1. Introduction

I2C as known as inter-integrated circuit is a bus interface used to communicate between devices. The communication speed of a standard I2C can serve up to 100kbits per second. In fast mode, the speed can serve up to 400kbits per second. However as the technology became advanced nowadays, the some of the I2C interface can provide the communication speed in high speed mode which up to 3.4Mbits per second.

I2C interface used two bus line to communicate. One of the bus line is called SDL (Serial Data line) which is used to transfer data while the other one is called SCL (Serial Clock Line) is used to transfer the clock pulse to trigger the data. In the communication with I2C interface, the devices are connected with the relationship between master and slave. There can be multiple master devices and multiple slave devices connected with only one I2C interface bus line, but only one master and one slave can communicate at the same time.

In master device, they can select one of the slave device to communicate if the device is connected in the same I2C interface bus line. Master devices can device whether send or receive data from a slave device. For a slave device, the interface are just waiting for the master to select them to get communication by matching the address sent from the master with their own address.

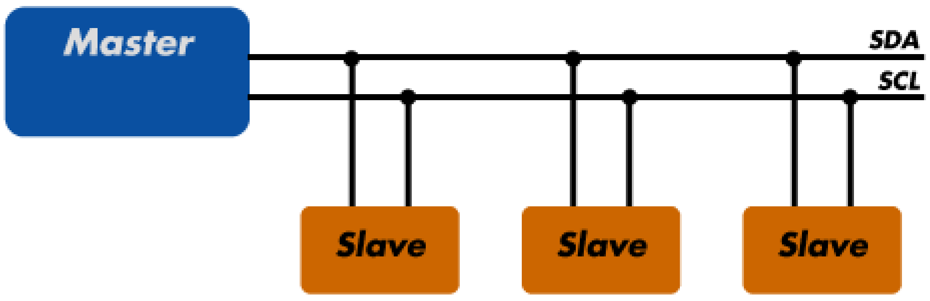


Figure 1.1

(Access from http://www.totalphase.com/support/articles/200349156-I2C-Background)

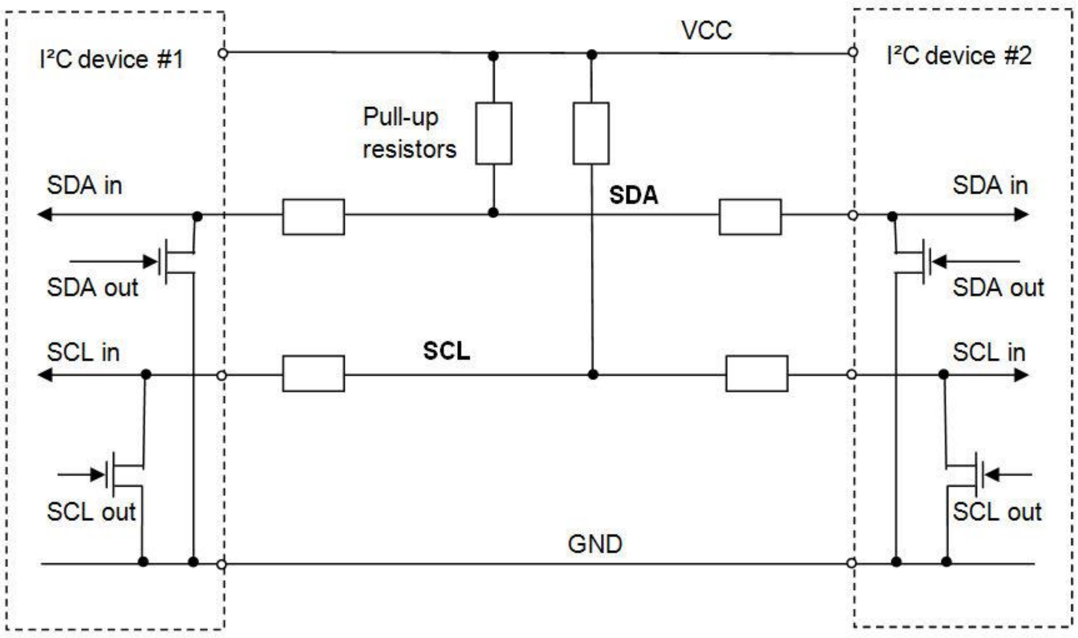


Figure 1.2

(Access from http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/)

Figure 1.2 shows the inner-circuit of the I2C interface. SDA and SCL are connected to the VCC source through the pull up resistor. The switch in the SDA and SCL will toggle to trigger the signal. In clock stretching, the slave device will closed the switch in SCL to pull the SCL signal to low in order to slow down the data from transmitting to prevent from overrun and underrun errors.

Protocol

There is 7 bit address and 10 bit address mode in I2C slave device. Normally 7 bit address mode was used while the 10 bit address mode is the enhanced version of 7 bit address mode, so that the master device able to connected with more slave devices in a single connection.

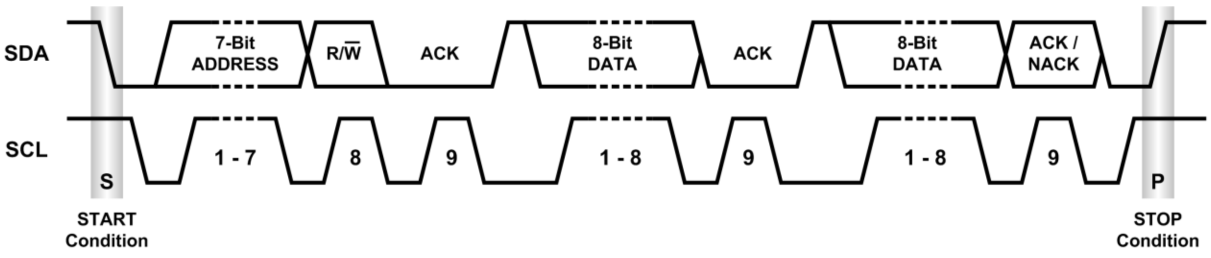


Figure 1.3

(Access from http://www.planetanalog.com/document.asp?doc_id=527900)

In 7 bit address mode, the master will generate a start bit by set the START bit in CR1 followed by the address byte. The address byte contains the 7 bit address and the LSB bit which the whether the master is going to transmit or receive data. The acknowledge bit was set to low by the slave device, the master will follow by send data or receive data from the slave.

When the master want to stop the communication the master should generate a stop condition by set the STOP bit to 1 if the master is transmitting data. While for master receiver mode, the master should give a NACK signal after receive the last data in order to send a stop condition to the slave.

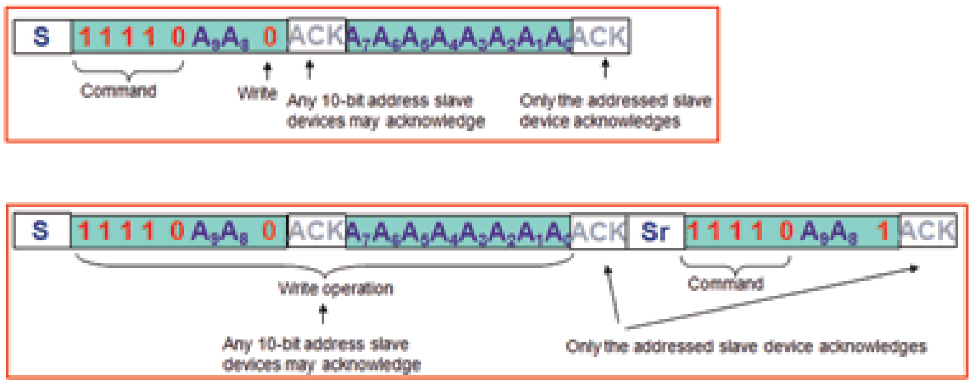

S
1 1 1 1 0 A9 A8 0
ACK
A7 A6 A5 A4 A3 A2 A1 A0
ACK
Command
Write
Any 10-bit address slave devices may acknowledge
Only the addressed slave device acknowledges
S
1 1 1 1 0 A9 A8 0
ACK
A7 A6 A5 A4 A3 A2 A1 A0
ACK
Sr
1 1 1 1 0 A9 A8 1
ACK
Write operation
Command
Any 10-bit address slave devices may acknowledge
Only the addressed slave device acknowledges

Figure 1.4

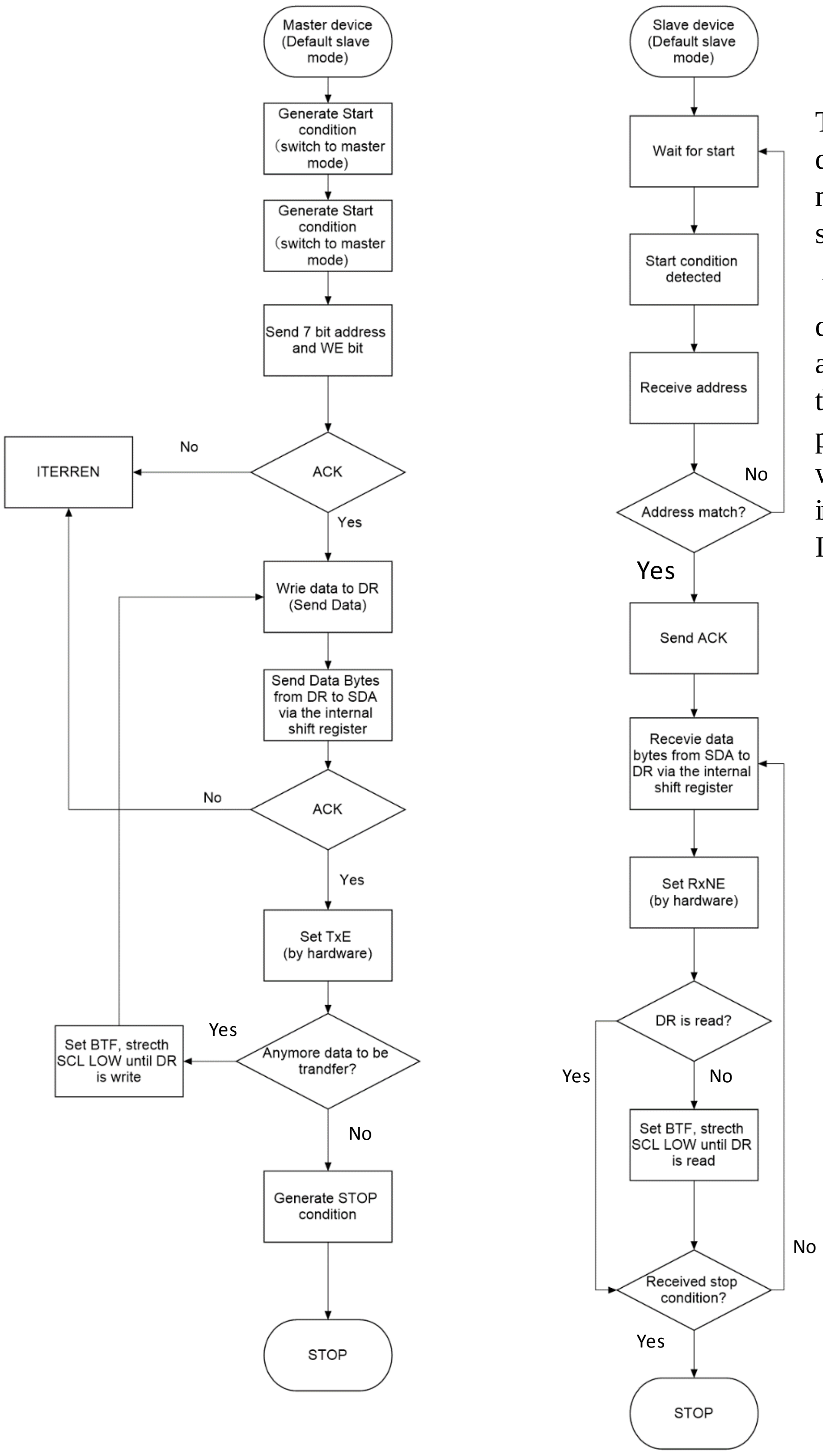
(Access from http://blog.testequipmentconnection.com/using-an-oscilloscope-to-debug-the-i2c-protocol)

S is the start condition, while Sr is the restart condition

In 10 bit mode, a header byte should be sent after the start bit was generated. The header bytes contains the header bits “11110XX0”, XX is the most two significant bits of the 10 bit address. Secondly, send the remaining 8 bit of the address. If the master decided to receive data from a slave device, the master required to repeat a start condition then send header bits “11110XX1” in order to change to receiver mode from transmitter mode.

2. Methodology

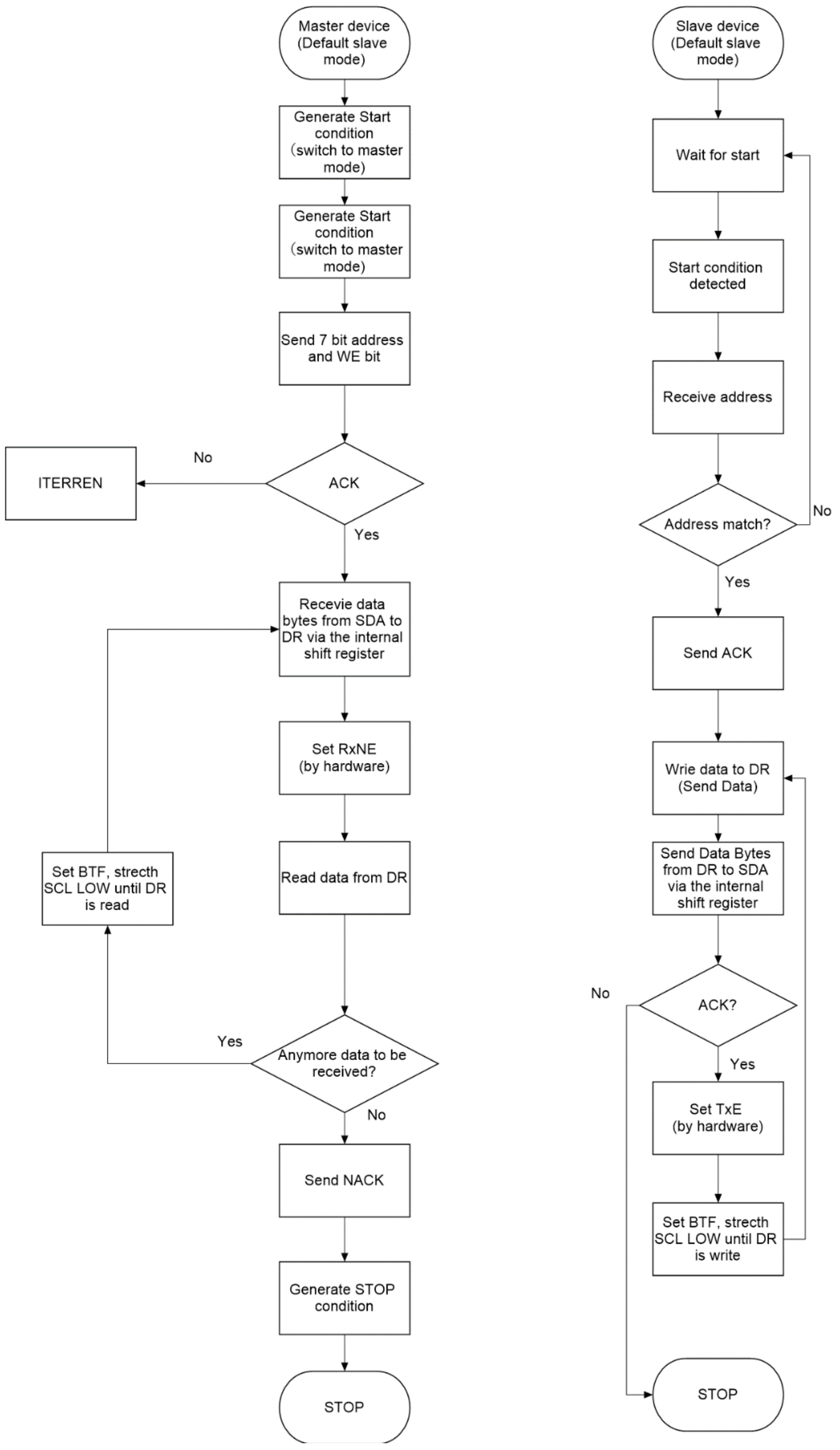
Flow chart



Master device (Default slave mode)
Generate Start condition (switch to master mode)
Generate Start condition (switch to master mode)
Send 7 bit address and WE bit
ACK
No
ITERREN
Yes
Wrie data to DR (Send Data)
Send Data Bytes from DR to SDA via the internal shift register
ACK
No
Yes
Set TxE (by hardware)
Anymore data to be trandfer?
Yes
Set BTF, strecth SCL LOW until DR is write
No
Generate STOP condition
STOP
Slave device (Default slave mode)
Wait for start
Start condition detected
Receive address
Address match?
No
Yes
Send ACK
Recevie data bytes from SDA to DR via the internal shift register
Set RxNE (by hardware)
DR is read?
Yes
No
Set BTF, strecth SCL LOW until DR is read
Received stop condition?
No
Yes
STOP

