AN13952

How to Use I3C in LPC86x

Rev. 1 — 26 July 2023 Application note

Document Information

Information	Content
Keywords	I3C, LPC86x
Abstract	This application note describes how to use I3C in LPC86x.

1 Introduction

This application note describes how to use I3C on LPC86x. It mainly includes the basic knowledge of I3C, the introduction of I3C peripherals of LPC86x, and the introduction of I3C SDK routines.

LPC86x is an Arm Cortex-M0+ based, low-cost, 32-bit MCU family operating at CPU frequencies of up to 60 MHz. LPC86x supports up to 64 kB of flash memory and 8 kB of SRAM. The peripheral complement of LPC86x includes a CRC engine, one I^2C -bus interface, one I3C-MIPI bus interface, up to three USARTs, up to two SPI interfaces, one multi-rate timer, one self-wake-up timer, two FlexTimers, one DMA, one 12-bit ADC, one analog comparator, function-configurable I/O ports through a switch matrix, one input-pattern match engine, and up to 54 general-purpose I/O pins.

2 I3C overview

2.1 I3C introduction

The I^2C bus has been popular in the embedded field for many years. Many kinds of sensors use the I^2C interface. I^2C is often used as a communication bus between the sensor and the processor. A processor can communicate with many I^2C devices and the communication bus has only two lines. This method of communication has been used for a long time.

With the development of the industry, the requirements for the communication method has become higher. Firstly, lower power consumption is required. Secondly, faster speed is needed. In addition, the interrupt signal should be implemented in the communication lines, without an additional interrupt signal line.

The I3C interface has been developed to address these sensor integration concerns, as well as those historically encountered while using I^2C, SPI, and UART in any product, by providing a fast, low-cost, low-power, managed, two-wire digital interface. The I3C interface is intended to improve upon the features of the I^2C interface, preserving backward compatibility. Compared with I^2C and UART ports, I3C has a higher communication speed. Compared with SPI, I3C requires fewer signal lines and supports in-band interrupts.

In the new I3C specification, the names of controller and target are no longer used and they are changed to controller and target.

2.2 I3C features

I3C has the following powerful features:

- Two-wire serial interface (up to 12.5 MHz)
- Can coexist with the I^2C protocol in a network
- Support legacy I^2C messaging
- Single data rate mode messaging
- Higher transmission bandwidth in the high data rate messaging modes
- Multi-controller capability
- In-band interrupt
- Hot-join
- Target reset without additional wires

2.3 I3C key features

There is a detailed introduction and description of I3C in the I3C specification. This application note mainly explains the basic key features.

2.3.1 I3C modes

I3C supports many transfer modes, which makes I3C more flexible.

- It supports **Legacy I^2C** target devices and messages.
- It supports the **I3C Single Data Rate (SDR) mode**, an enhanced version of the I^2C protocol. The SDR mode supports private messages, broadcast messages, and direct messages. Broadcast messages are sent to all targets and direct messages are sent to specific address targets.
- It supports **High Data Rate (HDR) modes**, which include four high-speed modes:
  - It supports the **Dual Data Rate (HDR-DDR) mode**. It uses the same signaling as the SDR mode, but it runs two times faster than SDR, because it operates the data on both edges of the SCL.
  - It supports the **Ternary Symbol Legacy (HDR-TSL) mode**, it uses ternary coding to get higher data rates, and it allows mixing the I^2C and I3C devices.
  - It supports the **Ternary Symbol Pure-bus (HDR-TSP) mode**. It is the same with as the HDR-TSL mode, but it accepts only I3C devices.
  - It supports the **Bulk Transport (HDR-BT) mode**. It uses single-lane, dual-lane, or quad-lane to get the highest possible throughput.

This application note mainly introduces the legacy I^2C mode and the SDR mode. They are the most basic and commonly used modes.

2.3.2 I3C Single Data Rate (SDR) mode

The SDR mode is the default mode of the I3C bus. Other modes and states must be entered from the SDR mode. Common Commands (CCCs), in-band interrupts, and dynamic address assignments are used in the SDR mode.

The I3C SDR mode has similar operations with the I^2C protocol. I3C devices and legacy I^2C devices can coexist in the same I3C network. The SDR mode also has its new features, which are not present in the I^2C protocol.

For the procedures and conditions that I3C shares with I^2C, the SDR mode closely follows the definitions in the I^2C specification. The I^2C traffic from an I3C controller to an I^2C target will be properly ignored by all I3C targets. The I3C traffic from an I3C controller to an I3C target will not be seen by most Legacy I^2C target devices, because the I^2C spike filter is opaque to I3C's higher clock speed.

2.3.3 I3C address header

The address header of I3C is derived from the START signal or the repeated START signal. It is the same with I^2C, it includes 7 bits of address, 1 bit of RnW, and 1 bit of ACK/NACK.

The address header derived from the START signal is arbitrable and uses the open-drain IO mode. The address header derived from the repeated START signal uses the push-pull mode.

On the address header stage, the targets can transmit some requests to the I3C controller, such as in-band interrupt, controller role request, and hot-join request.

2.3.4 Open-drain and push-pull mode

Open-drain and push-pull modes are I/O modes for I3C signal pins. In the open-drain mode, the rising and falling edges of the signal are very slow, which makes it difficult for the protocol to reach high speeds. In the push-pull mode, the I/O has a strong driving ability and it can reach a higher speed.

In the I^2C protocol, the SCL and SDA use the open-drain mode. To obtain a higher baud rate, the push-pull mode is used in the I3C mode.

In the address header stage, the I3C protocol still uses the open-drain mode to accept the request from the target and it allows the arbitration among targets.

In the ninth bit stage, the handshake of controller and target requires two modes to cooperate with each other.

2.3.5 Ninth bit of SDR

The ninth bit of SDR has some roles in different conditions.

- It is the same with the I^2C protocol. The target gives the ACK/NACK response to the controller after receiving the address value. The target can send the ACK with pulling down the SDA signal to respond to the SDR write from the controller. It also sends the ACK with leaving the SDA high-level state.
- The ninth data bit written by the controller is the parity of the preceding eight data bits. The target shall not drive the SDA line for the data written by the controller in the SDR mode. The ninth bit of SDR is called T-bit (T stands for transition).
- The ninth data bit allows target to end a controller read frame.

The ninth bit in the I3C protocol expresses more meanings and implements more functions.

2.3.6 Dynamic address assignment

At every system power-up, the controller assigns the dynamic address to all the targets on the bus. The dynamic address generates a priority ranking for the in-band interrupt.

The device receives the dynamic address from the controller using two methods:

- The controller uses the **Enter Dynamic Address Assignment (ENTDAA)** broadcast command code to assign the dynamic address. The target shall have a 48-bit provisioned ID.
- The controller uses the **Set All Addresses to Static Address (SETAASA)** broadcast command code to request all connected targets to use their static address as their dynamic address.

2.3.7 Common Command Codes (CCCs)

Common Command Codes (CCCs) are I3C's standardized command set. The I3C controller can use different CCC codes to implement specific functions.

There are two main kinds of CCCs: broadcast CCCs and direct CCCs.

Broadcast CCCs are valid for all devices. All devices should respond to the broadcast CCCs. Direct CCCs have address information and they only work on devices that match the address.

In the SDR mode, the CCC always starts with broadcast address 0x7E with writing direction. Table 1 lists some CCCs:

Table 1. Common Command Codes (CCCs)

Command code	CCC type	Command name	Description
0x07	Broadcast	Enter Dynamic Address Assignment (ENTDAA)	The controller has started the Dynamic Address Assignment procedure.
0x29	Broadcast	Set All Addresses to Static Addresses (SETAASA)	The controller tells every target with a static address to use it as the dynamic address.
0x87	Direct set	Set Dynamic Address from Static Address (SETDASA)	The controller assigns a dynamic address to a target with a known static address.
0x8D	Direct set	Get Provisioned ID (GETPID)	Gets the target's provisioned ID.

2.3.8 In-band interrupt

In the I3C protocol, the target can generate in-band interrupts through the clock and data lines, without wasting additional interrupt lines.

To request an in-band interrupt, an I3C target shall emit its address into the arbitrated address header following a START signal. If no START is forthcoming within the bus available condition, then the I3C target may issue a START by pulling the SDA line low and wait for the active controller to pull the SCL low.

The priority level controls the order in in-band interrupt requests. Targets with lower value addresses have higher priority levels in in-band interrupts.

After start, the target shall drive the SDA line with its own address, followed by an RnW bit with a value of 1’b1. If more than one target has issued an IBI request after the same START, then the active controller shall process those IBIs in a priority-level order. The target(s) that lost the arbitration may issue another IBI request, but they shall not do so until after the bus available condition.

2.4 I3C peripheral in LPC86X

A detailed introduction and description of the I3C is in the I3C specification. This application note mainly explains the basic key features.

3 I3C overview in LPC86X

3.1 I3C introduction

LPC86X has one controller/target I3C-MIPI bus interface. The I3C supports DDR. It is supported by the general-purpose DMA controller. The I3C peripheral supports the full I3C feature set, except for HDR Ternary modes (HDR-TSP and HDR-TSL).

