AN13238
How to Trigger and Detect Transmission Errors for LPC51U68 I2C

Rev. 0 — 04/2021

Application Note

Contents

1 Introduction

The LPC51U68 based on Arm® Cortex®-M0+ is a low-cost, low-power consumption, 32-bit Micro Controller Unit (MCU) family. It operates at frequencies of up to 100 MHz and supports up to 256 KB on-chip flash memory and up to 96 KB total SRAM composed of up to 64 KB main SRAM, plus an additional 32 KB SRAM. The on-chip peripherals in LPC51U68 includes one DMA controller, 48 General-Purpose I/O (GPIO) pins , one CRC engine, one 12-bit ADC, one 32-bit Real-Time Clock (RTC), one multiple-channel Multi-Rate 24-bit Timer (MRT), one Windowed WatchDog Timer (WWDT), eight Flexcomm interfaces which can be selected by software to be a USART, SPI, or I^{2}C interface.

The LPC51U68 I^{2}C bus contains the following features:

- Independent Master, Slave, and Monitor functions.
- Supporting the following bus speeds:
  - Standard mode, up to 100 kbits/s
  - Fast-mode, up to 400 kbits/s
  - Fast-mode Plus, up to 1 Mbits/s (on pins PIO0_23 and 24 or PIO0_25 and 26 that include specific I^{2}C support)
  - High speed mode, 3.4 Mbits/s as a Slave only (on pins PIO0_23 and 24 or PIO0_25 and 26 that include specific I^{2}C support)
- Supporting both Multi-master and Multi-master with Slave functions
- Supporting multiple I^{2}C slave addresses in the hardware
- Qualifying one slave address with a bit mask or an address range to respond to multiple I^{2}C bus addresses
- 10-bit addressing supported with software assistance
- Supporting System Management Bus (SMBus).
- Separated DMA requests for Master, Slave, and Monitor functions
- Waking up the device from deep-sleep mode, as no chip clocks are required to receive and compare an address as a Slave
- Automatic modes optionally allow less software overhead for some use cases

This document describes how to trigger and detect I^{2}C transmission errors including Start/Stop error, Arbitration Loss, SCL time-out, Event time-out on LPC51U68. I^{2}C transmission errors are triggered by introducing external glitch which is also called as interference.

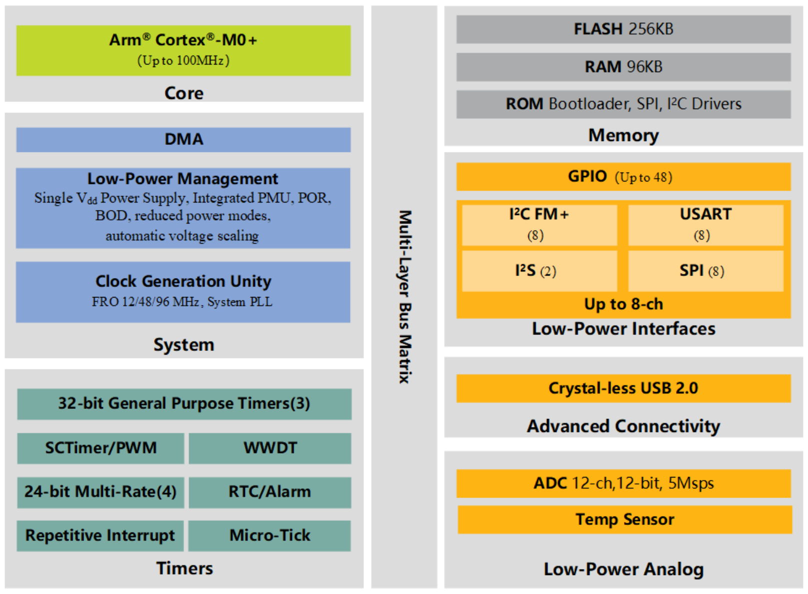


Figure 1. LPC51U68 block diagram

2 I2C bus introduction

I^{2}C is an 8-bit, bidirectional, serial multi-master bus. It supports four modes including:

- Standard mode up to 100 kbit/s
- Fast mode up to 400 kbit/s
- Fast mode Plus up to 1 Mbit/s
- High speed mode up to 3.4 Mbit/s

As shown in Figure 2, I^{2}C uses a Serial Clock Line (SCL) and a Serial Data Line (SDA) for data transfer. The SCL and SDA of master devices and slave devices are connected to the SCL and SDA of I^{2}C bus. For more details about I^{2}C bus specification, see *I^{2}C-bus specification and user manual* (document UM10204).

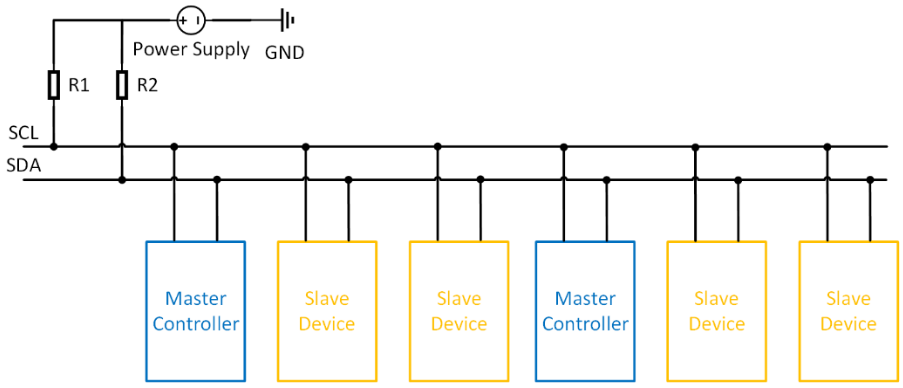


Figure 2. Multi-Master I^{2}C bus architecture

The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function. The master device initiates a transfer, generates clock signal, addresses the slave device, and terminates a transfer. Figure 3 illustrates the timing of I^{2}C bus data transmission.

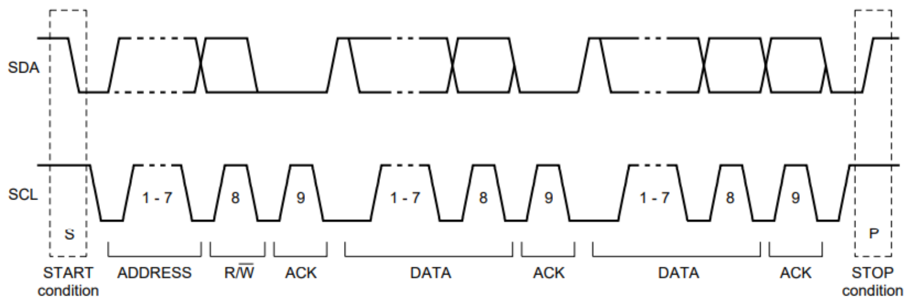


Figure 3. I^{2}C Bus Data Transmission

As shown in Figure 4, I^{2}C transactions begin with a START (S) and are terminated by a STOP (P). 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.

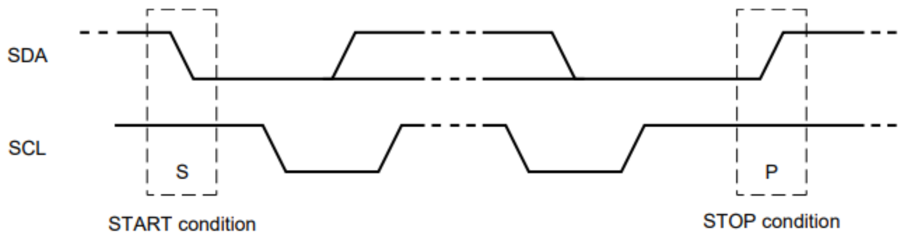


Figure 4. I^{2}C START and STOP conditions

3 System overview

As shown in Figure 5, the system described in this document consists of three parts including one master, one slave and one glitch generator.

The master and slave perform normal data transfer. They are implemented by two LPCXpresso51U68 (OM40005) evaluation boards.

The glitch generator is used to interfere with normal I^{2}C data transmission, resulting in transmission errors including Start/Stop error, Arbitration Loss, SCL time-out, Event time-out. The glitch generator is implemented by LPCXpresso55s69 (Revision A) evaluation board.

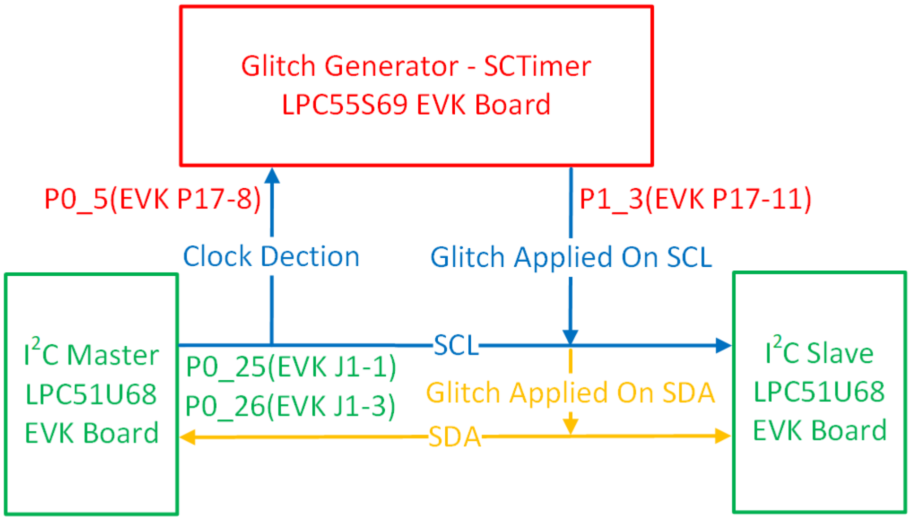


Figure 5. System block diagram with hard connection

The glitch generator is implemented by the state machine generated by LPC55S69 SCTimer.

NOTE

The glitch generator detects the serial clock on the SCL line and applies glitch on the SCL line or the SDA line according to the type of I²C transmission errors.

For specific details of I²C transmission errors, see I2C transmission error type.

