACK)

P - Stop

\bar{A} - Not Acknowledge (NACK)



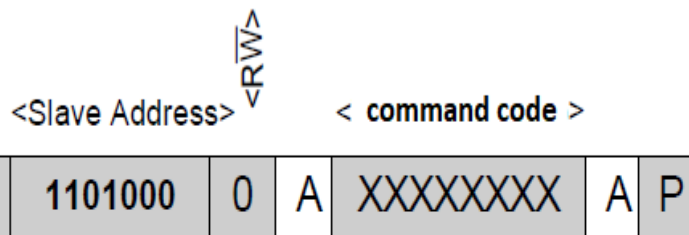
Master to slave



Slave to master

I2C transactions to read “length” bytes
of data from the slave

Data Write— Master sending command to slave



S - Start

A - Acknowledge (ACK)

P - Stop



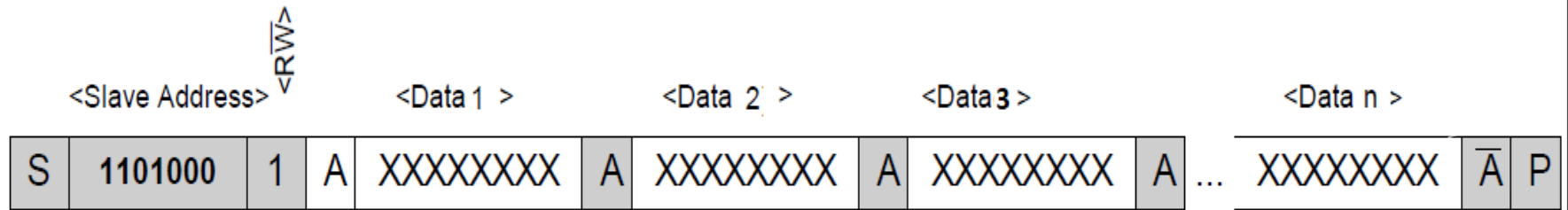
Master to slave



Slave to master

If length is ≤ 32 then only one "READ" Transaction

Data Read— Master reading response from the slave



S - Start

A - Acknowledge (ACK)

P - Stop

\bar{A} - Not Acknowledge (NACK)

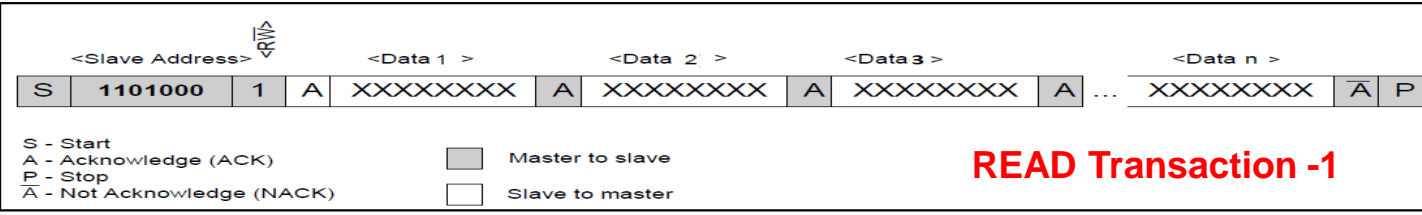


Master to slave

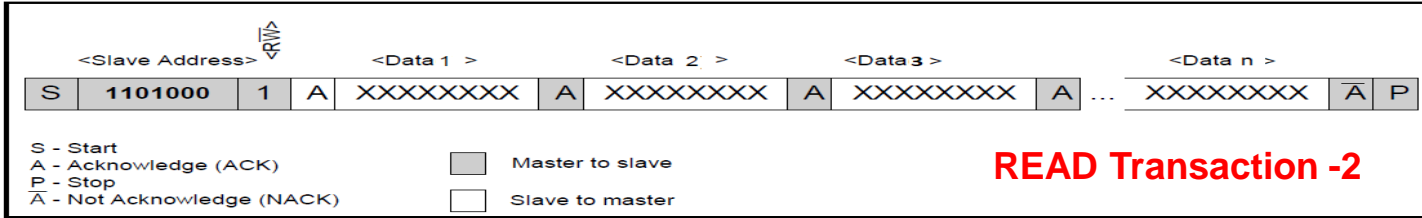


Slave to master

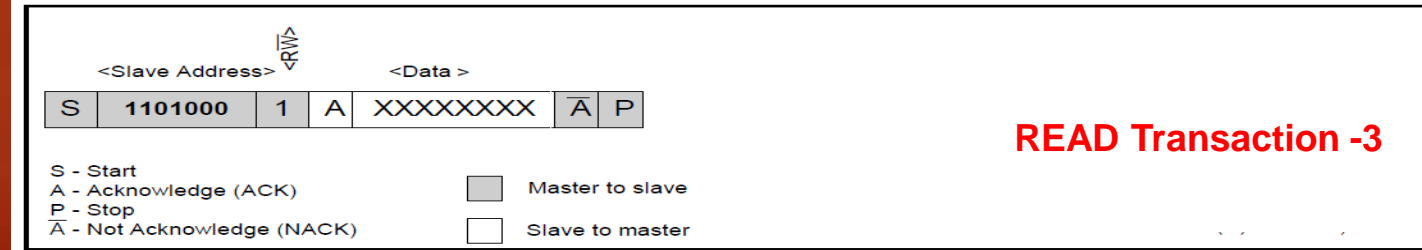
Data Read— Master reading response from the slave