3.2 I3C basic configuration

In LPC86X, the I3C reset, clock, interrupt, and DMA pins can be configured.

1. **Clock**

Table 2. Clock

Bit	Symbol	Value	Description	Reset value
23	I3C0	-	Enables clock to I3C	0
		0	Disable	
		1	Enable	

Enable the clock to the I3C in the **SYSAHBCLKCTRL0** register. This enables the register interface and the peripheral function clock. The peripheral clock source-select registers select the function of clock sources for the serial peripherals shown in the following list. The potential clock sources are the same for each peripheral. See Table 3.

- UART0 clock source-select register (UART0CLKSEL, address 0x4004 8090).
- UART1 clock source-select register (UART1CLKSEL, address 0x4004 8094).
- UART2 clock source-select register (UART2CLKSEL, address 0x4004 8098).
- I2C0 clock source-select register (I2C0CLKSEL, address 0x4004 80A4).
- I3C clock source-select register (I3CFCLKSEL, address 0x4004 80A8).
- I3C clock source-select register (I3CSLOWTCCLKSEL, address 0x4004 80AC).

- I3C clock source-select register (I3CSLOWCLKSEL, address 0x4004 80B0).
- SPI0 clock source-select register (SPI0CLKSEL, address 0x4004 80B4).
- SPI1 clock source-select register (SPI1CLKSEL, address 0x4004 80B8).

Table 3. Peripheral clock source-select registers

Bit	Symbol	Value	Description	Reset value
2:0	SEL	-	Peripheral clock source	0x7
		0x0	FRO	
		0x1	Main clock	
		0x2	FRG0 clock	
		0x3	FRG1 clock	
		0x4	FRO_DIV= FRO / 2	
		0x5	Reserved	
		0x6	Reserved	
		0x7	None	
31:3	-	-	Reserved	-

Select a clock source for the I3C function clock using the I3CFCLKSEL register.

Table 4. I3C clock divider register (I3CCLKDIV, address 0x4004 8078) bit description

Bit	Symbol	Value	Description	Reset value
7:0	I3C_FC LK_DIV	-	i3c_fclkfast clock divider	0x0
15:8	I3C_SL OW_TC _CLK_D IV	-	i3c_slow_tc_clkclock divider	0x0
23:16	I3C_SL OW_CL K_DIV	-	i3c_slow_clkdivider	0x0
31:24	-	-	Reserved	-

Select a clock divide for the I3C function clock using the I3CCLKDIV register.

2. **Reset**

Table 5. Reset

Bit	Symbol	Value	Description	Reset value
23	I3C_RST_N	-	I3C reset control	1
		0	Assert the I3C reset.	
		1	Clear the I3C reset.	

Clear the I3C peripheral reset in the PRESETCTRL0 register.

3. **Interrupt**

Table 6. Connection of interrupt sources to the NVIC

Interrupt number	Vector address	Name	Description	Flags
21	094	I3C_IRQ	I3C interface 0 interrupt	-

The I3C provides interrupts to the NVIC.

4. **Function pins**

Table 7. Function pins

Signal	I/O	Description
SCL	I/O	Serial clock
SDA	I/O	Serial clock
PUR	O	Pull-up resistance. There is internal pull-up resistance on the SDA, which is controlled by the I3C controller. If the internal pull-up is not enough, the PUR can be used to control an external pull-up resistance on SDA actively.

I3C has three function pins, which are clock pin SCL, data pin SDA, and Pull-Up Resistor (PUR) pin. Because LPC86X has a switch matrix, I3C function pins can be assigned on any pins (except for special pins) on LPC86X.

3.3 I3C baud rate

The I3C baud rate includes I^2C baud rate, open-drain baud rate, push-pull baud rate. It is more complicated than the setting of I^2C. This section describes the configuration of the baud rate in detail.

1. **Push-Pull Baud (PPBAUD):**
   - It sets the push-pull frequency as a divider from the FCLK. This frequency sets the SCL half-clock period baseline.
   - The formula for SCL in push-pull mode using PPBAUD and PPLOW is:
     SCL frequency (in MHz) = FCLK / ((2 + 2 * PPBAUD) + PPLOW)
   - If PPLOW is 0, then:
     SCL high width = SCL low width (in ns) = (1 + PPBAUD) * 1000 / FCLK (in MHz)
   - Examples:
     – If FCLK = 24 MHz, then PPBAUD = 0 yields 12 MHz (42.67 ns per half period).
     – If FCLK = 50 MHz, then PPBAUD = 1 yields 12.5 MHz (20 + 20 = 40 ns per half period).
2. **Push-Pull Low (PPLOW):**
   - It changes the duty cycle for push-pull. It indicates how many more FCLK cycles to use for low.
   - The formula for SCL in the push-pull mode using PPBAUD and PPLOW is:
     SCL freq (in MHz) = FCLK / ((2 + 2 * PPBAUD) + PPLOW)
   - If PPLOW is non-zero, then:
     SCL High (in ns) = (1 + PPBAUD) * 1000 / FCLK (in MHz)
     SCL Low (in ns) = (1 + PPBAUD + PPLOW) * 1000 / FCLK (in MHz)
   - For example, when FCLK is 50 MHz, PPBAUD is 1, and PPLOW is 1, then the periods are:
     (1 + 1) * 1000 / 50 = 20 + 20 = 40 ns high
     (1 + 1 + 1) * 1000 / 50 = 20 + 20 + 20 = 60 ns low

   This timing is equivalent to 10 MHz SCL, but this timing maintains the 40 ns high needed so I^2C devices do not see the high periods.
3. **Open-Drain Baud (ODBAUD):**
   - The number of PPBAUD periods to make up one I3C open-drain half-clock baseline.
     – If ODHPP is 0, the ODBAUD half-clock time period = (ODBAUD + 1) * PPBAUD high period.
     – If ODHPP is 0, the formula for SCL in open-drain mode is:

SCL (in MHz) = FCLK / (2 + 2 * PPBAUD) * (ODBAUD + 1)

- The target to get an open-drain half-clock period is 200 ns.
- For example:
  - If PPBAUD yields 12.5 MHz (40 ns per PPBAUD period), then ODBAUD = 4 can be used to get 200 ns.
  - If PPBAUD = 1, FCLK = 50 MHz, and ODBAUD = 4, then open drain is:
    SCL = 50 / (2 + 2 * 1) * 5 = 2.5 MHz

4. **Open-Drain High Push-Pull (ODHPP):**
   - An optional field that allows the I3C open-drain to be long low and short high. The high period of SCL is the PPBAUD period. This period leaves enough time for the pull-up resistor to pull the SDA high when SCL is low. It is also quick when SCL is high and there are no changes.
   - ODBAUD low half-clock time period = (ODBAUD + 1) * PPBAUD high period.
   - ODBAUD high half-clock time period = PPBAUD high period.
   - If ODHPP is 1, the formula for SCL in Open-Drain mode is:
     SCL (in MHz) = FCLK / (1 + PPBAUD) * (ODBAUD + 1) + (1 + PBAUD)
   - For example:
     - If PPBAUD produces 12.5 MHz (40 ns per PPBAUD period), then ODBAUD = 4 and ODHPP = 1. These settings provide a high period of 40 ns and a low period of 200 ns.
     - If PPBAUD = 1, FCLK = 50 MHz, and ODBAUD = 4 then open drain SCL = 50 / (1 + 1) * 5 + (1 + 1) = 4.16 MHz.
5. **I^{2}C Baud (I2CBAUD):**
   - It indicates how many ODBAUD periods are required to communicate with I^{2}C devices.
   - For example, if ODBAUD gives 200 ns and the goal is Fm+ (Fast Mode, 1 MHz), then the sum must be 1 µs.
   - I2CBAUD acts differently for odd and even values. For example, if I2CBAUD = 3, it gives 3 ODBAUD periods low and 2 ODBAUD periods high. If I2CBAUD = 4, then it gives 3 ODBAUD periods for low and 3 ODBAUD periods for high. Also, if I2CBAUD = 3, this yields 200 * 3 = 600 ns and 200 * 2 = 400 ns, with the sum 600 + 400 = 1000 ns = 1 µs.
6. **Summary:**
   - The relationship and calculation if the I3C baudrate is shown in Figure 1.