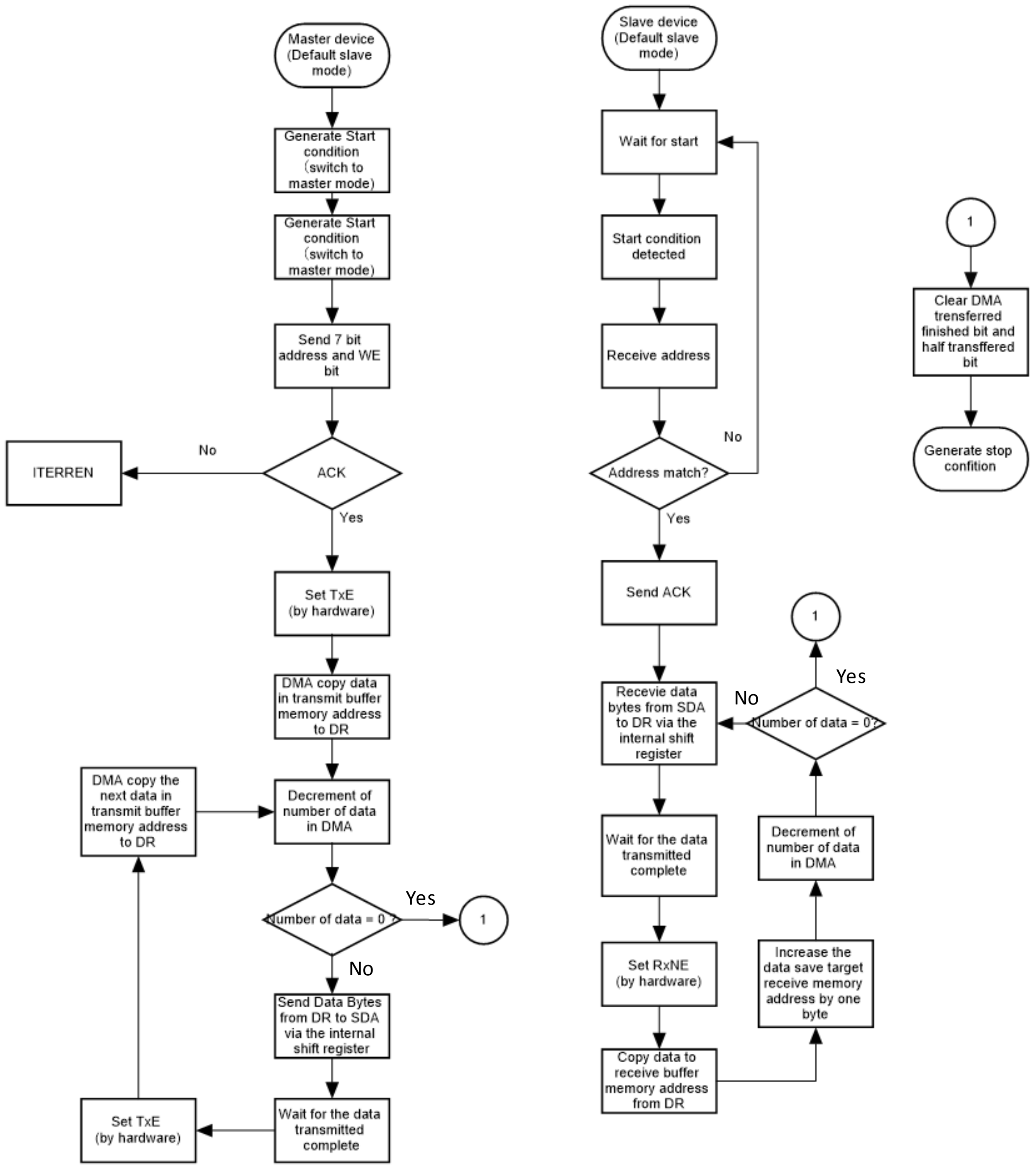
This is the code design flow for master transmit and slave receive mode.

When the master does not receive acknowledge from the slave, the program counter will jump to interrupt (If ITERREN = 1).



Master device (Default slave mode)
Generate Start condition （switch to master mode)
Generate Start condition （switch to master mode)
Send 7 bit address and WE bit
ACK
No
ITERREN
Yes
Recevie data bytes from SDA to DR via the internal shift register
Set RxNE (by hardware)
Read data from DR
Set BTF, strecth SCL LOW until DR is read
Anymore data to be received?
Yes
No
Send NACK
Generate STOP condition
STOP
Slave device (Default slave mode)
Wait for start
Start condition detected
Receive address
Address match?
No
Yes
Send ACK
Wrie data to DR (Send Data)
Send Data Bytes from DR to SDA via the internal shift register
ACK?
No
Yes
Set TxE (by hardware)
Set BTF, strecth SCL LOW until DR is write
STOP

This is the code design flow chart for master receive and slave transmit mode. When the master want to stop, it should send acknowledge in order to let the slave stop the slave from transmitting data to the master and send a STOP condition to release the connection. However, if the slave receive any NACK, the slave will stop sending data.

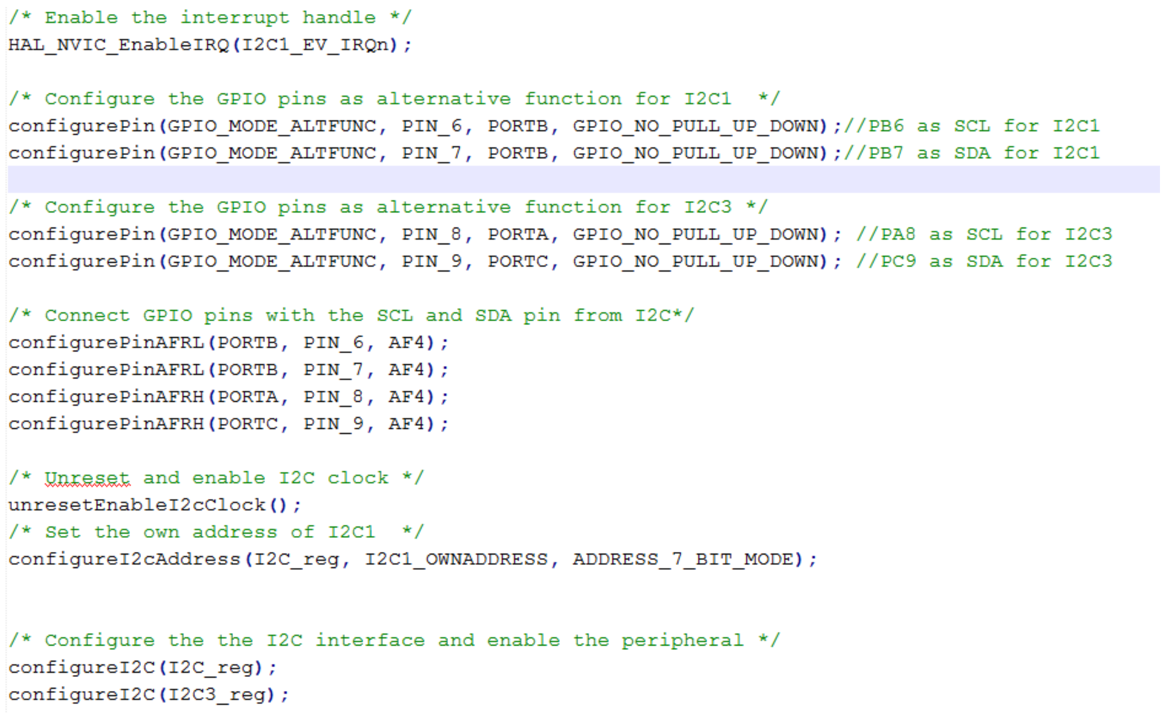


Master device (Default slave mode)
Generate Start condition (switch to master mode)
Generate Start condition (switch to master mode)
Send 7 bit address and WE bit
ACK
No
ITERREN
Yes
Set TxE (by hardware)
DMA copy data in transmit buffer memory address to DR
Decrement of number of data in DMA
DMA copy the next data in transmit buffer memory address to DR
Number of data = 0?
Yes
1
No
Send Data Bytes from DR to SDA via the internal shift register
Wait for the data transmitted complete
Set TxE (by hardware)
Slave device (Default slave mode)
Wait for start
Start condition detected
Receive address
Address match?
No
Yes
Send ACK
Recevie data bytes from SDA to DR via the internal shift register
No
Number of data = 0?
Yes
1
Wait for the data transmitted complete
Set RxNE (by hardware)
Copy data to receive buffer memory address from DR
Increase the data save target receive memory address by one byte
Decrement of number of data in DMA
1
Clear DMA trensferred finished bit and half transffered bit
Generate stop confition

This figure show the flow of the process of transmit and receive data using DMA. To initiate the communication, the master device was generated a start condition. After that, the address was set to the master's data register by using software to select the slave device. By doing this, the bus line was being busy and the TxE flag in the master device was set. The DMA copied the data from the following transmit buffer memory address to the Data Register of the I2C3. The data was sent to the slave device. In the slave device, after the slave received the data and set the RxNE flag, the DMA copied the data from the Data Register to the receive buffer memory.

When the first byte of data was sent via DMA, the program increment by a byte from the original memory address in order to send the next byte. Each data byte was sent would decrease the value in Number of Data Register in DMA by one. When the number of data was zero which meant all the required number of data was transferred from memory to I2C Data Register, the program was jumped to interrupt to clear the flags and generate stop condition to release the communication.

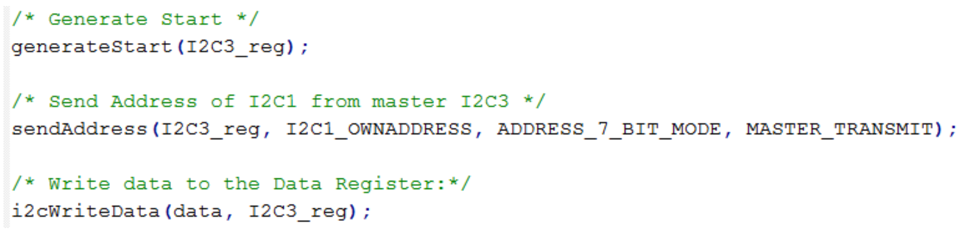
The main() function



```
/* Enable the interrupt handle */
HAL_NVIC_EnableIRQ(I2C1_EV_IRQn);

/* Configure the GPIO pins as alternative function for I2C1  */
configurePin(GPIO_MODE_ALTFUNC, PIN_6, PORTB, GPIO_NO_PULL_UP_DOWN);//PB6 as SCL for I2C1
configurePin(GPIO_MODE_ALTFUNC, PIN_7, PORTB, GPIO_NO_PULL_UP_DOWN);//PB7 as SDA for I2C1

/* Configure the GPIO pins as alternative function for I2C3 */
configurePin(GPIO_MODE_ALTFUNC, PIN_8, PORTA, GPIO_NO_PULL_UP_DOWN); //PA8 as SCL for I2C3
configurePin(GPIO_MODE_ALTFUNC, PIN_9, PORTC, GPIO_NO_PULL_UP_DOWN); //PC9 as SDA for I2C3

/* Connect GPIO pins with the SCL and SDA pin from I2C*/
configurePinAFRL(PORTB, PIN_6, AF4);
configurePinAFRL(PORTB, PIN_7, AF4);
configurePinAFRH(PORTA, PIN_8, AF4);
configurePinAFRH(PORTC, PIN_9, AF4);

/* Unreset and enable I2C clock */
unresetEnableI2cClock();
/* Set the own address of I2C1  */
configureI2cAddress(I2C_reg, I2C1_OWNADDRESS, ADDRESS_7_BIT_MODE);


/* Configure the the I2C interface and enable the peripheral */
configureI2C(I2C_reg);
configureI2C(I2C3_reg);
```

Figure 2.1

Figure 2.1 shows the code for the main function.

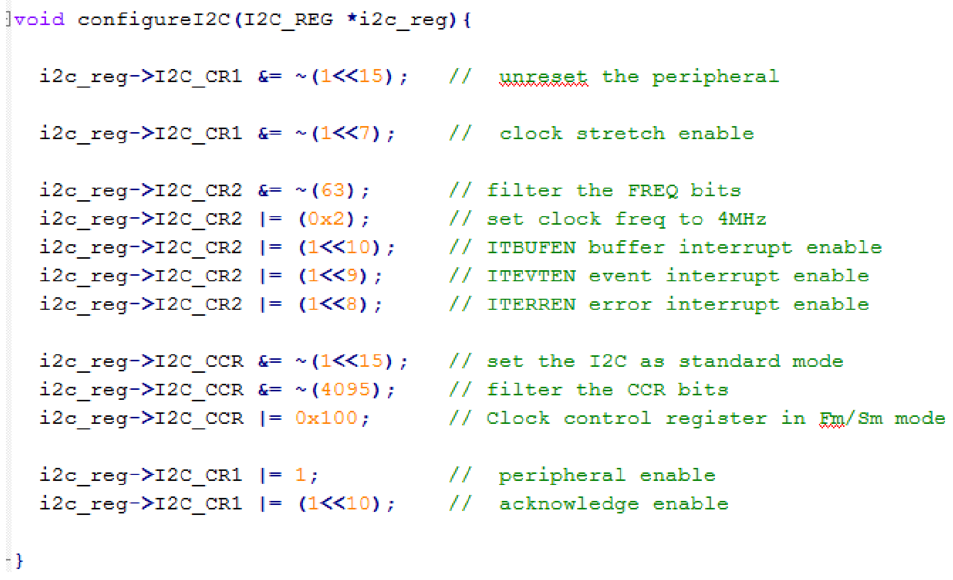


```
/* Generate Start */
generateStart(I2C3_reg);

/* Send Address of I2C1 from master I2C3 */
sendAddress(I2C3_reg, I2C1_OWNADDRESS, ADDRESS_7_BIT_MODE, MASTER_TRANSMIT);

/* Write data to the Data Register:*/
i2cWriteData(data, I2C3_reg);
```

Figure 2.2

Figure 2.2 shows the code for starting the communication between the master (I2C3) and the slave (I2C1).