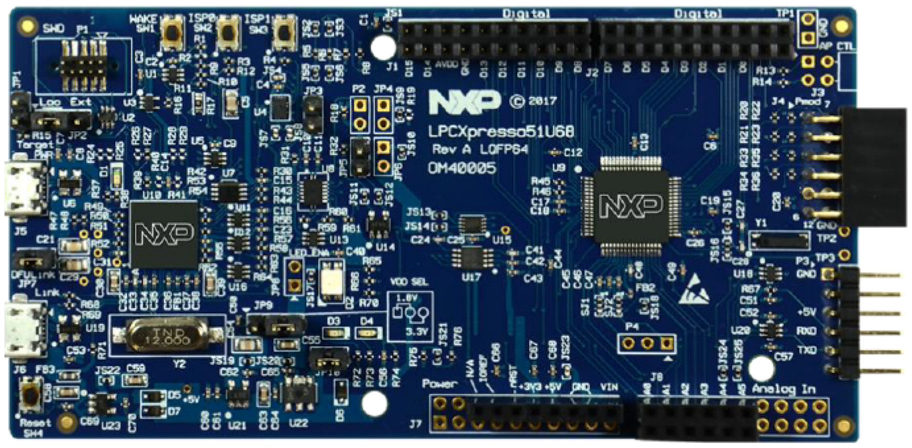


Figure 6. LPCXpresso51U68 evaluation board

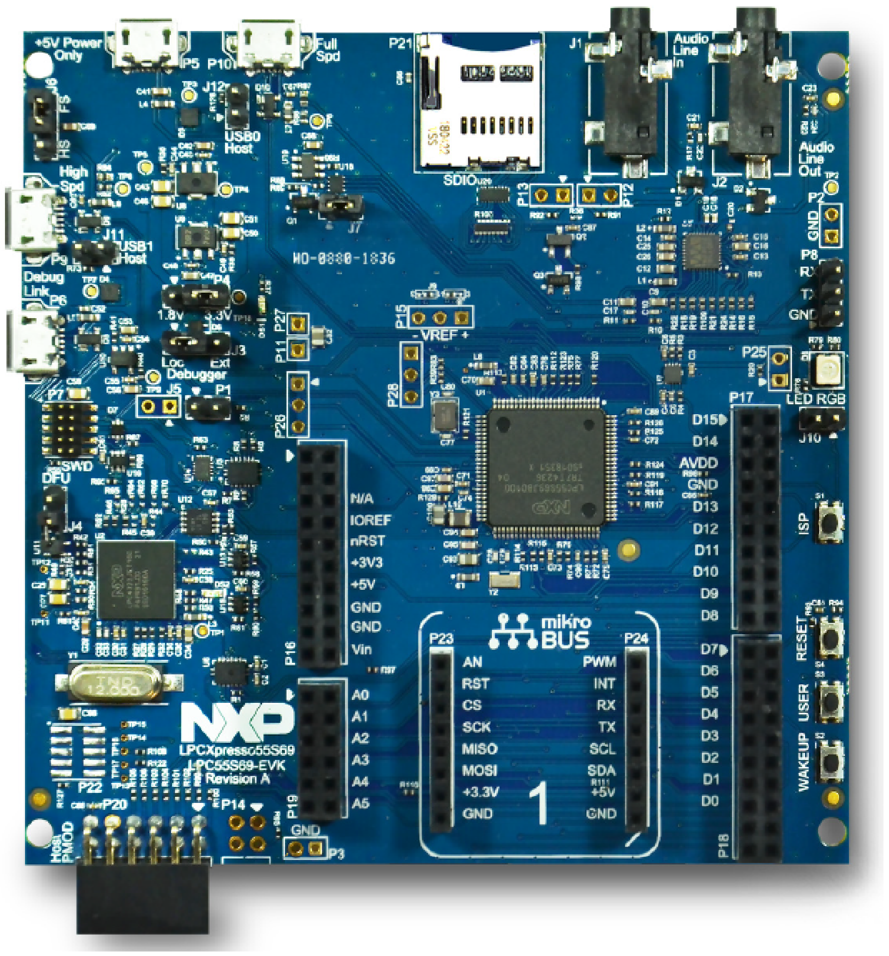


Figure 7. LPCXpresso55S69 evaluation board

4 Hardware configurations

Table 1, Table 2, and Table 3 describe the detailed hardware configurations of I^{2}C master, I^{2}C slave and glitch generator.

Table 1. I^{2}C Master hardware configurations

Part	Port	Pin	Mark on Evk board	Alternative function	Description
LPC51U68	0	25	J1-1	FC4_RTS_**SCL**_SSEL1	SCL line
	0	26	J1-3	FC4_CTS_**SDA**_SSEL0	SDA line
	0	29	J2-5	**PIO0_29**	Start/Stop error indicator LED
	0	2	J8-1	**PIO0_2**	Event time-out indicator LED
	0	3	J8-3	**PIO0_3**	other errors indicator LED

Table continues on the next page...

Table 1. I^{2}C Master hardware configurations (continued)

Part	Port	Pin	Mark on Evk board	Alternative function	Description
	1	9	J8-5	**PIO1_9**	SCL time-out indicator LED
	1	10	J8-8	**PIO1_10**	Arbitration Loss indicator LED

Table 2. I^{2}C Slave hardware configurations

Part	Port	Pin	Mark on Evk board	Alternative function	Description
LPC51U68	0	25	J1-1	FC4_RTS_**SCL**_SSEL1	SCL line
	0	26	J1-3	FC4_CTS_**SDA**_SSEL0	SDA line

Table 3. I^{2}C Glitch generator hardware configurations

Part	Port	Pin	Mark on Evk board	Alternative function	Description
LPC55S69	0	5	P17-8	**SCT0_GPI5**	Detect clock on SCL line
	1	3	P17-11	**SCT0_OUT4**	Apply glitch on SCL line or SDA line

5 I2C transmission error type

5.1 Start/Stop error

According to *I^{2}C-bus specification and user manual* (document UM10204), all transactions begin with a START (S) and are terminated by a STOP (P). The bus is considered to be busy after a START condition. The bus is considered to be free again a certain time after a STOP condition. In other words, if a Start or Stop condition is detected when bus is busy, it can be considered as a **Start/Stop error**.

5.2 Arbitration loss

According to *I^{2}C-bus specification and user manual* (document UM10204), arbitration which is only applicable to multi-master ensures that only one master is allowed to control bus and the data transfer is not corrupted, if more than one master simultaneously tries to control the bus. After the Start condition being issued, arbitration is then required to determine which master will complete its transmission.

The arbitration is performed bit by bit. For a specific bit, each bus master must check whether the bit data to be sent matches the current SDA line status or not. If they match, continue to compare the next bit until the last bit. Otherwise, the master will lose control of the bus, that is, an **Arbitration Loss**.

5.3 Time-out

LPC51U68 I^{2}C supports time-out feature and supports two timeout types, **SCL time-out** and **Event time-out**.

SCL time-out is reflected by the **SCLTIMEOUT** flag in the STAT register and is asserted when the SCL signal remains low longer than the time configured in the `TIMEOUT` register.

Event time-out is reflected by the **EVENTTIMEOUT** flag in the `STAT` register, the time between bus events governs the time-out check. These events include Start, Stop, and all changes on the I^{2}C clock (SCL). This time-out is asserted when the time between any of these events is longer than the time configured in the `TIMEOUT` register.

6 Glitch generation for I2C transmission error

SCTimer is used to implement glitch generator. As the peripheral is only available in NXP MCUs, the state configurable timer (SCTimer/PWM) is available on all LPC55S69 devices. It can work like traditional timer as a timer/counter. However, the state machine function can be used to detect serial clock on SCL line, which is the most outstanding feature of SCTimer/PWM and greatly enhances the configuration and control flexibility of LPC devices.

6.1 Glitch generation for Start/Stop error

According to the description on Start/Stop error in I2C transmission error type, the bus is busy after the master issues a Start condition. Therefore, a Start or Stop condition is generated when the bus is busy is regarded as a Start/Stop error.

Figure 8 shows the state transition diagram for Start/Stop error. **State 3** has checked the second rising edge of the I^{2}C serial clock and then delays for a period of time, during the SCL high level period, pull down SDA. A Start condition is generated when I^{2}C bus is busy since the master is addressing slave devices. Therefore, a Start/Stop error occurs.

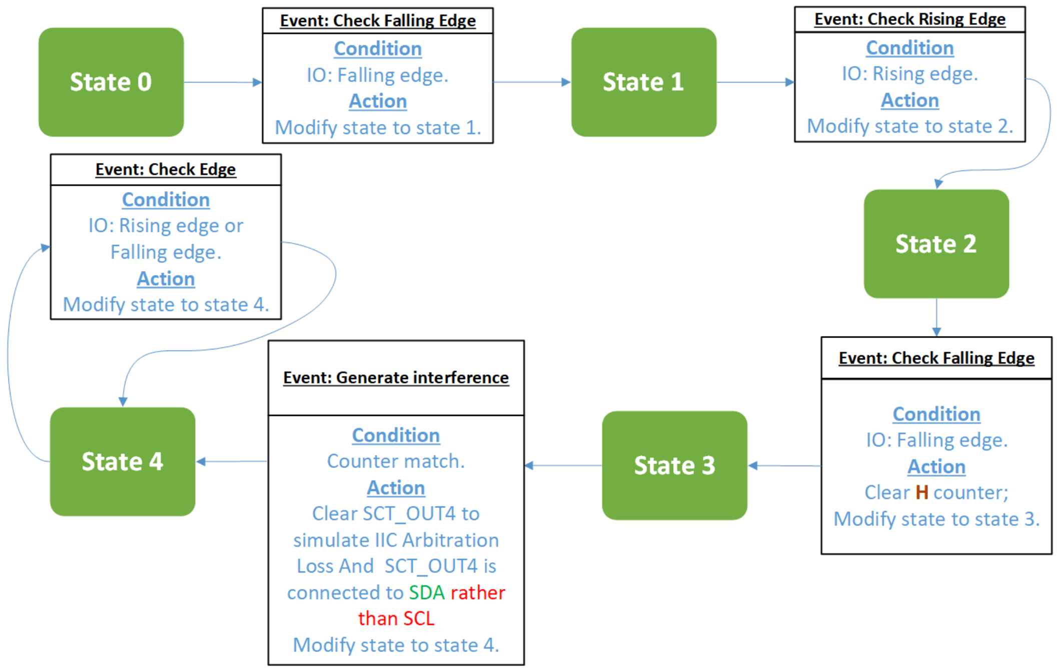


Figure 8. State transition for Start/Stop error

Figure 9 shows the waveforms captured from SCL line, SDA line and glitch generator output. Due to the output of the glitch generator, a Start condition is generated during the high level of the second I^{2}C serial clock, which is consistent with the setting of the state machine.