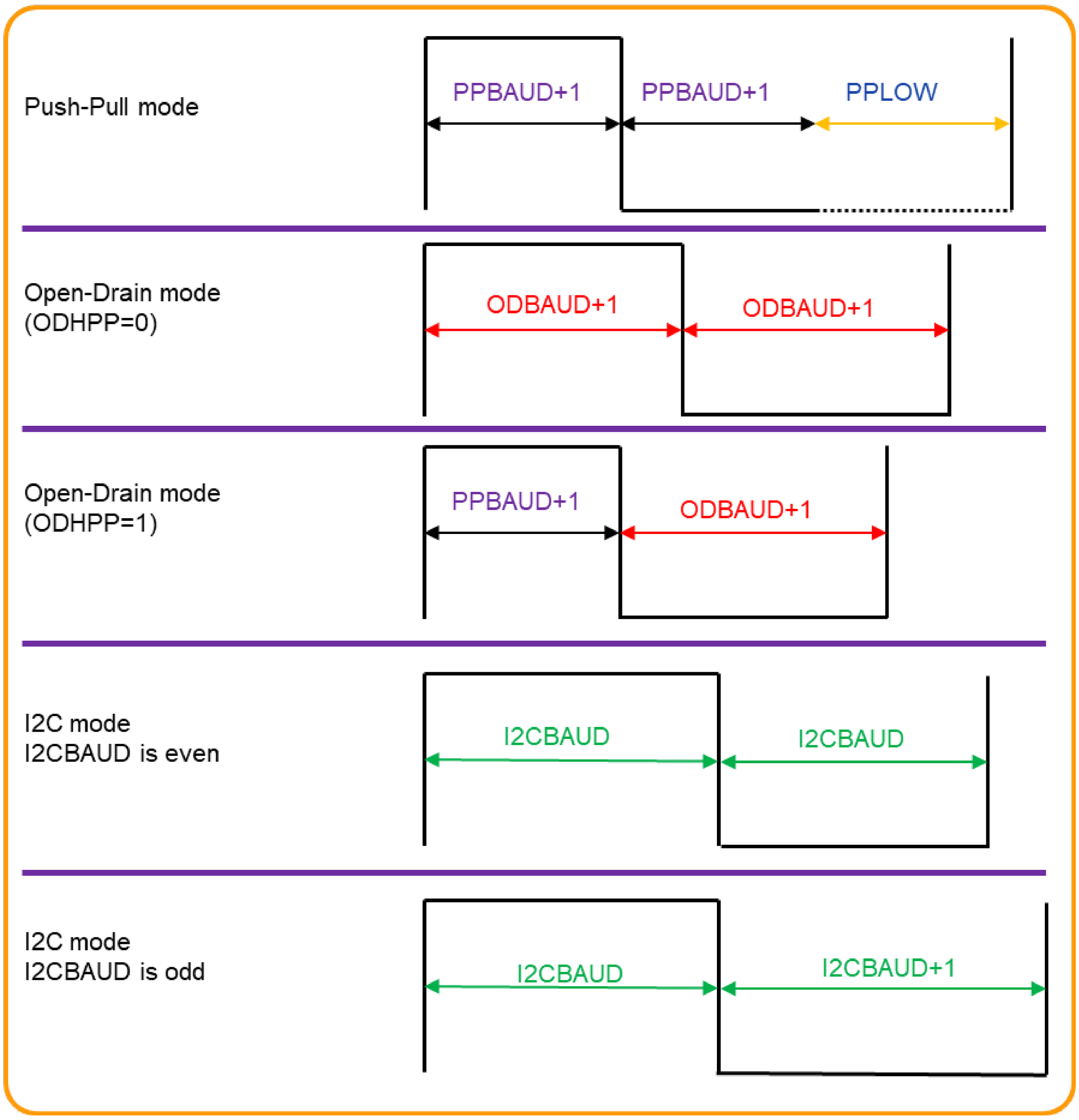

Figure 1. Summary

3.4 I3C registers

3.4.1 Controller registers

Table 8 shows controller registers.

Table 8. Controller registers

Name	Offset	Description
MCONFIG	0x0	Controller configuration.
MCTRL	0x84	Controller main control.

Table 8. Controller registers...*continued*

Name	Offset	Description
MSTATUS	0x88	Controller status, including interrupt causes.
MIBIRULES	0x8C	Rules for in-band interrupt use.
MINTSET	0x90	Controller interrupt set/enable.
MINTCLR	0x94	Controller interrupt clear/disable.
MINTMASKED	0x98	Masked interrupts for controller.
MERRWARN	0x9C	Controller errors and warnings.
MDMACTRL	0xA0	Controller DMA control.
MDATACTRL	0xAC	Controller data control.
MWDATAB	0xB0	Controller write data byte.
MWDATABE	0xB4	Controller write data byte end.
MWDATAH	0xB8	Controller write data half-word.
MWDATAHE	0xBC	Controller write data byte end.
MRDATAB	0xC0	Controller read data byte.
MRDATAH	0xC8	Controller read data half-word.
MWMSG_SDR_CONTROL	0xD0	Controller write message control.
MWMSG_SDR_DATA	0xD0	Controller write message data.
MRMSG_SDR	0xD4	Controller read message in SDR mode.
MWMSG_DDR_CONTROL	0xD8	Controller write message control in DDR mode.
MWMSG_DDR_DATA	0xD8	Controller write message data in DDR mode.
MRMSG_DDR	0xDC	Controller read message in DDR mode.
MDYNADDR	0xE4	Controller dynamic address.

3.4.2 Target registers

Table 9 shows target registers.

Table 9. Target registers

Name	Offset	Description
SCONFIG	0x4	Slave configuration.
SSTATUS	0x8	Slave status register.
SCTRL	0xC	Slave control.
SINTSET	0x10	Slave interrupt set/enable.
SINTCLR	0x14	Slave interrupt clear/disable.
SINTMASKED	0x18	Masked interrupts for slave.
SERRWARN	0x1C	Slave errors and warnings.
SDMACTRL	0x20	Slave DMA control.

Table 9. Target registers*...continued*

Name	Offset	Description
SDATACTRL	0x2C	Slave Data control.
SWDATAB	0x30	Slave write data byte.
SWDATABE	0x34	Slave write data byte end.
SWDATAH	0x38	Slave write data half-word.
SWDATAHE	0x3C	Slave write data half-word end.
SRDATAB	0x40	Slave read data byte.
SRDATAH	0x48	Slave read data half-word.
SCAPABILITIES	0x60	Slave capabilities.
SDYNADDR	0x64	Slave dynamic address.
SMAXLIMITS	0x68	Slave maximum limits.
SIDPARTNO	0x6C	Slave ID part number.
SIDEXT	0x70	Slave ID extension.
SVENDORID	0x74	Slave vendor ID.
STCCLOCK	0x78	Slave time control clock.
SMSGMAPADDR	0x7C	Slave message-mapped address.
SID	0xFFC	Slave module ID.

3.5 Dynamic address assignment

Compared with the I^{2}C protocol, one feature of the I3C protocol is that the controller can assign a dynamic address. The dynamic address assignment allows the controller to set the address priority of each target and it also allocates addresses for newly added targets at any time. This method allows the controller not to know the static address of the target in advance, but to just read the PID of the target through the command and then set the corresponding address according to the PID. The subsequent communication will use the dynamic address.

- PROCESSDAA: If it is not in the Dynamic Address Assignment (DAA) mode, it issues START, 7E, and ENTDAA and then it emits 7E/R to process each slave. It stops just before the new Dynamic Address (DA) is to be emitted. The next Process DAA request uses the Addr field as the new DA to assign. If NACKed on the 7E/R, then the interrupt will indicate this situation and a STOP is emitted.

After initializing the controller, the user can implement dynamic address assignment using the PROCESSDAA request in the controller Main Control (MCTRL) register.

After sending the DAA sequence, the user can write the MWDATAB with the dynamic address. If the PID cannot be read out, the user can re-operate according to the error flag.

After the dynamic address is assigned successfully, the controller can use the dynamic address to communicate with the targets.

3.6 Controller send SDR message

The I3C baudrate includes the I^{2}C baudrate (open-drain).

3.7 DMA usage in I3C

The I3C baud rate can reach up to 12 Mbit/s. DMA is necessary to obtain the highest baudrate for 60 MHz system clock in LPC86X.

Table 10. DMA requests

DMA channel #	Request input	DMA trigger multiplexer
13	I3C0_TX_DMA	DMA_ITRIG_INMUX13
14	Reserved	DMA_ITRIG_INMUX14
15	Reserved	DMA_ITRIG_INMUX15

DMA requests are directly connected to the peripherals. The I30_RX_DMA is connected to DMA channel 12 and the I3C0_TX_DMA is connected to DAM channel 13.

Table 11. DMA requests

DMA channel #	Request input	DMA trigger multiplexer
10	I2C0_RX_DMA	DMA_ITRIG_INMUX10
11	I2C0_TX_DMA	DMA_ITRIG_INMUX11
12	I3C0_RX_DMA	DMA_ITRIG_INMUX12

The I3C has a data FIFO, which can store 8 words totally. For transmitting the data, the DMA request is sent if the TX FIFO is not full. For receiving the data, the DMA request is sent if the RX FIFO is not empty.

After configuring the I3C and DMA, use software to start the DMA and use an I3C peripheral request to push the DMA transferring.

Table 12. DMA configuration

25:16	XFERCOUNT	The total number of transfers to be performed, minus one encoded. The number of bytes transferred is: (XFERCOUNT + 1) x data width (as defined by the WIDTH field). **Remark:** The DMA controller uses this bit field during the transfer to count down. It cannot be used by software to read back the size of the transfer, for instance, in an interrupt handler. • 0x0 = a total of 1 transfer is performed. • 0x1 = a total of 2 transfers is performed. • 0x3FF = a total of 1024 transfers is performed.	0

In DMA configuration, the maximum number of transfers is 1024. If transferring two bytes for every I3C request, 2048 bytes can be transferred totally for every DMA descriptor. More data can be transferred using a linked descriptor.