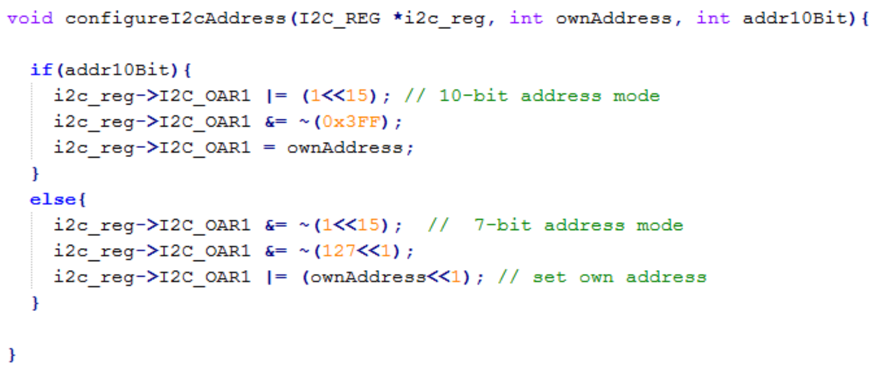
Configure I2C function



```
void configureI2C(I2C_REG *i2c_reg){

  i2c_reg->I2C_CR1 &= ~(1<<15);   //  unreset the peripheral

  i2c_reg->I2C_CR1 &= ~(1<<7);    //  clock stretch enable

  i2c_reg->I2C_CR2 &= ~(63);      // filter the FREQ bits
  i2c_reg->I2C_CR2 |= (0x2);      // set clock freq to 4MHz
  i2c_reg->I2C_CR2 |= (1<<10);    // ITBUFEN buffer interrupt enable
  i2c_reg->I2C_CR2 |= (1<<9);     // ITEVTEN event interrupt enable
  i2c_reg->I2C_CR2 |= (1<<8);     // ITERREN error interrupt enable

  i2c_reg->I2C_CCR &= ~(1<<15);   // set the I2C as standard mode
  i2c_reg->I2C_CCR &= ~(4095);    // filter the CCR bits
  i2c_reg->I2C_CCR |= 0x100;      // Clock control register in Fm/Sm mode

  i2c_reg->I2C_CR1 |= 1;          //  peripheral enable
  i2c_reg->I2C_CR1 |= (1<<10);    //  acknowledge enable

}
```

Figure 2.3

This figure shows the code for configuration of the I2C.

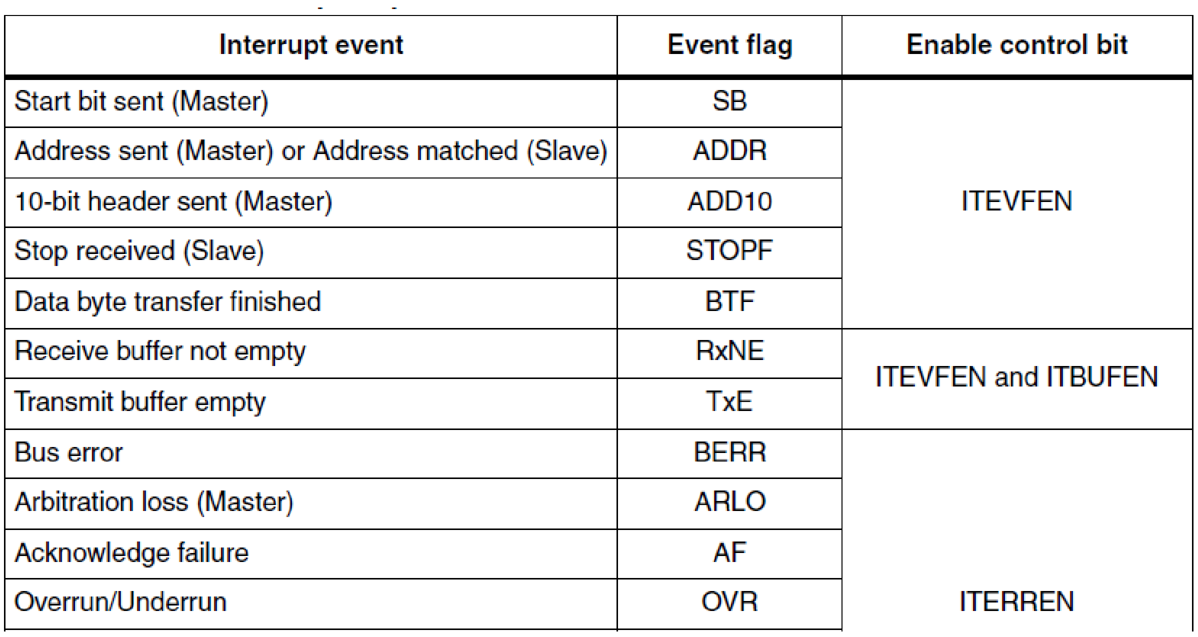


```
void configureI2cAddress(I2C_REG *i2c_reg, int ownAddress, int addr10Bit){

  if(addr10Bit){
    i2c_reg->I2C_OAR1 |= (1<<15); // 10-bit address mode
    i2c_reg->I2C_OAR1 &= ~(0x3FF);
    i2c_reg->I2C_OAR1 = ownAddress;
  }
  else{
    i2c_reg->I2C_OAR1 &= ~(1<<15);  //  7-bit address mode
    i2c_reg->I2C_OAR1 &= ~(127<<1);
    i2c_reg->I2C_OAR1 |= (ownAddress<<1); // set own address
  }

}
```

Figure 2.4

Figure 2.4 shows the configuration of the address of the I2C. This configuration only required for the slave device.

I2C Interrupt and Event



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	

ITEVFEN and ITBUFEN flags

When the program was went to the interrupt, in order to clear the interrupt flags, the following sequences were performed instead of write 0 to the flowing bit in the status register.

SB = 1, cleared by reading SR1 register followed by writing DR register with Address.
ADDR = 1, clear by reading SR1 followed by SR2.
TxE = 1, shift register not empty, .data register empty, cleared by writing DR register.
BTF = 1, Cleared by software by either a read or write in the DR register or by hardware after a start or a stop condition in transmission.
RxNE = 1 cleared by reading DR register.
STOPF = 1, cleared by reading SR1 register followed by writing to the CR1 register
ADD10 = 1, cleared by reading SR1 register followed by writing DR register.
STOPF = 1, cleared by reading SR1 register followed by writing to the CR1 register
AF = 1; AF is cleared by writing ‘0’ in AF bit of SR1 register.

ITERREN flags

Bus error – Device detected an external Stop or Start condition during an address or a data transfer.

Arbitration loss - the I2C interface detects an arbitration lost condition. Arbitration loss is defined by loss of right to control. In I2C, when the master loss the arbitration which meant loss the control of the bus line. This happened when two master devices generate start condition in the same time.

Acknowledge failure – Master device detected a nonacknowledge bit. Master then should generate a Start or Stop condition.

Overrun - When another new byte of data arrived before the previous byte was been read from the Data Register. The last received byte is lost and replaced by the new byte of data.

Underrun – Same data was sent as the new data was not updated to the Data Register of the transmitter to transmit.

While the ITERREN flags was cleared by writing 0 to the following bit in the Status Register 1 of the I2C.

Configuration for I2C

I2C Control Register 1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWRST	Res.	ALERT	PEC	POS	ACK	STOP	START	NO STRETCH	ENGC	ENPEC	ENARP	SMB TYPE	Res.	SMBU S	PE
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw		rw	rw

ACK was set to enable acknowledgement.

NOSTRETCH was reset to enable clock stretching.

PE was set to enable the peripheral. (This bit must be enable at last after configure the other register.)

I2C Control Register 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			LAST	DMA EN	ITBUF EN	ITEVTE N	ITERR EN	Reserved		FREQ[5:0]					
			rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

ITBUFEN, ITEVEN and ITERREN were set to enable all interrupts.

Set FREQ to 4MHz as this experiment was run in standard mode.

I2C Own Address Register 1

This register need to be configured for slave devices only.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD MODE	Reserved					ADD[9:8]		ADD[7:1]							ADD0
rw						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

ADD[7:1] indicated the slave address was assigned with a fixed value. Only ADD[7:1] was assign for 7 bit address mode.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F/S	DUTY	Reserved		CCR[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CRR[11:0] was set with value 0x100 to generate about 8k of SCL frequency.

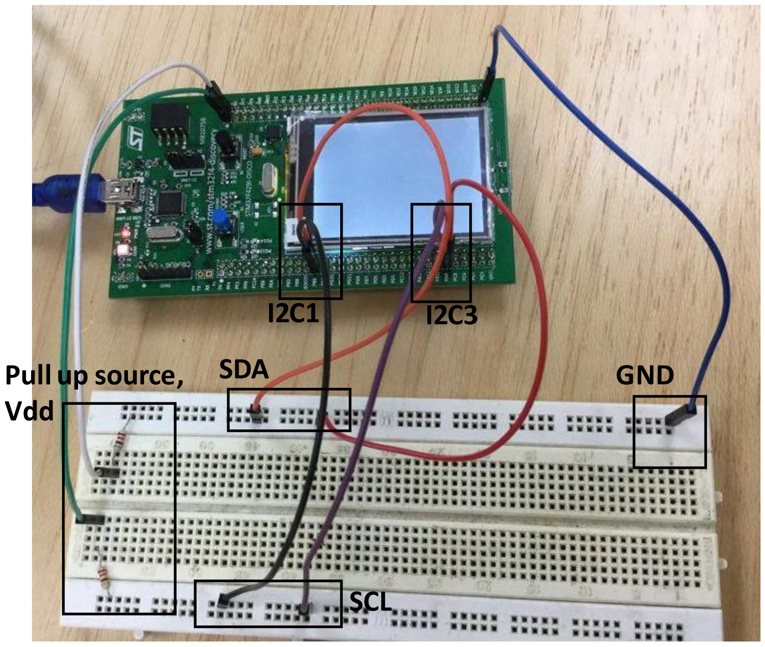
Configuration for GPIO

PB6 and PB7 were configured as SCL and SDA of I2C1.

PA8 and PC9 were configured as SCL and SDA of I2C3.

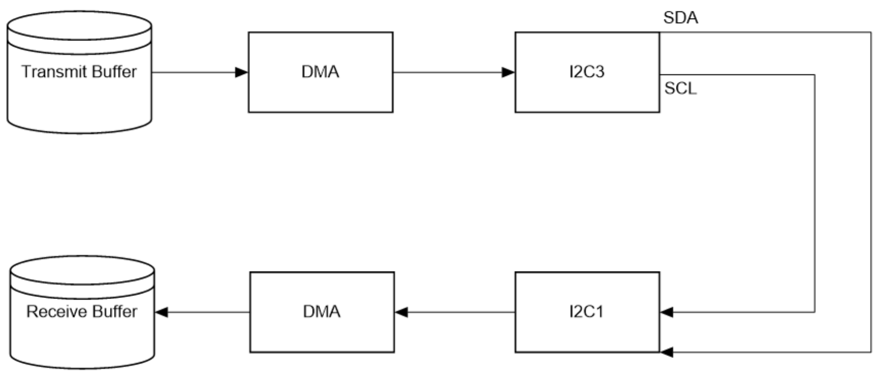
Set all the port as alternate function mode, open drain, no pull up pull down and high speed mode.

Set the alternative function register (AFRH or AFRL) according to the alternate function mapping table for STM32F429xx device which may found in http://www.sciencezero.org/index.php?title=STM32F429_Microcontroller.



This figure shown the way of the connection of the external pull-up voltage and the two I2C interfaces which were the I2C1 and I2C3.

Configuration for communication using DMA (Direct Access Memory)



The figure was showing how the DMA was used to transmit data from memory (Transmit Buffer) to another memory (Receive buffer) via I2C interfaces.

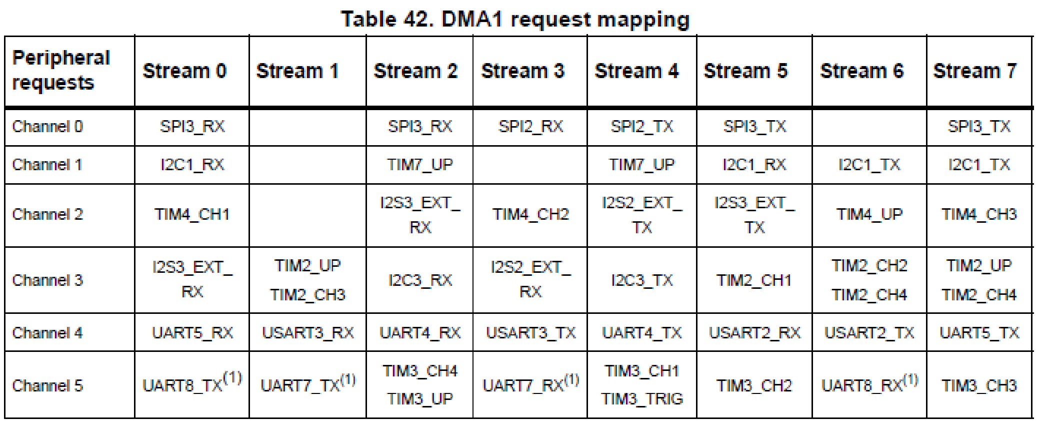


Table 42. DMA1 request mapping

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_ RX	TIM4_CH2	I2S2_EXT_ TX	I2S3_EXT_ TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_ RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_ RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3

Table shown the DMA1 request mapping. As I2C3 was used to transmit data while I2C was used to receive data, so I2C3_TX and I2C_RX were selected:

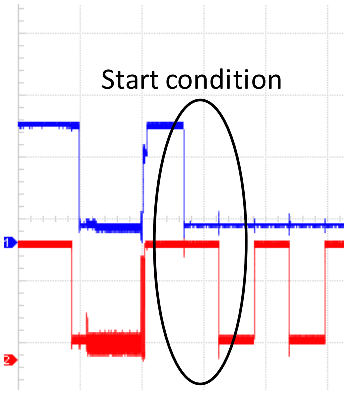
I2C3_TX: Stream 4, Channel 3

I2C1_RX: Stream 5, Channel 1

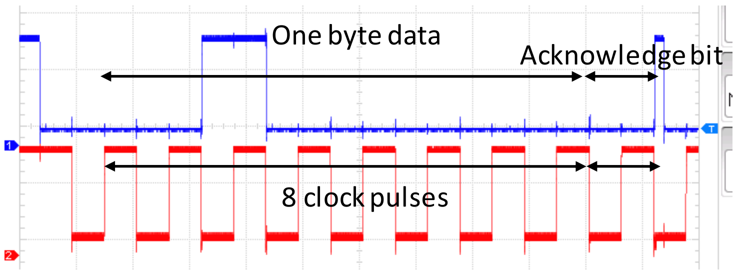
The number of size of data to transfer was a byte. This was because I2C was transmitting data by byte to byte. Data more to one byte was cut off and only the least 8 bit was successful be transferred. Memory increment mode was enabled which mean when the data was sent from the first address of the memory, the follow data in the second address will be sent. This applied for receiving data too.

3. Result

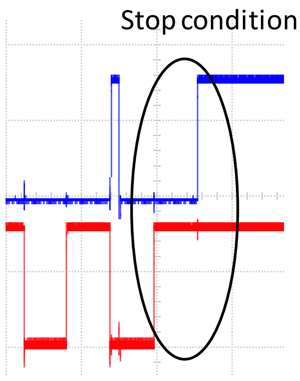
In the whole experiment, I2C3 was configured as the master device will the I2C1 was configured as slave device.



Every communication of the device using the I2C interface was starting with a start condition which was generated by the master device.



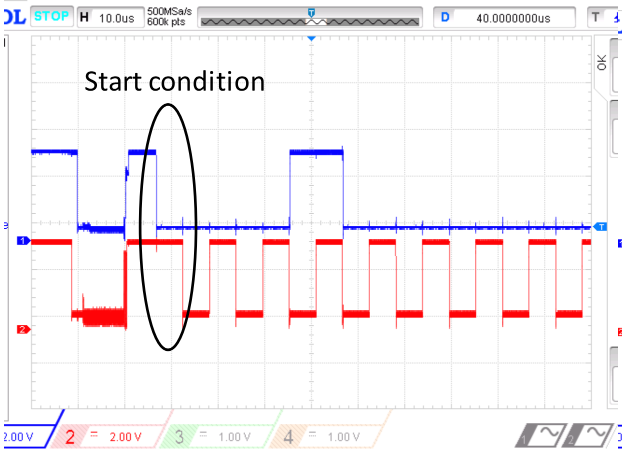
After the start bit, followed by byte data and the acknowledge bit. The 8 clock pulses will be triggered again to transmit the next data byte.