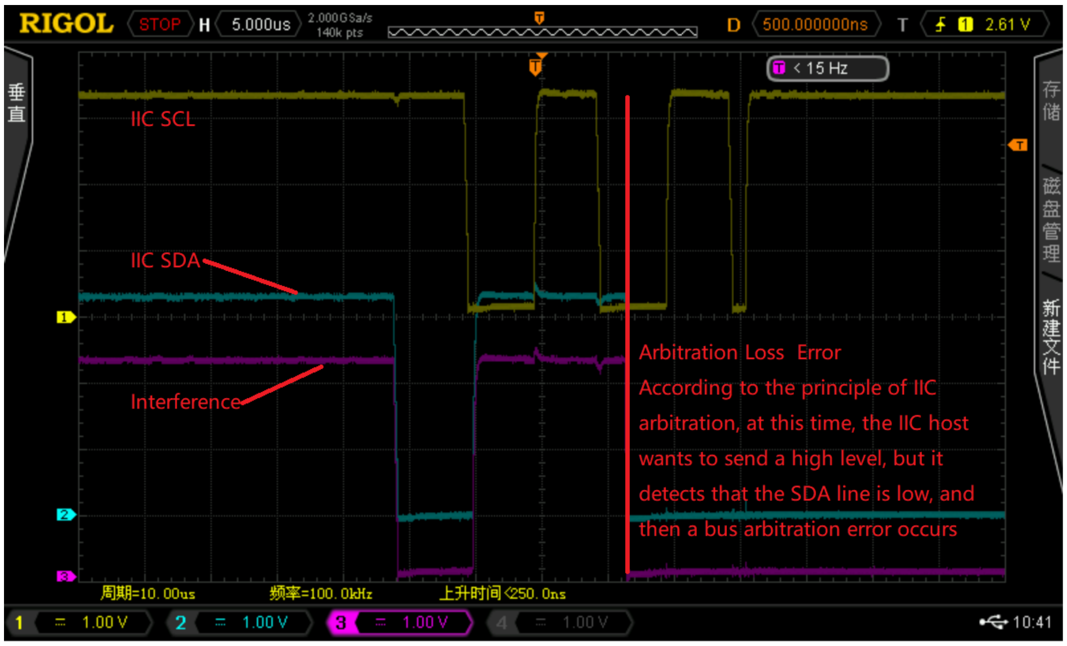


Figure 9. Waveforms for Start/Stop error

6.2 Glitch generation for arbitration loss

According to the description on Arbitration Loss in I2C transmission error type, in the case of multi-master, when certain data bit to be sent by the master is inconsistent with the state of the SDA bus, the master loses control of the bus. At this time, an Arbitration Loss occurs.

Figure 10 shows the state transition diagram for Arbitration Loss. **State 2** has checked the second falling edge of the I^{2}C serial clock and then delays for a period of time, during the SCL low level period, pull down SDA. The slave address is configured as 0x7E and is sent in Most Significant Bit (MSB) mode. In other words, the master wants to send a logic high to SDA during the second serial clock. However, the current state of SDA is logic low. Therefore, an Arbitration Loss occurs.

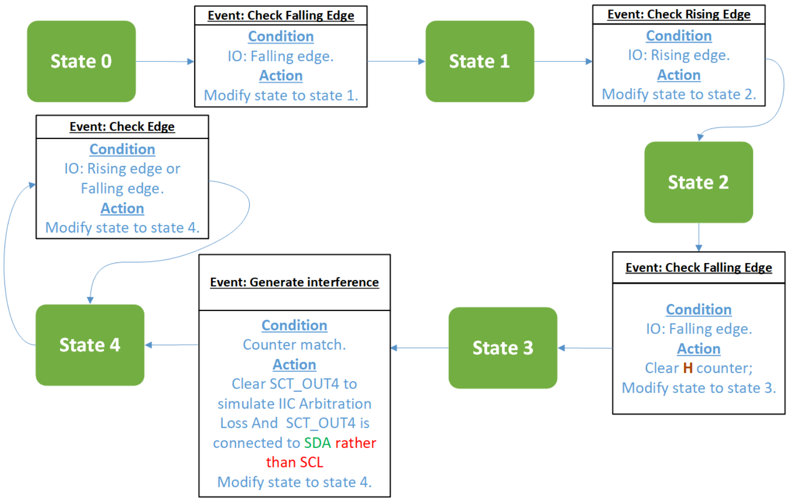
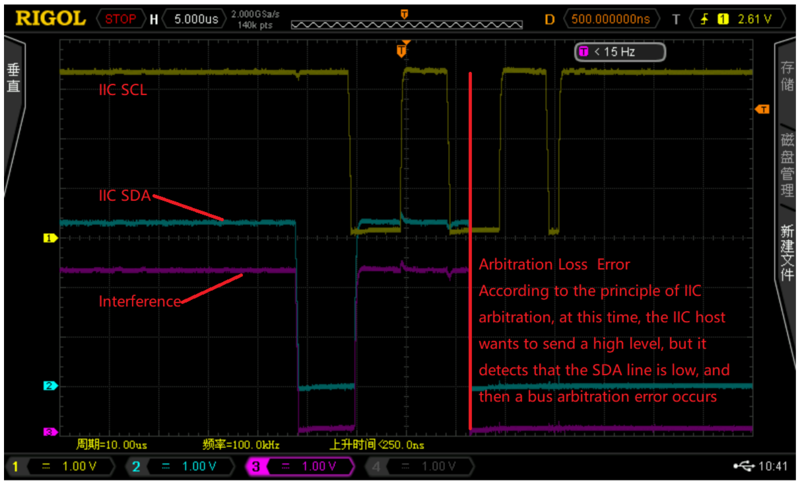


Figure 10. State transition for arbitration loss

Figure 11 shows the waveforms captured from SCL line, SDA line and glitch generator output. Due to the output of the glitch generator, a mismatch between address bit status which master wants to send and current state of SDA line is generated during the low level of the second I^{2}C serial clock, which is consistent with the setting of the state machine.



Arbitration Loss Error
According to the principle of IIC arbitration, at this time, the IIC host wants to send a high level, but it detects that the SDA line is low, and then a bus arbitration error occurs

Figure 11. Waveforms for arbitration loss

6.3 Glitch generation for SCL time-out

According to the description on SCL time-out in I2C transmission error type, when the SCL signal remains low level longer than the time configured in the TIMEOUT register, an SCL time-out occurs.

Figure 12 shows the state transition diagram for SCL time-out. **State 4** has checked the third falling edge of the I^{2}C serial clock and pull down SCL. Since SCL will always be driven low and the duration of low level will be longer than the time configured in the TIMEOUT register, an SCL time-out occurs.

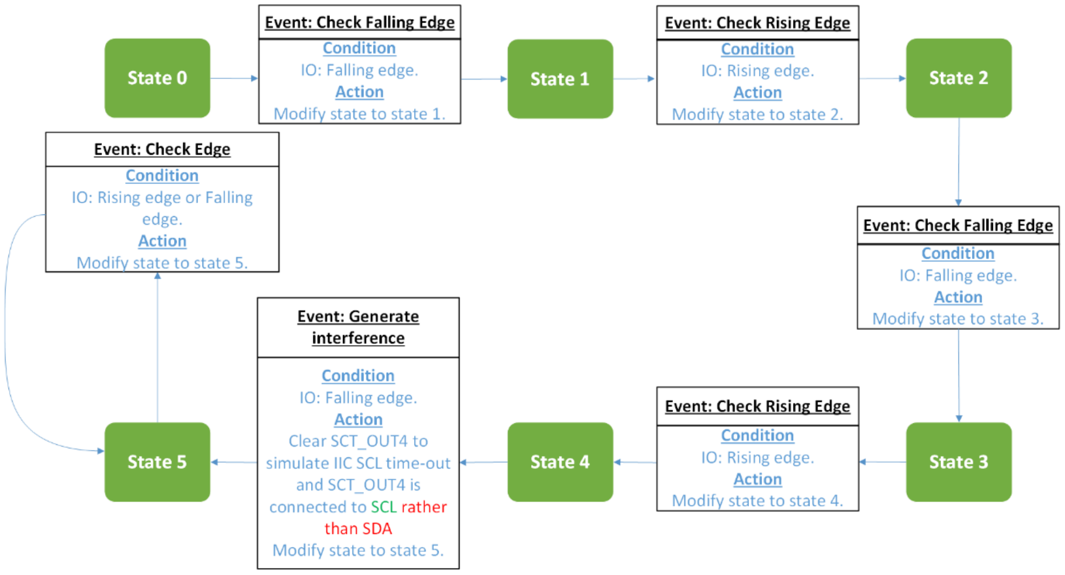


Figure 12. State transition for SCL time-out

Figure 13 shows the waveforms captured from SCL line, SDA line and glitch generator output. Due to the output of the glitch generator, SCL is kept at low level, which is consistent with the setting of the state machine.

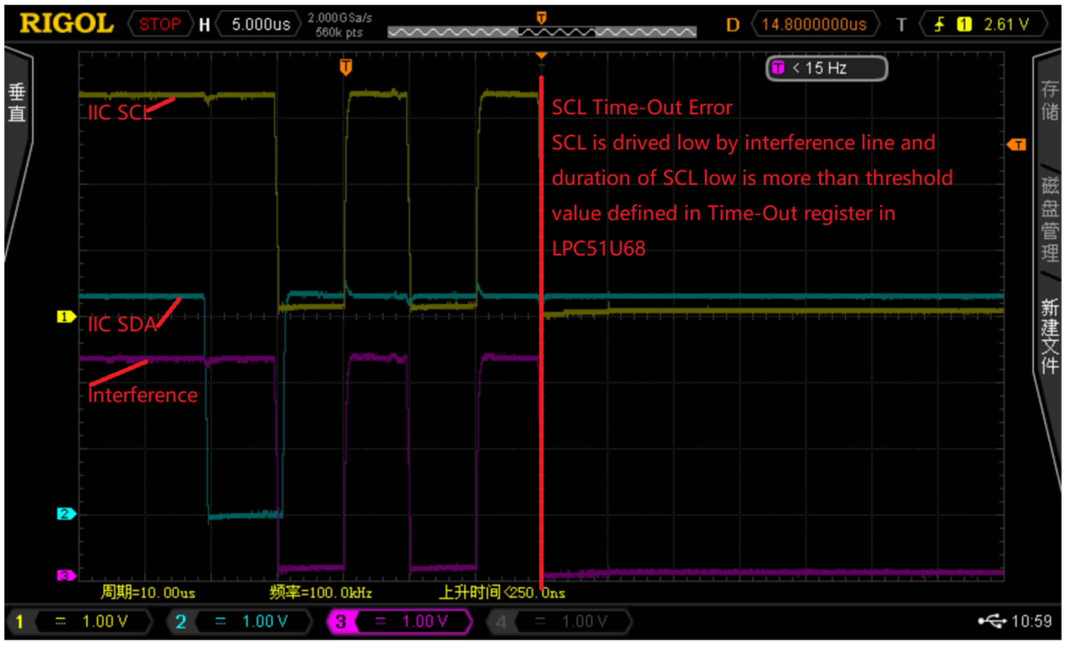


Figure 13. Waveforms for SCL time-out

6.4 Glitch generation for event time-out

According to the description on Event time-out in I2C transmission error type, when the time interval between any of bus events is longer than the time configured in the TIMEOUT register, an Event time-out occurs.