3.8 In-Band Interrupt (IBI) handling

The I3C baud rate includes the I^2C baud rate (open-drain).

4 LPC86X SDK I3C examples introduction

There are four kinds of I3C examples in the LPC86X SDK:

- master_read_sensor_icm42688p
- interrupt_b2b_transfer
- interrupt_b2b

- polling_b2b_transfer
- dma_b2b_transfer

This application note introduces their implementation methods and steps.

4.1 Controller reads the I3C sensor data

4.1.1 Clock

The FRO clock is used as the clock source of I3C. The frequency division value is set to 4. The FRO clock is set to 60 MHz. The system clock also uses the FRO clock. The I3C function clock is 15 MHz.

The code snippet is as follows:

```
CLOCK_Select(kI3C_Clk_From_Fro);
CLOCK_SetI3CFClkDiv(4);

BOARD_InitPins();
BOARD_BootClockFRO60M();
BOARD_InitDebugConsole();
```

4.1.2 I3C controller initialization

The code in this stage sets the MCONFIG and MDATACTRL registers.

1. Enable the controller to operate.
2. Enable the timeout function. If the controller has been left in a state other than STOP for more than 100 us, then the time-out will send an MERRWARN event.
3. See the I3C specifications. A high keeper is a weak pull-up type device used when SDA (and sometimes SCL) is in high-Z, with respect to all devices. A high keeper is used for the controller-to-target and target-to-controller bus handoff, as well as when the bus is idle (optionally). In the LPC86X SDK example, the PUR (Pull-Up Resistor) is used as the high-keeper method.
4. Set the open-drain stop. If ODSTOP=1, then the STOP is emitted at open-drain speeds, even for I3C messages. This can be useful for legacy devices to ensure that the legacy devices see the STOP.
5. Set the open-drain high period with the push-pull mode. If ODHPP=1, then the open-drain high should be 1 PPBAUD for I3C messages. Otherwise, the ODHPP should be the same value as ODBAUD. Set the high period and low period with different values, so that I^{2}C devices do not see the high level.
6. Set the watermark of the TX integer trigger level with a trigger value of one less than full or less. As long as there is a space in the TX buffer, the TX trigger starts. Set the watermark of the RX integer trigger level with the trigger on not empty. As long as there is data in the RX buffer, the trigger starts. The watermark does not work for the DMA in LPC86X.
7. The push-pull baud rate is set to 12.5 Mbit/s, the open-drain baudrate is set to 4 Mbit/s, and the I^{2}C baudrate is set to 400 kbit/s. The calculation method of the baud rate is a bit complicated. See the clock description in the previous chapter. The baud rate of push-pull is relatively high, but it must be an integer multiple of the clock source. Therefore, some baud rate setting results are inconsistent with the parameters.

```
I3C_MasterGetDefaultConfig(&masterConfig);
masterConfig.baudRate_Hz.i2cBaud             = EXAMPLE_I2C_BAUDRATE;
masterConfig.baudRate_Hz.i3cPushPullBaud     = 12500000U;
masterConfig.baudRate_Hz.i3cOpenDrainBaud    = 400000U;
masterConfig.enableOpenDrainStop             = false;
I3C_MasterInit(EXAMPLE_MASTER, &masterConfig, I3C_MASTER_CLOCK_FREQUENCY);
```

4.1.3 I3C controller handle

The code creates a driver handle for a non-blocking transfer, which includes:

- The transfer state machine current state.
- The byte count in the current state.
- It is read-term configured.
- The current transfer info.
- The target address that requests the IBI.
- The pointer to the IBI buffer to keep the IBI bytes.
- The IBI payload size.
- The IBI type.
- The callback functions pointer.
- The application data passed to the callback.

The application code initializes the callback and user data, sets the IRQ handler, clears all the flags, resets the FIFO, and enables the interrupt. The MDATACTRL, MERRWARN, MSTATUS, and MINTSET registers are configured.

Write a logic 1 to the FLUSHTB and FLUSHFB bit fields in MDATACTRL to make the buffer empty. The MERRWARN register provides I3C errors and warnings flags. Use this window to see what is wrong with the bus or state machine during operations.

MINTSET is used to set the interrupts of I3C.

```
void I3C_MasterTransferCreateHandle(I3C_Type *base,
                                    i3c_master_handle_t *handle,
                                    const i3c_master_transfer_callback_t
 *callback,
                                    void *userData)
{
    uint32_t instance;

    assert(NULL != handle);

    /* Clear out the handle. */
    (void)memset(handle, 0, sizeof(*handle));

    /* Look up instance number */
    instance = I3C_GetInstance(base);

    /* Save base and instance. */
    handle->callback = *callback;
    handle->userData = userData;

    /* Save this handle for IRQ use. */
    s_i3cMasterHandle[instance] = handle;

    /* Set irq handler. */
    s_i3cMasterIsr = I3C_MasterTransferHandleIRQ;

    /* Clear all flags. */
    I3C_MasterClearErrorStatusFlags(base, (uint32_t)kMasterErrorFlags);
    I3C_MasterClearStatusFlags(base, (uint32_t)kMasterClearFlags);
    /* Reset fifos. These flags clear automatically. */
    base->MDATACTRL |= I3C_MDATACTRL_FLUSHTB_MASK | I3C_MDATACTRL_FLUSHFB_MASK;
    /* Enable NVIC IRQ, this only enables the IRQ directly connected to the
 NVIC.
     In some cases the I3C IRQ is configured through INTMUX, user needs to
 enable
     INTMUX IRQ in application code. */
    (void)EnableIRQ(kI3cIrqs[instance]);

    /* Clear internal IRQ enables and enable NVIC IRQ. */
    I3C_MasterEnableInterrupts(base, (uint32_t)kMasterIrqFlags);
}
```

4.1.4 Register IBI and assigning dynamic address

Register the in-band interrupt for the sensor address. Write the sensor address to the register. If NOBYTE = 0, then the ADDRx fields refer to targets with a mandatory IBI byte.

```
i3c_register_ibi_addr_t ibiRecord = {.address = {slaveAddr}, .ibiHasPayload =
 true};
I3C_MasterRegisterIBI(EXAMPLE_MASTER, &ibiRecord);
result = I3C_MasterProcessDAA(EXAMPLE_MASTER, addressList, sizeof(addressList));
```

The steps of the dynamic address assignment are as follows:

- Clear all the flags.
- Disable the interrupt.
- Emit the process DAA request.
- Get the VID, part number, BCR, and DCR from the target.
- Write the MWDATAB with the address to be assigned.
- Re-emit the process DAA request for more targets.
- Clear the flags and enable the interrupt.

After finishing the dynamic address assignment, all the targets have their dynamic addresses. The controller can use the dynamic address to communicate with targets.

4.1.5 Sensor operations and I3C state machine

The operation of the sensor includes initialization, enablement, configuration registers, and so on. The middle layer is mainly implemented by two functions:

- I3C_WriteSensor
- I3C_ReadSensor

These functions are implemented by the I3C_MasterTransferNonBlocking low-level driver routine. In this application note, it mainly introduces the implementation of the I3C low-level driver.

The non-blocking transfer code snippet is as follows:

```
/* Set register data */
masterXfer.slaveAddress   = deviceAddress;
masterXfer.direction      = kI3C_Read;
masterXfer.busType        = kI3C_TypeI3CSdr;
masterXfer.subaddress     = regAddress;
masterXfer.subaddressSize = 1;
masterXfer.data           = regData;
masterXfer.dataSize       = dataSize;
masterXfer.flags          = kI3C_TransferDefaultFlag;
```

```
result = I3C_MasterTransferNonBlocking(EXAMPLE_MASTER, &g_i3c_m_handle,
 &masterXfer);
```

The non-blocking transfer descriptor structure (_i3c_master_transfer) defines the following items:

- The bit mask of options for the transfer.
- The 7-bit target address.
- The direction (kI3C_Read or kI3C_Write).
- The sub address.
- The length of the sub address to send (in bytes).
- The pointer to the data to transfer.
- The number of bytes to transfer.
- The bus type.
- The IBI response during the transfer.

The I3C_MasterTransferNonBlocking function mainly implements the initialization of the state machine. The state machine contains the following states:

- kIdleState
- kIBIWonState
- kSlaveStartState
- kSendCommandState
- kWaitRepeatedStartCompleteState
- kTransferDataState
- kStopState
- kWaitForCompletionState

The running state machine is implemented in the I3C_RunTransferStateMachine function. This function is the most important implementation function called by the interrupt function.

At the beginning, sending the address header and sub address must call the interrupt function to run the state machine.

The I3C_RunTransferStateMachine function contains the processing steps of all states and it responds differently in different states. To increase the speed of the I3C, you can optimize the steps in this function.

1. According to the flag bits of I3C, if it is judged to be kSlaveStartState, then the state machine starts the AUTOIBI request in the MCTRL I3C register. Change the state to kIBIWonState to receive the IBI payload data.
2. When the state machine enters kSendCommandState, the sub-address data are sent by the I3C controller. A repeated start signal can be sent in the kWaitRepeatedStartCompleteState state.
3. The kTransferDataState state is the key state in the state machine, because most data transferring is implemented in this state. If the controller writes the data to the bus, the application code writes the MWDATAB register and writes MWDATABE for the last byte. If the controller reads the data from the bus, the application code gets the data from the MRDATAB register.
4. At the end of all operations, the stop signal is sent in the kStopState state.