Data Read— Master reading response from the slave



Data Read— Master reading response from the slave



Common Problems in I2C and Debugging Tips

Copyright © 2019 Bharati Software



GND

SDA & SCL



Tip

Whenever you face problem in I2C , probe the SDA and SCL line after I2C initialization. It must be held at HIGH level.

Problem-1: SDA and SCL line not held HIGH Voltage after I2C pin initialization

Reason-1:

Not activating the pullup resistors if you are using the internal pull up resistor of an I/O line

Debug Tip:

worth checking the configuration register of an I/O line to see whether the pullups are really activated or not, best way is to dump the register contents.

Problem-2: ACK failure

Reason-1:

Generating the address phase with wrong slave address

Debug Tip:

verify the slave address appearing on the SDA line by using logic analyser.

Problem-2: ACK failure

Reason-2:

Not enabling the ACKing feature in the I2C control register

Debug Tip:

Cross check the I2C Control register ACK enable field

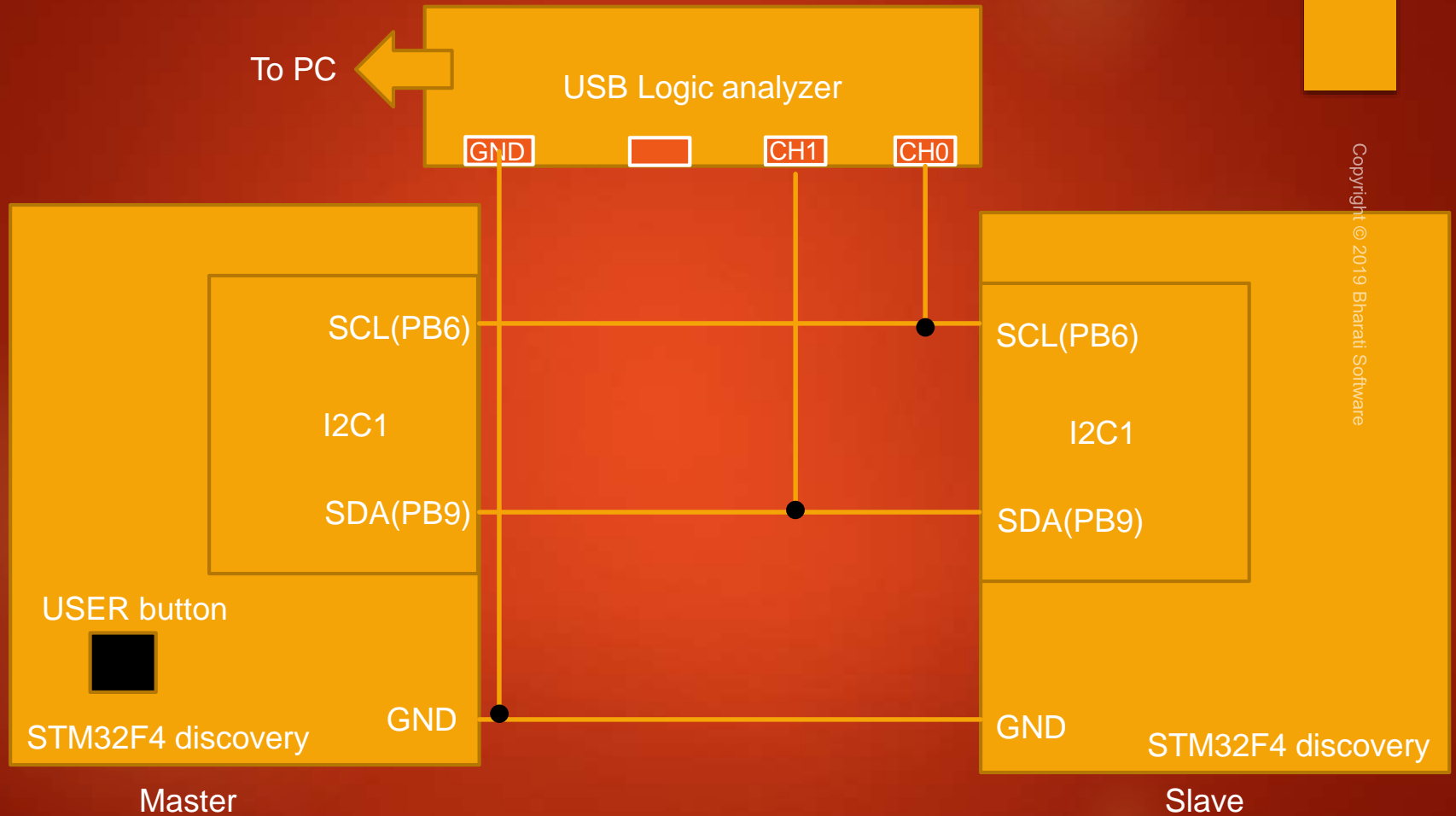
Problem-3: Master is not producing the clock