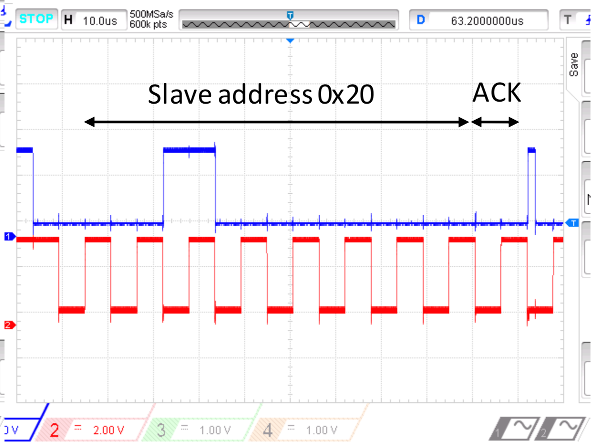


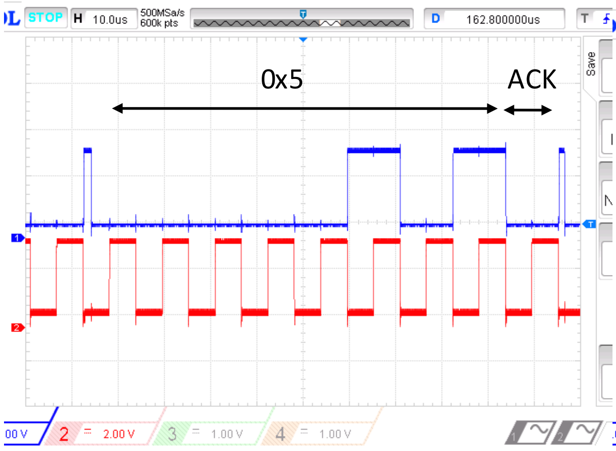
The stop condition was generated when the master device want to stop the data transmitting. When this condition was generated, the bus line was freed and the master switched to slave device.

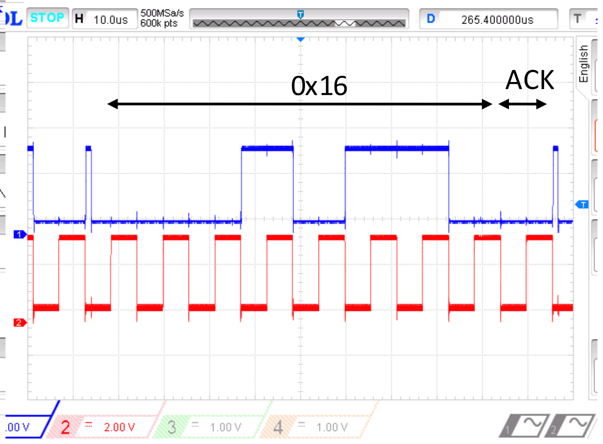
Send 0x5 and 0x16 from master to slave with address 7 bit 0x20.

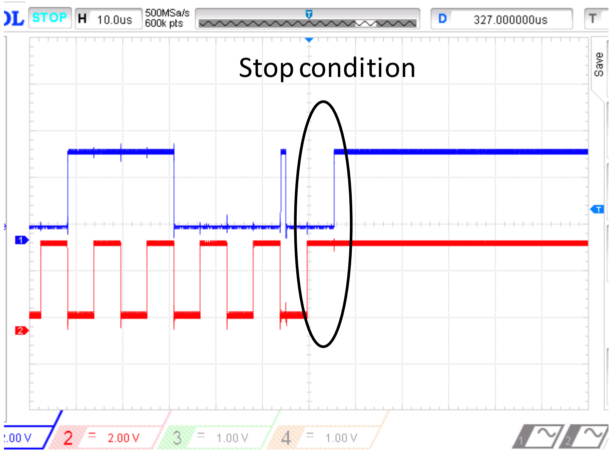
The following figure shows the waveform of transmitting data 0x5 and 0x16 from master to slave with address 7 bit 0x20.





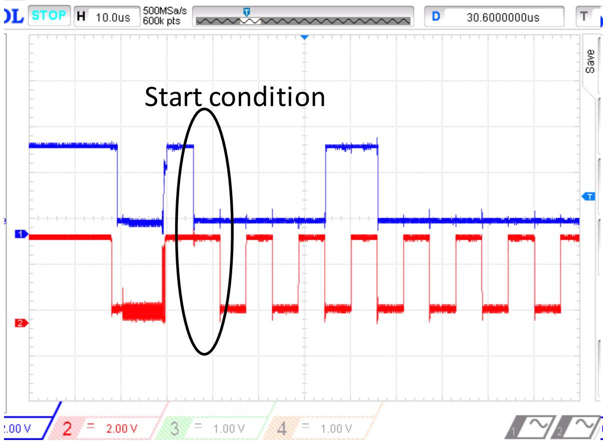


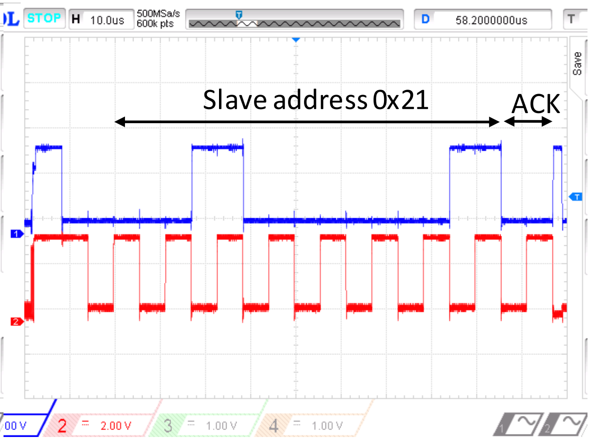


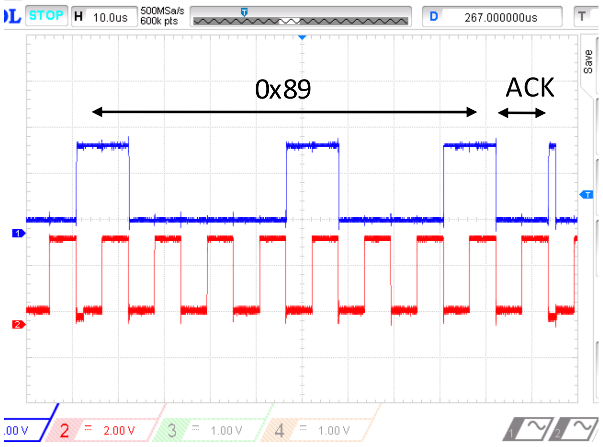


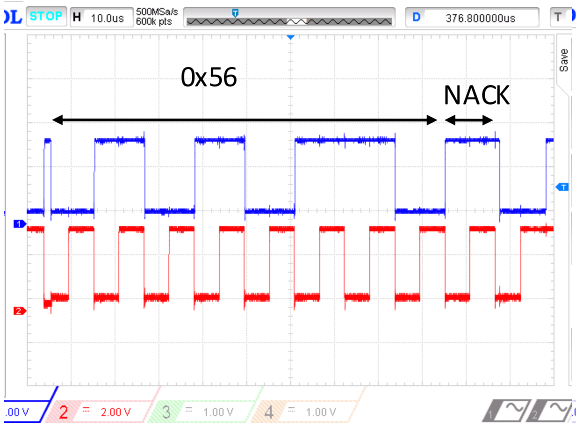
Receive 0x89, and 0x56 from slave device with 7 bit address 0x21 to master device.

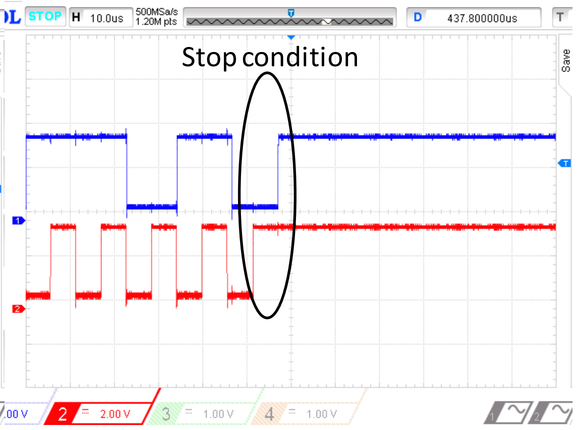
The following figure shows the waveform of transmitting data 0x89, and 0x56 from slave with address 7 bit 0x21 to the master device.