A complete I3C data frame includes several different signal states. These stages are regularly combined to achieve different functions. The I3C register gives the interface, shows different states through different flags, and transmits signals and data through control registers and data registers. A state machine is needed in the driver layer. These states can be combined to complete the given tasks.

Using a non-blocking mode to handle the I3C bus signals, the Arm core can do other tasks in the I3C processing gap. However, for the 60-MHz system clock, dealing with I3C timing at 12.5 Mbit/s is already very difficult, leaving no time for other tasks. The advantages of the interrupt mode are not fully utilized.

4.1.6 I3C interrupt

The interrupt is the feedback from the bottom layer of the peripheral to the upper layer and many status types can generate interrupts. After the driver function enters the interrupt, it must read the flags from the state register to determine which state the peripheral is jumping in.

In the interrupt function (in addition to running the state machine), additional processing must be done according to different states, which is a supplement to the state machine. For example, turn off the TX ready flag interrupt in the kIdleState state and send the stop signal in the kStatus_Success state.

4.2 I3C interrupt transfer with state machine

Similarly to the master_read_sensor_icm42688p example, the interrupt_b2b_transfer example uses the interrupt mode. Therefore, the controller data-transmission mechanism is the same. The controller runs the state machine in an interrupt handler and sends different signal or data timings in different states.

This chapter does not explain the behavior of the controller too much. It focuses mainly on the operation of the target.

4.2.1 Target initialization

The most important thing in the initialization of the target is to set the static address, vendor ID, and frequency. Of course, the application code must also set the BCR, DCR, part number, maximum read and write length, and so on, which have given the default values in the default configuration.

```
I3C_SlaveGetDefaultConfig(&slaveConfig);

slaveConfig.staticAddr = I3C_MASTER_SLAVE_ADDR_7BIT;
slaveConfig.vendorID   = 0x123U;
slaveConfig.offline    = false;

I3C_SlaveInit(EXAMPLE_SLAVE, &slaveConfig, I3C_SLAVE_CLOCK_FREQUENCY);
```

The main capabilities' registers involved are as follows:

Table 13. Main capabilities' registers

SMAXLIMITS	0x68	Slave Maximum Limits.	Capabilities
SIDPARTNO	0x6C	Slave ID Part Number.	Capabilities
SIDEXT	0x70	Slave ID Extension.	Capabilities
SVENDORID	0x74	Slave Vendor ID.	Capabilities

The SCONFIG register configures the target. This register defines the static address field, enables the target, and performs some other functions. For details, see the description of the UM register.

4.2.2 Create target handle

Before starting to transfer data, the application must do some preparation. In the I3C_SlaveTransferCreateHandle routine, the application sets the callback routine. The FIFO transmit value and FIFO receive value of the I3C can be read in the SCAPABILITIES register. Knowing the capacity of the FIFO, the application can plans the number of bytes to be sent and received. Then, the application disables the interrupt for the moment and enables the NVIC. Later, as long as the I3C internal interrupt is enabled, the relevant event will generate an I3C interrupt.

```
/* Look up instance number */
```

```
instance = I3C_GetInstance(base);

/* Save base and instance. */
handle->callback = callback;
handle->userData = userData;

/* Save Tx FIFO Size. */
handle->txFifoSize =
2U << ((base->SCAPABILITIES & I3C_SCAPABILITIES_FIFOTX_MASK) >>
 I3C_SCAPABILITIES_FIFOTX_SHIFT);

/* Save this handle for IRQ use. */
s_i3cSlaveHandle[instance] = handle;

/* Set irq handler. */
s_i3cSlaveIsr = I3C_SlaveTransferHandleIRQ;

/* Clear internal IRQ enables and enable NVIC IRQ. */
I3C_SlaveDisableInterrupts(base, (uint32_t)kSlaveIrqFlags);
(void)EnableIRQ(kI3cIrqs[instance]);
```

In the application, the target first receives the data sent by the controller. Before receiving data, initialize the receiving buffer and enable related interrupts.

```
/*! IRQ sources enabled by the non-blocking transactional API. */
kSlaveIrqFlags = kI3C_SlaveBusStartFlag | kI3C_SlaveMatchedFlag |
 kI3C_SlaveBusStopFlag | kI3C_SlaveRxReadyFlag |
                 kI3C_SlaveDynamicAddrChangedFlag | kI3C_SlaveReceivedCCCFlag |
 kI3C_SlaveErrorFlag |
                 kI3C_SlaveHDRCommandMatchFlag | kI3C_SlaveCCCHandledFlag |
 kI3C_SlaveEventSentFlag,
```

4.2.3 Target data transmission in interrupt mode

In the I3C protocol, the behavior of the target is guided by the controller. Therefore, the target does not need a very complicated state machine. It only needs to respond to different interrupts according to the bus signal and do different preparations.

The bus information can be displayed in the SSTATUS register and this register can be read to obtain relevant status information after an interrupt is generated. If there is an error on the bus or the module, the relevant error information can be obtained from the SERRWARN register. After entering the interrupt, the first thing to do is to read these two registers to obtain more accurate interrupt information.

There are many flags that can cause interrupts and the corresponding interrupts can be set in the I3C SINTSET register. The SINTMASKED register returns the status of enabled interrupts (the value of Target Status (SSTATUS) ANDed with the value of Target Interrupt Set (SINTSET)). Not all status flags generate interrupts in the application. Only the enabled flags generate interrupts.

Figure 2 is the processing flow in the I3C target interrupt, and the brown rectangles are the detailed execution of each state.

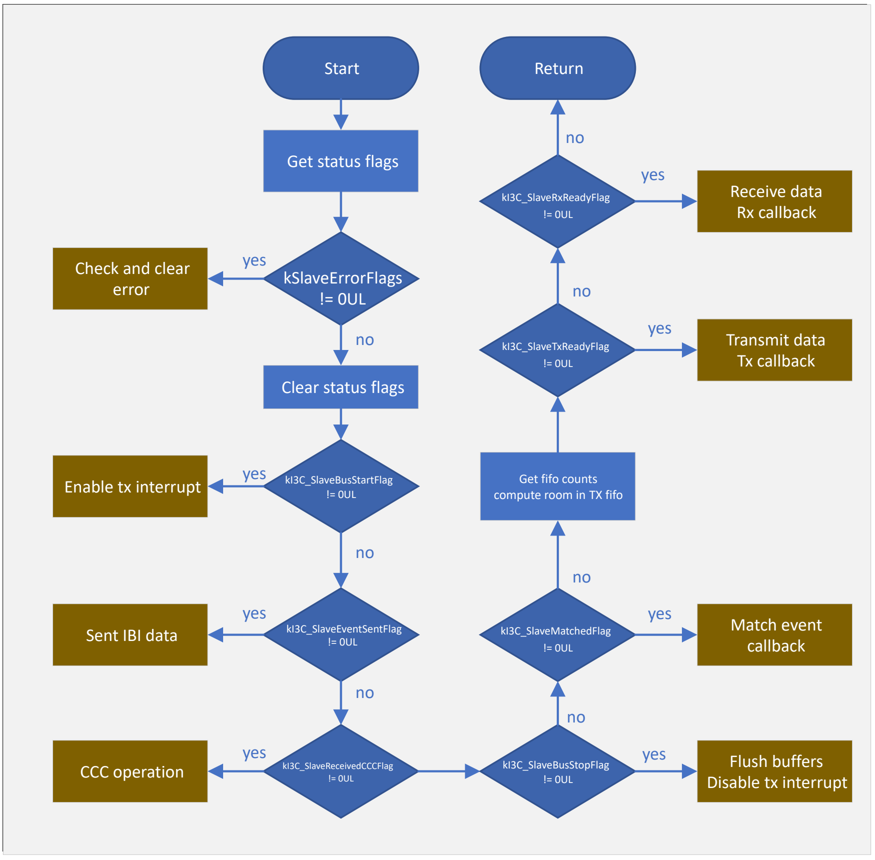
Start
Get status flags
kSlaveErrorFlags != 0UL
yes
Check and clear error
no
Clear status flags
kI3C_SlaveBusStartFlag != 0UL
yes
Enable tx interrupt
no
kI3C_SlaveEventSentFlag != 0UL
yes
Sent IBI data
no
kI3C_SlaveReceivedCCCFlag != 0UL
yes
CCC operation
kI3C_SlaveBusStopFlag != 0UL
yes
Flush buffers Disable tx interrupt
no
kI3C_SlaveMatchedFlag != 0UL
yes
Match event callback
no
Get fifo counts compute room in TX fifo
kI3C_SlaveTxReadyFlag != 0UL
yes
Transmit data Tx callback
no
kI3C_SlaveRxReadyFlag != 0UL
yes
Receive data Rx callback
no
Return

Figure 2. Target data transmission in interrupt mode

In the i3c_slave_callback routine, the application code configures the buffer and its length for sending and receiving. These operations are supplementary to the interrupt function. The application sets g_slaveCompletionFlag when the operation is complete.