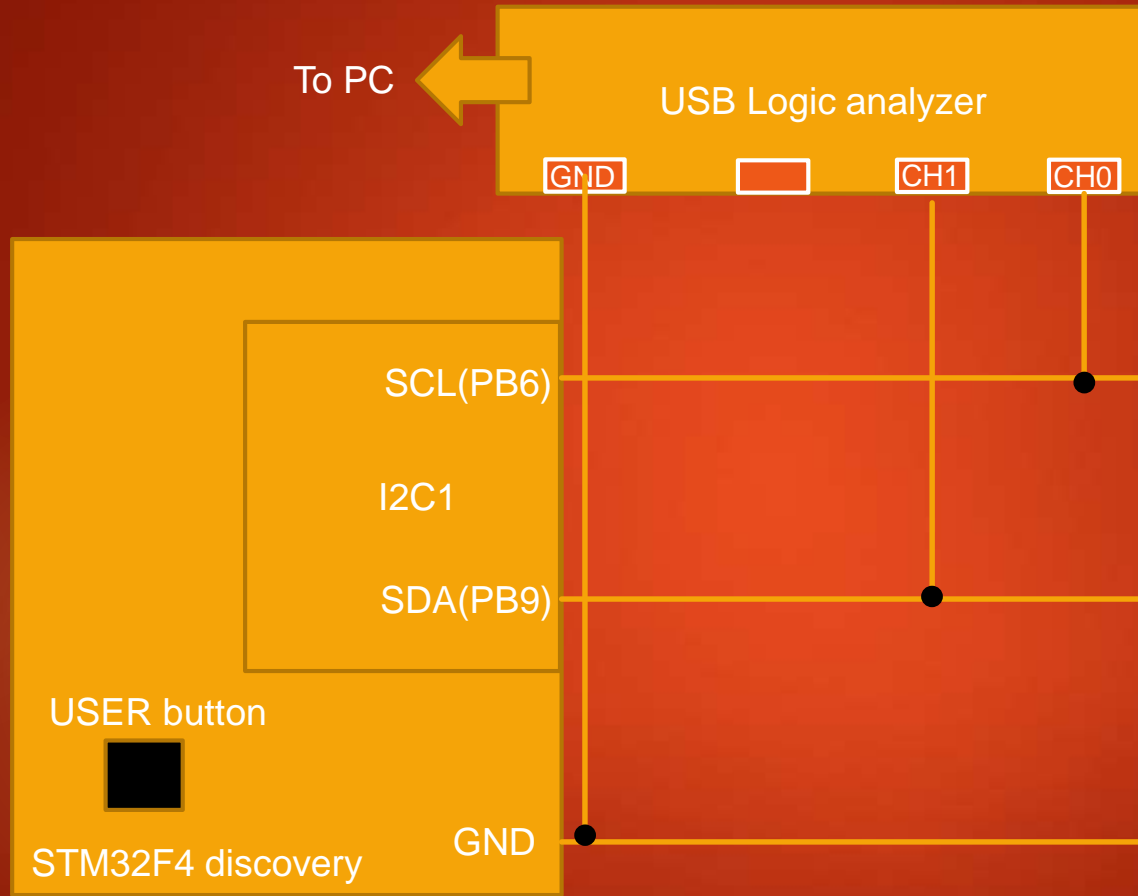
Debug Tip 1 :

First Check whether I2C peripheral clock is enabled and set to at least 2MHz to produce standard mode i2c serial clock frequency

Debug Tip 2 :

Check whether GPIOs which you used for SCL and SDA functionality are configured properly for the alternate functionality





Master

Slave

I2C Clock Stretching

Clock Stretching simply means that holding the clock to 0 or ground level.

The moment clock is held at low, then the whole I2C interface pauses until clock is given up to its normal operation level.

So, What is the use of clock stretching ?

I2C devices, either Master or Slave, uses this feature **to slow down the communication** by stretching SCL to low, **which prevents the clock to Rise high again** and the i2c communication stops for a while

There are situations where an **I2C slave is not able to co-operate with the clock speed** given by the master and needs to slow down a little.

If slave needs time, then it takes the advantage of clock stretching , and by holding clock at low, it momentary pauses the I2C operation.

Modifying I2C application to send/receive more than 32 bytes