Figure 14 shows the state transition diagram for Event time-out. **State 0** drives SDA to low level when has checked the first falling edge of the I^{2}C serial clock and then delays for a period of time, drives SDA to high level during SCL high level. This is a Stop condition.

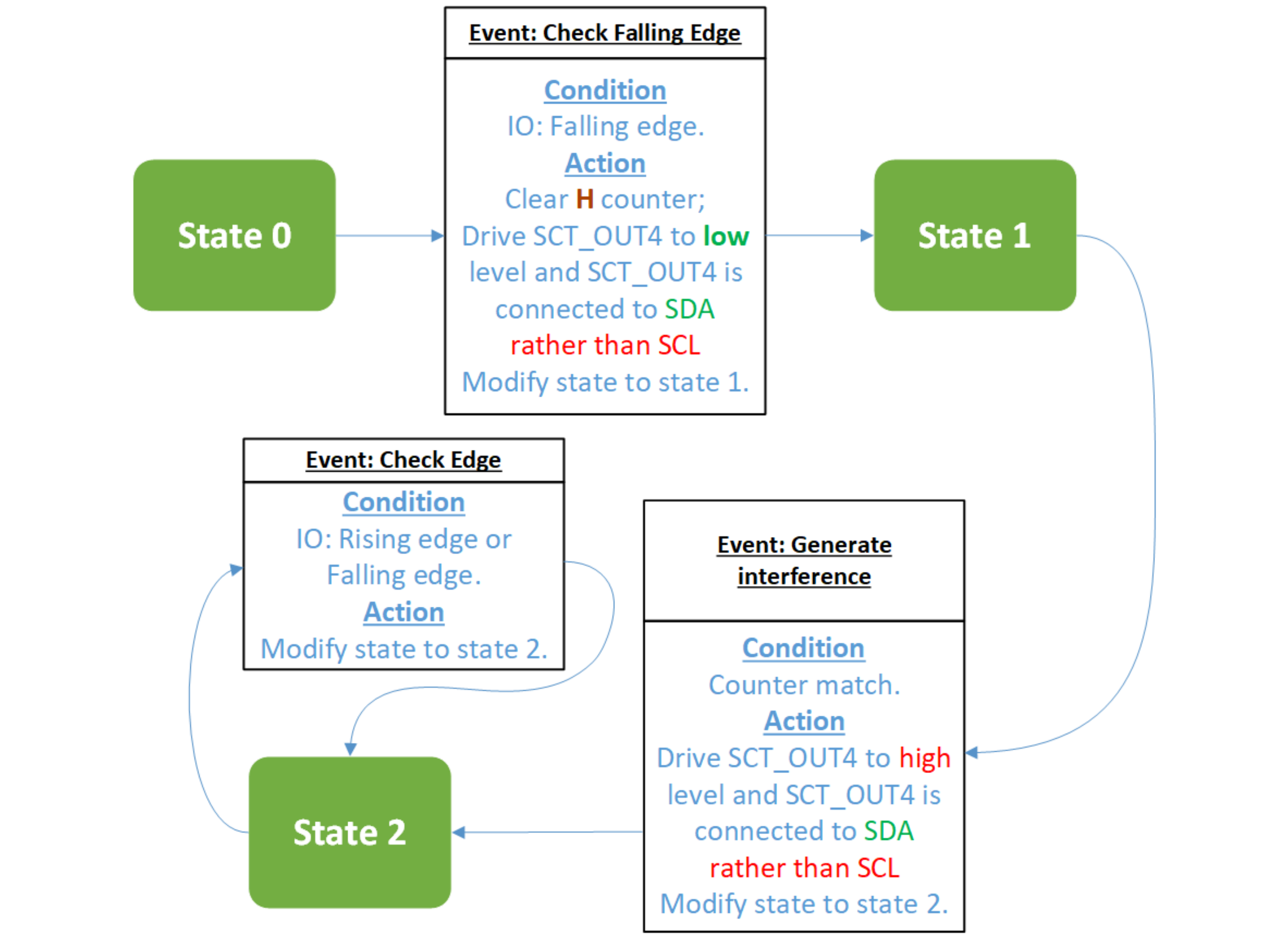

Event: Check Falling Edge
Condition
IO: Falling edge.
Action
Clear H counter;
Drive SCT_OUT4 to low level and SCT_OUT4 is connected to SDA rather than SCL
Modify state to state 1.
State 0
State 1
Event: Check Edge
Condition
IO: Rising edge or Falling edge.
Action
Modify state to state 2.
Event: Generate interference
Condition
Counter match.
Action
Drive SCT_OUT4 to high level and SCT_OUT4 is connected to SDA rather than SCL
Modify state to state 2.
State 2

Figure 14. State transition for event time-out

Figure 15 shows the waveforms captured from SCL line, SDA line and glitch generator output. Due to the output of the glitch generator, the time interval between two adjacent Start/Stop events is 10.68 μs. It is longer than the time with the value of 10.67 μs configured in the TIMEOUT register, as described in Time-out threshold calculation.

Error Detect line in Figure 15 is used to indicate the detection to Start/Stop error and Event time-out. This line is initialized to low level. One rising edge appears on this line when Start/Stop error occurs and one falling edge appears on this line when Event time-out occurs.

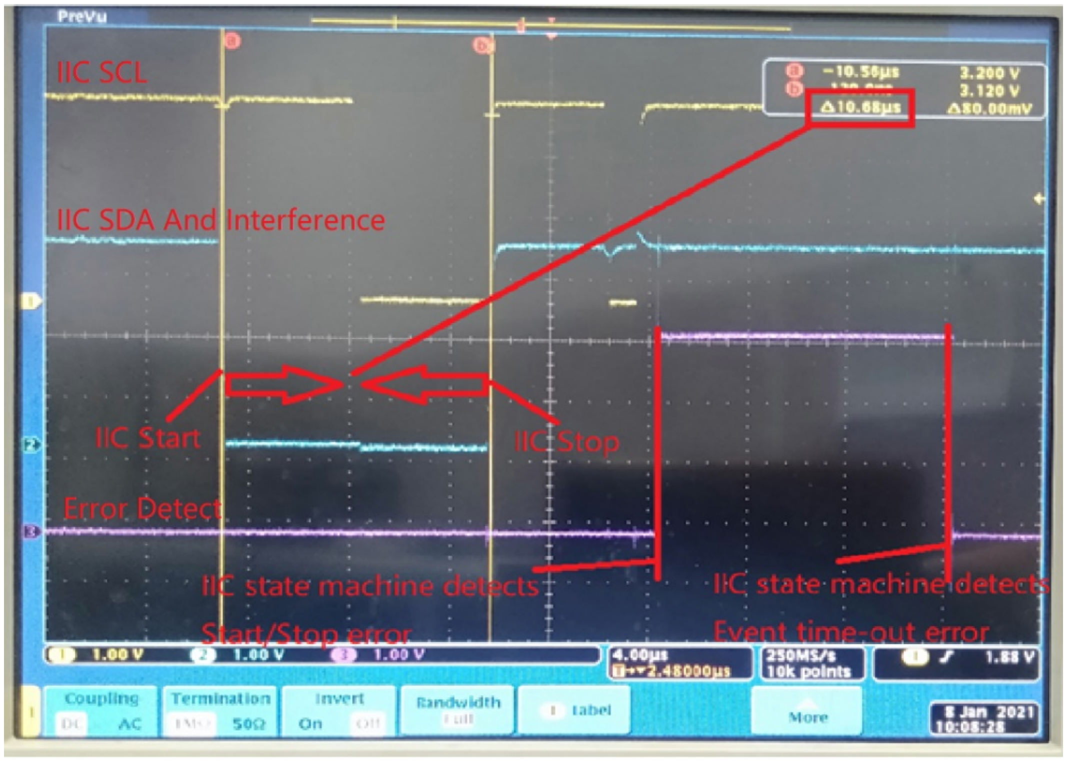


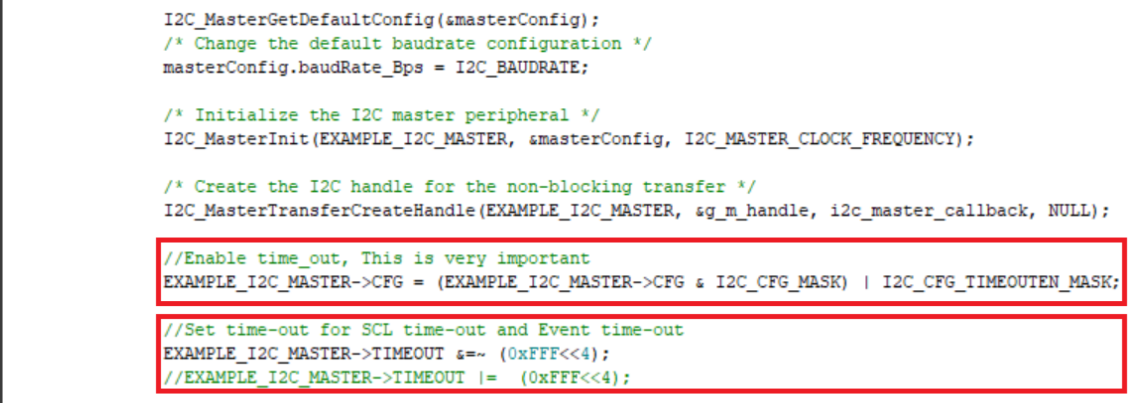
Figure 15. Waveforms for event time-out

7 I2C transmission error detection

7.1 SDK support

The SDK version used in this document is 2.8.2. The SDK with this version doesn't support I^{2}C transmission error detection. To detect I^{2}C transmission errors, follow the steps below and all modifications are specific to I^{2}C master.