After receiving the data sent by the controller, the target will return the received data to the controller. The completion of each data transmission mainly depends on the judgment of g_slaveCompletionFlag.

4.3 I3C interrupt transfer without state machine

To reduce the complexity of software logic, the SDK also provides a simplified version of the interrupt mode transfer example (interrupt_b2b). The interrupt_b2b example also uses the interrupt mode, but without the complex state machine. This method is more convenient for users to modify the code as needed. It can realize basic functions, such as in-band interrupt, sending data, and receiving data.

4.3.1 Interrupt_b2b for controller

Same as with the master_read_sensor_icm42688p example, the initialization of the controller will configure the baud rate and other default values.

The controller is responsible for the assignment of dynamic addresses and the existing addresses of targets must be reset before the address is allocated each time. 0x06 is a CCC command about the Reset Dynamic Address Assignment (RSTDAA).

```
/* Reset dynamic address before DAA */

I3C_MasterStart(EXAMPLE_MASTER, kI3C_TypeI3CSdr, 0x7EU, kI3C_Write);
```

```
uint8_t cmdCode = 0x06U;
result          = i3c_master_transferBuff(&cmdCode, 1U, kI3C_Write);

if (kStatus_Success != result)
{
    return result;
}

I3C_MasterStop(EXAMPLE_MASTER);
```

The operation of the dynamic address assignment is the same as the master_read_sensor_icm42688p example.

In this example, the routines for sending start and stop signals are listed separately and the address can be directly set in the parameters of the function when sending the start address header.

The data transfer operation is performed in the i3c_master_transferBuff function. The first thing to do is to enable different interrupts according to the read and write directions. If it is a write operation, enable the TX ready interrupt, and if it is a read operation, enable the RX ready interrupt. To detect error conditions, the error flag interrupt is also turned on.

```
if (dir == kI3C_Write)
{
    I3C_MasterEnableInterrupts(EXAMPLE_MASTER, (uint32_t)(kI3C_MasterTxReadyFlag
 | kI3C_MasterErrorFlag));
}
else
{
    I3C_MasterEnableInterrupts(EXAMPLE_MASTER, (uint32_t)(kI3C_MasterRxReadyFlag
 | kI3C_MasterErrorFlag));
}

uint32_t timeout = 0U;
/* Wait for transfer completed. */
while ((!g_masterCompletionFlag) && (g_completionStatus == kStatus_Success) &&
 (++timeout < I3C_TIME_OUT_INDEX))
{
    __NOP();
}
```

Interrupt processing is executed in the I3C0_IRQHandler function. The flow chart is in Figure 3.

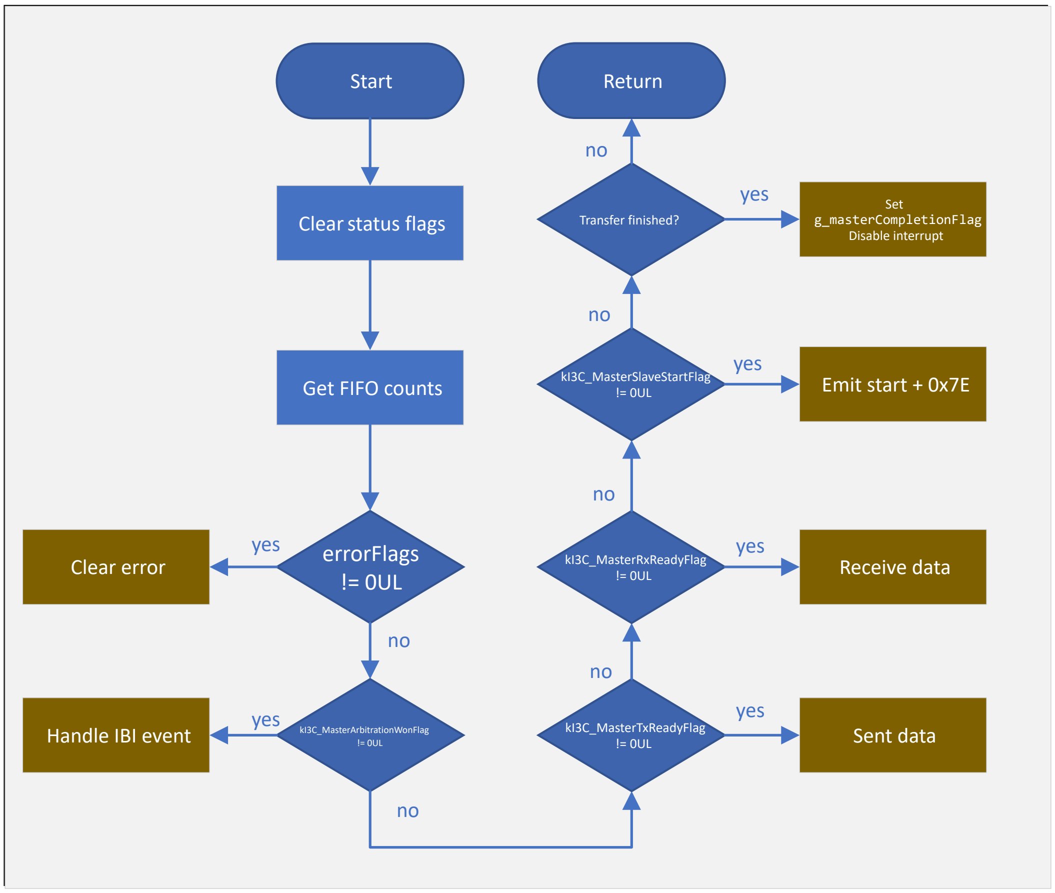

Figure 3. Flow chart

4.3.2 Interrupt_b2b for target

The initialization of target is the same as with the Interrupt_b2b_transfer example. After the target is initialized, the application enables the interrupt about event sending, match flag, and bus stop flag. The target starts to prepare to receive the data sent by the controller.

As an interrupt transmission method, set the receive flag interrupt and error flag interrupt of the target. When the completion flag is set, it means all the data has been received in the buffer. At this time, you can turn off the interrupt and wait for the next operation. The interrupt function of the target is not as complicated as that of the controller. It only judges several states, and then sends and receives several bytes of data. Because the target does not need to be responsible for all signal processing like the controller, most of the cases are only passive data transmissions in the appropriate state. Interrupt processing is executed in the I3C0_IRQHandler function. The flow chart is in Figure 4.

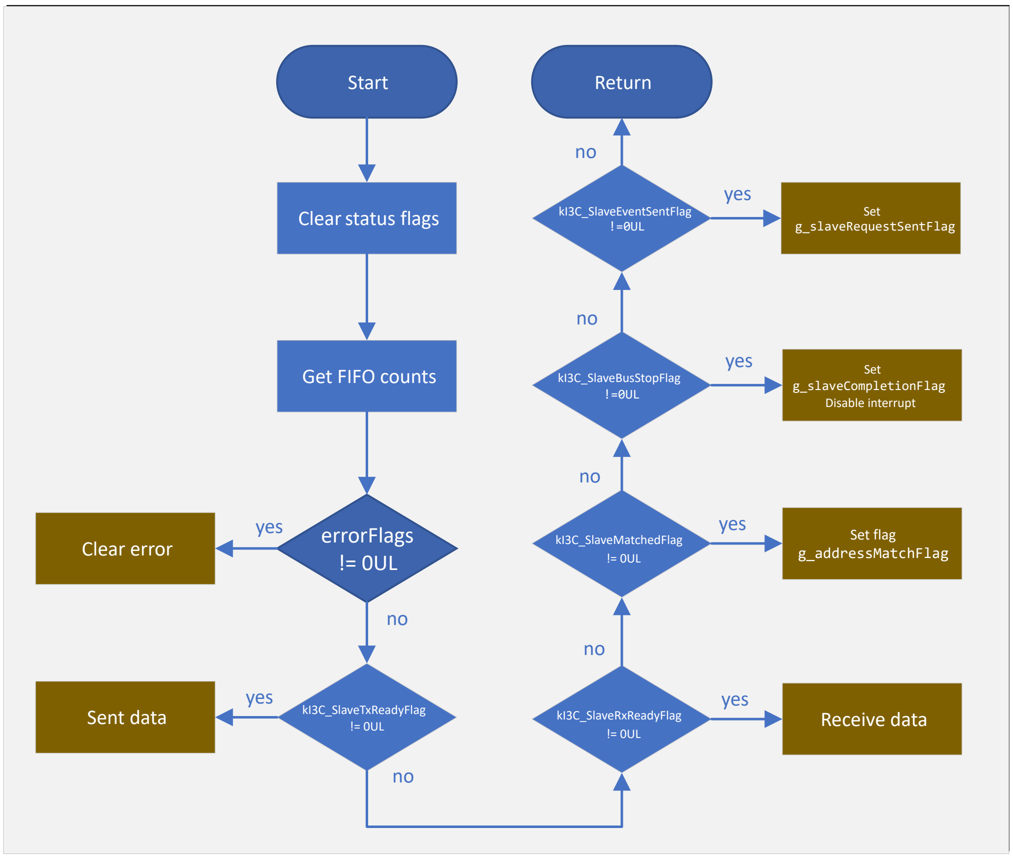
Start
Return
Clear status flags
Get FIFO counts
errorFlags != 0UL
yes
Clear error
no
kI3C_SlaveTxReadyFlag != 0UL
yes
Sent data
no
kI3C_SlaveRxReadyFlag != 0UL
yes
Receive data
no
kI3C_SlaveMatchedFlag != 0UL
yes
Set flag g_addressMatchFlag
no
kI3C_SlaveBusStopFlag !=0UL
yes
Set g_slaveCompletionFlag Disable interrupt
no
kI3C_SlaveEventSentFlag !=0UL
yes
Set g_slaveRequestSentFlag
no

Figure 4. Flow chart

The application example also implements the in-band function. The target requests the IBI interrupt and sends the first byte in the RX buffer to the controller.