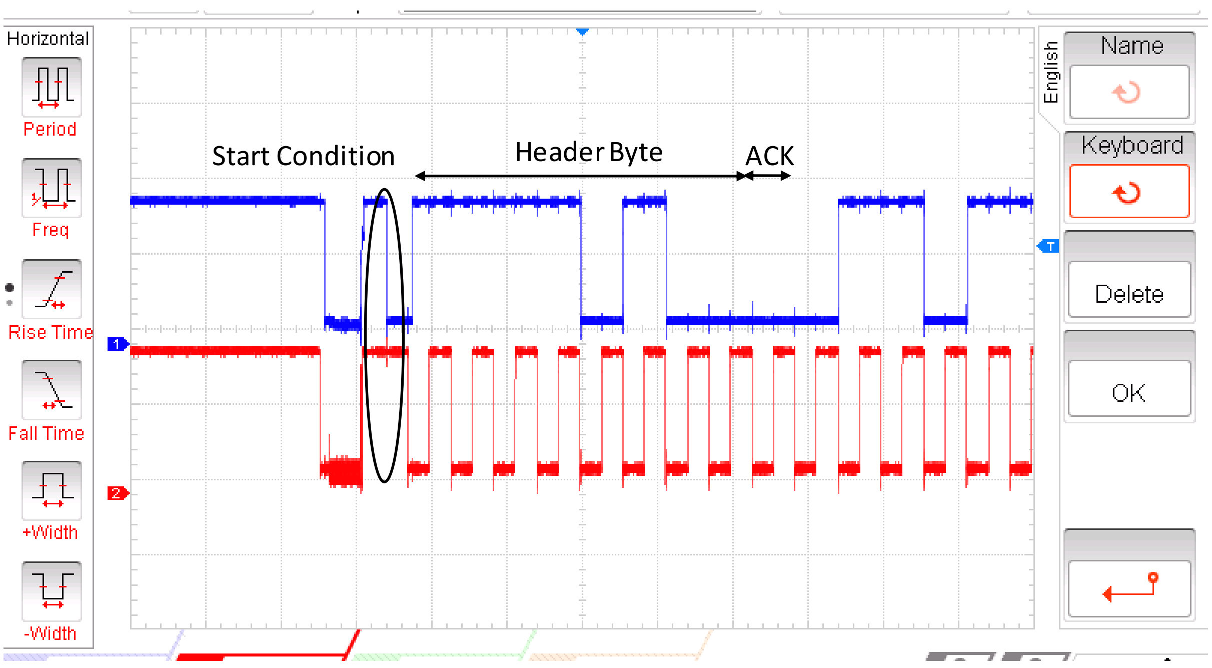


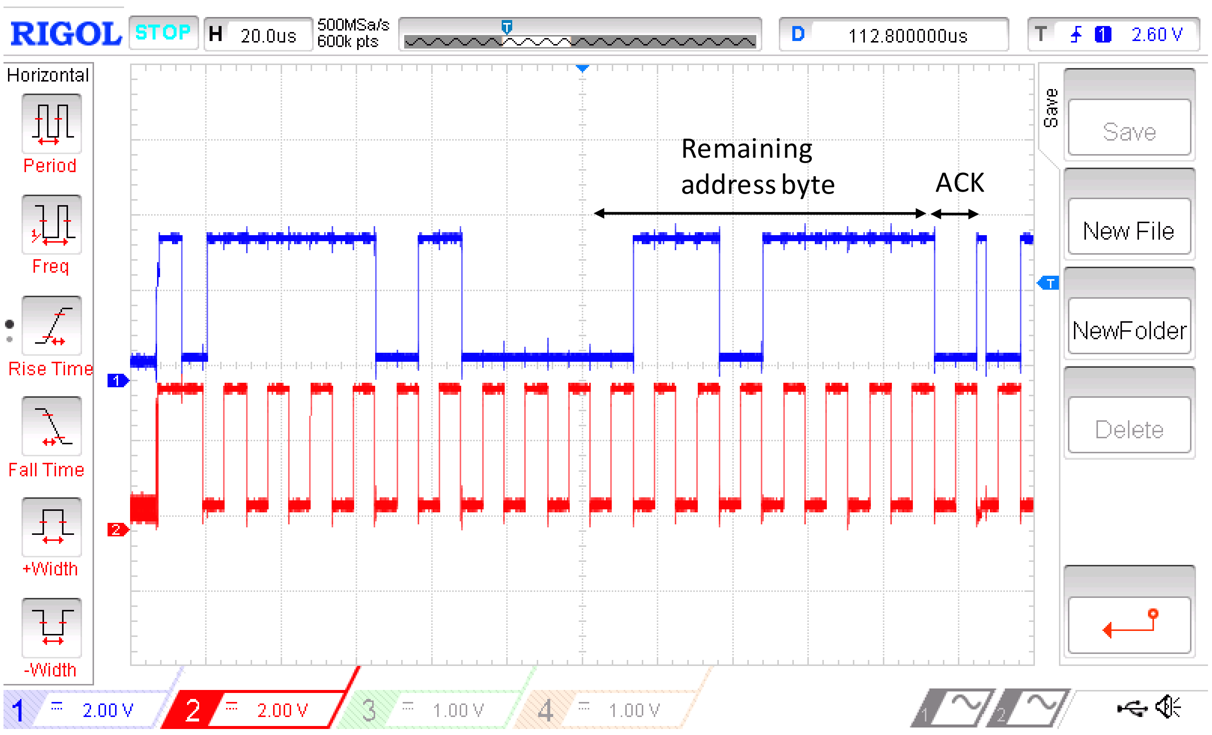


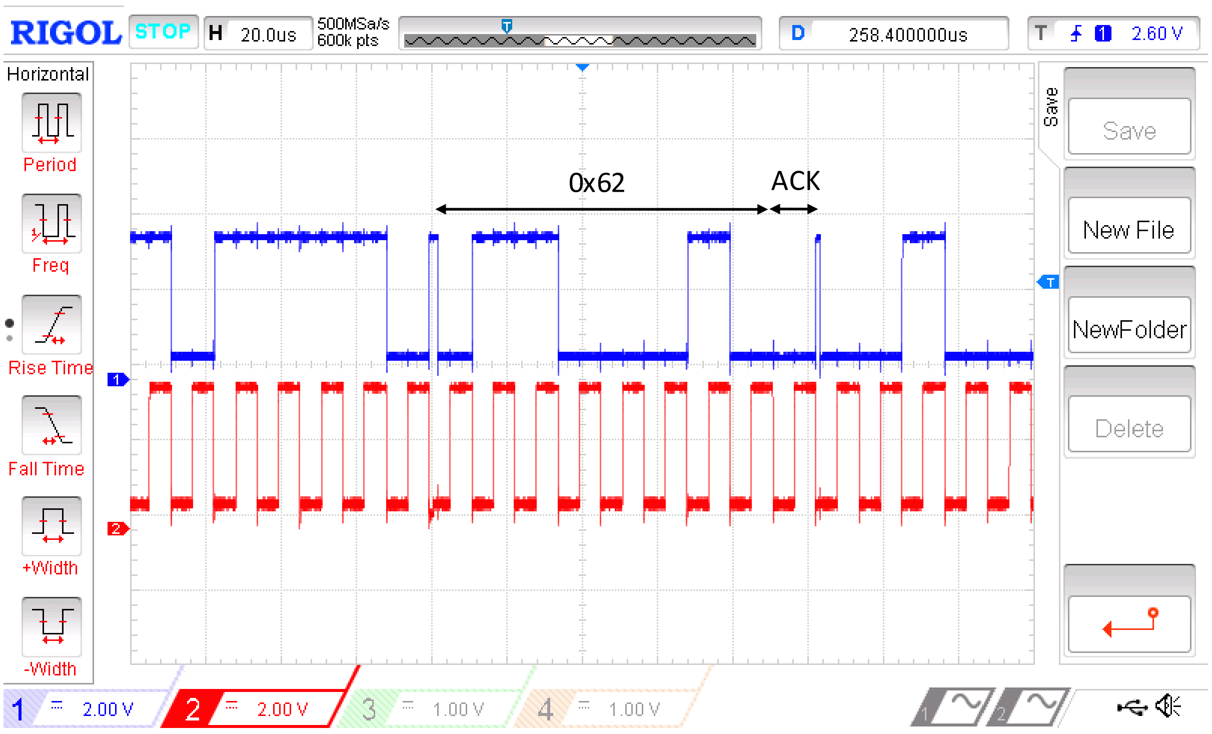


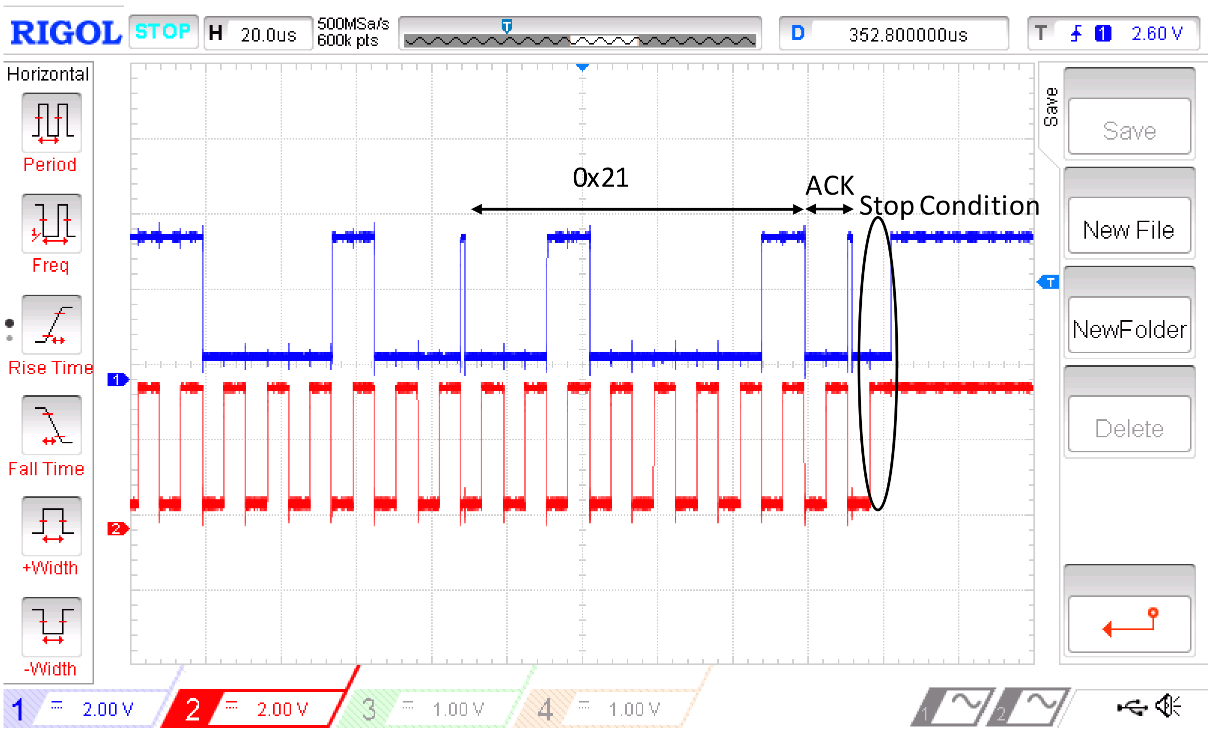
<u>Send 0x62 and 0x21 from master device to slave device with 10 bit address mode.</u>

The following figure shows the signal waveform of transmitting data 0x46 and 0x 28 from master device to slave device in 10 bit address mode with address 0x26F.



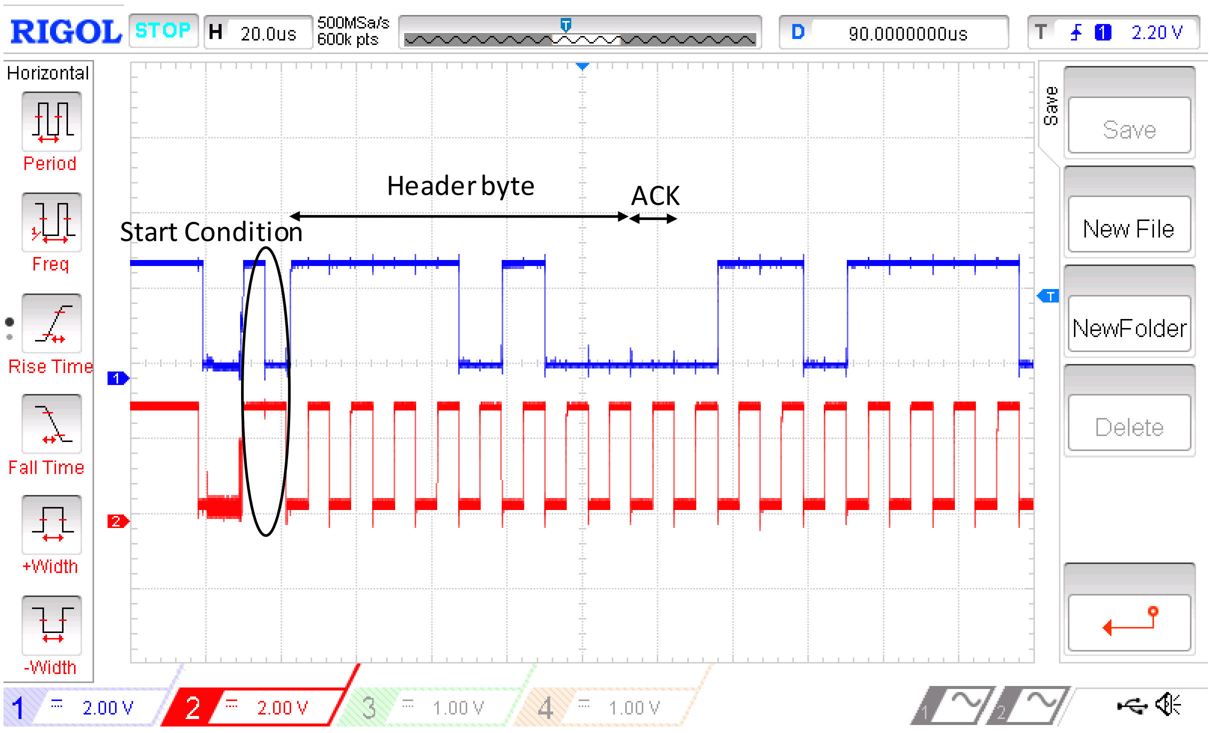


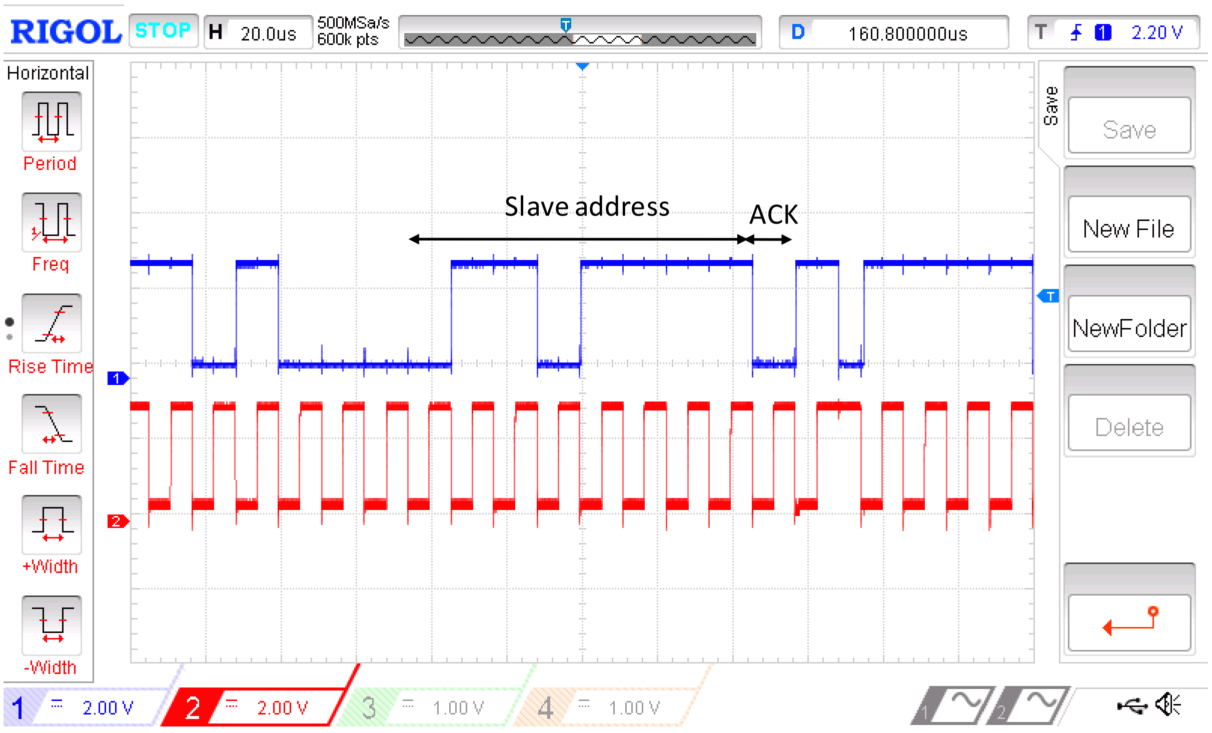

RIGOL
STOP
H 20.0us
500MSa/s
600k pts
D 258.400000us
T 2.60 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
0x62
ACK
Save
Save
New File
NewFolder
Delete
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

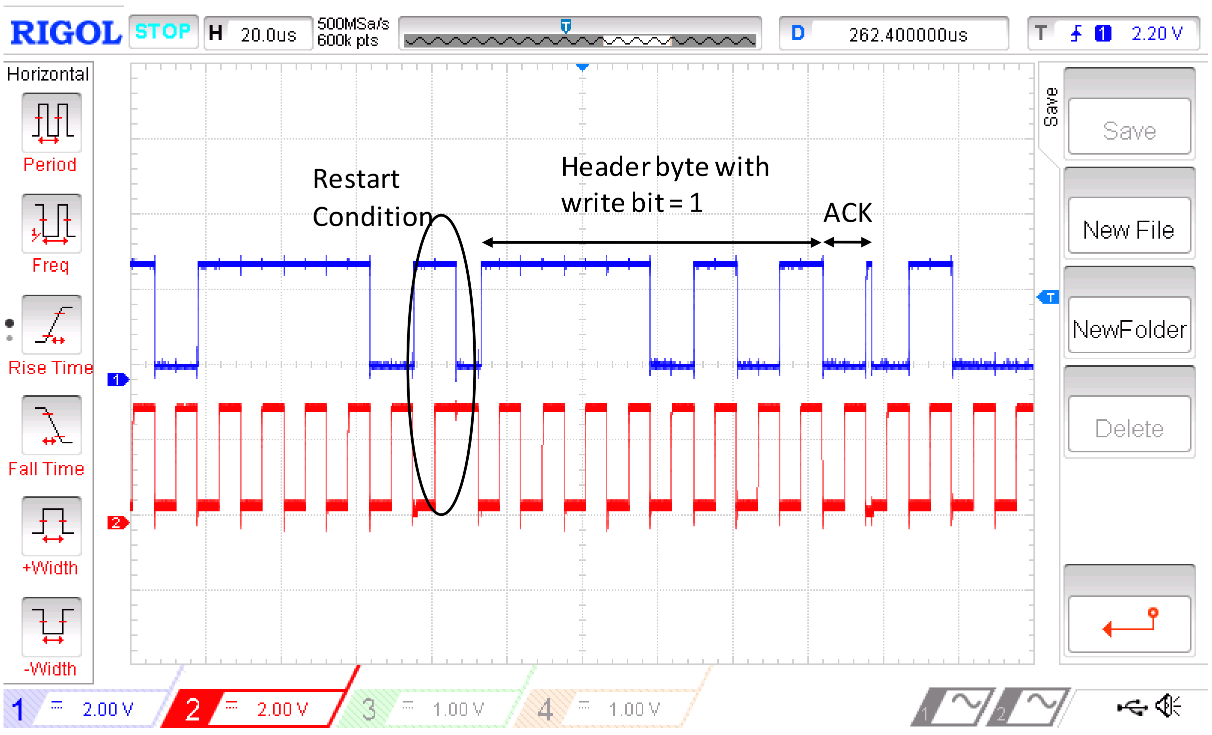

RIGOL
STOP
H 20.0us
500MSa/s
600k pts
D 352.800000us
T 2.60 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
0x21
ACK
Stop Condition
Save
Save
New File
NewFolder
Delete
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

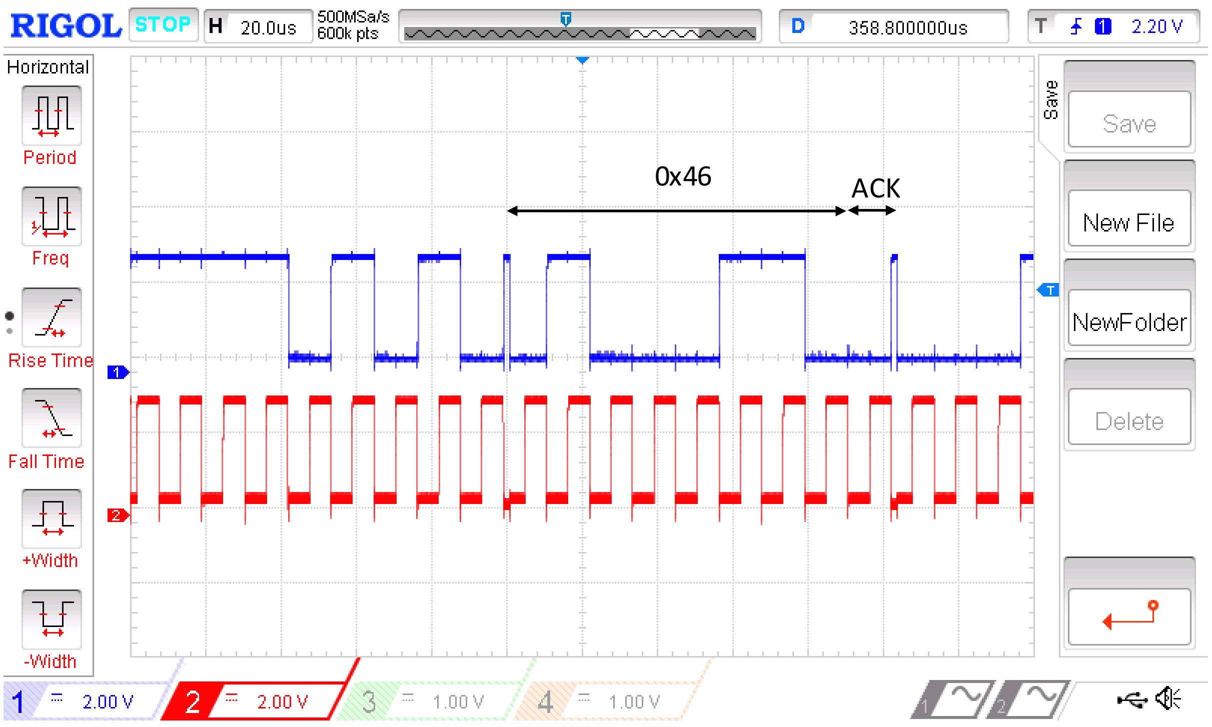
Receive 0x46 and 0x18 from slave device to master device in 10 bit address mode.