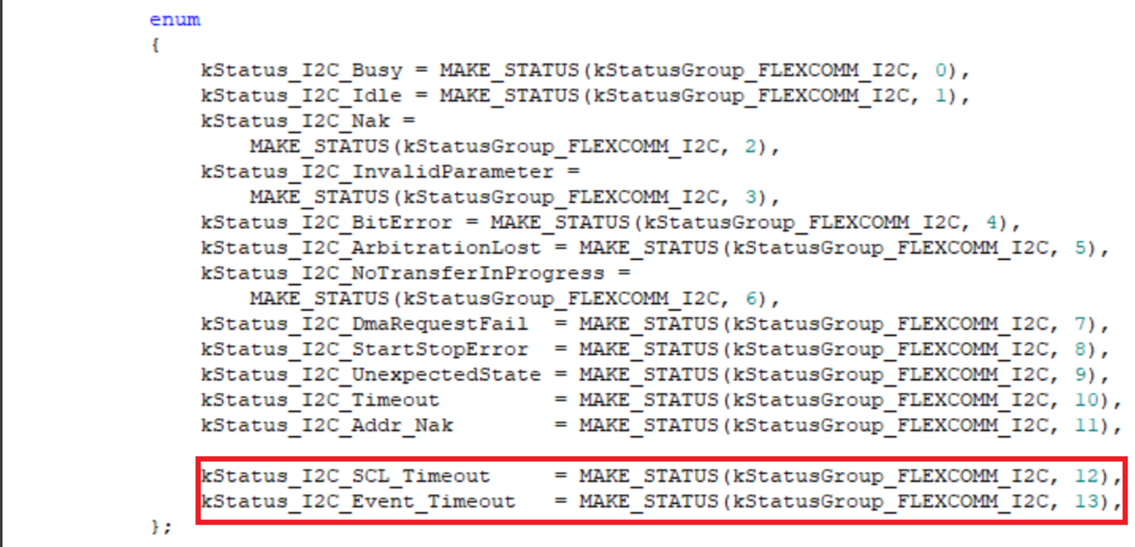
1. Download LPC51U68 SDK version 2.8.2.
2. Open I^{2}C master demo project located in *\boards\lpcxpresso51u68\driver_examples\i2c\interrupt_b2b_transfer\master*.
3. By default, the I^{2}C timeout function is disabled and needs to be enabled by software. In addition, set the timeout threshold of the TIMEOUT register to the minimum.



```
I2C_MasterGetDefaultConfig(&masterConfig);
/* Change the default baudrate configuration */
masterConfig.baudRate_Bps = I2C_BAUDRATE;

/* Initialize the I2C master peripheral */
I2C_MasterInit(EXAMPLE_I2C_MASTER, &masterConfig, I2C_MASTER_CLOCK_FREQUENCY);

/* Create the I2C handle for the non-blocking transfer */
I2C_MasterTransferCreateHandle(EXAMPLE_I2C_MASTER, &g_m_handle, i2c_master_callback, NULL);

//Enable time_out, This is very important
EXAMPLE_I2C_MASTER->CFG = (EXAMPLE_I2C_MASTER->CFG & I2C_CFG_MASK) | I2C_CFG_TIMEOUTEN_MASK;

//Set time-out for SCL time-out and Event time-out
EXAMPLE_I2C_MASTER->TIMEOUT &=~ (0xFFF<<4);
//EXAMPLE_I2C_MASTER->TIMEOUT |=  (0xFFF<<4);
```

Figure 16. Enable time-out and set time-out threshold

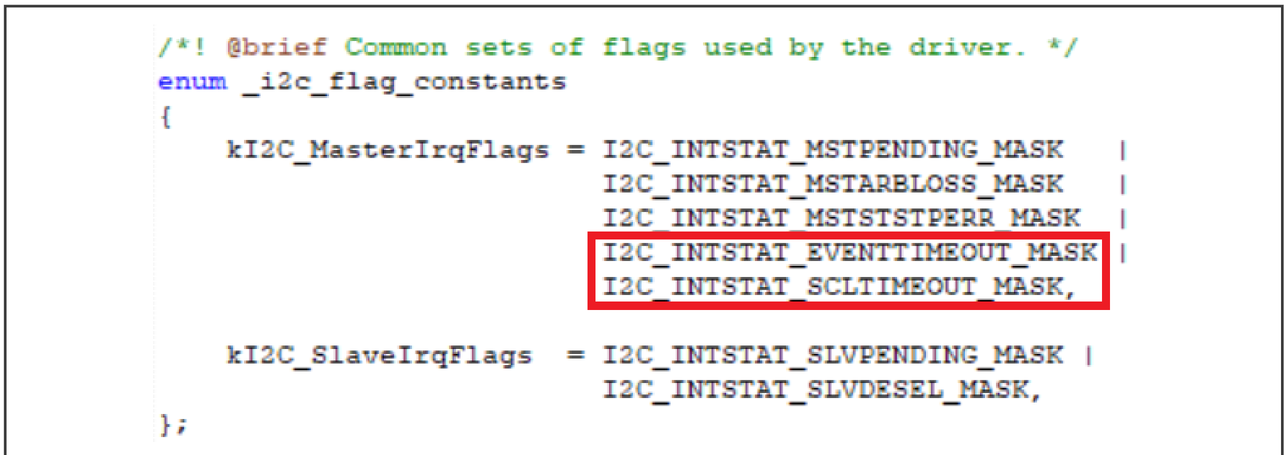
4. The macro `kStatus_I2C_Timeout` in the SDK can't distinguish whether it is SCL time-out or Event time-out. For this reason, two macros are added to distinguish time-out types, as shown in the red square box in Figure 17.



```
enum
{
    kStatus_I2C_Busy = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 0),
    kStatus_I2C_Idle = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 1),
    kStatus_I2C_Nak =
        MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 2),
    kStatus_I2C_InvalidParameter =
        MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 3),
    kStatus_I2C_BitError = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 4),
    kStatus_I2C_ArbitrationLost = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 5),
    kStatus_I2C_NoTransferInProgress =
        MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 6),
    kStatus_I2C_DmaRequestFail  = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 7),
    kStatus_I2C_StartStopError  = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 8),
    kStatus_I2C_UnexpectedState = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 9),
    kStatus_I2C_Timeout         = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 10),
    kStatus_I2C_Addr_Nak        = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 11),

    kStatus_I2C_SCL_Timeout     = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 12),
    kStatus_I2C_Event_Timeout   = MAKE_STATUS(kStatusGroup_FLEXCOMM_I2C, 13),
};
```

Figure 17. Add Macros to distinguish I^{2}C time-out type

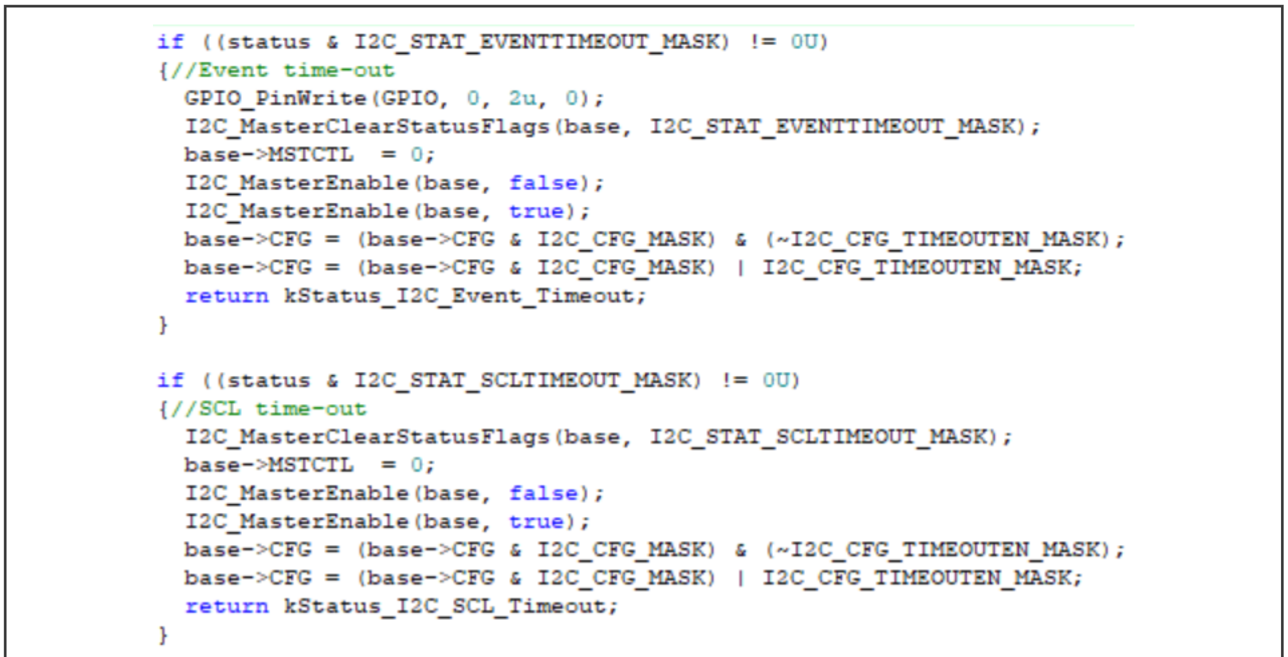
5. By default, the interrupt flag of the I^{2}C master in the SDK does not include SCL time-out and Event time-out. It is necessary to add supports for these two time-out types in the red square box.



```
/*! @brief Common sets of flags used by the driver. */
enum _i2c_flag_constants
{
    kI2C_MasterIrqFlags = I2C_INTSTAT_MSTPENDING_MASK    |
                          I2C_INTSTAT_MSTARBLOSS_MASK    |
                          I2C_INTSTAT_MSTSTSTPERR_MASK   |
                          I2C_INTSTAT_EVENTTIMEOUT_MASK  |
                          I2C_INTSTAT_SCLTIMEOUT_MASK,

    kI2C_SlaveIrqFlags  = I2C_INTSTAT_SLVPENDING_MASK |
                          I2C_INTSTAT_SLVDESEL_MASK,
};
```