The in-band function can be realized by setting the SCTRL register, which will enable the IBI value and fill the IBIDATA value.

```
void I3C_SlaveRequestIBIWithData(I3C_Type *base, uint8_t *data, size_t dataSize)
{
    assert((dataSize > 0U) && (dataSize <= 8U));

    uint32_t ctrlValue;

#if (defined(I3C_IBIEXT1_MAX_MASK) && I3C_IBIEXT1_MAX_MASK)
    if (dataSize > 1U)
    {
        ctrlValue = I3C_IBIEXT1_EXT1(data[1]);
        if (dataSize > 2U)
        {
            ctrlValue |= I3C_IBIEXT1_EXT2(data[2]);
        }
        if (dataSize > 3U)
        {
            ctrlValue |= I3C_IBIEXT1_EXT3(data[3]);
        }
        ctrlValue |= I3C_IBIEXT1_CNT(dataSize - 1U);
        base->IBIEXT1 = ctrlValue;
    }
    if (dataSize > 4U)
    {
        ctrlValue = I3C_IBIEXT2_EXT4(data[4]);
        if (dataSize > 5U)
        {
            ctrlValue |= I3C_IBIEXT2_EXT5(data[5]);
        }
        if (dataSize > 6U)
        {
            ctrlValue |= I3C_IBIEXT2_EXT6(data[6]);
        }
        if (dataSize > 7U)
        {
            ctrlValue |= I3C_IBIEXT2_EXT7(data[7]);
        }
        base->IBIEXT2 = ctrlValue;
    }
#endif

    ctrlValue = base->SCTRL;
#if (defined(I3C_IBIEXT1_MAX_MASK) && I3C_IBIEXT1_MAX_MASK)
    ctrlValue &= ~(I3C_SCTRL_EVENT_MASK | I3C_SCTRL_IBIDATA_MASK |
 I3C_SCTRL_EXTDATA_MASK);
    ctrlValue |= I3C_SCTRL_EVENT(1U) | I3C_SCTRL_IBIDATA(data[0]) |
 I3C_SCTRL_EXTDATA(dataSize > 1U);
#else
    ctrlValue &= ~(I3C_SCTRL_EVENT_MASK | I3C_SCTRL_IBIDATA_MASK);
    ctrlValue |= I3C_SCTRL_EVENT(1U) | I3C_SCTRL_IBIDATA(data[0]);
#endif
    base->SCTRL = ctrlValue;
}
```

After the controller receives the IBI interrupt, it sends the address header signal. The target sends its own address according to the clock sent by the host and participates in address arbitration. After the address is sent, the controller will identify which target generated the interrupt and can receive the following data on the bus.

4.4 I3C polling transfer

Different from the interrupt_b2b_transfer example, polling_b2b_transfer adopts the method of sequential execution, that is, after one step is completed, the next step is executed. The CPU has been waiting for the execution of each step to complete, and cannot do other tasks. This is also called a blocking method.

4.4.1 Controller operations

Same as with the interrupt mode, the I3C peripheral must be initialized before the data transmission. The user mainly configures the baud rate and other configurations use the default values. The controller registers the IBI function and can receive the in-band interrupt of the target at any time.

```
I3C_MasterGetDefaultConfig(&masterConfig);
masterConfig.baudRate_Hz.i2cBaud          = EXAMPLE_I2C_BAUDRATE;
masterConfig.baudRate_Hz.i3cPushPullBaud  = EXAMPLE_I3C_PP_BAUDRATE;
masterConfig.baudRate_Hz.i3cOpenDrainBaud = EXAMPLE_I3C_OD_BAUDRATE;
masterConfig.enableOpenDrainStop          = false;
I3C_MasterInit(EXAMPLE_MASTER, &masterConfig, I3C_MASTER_CLOCK_FREQUENCY);

memset(&masterXfer, 0, sizeof(masterXfer));
```

The blocking mode requires the parameters provided by the user to be basically the same as for the non-blocking mode, including the target address, data transmission direction, bus type, data buffer and data length, and so on.

```
/* subAddress = 0x01, data = g_master_txBuff - write to slave.
start + slaveaddress(w) + subAddress + length of data buffer + data buffer +
 stop*/
uint8_t deviceAddress      = 0x01U;
masterXfer.slaveAddress    = I3C_MASTER_SLAVE_ADDR_7BIT;
masterXfer.direction       = kI3C_Write;
masterXfer.busType         = kI3C_TypeI2C;
masterXfer.subaddress      = (uint32_t)deviceAddress;
masterXfer.subaddressSize = 1;
masterXfer.data            = g_master_txBuff;
masterXfer.dataSize        = I3C_DATA_LENGTH;
masterXfer.flags           = kI3C_TransferDefaultFlag;

result = I3C_MasterTransferBlocking(EXAMPLE_MASTER, &masterXfer);
```

The I3C_MasterTransferBlocking routine performs a controller-polling transfer on the I^{2}C /I3C bus. Clear the used flags, reset the FIFO, and turn off interrupts. Send the start signal and the sub-address. Depending on the direction of the data transmission, send data or receive data. Clear all flags and enable interrupts. The flowchart is in Figure 5.

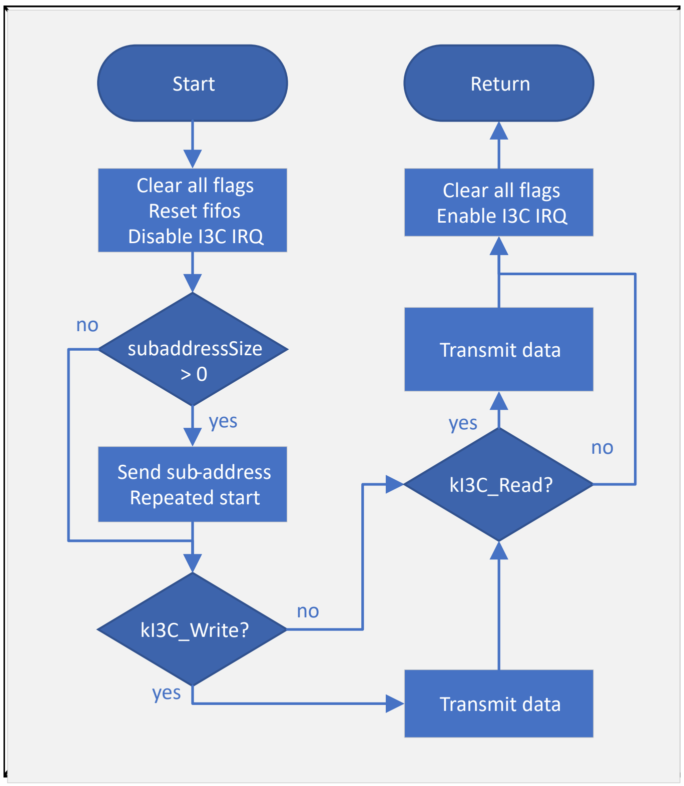

Figure 5. Flow chart

Every time the I3C_MasterTransferBlocking function is executed to transfer data, the masterXfer parameter must be reconfigured. The operation of dynamic address assignment is the same as with the interrupt transfer method.

4.4.2 Target operations

In the polling_b2b_transfer example, the target also uses the interrupt method, because the target receives signals passively and it is more appropriate to use the interrupt method. If the polling method is used, the target must wait for every signal from the controller and it cannot perform other tasks. See the target operations in the interrupt_b2b_transfer example.

4.5 I3C transfer with DMA

The bus speed of I3C is as high as 12.5 MHz. On the LPC86X with a main frequency of 60 MHz, it is very bandwidth-intensive to perform data transmission through the Arm core.

It is very important to use DMA to transfer data. Fortunately, the SDK of LPC86X provides the dma_b2b_transfer example, which can be used as a reference.

This application note does not introduce the configuration of DMA. See the relevant chapters in the *LPC86X User Manual* (document UM11607). On the I3C side, it is mainly about the MDMACTRL and SDMACTRL configuration registers.