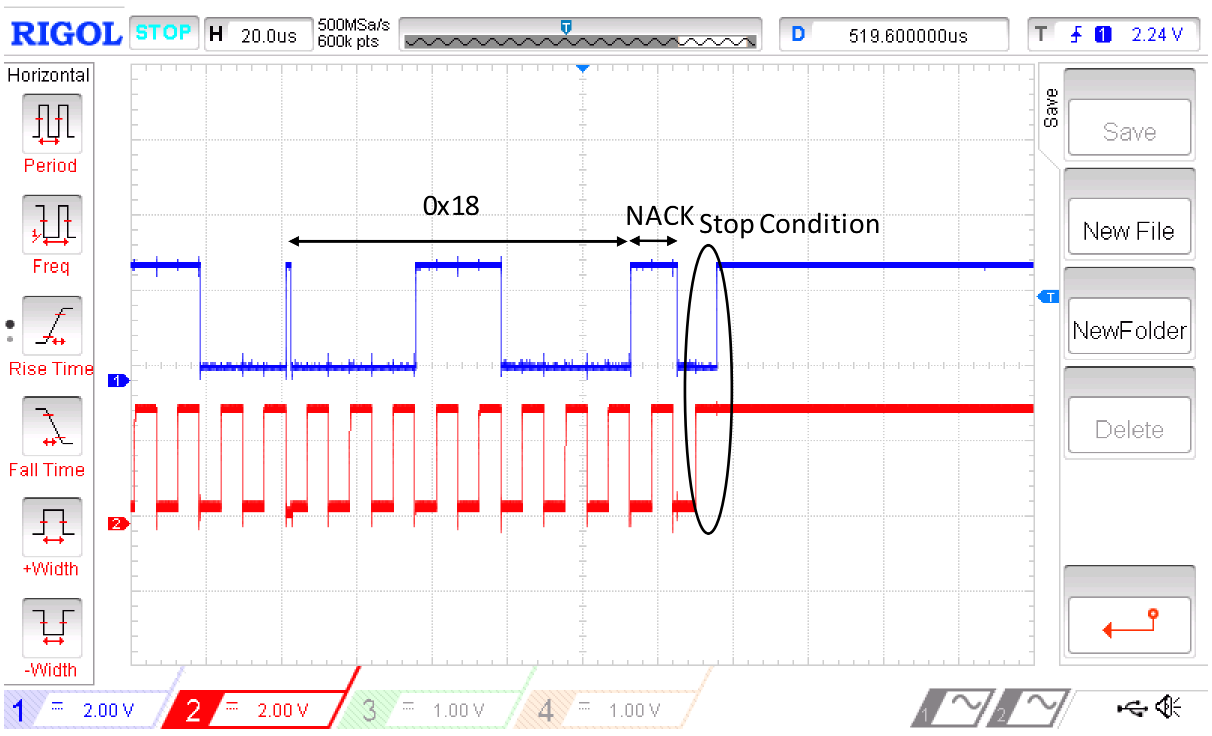
The following figure shows the signal waveform of transmitting data 0x46 and 0x 28 from slave device with 10 bit address mode and address of 0x26F to master device.





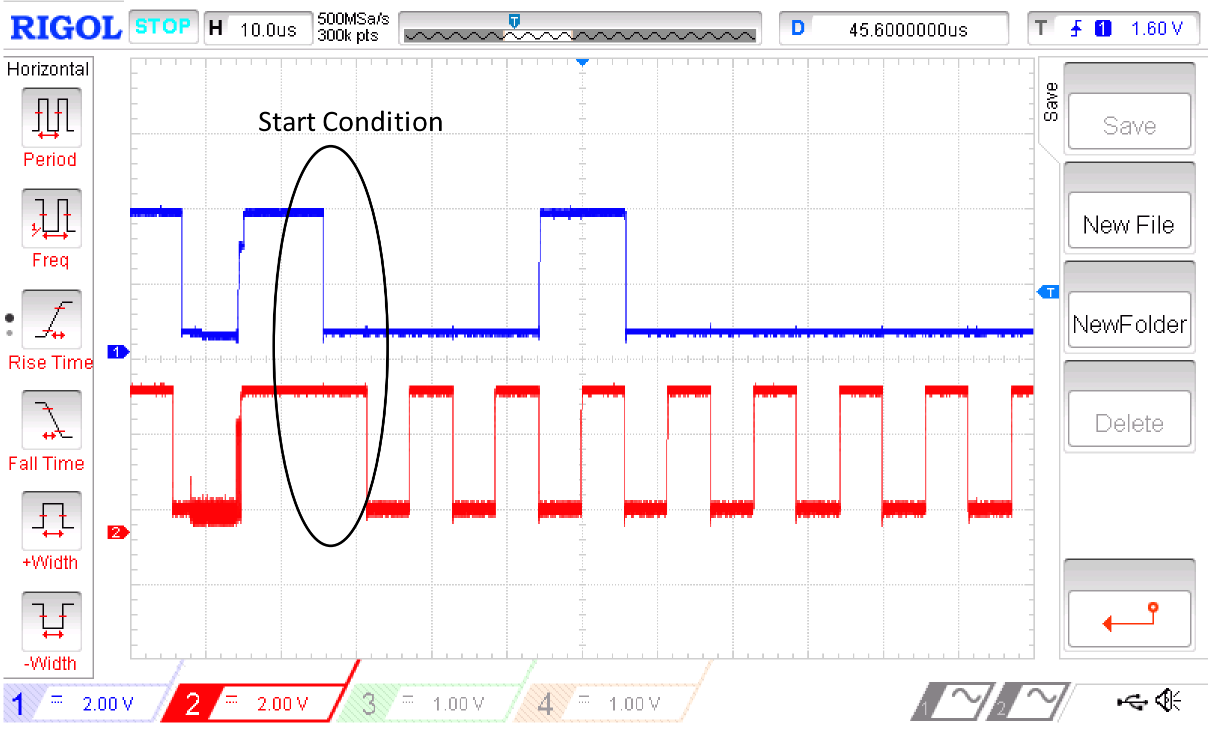

RIGOL
STOP
H 20.0us
500MSa/s
600k pts
D 262.400000us
T 2.20 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
Save
Save
New File
NewFolder
Delete
Restart Condition
Header byte with write bit = 1
ACK
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

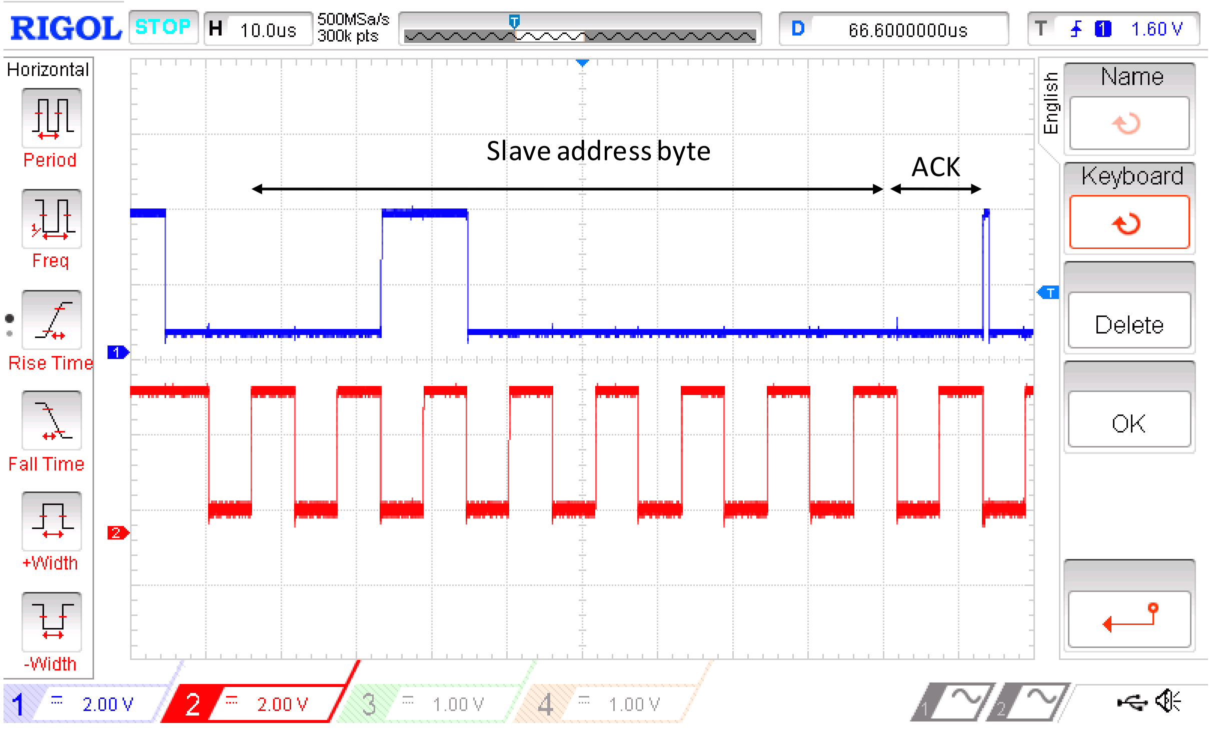

RIGOL
STOP
H 20.0us
500MSa/s
600k pts
D 358.800000us
T 2.20 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
Save
Save
New File
NewFolder
Delete
0x46
ACK
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

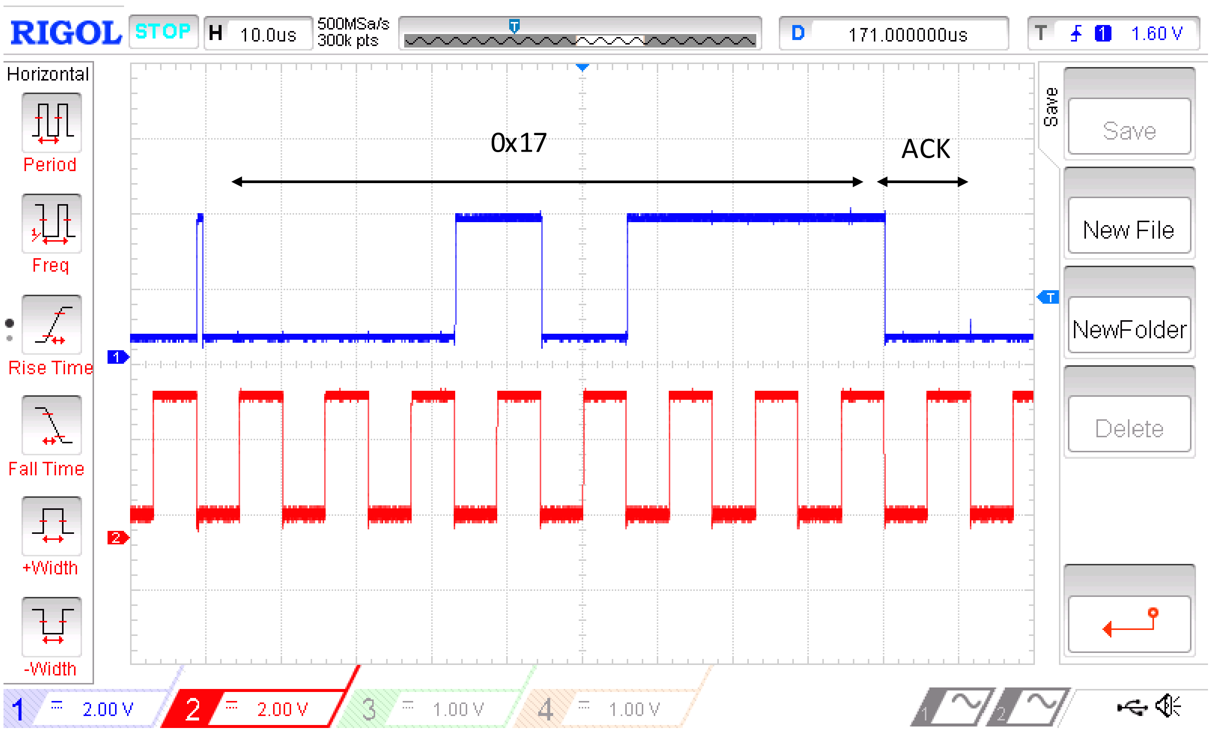

RIGOL
STOP
H 20.0us
500MSa/s
600k pts
D 519.600000us
T 2.24 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
0x18
NACK
Stop Condition
Save
Save
New File
NewFolder
Delete
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

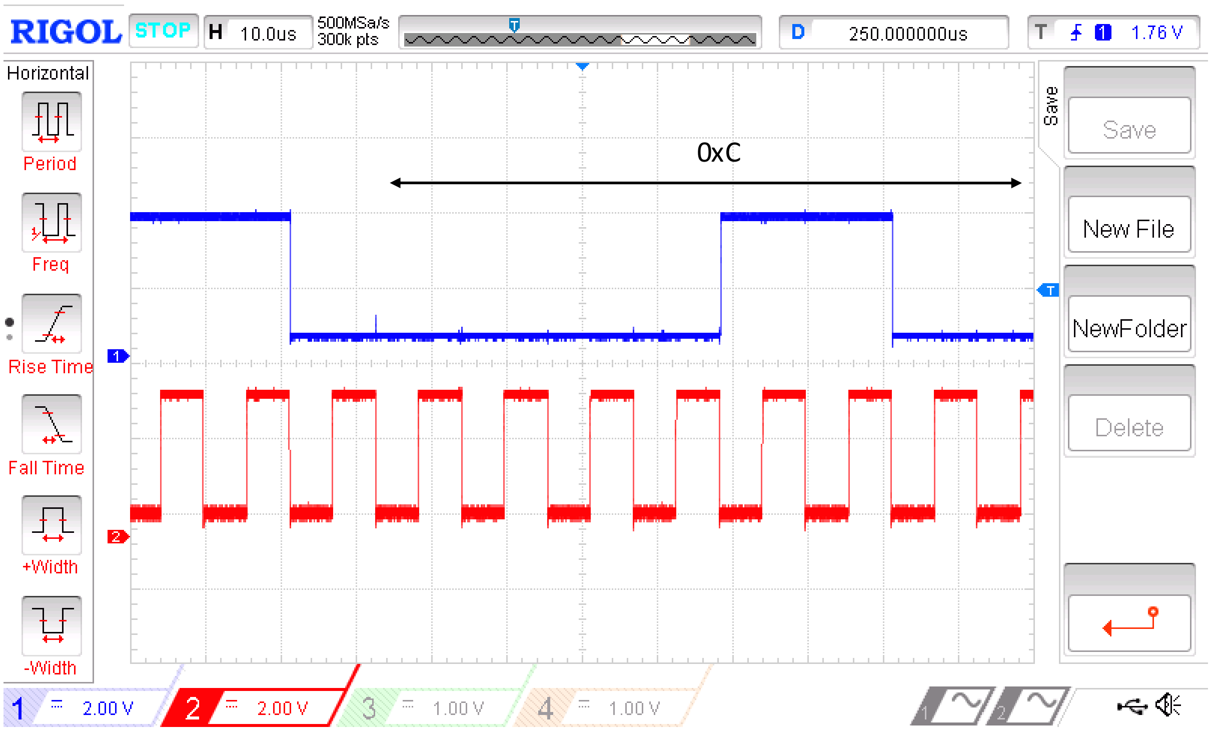
Transmission using DMA from I2C3 to I2C1.