Figure 18. I²C master interrupt flag support

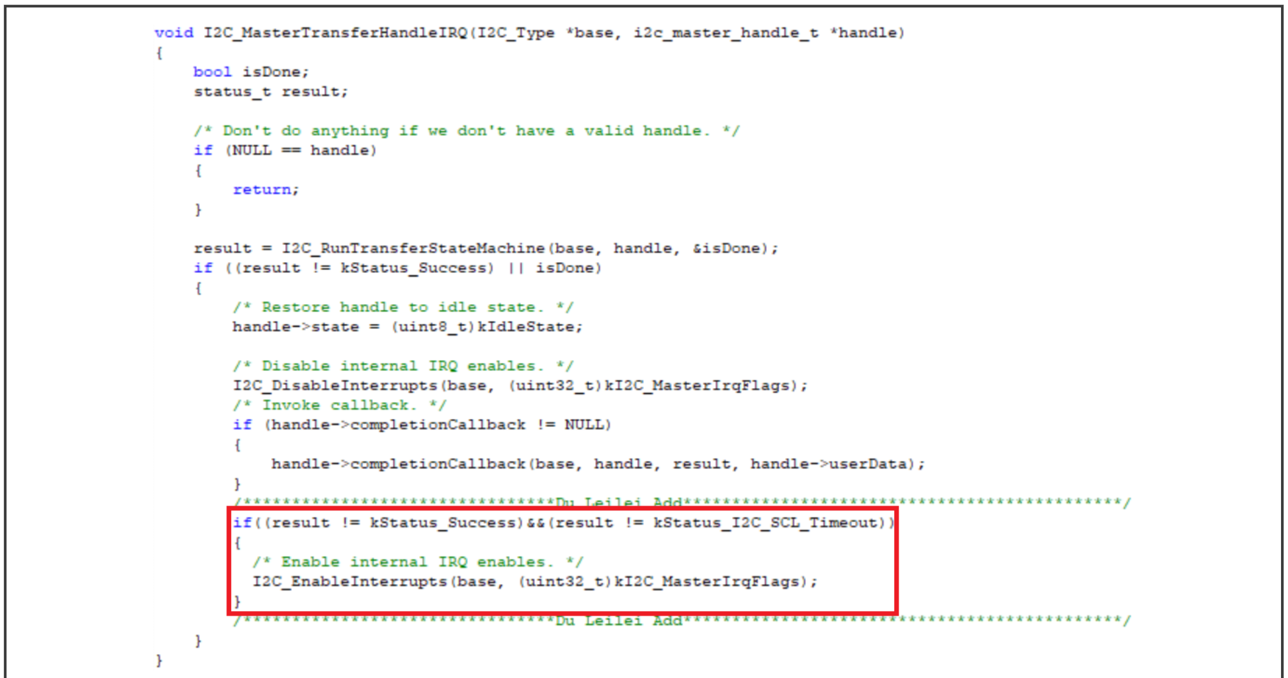
6. Add processing branches for SCL time-out and Event time-out in the `I2C_RunTransferStateMachine` function.



```
if ((status & I2C_STAT_EVENTTIMEOUT_MASK) != 0U)
{//Event time-out
  GPIO_PinWrite(GPIO, 0, 2u, 0);
  I2C_MasterClearStatusFlags(base, I2C_STAT_EVENTTIMEOUT_MASK);
  base->MSTCTL  = 0;
  I2C_MasterEnable(base, false);
  I2C_MasterEnable(base, true);
  base->CFG = (base->CFG & I2C_CFG_MASK) & (~I2C_CFG_TIMEOUTEN_MASK);
  base->CFG = (base->CFG & I2C_CFG_MASK) | I2C_CFG_TIMEOUTEN_MASK;
  return kStatus_I2C_Event_Timeout;
}

if ((status & I2C_STAT_SCLTIMEOUT_MASK) != 0U)
{//SCL time-out
  I2C_MasterClearStatusFlags(base, I2C_STAT_SCLTIMEOUT_MASK);
  base->MSTCTL  = 0;
  I2C_MasterEnable(base, false);
  I2C_MasterEnable(base, true);
  base->CFG = (base->CFG & I2C_CFG_MASK) & (~I2C_CFG_TIMEOUTEN_MASK);
  base->CFG = (base->CFG & I2C_CFG_MASK) | I2C_CFG_TIMEOUTEN_MASK;
  return kStatus_I2C_SCL_Timeout;
}
```

Figure 19. Add processing branches for I²C time-out

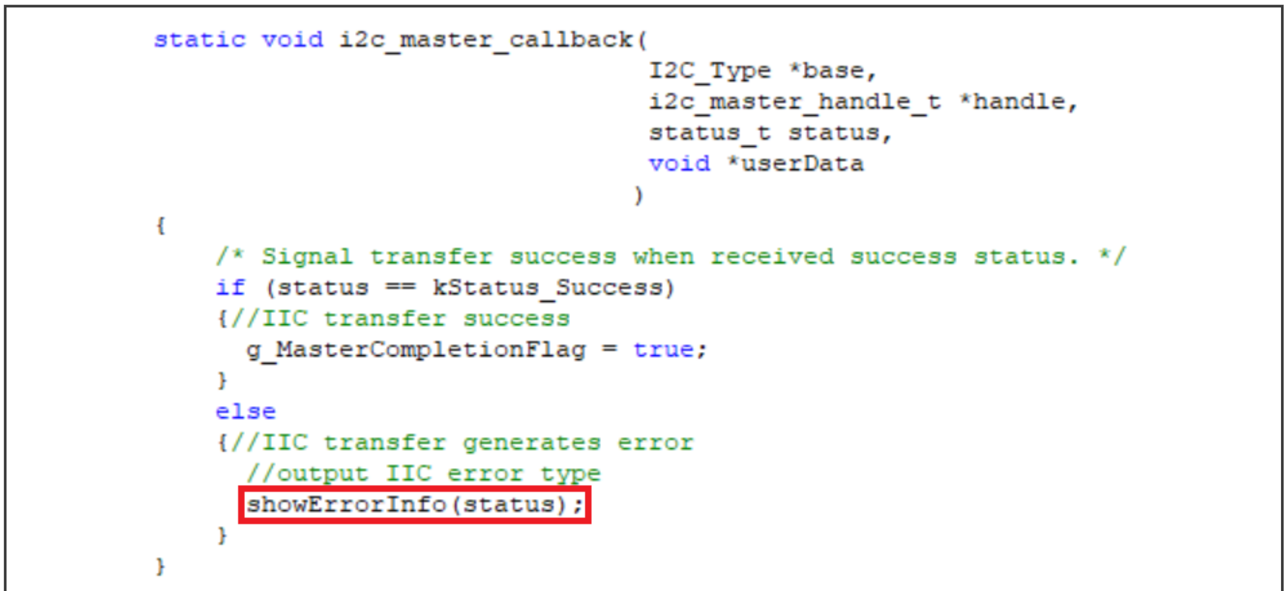
7. This step is very important for detecting I²C Event time-out. If this step is missing, Event time-out can't be detected. The reason for this step is described in Event time-out detection.



```
void I2C_MasterTransferHandleIRQ(I2C_Type *base, i2c_master_handle_t *handle)
{
    bool isDone;
    status_t result;

    /* Don't do anything if we don't have a valid handle. */
    if (NULL == handle)
    {
        return;
    }

    result = I2C_RunTransferStateMachine(base, handle, &isDone);
    if ((result != kStatus_Success) || isDone)
    {
        /* Restore handle to idle state. */
        handle->state = (uint8_t)kIdleState;

        /* Disable internal IRQ enables. */
        I2C_DisableInterrupts(base, (uint32_t)kI2C_MasterIrqFlags);
        /* Invoke callback. */
        if (handle->completionCallback != NULL)
        {
            handle->completionCallback(base, handle, result, handle->userData);
        }
        /********************************Du Leilei Add*********************************************/
        if((result != kStatus_Success)&&(result != kStatus_I2C_SCL_Timeout))
        {
          /* Enable internal IRQ enables. */
          I2C_EnableInterrupts(base, (uint32_t)kI2C_MasterIrqFlags);
        }
        /********************************Du Leilei Add*********************************************/
    }
}
```

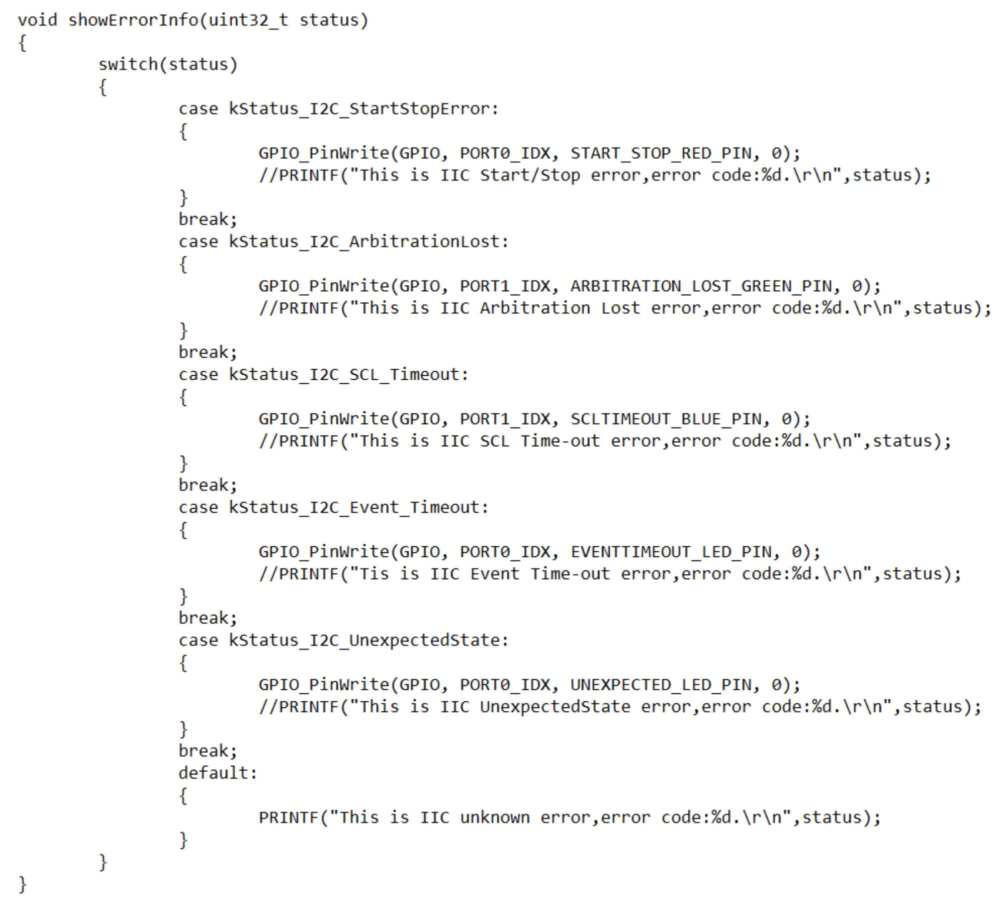
Figure 20. Modifications to detect event time-out