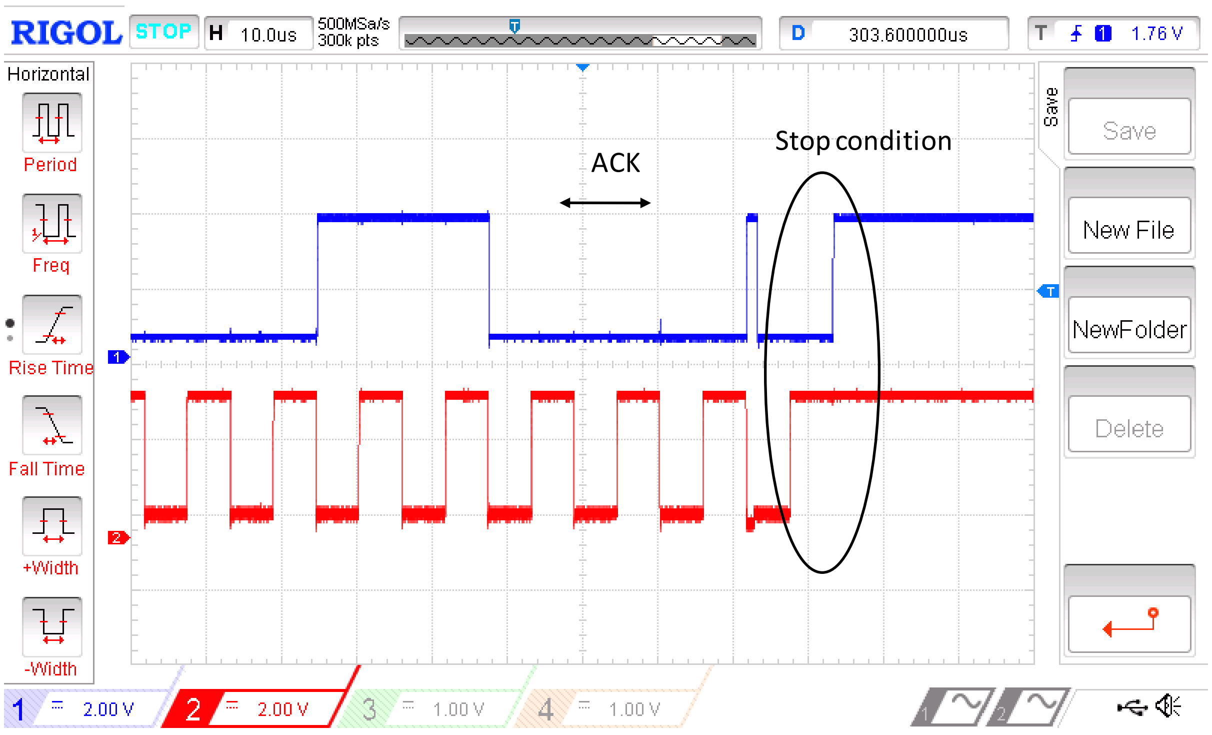
The following figure shows the transmitting of data 0x17 and 0xC from a memory buffer to the slave device through a master device using DMA. The address of the slave device was 0x20.






RIGOL
STOP
H 10.0us
500MSa/s
300k pts
D 171.000000us
T 1.60 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
Save
Save
New File
NewFolder
Delete
0x17
ACK
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V


RIGOL
STOP
H 10.0us
500MSa/s
300k pts
D 250.000000us
T 1.76 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
Save
Save
New File
NewFolder
Delete
0xC
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

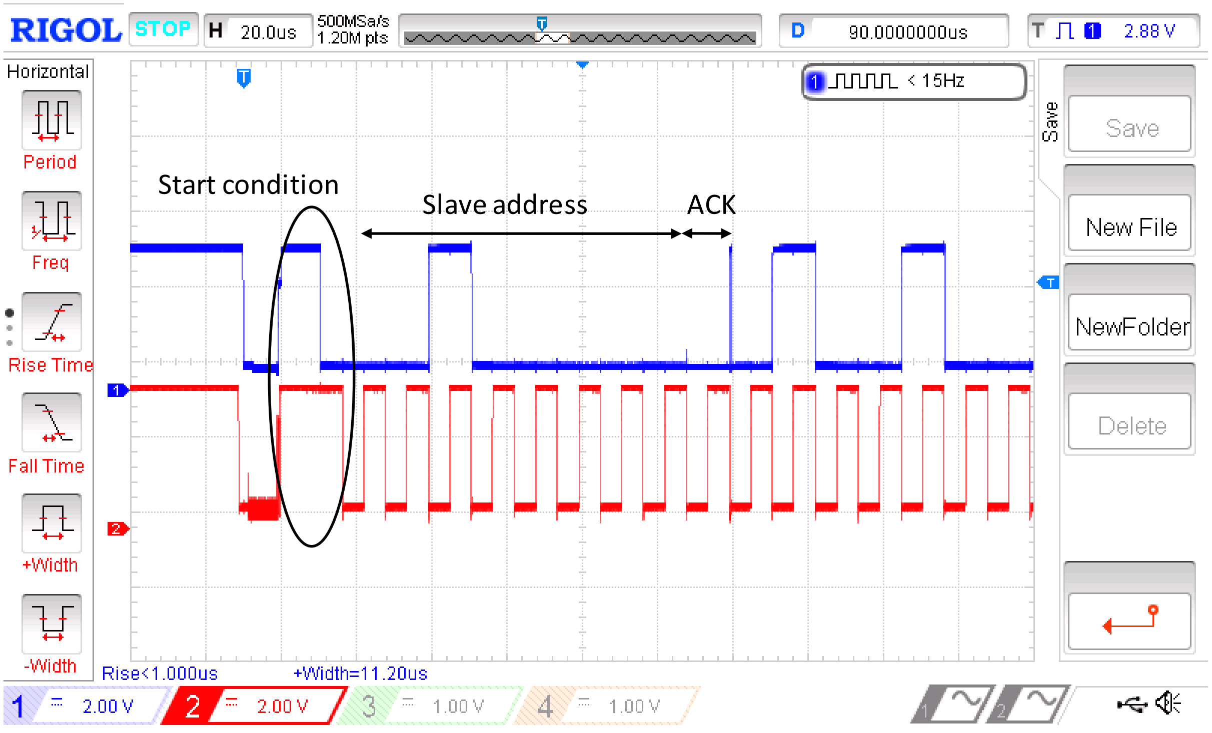

RIGOL
STOP
H 10.0us
500MSa/s
300k pts
D 303.600000us
T 1.76 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
ACK
Stop condition
Save
Save
New File
NewFolder
Delete
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

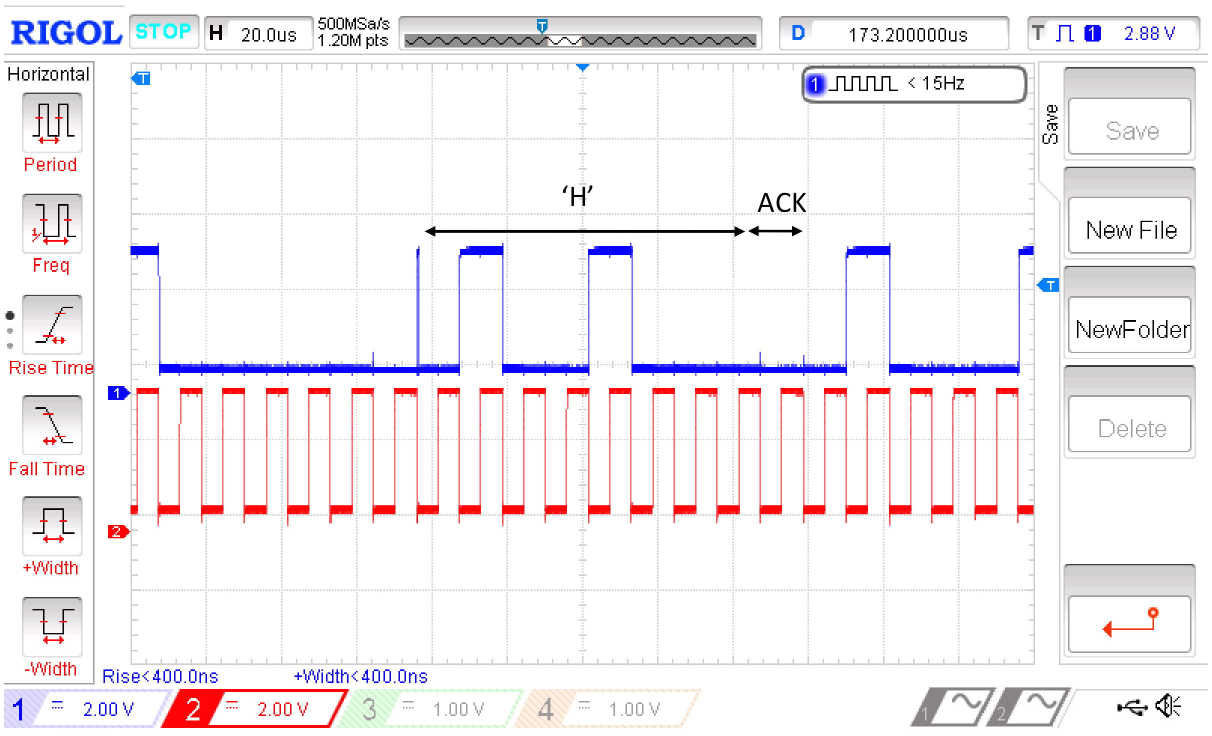
Transfer message "HELLO" ASCII characters from memory to another memory using DMA via I2C interface.

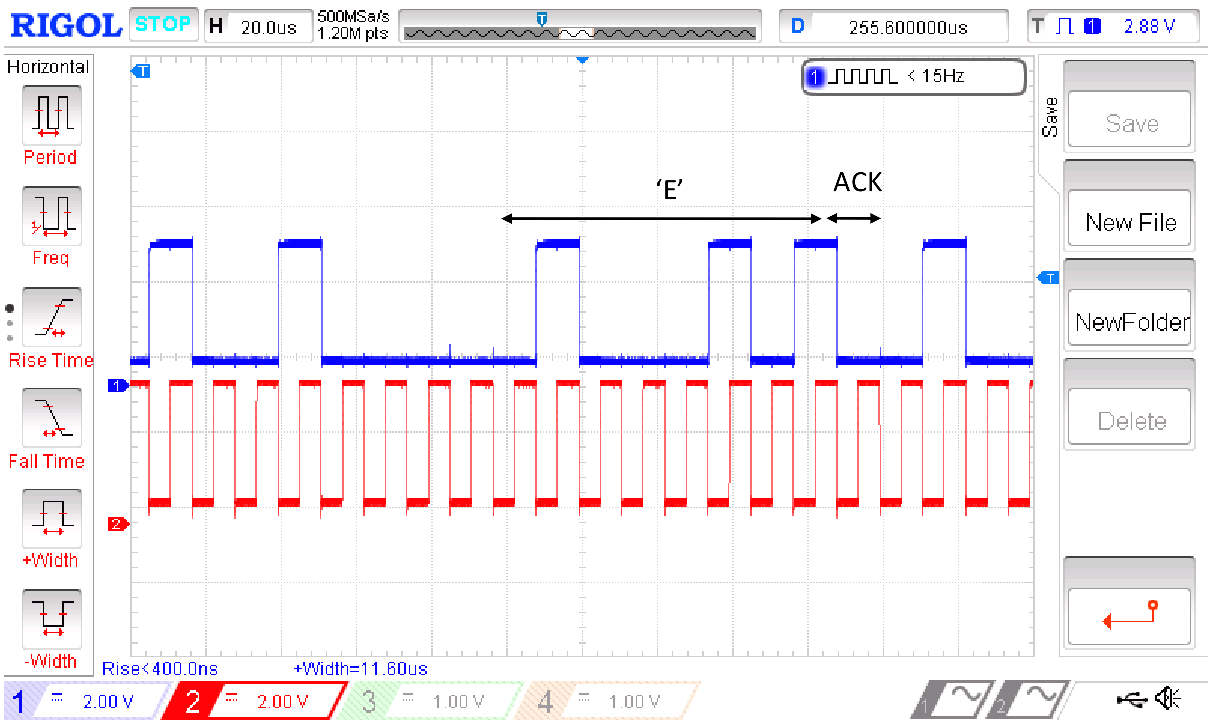
ASCII character	Hex	Binary
H	0x48	01001000
E	0x45	01000101
L	0x4c	01001100
L	0x4c	01001100
O	0x4f	01001111

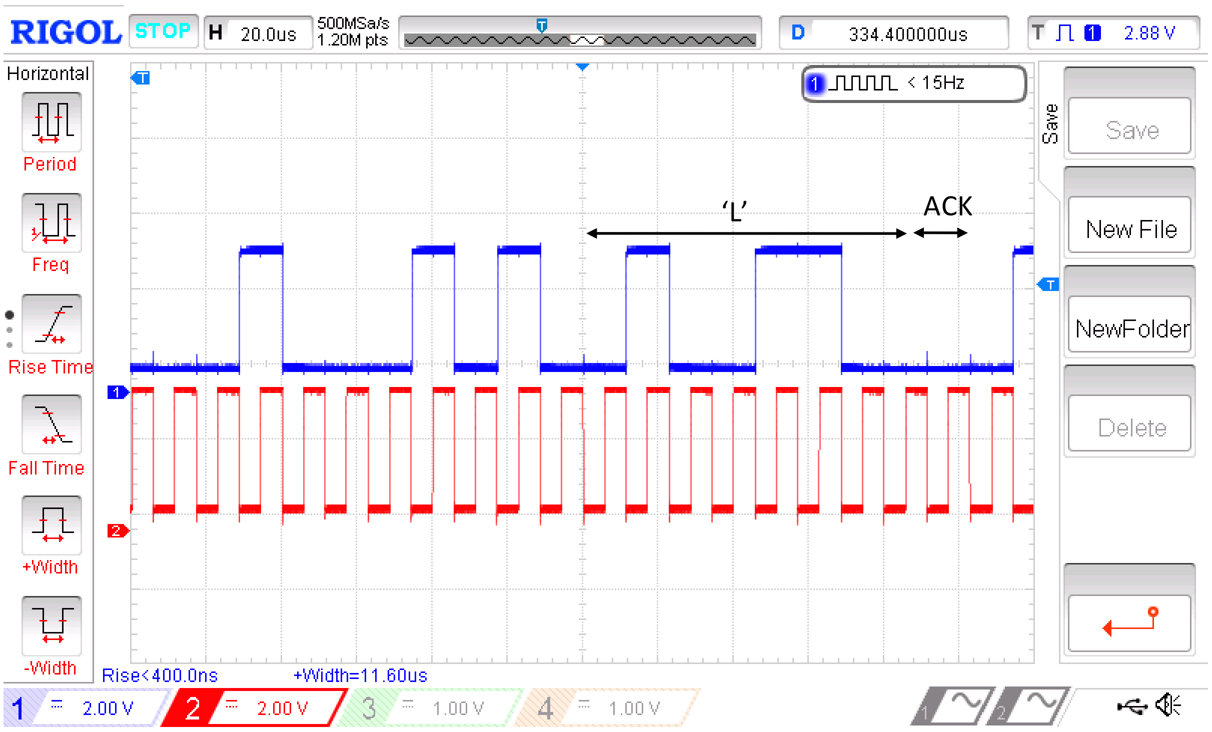
Convert ASCII character to Binary to verify the signal