8. Add I^{2}C error output function to I^{2}C interrupt callback function marked as `i2c_master_callback`.



```
static void i2c_master_callback(
                                I2C_Type *base,
                                i2c_master_handle_t *handle,
                                status_t status,
                                void *userData
                               )
{
    /* Signal transfer success when received success status. */
    if (status == kStatus_Success)
    {//IIC transfer success
      g_MasterCompletionFlag = true;
    }
    else
    {//IIC transfer generates error
      //output IIC error type
      showErrorInfo(status);
    }
}
```

Figure 21. Add I^{2}C error display function



```
void showErrorInfo(uint32_t status)
{
        switch(status)
        {
                case kStatus_I2C_StartStopError:
                {
                        GPIO_PinWrite(GPIO, PORT0_IDX, START_STOP_RED_PIN, 0);
                        //PRINTF("This is IIC Start/Stop error,error code:%d.\r\n",status);
                }
                break;
                case kStatus_I2C_ArbitrationLost:
                {
                        GPIO_PinWrite(GPIO, PORT1_IDX, ARBITRATION_LOST_GREEN_PIN, 0);
                        //PRINTF("This is IIC Arbitration Lost error,error code:%d.\r\n",status);
                }
                break;
                case kStatus_I2C_SCL_Timeout:
                {
                        GPIO_PinWrite(GPIO, PORT1_IDX, SCLTIMEOUT_BLUE_PIN, 0);
                        //PRINTF("This is IIC SCL Time-out error,error code:%d.\r\n",status);
                }
                break;
                case kStatus_I2C_Event_Timeout:
                {
                        GPIO_PinWrite(GPIO, PORT0_IDX, EVENTTIMEOUT_LED_PIN, 0);
                        //PRINTF("Tis is IIC Event Time-out error,error code:%d.\r\n",status);
                }
                break;
                case kStatus_I2C_UnexpectedState:
                {
                        GPIO_PinWrite(GPIO, PORT0_IDX, UNEXPECTED_LED_PIN, 0);
                        //PRINTF("This is IIC UnexpectedState error,error code:%d.\r\n",status);
                }
                break;
                default:
                {
                        PRINTF("This is IIC unknown error,error code:%d.\r\n",status);
                }
        }
}
```

Figure 22. I²C error output function

7.2 Time-out threshold calculation

The I²C timeout threshold is determined by timeout register marked as **TIMEOUT**. Figure 23 shows the descriptions of the TIMEOUT register bit field and I²C function clock.

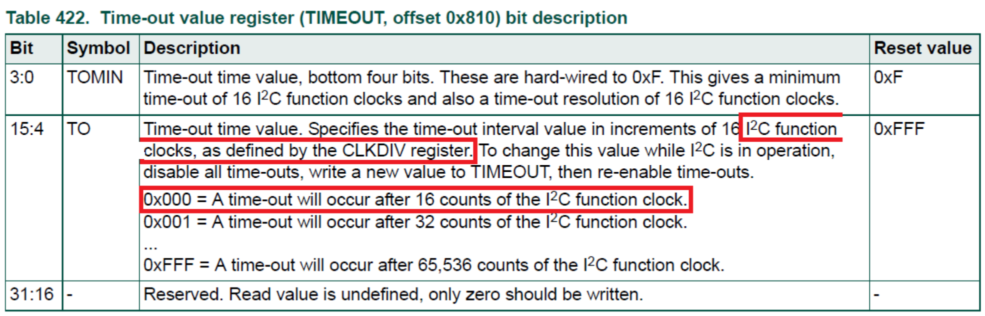


Table 422. Time-out value register (TIMEOUT, offset 0x810) bit description

Bit	Symbol	Description	Reset value
3:0	TOMIN	Time-out time value, bottom four bits. These are hard-wired to 0xF. This gives a minimum time-out of 16 I²C function clocks and also a time-out resolution of 16 I²C function clocks.	0xF
15:4	TO	Time-out time value. Specifies the time-out interval value in increments of 16 I²C function clocks, as defined by the CLKDIV register. To change this value while I²C is in operation, disable all time-outs, write a new value to TIMEOUT, then re-enable time-outs. 0x000 = A time-out will occur after 16 counts of the I²C function clock. 0x001 = A time-out will occur after 32 counts of the I²C function clock. ... 0xFFF = A time-out will occur after 65,536 counts of the I²C function clock.	0xFFF
31:16	-	Reserved. Read value is undefined, only zero should be written.	-

Figure 23. TIMEOUT register

Figure 24 shows the bit field arrangement of the **CLKDIV** register.

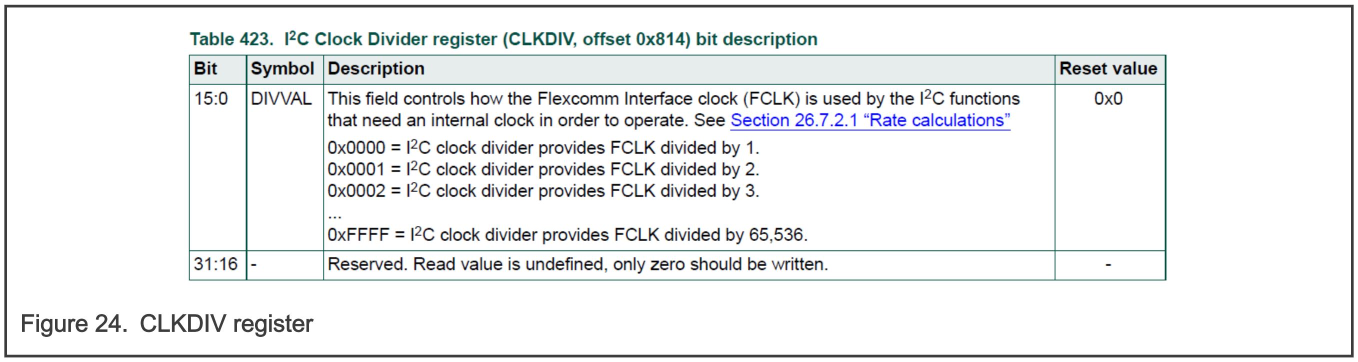


Table 423. I²C Clock Divider register (CLKDIV, offset 0x814) bit description

Bit	Symbol	Description	Reset value
15:0	DIVVAL	This field controls how the Flexcomm Interface clock (FCLK) is used by the I²C functions that need an internal clock in order to operate. See Section 26.7.2.1 "Rate calculations" 0x0000 = I²C clock divider provides FCLK divided by 1. 0x0001 = I²C clock divider provides FCLK divided by 2. 0x0002 = I²C clock divider provides FCLK divided by 3. ... 0xFFFF = I²C clock divider provides FCLK divided by 65,536.	0x0
31:16	-	Reserved. Read value is undefined, only zero should be written.	-

Figure 24. CLKDIV register

As shown in Figure 16, the bit field [15:4] in the TIMEOUT register is set to **0**.

As shown in Figure 23, the timeout threshold is 16 I^{2}C function clocks, and the I^{2}C function clock is determined by the value of the CLKDIV register and the I^{2}C module clock.

As shown in Figure 25, the I^{2}C module clock frequency is set to 12 MHz in this document, and CLKDIV is 7.

As described above, the timeout threshold can be calculated as follows:

- I^{2}C clock divider value is CLKDIV + 1 = 8
- I^{2}C function clock is 12MHz / 8 = 1.5 MHz
- I^{2}C timeout threshold is 16*(1/(1.5 MHz)) = 16 * ((2/3) µs) = **10.67** µs

7.3 Event time-out detection

As described in I2C transmission error type. Event time-out is asserted when the time interval between any of bus events is longer than the time configured in the `TIMEOUT` register. Bus events include Start, Stop, and all changes on the I^{2}C clock (SCL).

A key point is that Event time-out may be accompanied by Start/Stop error, and Start/Stop error precedes Event time-out. Therefore, I^{2}C interrupt can't be disabled after Start/Stop error is detected, otherwise Event time-out will not be detected. This is why the I^{2}C interrupt is enabled in Step 7 in SDK support.

7.4 Detection output

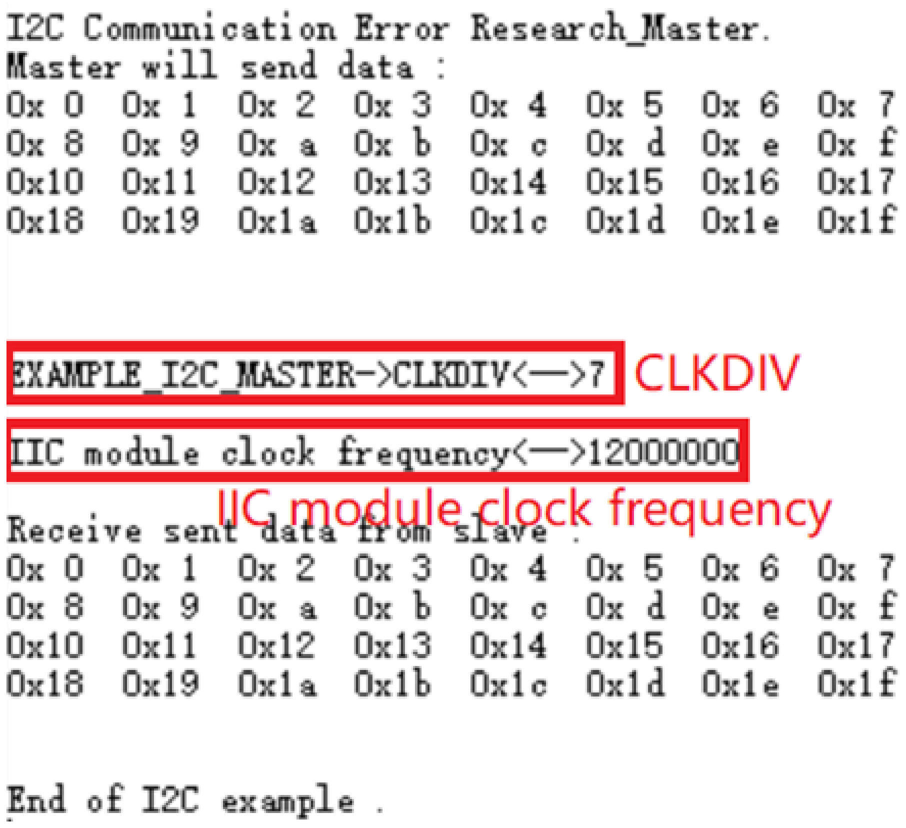
This document supports two ways to observe detection output of I^{2}C transmission error:

- To print the I^{2}C transmission error type to the console on the PC side through UART
- To light up the LEDs connected to different IO pins when different types of I^{2}C transmission errors occur

Table 1 describes the correspondence between I^{2}C transmission error type and IO pin.

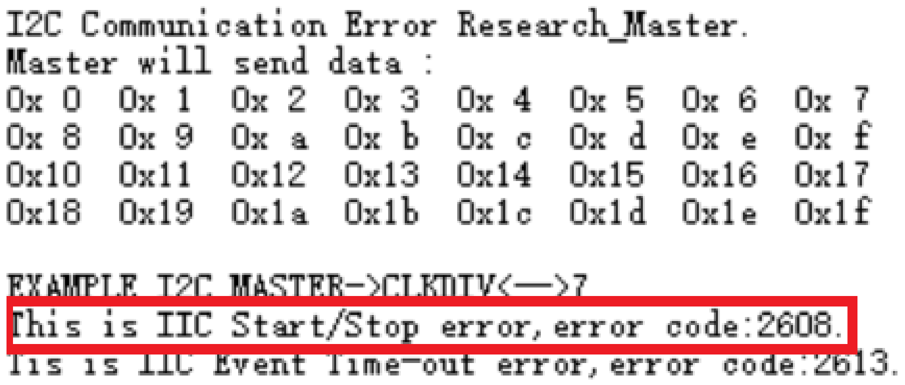
Figure 25 shows the normal transmission between I^{2}C master and I^{2}C slave. Master sends data bytes from `0x00` to `0x1f` and receives data bytes which are the same as the data bytes sent to slave.

Figure 26, Figure 27, Figure 28, and Figure 29 show the detection outputs for Start/Stop error, Arbitration Loss, SCL time-out and Event time-out.



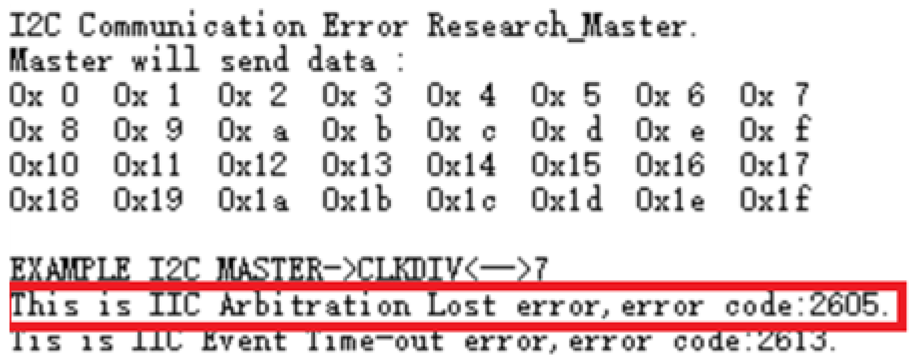
```
I2C Communication Error Research_Master.
Master will send data :
0x 0   0x 1   0x 2   0x 3   0x 4   0x 5   0x 6   0x 7
0x 8   0x 9   0x a   0x b   0x c   0x d   0x e   0x f
0x10   0x11   0x12   0x13   0x14   0x15   0x16   0x17
0x18   0x19   0x1a   0x1b   0x1c   0x1d   0x1e   0x1f


EXAMPLE_I2C_MASTER->CLKDIV<—>7

IIC module clock frequency<—>12000000

Receive sent data from slave :
0x 0   0x 1   0x 2   0x 3   0x 4   0x 5   0x 6   0x 7
0x 8   0x 9   0x a   0x b   0x c   0x d   0x e   0x f
0x10   0x11   0x12   0x13   0x14   0x15   0x16   0x17
0x18   0x19   0x1a   0x1b   0x1c   0x1d   0x1e   0x1f


End of I2C example .
```

Figure 25. Normal I²C transmission



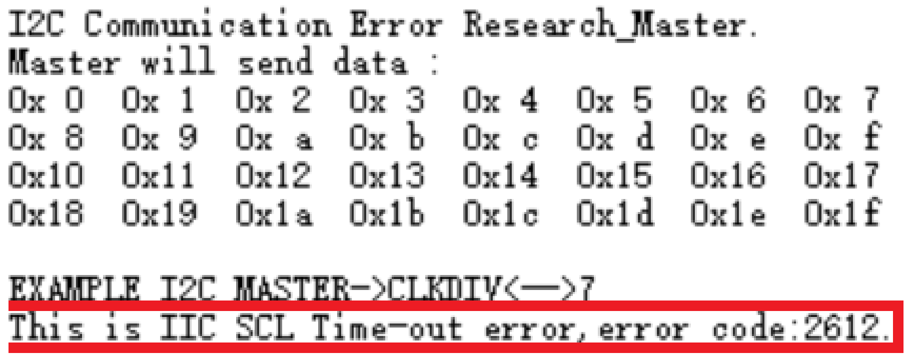
```
I2C Communication Error Research_Master.
Master will send data :
0x 0   0x 1   0x 2   0x 3   0x 4   0x 5   0x 6   0x 7
0x 8   0x 9   0x a   0x b   0x c   0x d   0x e   0x f
0x10   0x11   0x12   0x13   0x14   0x15   0x16   0x17
0x18   0x19   0x1a   0x1b   0x1c   0x1d   0x1e   0x1f

EXAMPLE_I2C_MASTER->CLKDIV<—>7
This is IIC Start/Stop error, error code:2608.
Tis is IIC Event Time-out error, error code:2613.
```

Figure 26. Detection output for Start/Stop error



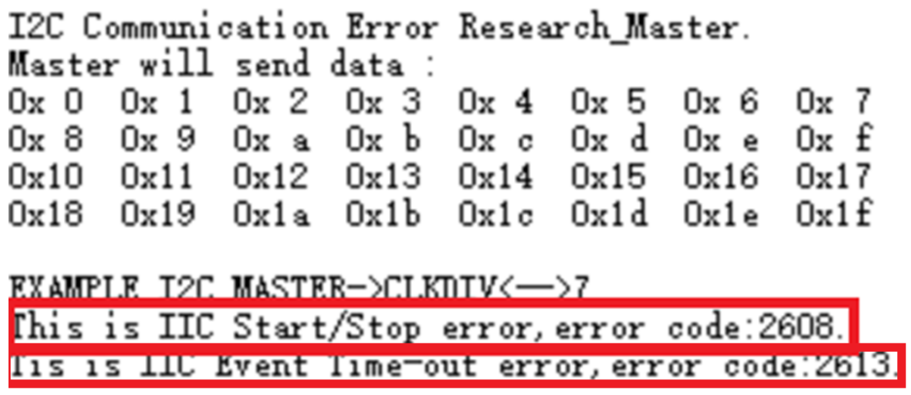
```
I2C Communication Error Research_Master.
Master will send data :
0x 0  0x 1  0x 2  0x 3  0x 4  0x 5  0x 6  0x 7
0x 8  0x 9  0x a  0x b  0x c  0x d  0x e  0x f
0x10  0x11  0x12  0x13  0x14  0x15  0x16  0x17
0x18  0x19  0x1a  0x1b  0x1c  0x1d  0x1e  0x1f

EXAMPLE I2C MASTER->CLKDIV<—>7
This is IIC Arbitration Lost error, error code:2605.
Tis is IIC Event Time-out error, error code:2613.
```

Figure 27. Detection output for arbitration loss



```
I2C Communication Error Research_Master.
Master will send data :
0x 0  0x 1  0x 2  0x 3  0x 4  0x 5  0x 6  0x 7
0x 8  0x 9  0x a  0x b  0x c  0x d  0x e  0x f
0x10  0x11  0x12  0x13  0x14  0x15  0x16  0x17
0x18  0x19  0x1a  0x1b  0x1c  0x1d  0x1e  0x1f

EXAMPLE I2C MASTER->CLKDIV<—>7
This is IIC SCL Time-out error, error code:2612.
```

Figure 28. Detection output for SCL time-out



```
I2C Communication Error Research_Master.
Master will send data :
0x 0  0x 1  0x 2  0x 3  0x 4  0x 5  0x 6  0x 7
0x 8  0x 9  0x a  0x b  0x c  0x d  0x e  0x f
0x10  0x11  0x12  0x13  0x14  0x15  0x16  0x17
0x18  0x19  0x1a  0x1b  0x1c  0x1d  0x1e  0x1f

EXAMPLE I2C MASTER->CLKDIV<—>7
This is IIC Start/Stop error, error code:2608.
Tis is IIC Event Time-out error, error code:2613.
```

Figure 29. Detection output for event time-out

8 How to use LPC51U68 I2C glitch generation and detection demo

To use this demo, perform the following steps:

1. Establish hardware environment for this demo as described in Hardware configurations.
2. Open I^{2}C master code project located in *I2C_Transfer_With_Error_Detection\boards\lpcxpresso51u68\driver_examples\i2c\interrupt_b2b_transfer\master\mdk* and compile it to generate an executable file.

3. Open I^{2}C slave code project located in *I2C_Transfer_With_Error_Detection\boards\lpcxpresso51u68\driver_examples\i2c\interrupt_b2b_transfer\slave\mdk* and compile it to generate an executable file.
4. Open glitch generator code project located in *I2C_Glitch_Generator\boards\lpcxpresso55s69\driver_examples\sctimer\edge_check_match\cm33_core0\mdk* and compile it to generate an executable file.
5. Program executable file for I^{2}C master, I^{2}C slave and glitch generator to corresponding evaluation boards mentioned in System overview.
6. Open two UART terminals on PC side corresponding to master and slave. The settings are as below:
   - Baudrate:115200
   - Data bits:8
   - Parity Check: No
   - Flow Control: No
7. Run the following in sequence:
   a. Glitch generator application
   b. I^{2}C slave application
   c. I^{2}C master application

9 Summary

This document introduces:

- I^{2}C bus
- composition of the system
- role of each component
- primary transmission errors of I^{2}C bus including Start/Stop error, Arbitration Loss, SCL time-out and Event time-out
- how to design glitch generator using SCTimer state machine
- how to detect I^{2}C transmission errors triggered by glitch
- how to use generation and detection demo for the I^{2}C transmission error

Demo code projects are attached to this document. For more technical details, see AN13238SW.

Date of release: 04/2021
Document identifier: AN13238