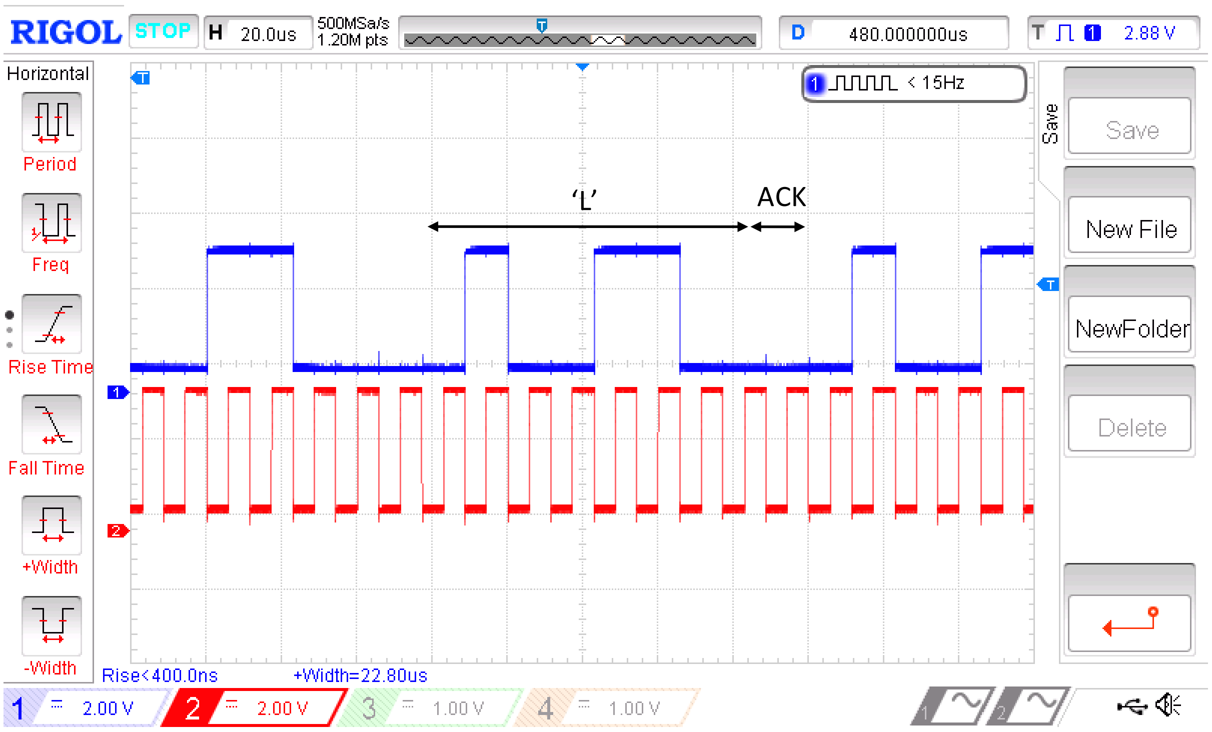
The following figure shows the transmitting of data "HELLO" from the transmit buffer memory to the receive buffer memory using DMA. The data in the transmit buffer was sent out via the I2C3 while the data received via the I2C1 and store in the receive buffer memory.

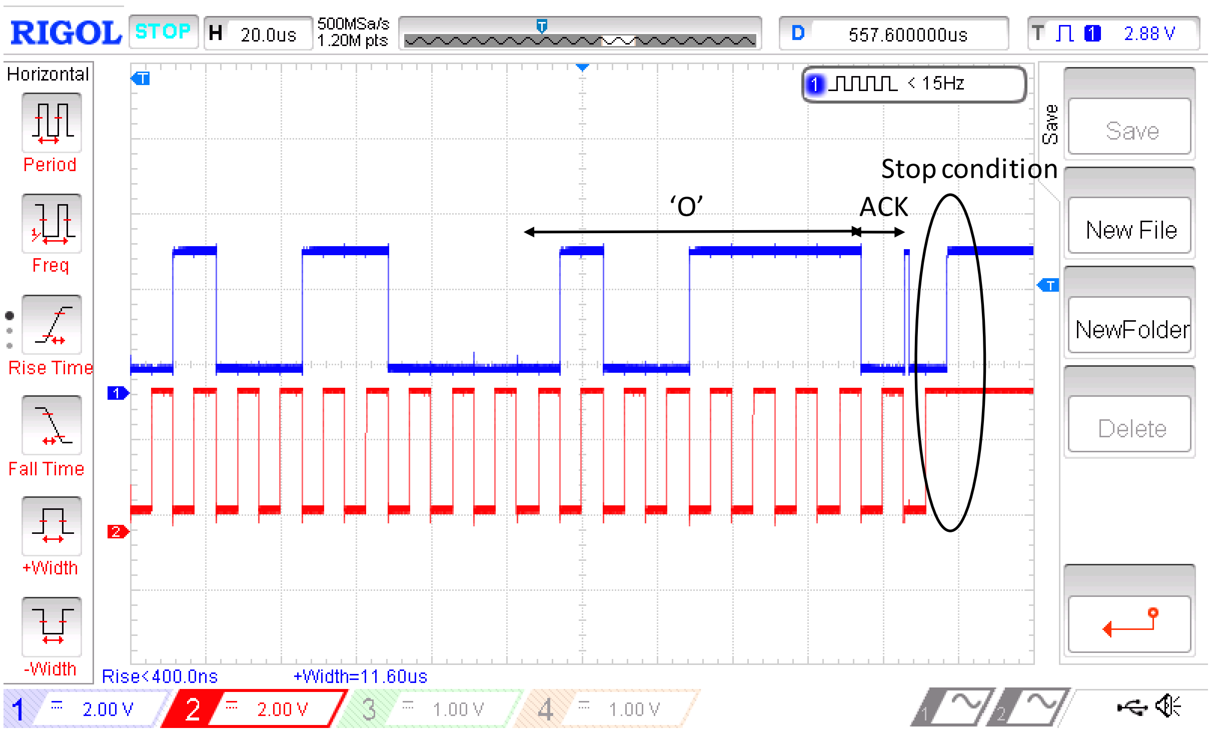



RIGOL
STOP
H 20.0us
500MSa/s
1.20M pts
D 173.200000us
T 2.88 V
< 15Hz
'H'
ACK
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
Save
Save
New File
NewFolder
Delete
Rise<400.0ns
+Width<400.0ns
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V

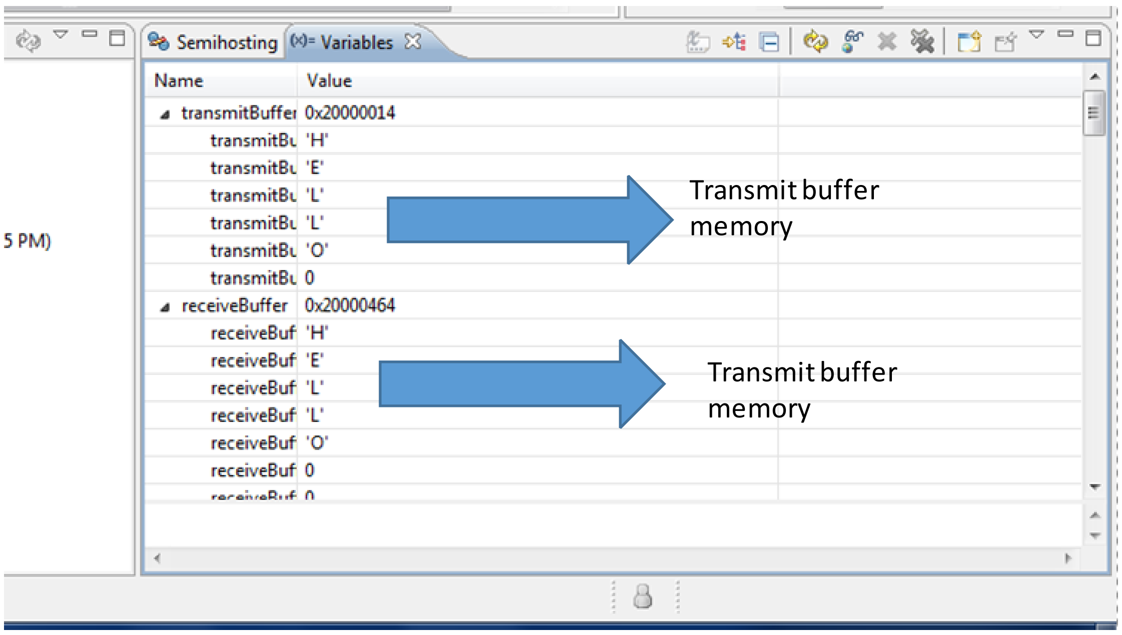

RIGOL
STOP
H 20.0us
500MSa/s
1.20M pts
D 255.600000us
T 2.88 V
< 15Hz
'E'
ACK
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
Save
Save
New File
NewFolder
Delete
Rise<400.0ns
+Width=11.60us
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V


RIGOL
STOP
H 20.0us
500MSa/s
1.20M pts
D 334.400000us
T 2.88 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
< 15Hz
'L'
ACK
Save
Save
New File
NewFolder
Delete
Rise<400.0ns
+Width=11.60us
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V


RIGOL
STOP
H 20.0us
500MSa/s
1.20M pts
D 480.000000us
T 2.88 V
Horizontal
Period
Freq
Rise Time
Fall Time
+Width
-Width
< 15Hz
'L'
ACK
Save
Save
New File
NewFolder
Delete
Rise<400.0ns
+Width=22.80us
1 2.00 V
2 2.00 V
3 1.00 V
4 1.00 V



The receive buffer memory was observed using debugging tool to check the data was transmitted from transmit buffer to receive buffer memory.



Name	Value
transmitBuffer	0x20000014
transmitBu	'H'
transmitBu	'E'
transmitBu	'L'
transmitBu	'L'
transmitBu	'O'
transmitBu	0
receiveBuffer	0x20000464
receiveBuf	'H'
receiveBuf	'E'
receiveBuf	'L'
receiveBuf	'L'
receiveBuf	'O'
receiveBuf	0
receiveBuf	0

4. Discussion

From the observation of the result, once the start condition was generated, the first byte data which was the address of the slave will be sent. Every data byte had an acknowledge bit at the end of the byte. The acknowledge bit was controlled by the receiver. When the receiver was successfully receive the data, it pull the acknowledge bit to low, else the bit will remain high. When the master device was receiving, the master device purposely generate a non-acknowledge(NACK) in order to get back the control of the line to send the stop condition. The reason of a master device have to generate the stop condition to close the communication instead of just leave it there after finishing sent or received data is because so that the master device are able to switch back to the slave device and allow the other master device to communicate with it.

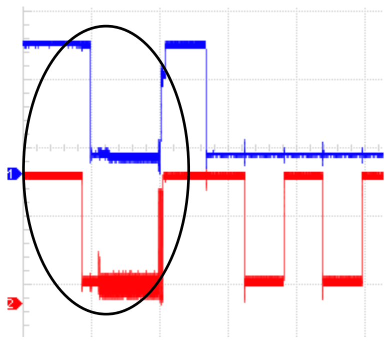


Figure 4.1

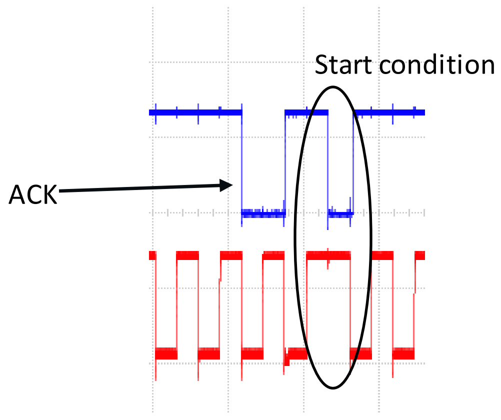


Figure 4.2

From Figure 4.1, a short time of low signal with noise was observed before the start condition was generated. This condition was not related with the start bit, this uncertainty signal was caused by during the configuration of the GPIO port. This situation was detected in every experiment as the signal capture of every experiment was started with the configuration of the GPIO port and I2C. Figure 4.2 shows a start condition which was generated after a byte of data. The low signal in SDA was caused by the acknowledgement of the receiver.

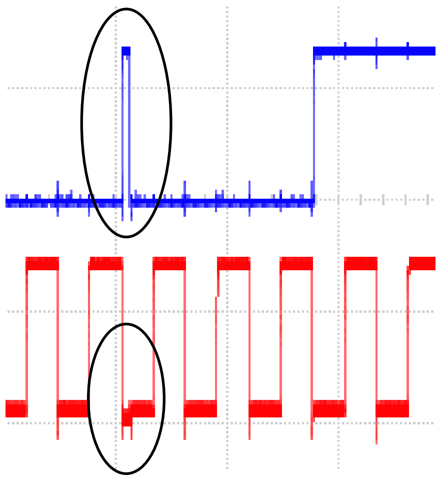


Figure 4.3

In the waveform some of the glitch like wave was observed as shown in the Figure 4.3. This glitch was caused by the clock stretching from the slave device. It only happen between transmissions of two bytes data to prevent overrun and underrun error. For example, during data transmission from master device to slave device, after the slave device received the first byte of data, the slave device actually need time to read the data from the Data Register, it consumed time. So, the slave device will pull the SCL to low to hold the data from transmitting to prevent overrun error occur.

From the GPIO configuration, no pull up and pull down was selected as external pull up voltage was used.

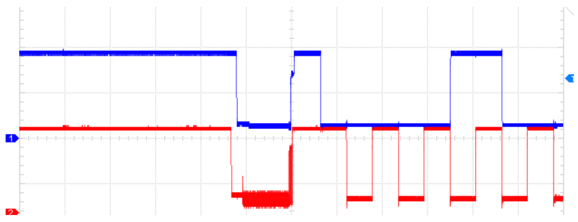


Figure 4.4

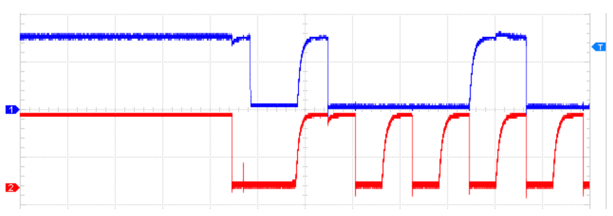


Figure 4.5

Figure 4.4 shown the signals with the external pull up voltage source while Figure 4.3 shown the signals with internal pull up source. The reason may cause the shark like wave shown in Figure 4.5 was the internal pull up source not fast enough to charge up the voltage. As the maximum allowed injected current for the I/O pin was only 25mA which given in the datasheet
(http://www.st.com/web/en/resource/technical/document/datasheet/DM00071990.pdf), while the external pull up voltage was 3V, by using Ohm’s law calculation R = V/I, 120 Ohms was the minimum resistor used for the experiment. However, 200 Ohm resistor was used to reduce the power source.

In the transmission using DMA, the data need to be send to the slave device were store in a transmit buffer memory. In this experiment, the data was transmitted were 0x17 and 0xC. After generated the start condition, the slave address were send manually by setting the Data Register of the master device. So that the TxE flag was set after sent the address of the slave device. When TxE flag was set, the data in the buffer memory will automatically sent the according data 0x17 to the Data Register of the master device and transmit to the slave device. When the TxE flag was set, the second data 0xC was transmitted.

For the experiment to transmit “HELLO” message data from one memory to another memory, the characters in the message were converted to binary form according to the America Standard Code II (ASCII) code.

5. Conclusion

In communication between devices using I2C interface, there is a relationship between master and slave devices. Master device take control of the bus line, it decide which slave device to communicate with by sending the correct address to the slave device. The master master then can decide to send or receive data from the slave. However only one master and and one slave can be communicate in a single bus line. A start condition was generated by the the master device indicated the start of the communication and a stop condition was generated also by the master device indicated the stop condition.

6. Reference

IndianYouthful, (2013). I2C EEPROM Interfacing with STM32F4 Discovery
http://www.electroons.com/blog/2013/01/hello-world/

TILZ0R (2014). LIBRARY 09- I2C FOR STM32F4
http://stm32f4-discovery.com/2014/05/library-09-i2c-for-stm32f4xx/

I2C Info – I2C Bus, Interface and Protocol. I2C Bus Specification
http://i2c.info/i2c-bus-specification