4.5.1 Controller operations

1. I3C controller initialization - same as with the interrupt mode, the application code must configure the I3C baudrate and other default values. See the interrupt mode chapter description.
2. DMA initialization - the initialization of DMA mainly includes enabling the DMA clock, resetting the DMA module, assigning the address of the DMA descriptor table to the register SRAMBASE, and enabling the DMA peripheral.
3. DMA configuration - enable the corresponding DMA channel for I3C TX and RX. Create a handle for each of them. Set the interrupt handler and set the callback functions for sending and receiving. Clear the status and error flags for I3C. Enable the I3C interrupt. When using DMA to transfer, enable the I3C interrupt. Because DMA only helps with transferring data, other signal conditions still require I3C interrupts to process.
4. Non-blocking DMA transaction.

The SDK creates the following functions specifically for DMA transfers:

- I3C_MasterTransferCreateHandleDMA
- I3C_MasterTransferDMA
- I3C_MasterInitTransferStateMachineDMA
- I3C_MasterTransferDMAHandleIRQ
- I3C_MasterRunTransferStateMachineDMA

The call relationship between them is shown in Figure 6.

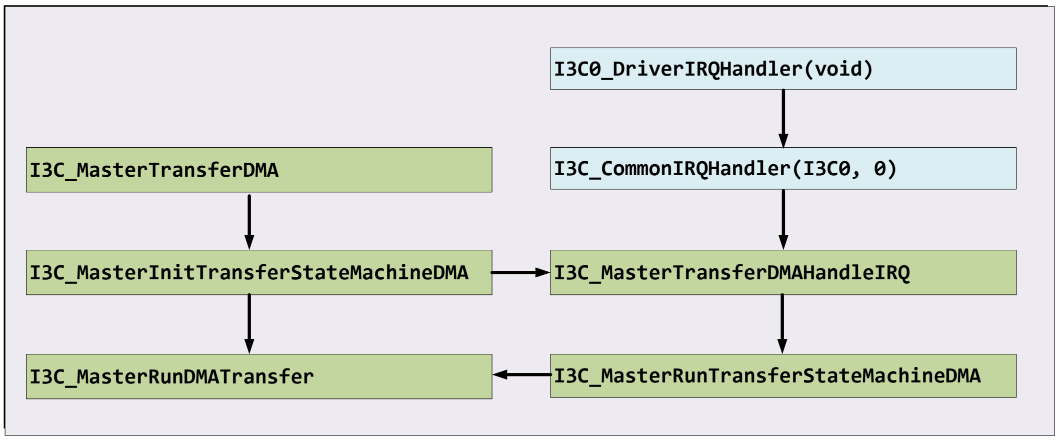
I3C0_DriverIRQHandler(void)
I3C_MasterTransferDMA
I3C_CommonIRQHandler(I3C0, 0)
I3C_MasterInitTransferStateMachineDMA
I3C_MasterTransferDMAHandleIRQ
I3C_MasterRunDMATransfer
I3C_MasterRunTransferStateMachineDMA

Figure 6. Call relationship

Similar to the example for interrupt transfers, DMA transfers also require a state machine architecture. The state machine calls back once when the transmission starts and it is called in the interrupt later.

For sending start, stop, and restart, signals and processing IBI signals, the controller executes directly in the interrupted state machine. For address sending and data sending and receiving, it is handled by calling the I3C_MasterRunDMATransfer function. In the I3C_MasterRunDMATransfer function, it is used mainly to configure the DMA channel and enable the transfer.

The DMA must configure the number of bytes transferred in advance. For controller sending, this is the active behavior of the controller and the number of bytes transferred by DMA can be configured in advance. For controller reception, it must be aligned with the target in advance. In the DMA example, the target sends data to the controller through the IBI interrupt and this data provides the number of bytes that the target sends to the controller.

4.5.2 Target operations

The target's DMA transfer code architecture is similar to the interrupt transfer mode and there is no state machine as a controller. However, the bus signal event is passively processed in the interrupt. For data transfer, the I3C_SlaveTransferDMA function is used.

In the I3C_MasterTransferHandleIRQ interrupt function, the following flags are processed:

- kSlaveErrorFlags
- kI3C_SlaveEventSentFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveMatchedFlag

They all eventually jump to the i3c_slave_callback callback function to run. In the i3c_slave_callback function, not all states are processed. Only kI3C_SlaveCompletionEvent and kI3C_SlaveRequestSentEvent are processed. The operations performed by the target are not as complicated as the controller. Figure 7 shows the execution flow chart of the function I3C_MasterTransferHandleIRQ.

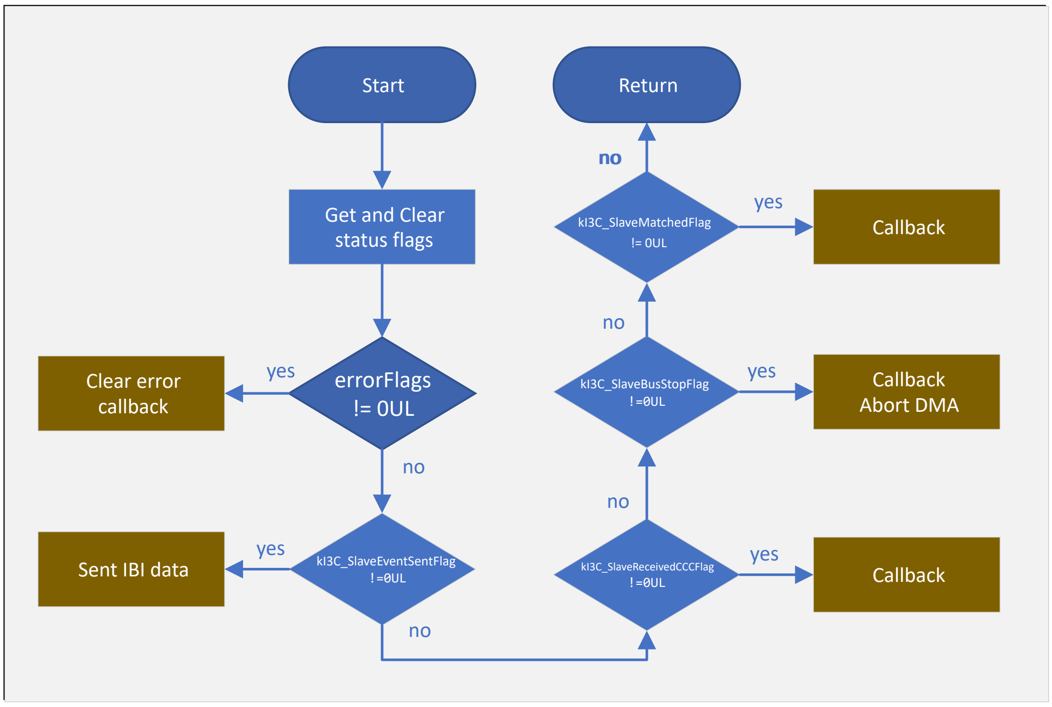
Start
Return
Get and Clear status flags
errorFlags != 0UL
Clear error callback
yes
no
kI3C_SlaveEventSentFlag !=0UL
Sent IBI data
yes
no
kI3C_SlaveReceivedCCCFlag !=0UL
yes
Callback
no
kI3C_SlaveBusStopFlag !=0UL
yes
Callback Abort DMA
no
kI3C_SlaveMatchedFlag != 0UL
yes
Callback
no

Figure 7. I3C_MasterTransferHandleIRQ flow chart

The data transfer about DMA is implemented through the I3C_SlaveTransferDMA function as follows:

```
/* Start slave non-blocking transfer. */
memset(g_slave_rxBuff, 0, I3C_DATA_LENGTH);
slaveXfer.rxData     = g_slave_rxBuff;
slaveXfer.rxDataSize = I3C_DATA_LENGTH;
I3C_SlaveTransferDMA(EXAMPLE_SLAVE, &g_i3c_s_handle, &slaveXfer,
 kI3C_SlaveCompletionEvent);
```

In the I3C_SlaveTransferDMA function, the DMA is prepared according to the data transfer direction.

```
if ((transfer->txData != NULL) && (transfer->txDataSize != 0U))
{
    I3C_SlavePrepareTxDMA(base, handle);
    txDmaEn = true;
    width   = 2U;
}

if ((transfer->rxData != NULL) && (transfer->rxDataSize != 0U))
{
    I3C_SlavePrepareRxDMA(base, handle);
    rxDmaEn = true;
    width   = 1U;
```

Sending and receiving are realized by the I3C_MasterSend and I3C_MasterReceive functions. For sending data, write the data into the MWDATAB or MWDATAH register and write the last data into MWDATABE or MWDATAHE when the TX ready flag is set to one.

5 Summary

LPC86X provides an I3C to facilitate communication with devices with an I3C interface. Its speed is between I^2C and SPI and it has many unique features.

In the I3C specification, the speed of the I3C can be as high as 12.5 MHz. On LPC86X, such a high speed can be achieved through DMA transmission. If you use the interrupt and polling mode, the speed may be relatively low. The interrupt mode is more flexible. To increase the speed, you can optimize the code with removing some conditions and state judgments. However, it must be ensured that missing states will not occur and it must be used according to specific applications.

In the process of I3C debugging (in addition to online debugging of the code), the best way is to check the status of the bus in real time using a logic analyzer.

6 Revision history

Table 14 summarizes the changes done to this document since the initial release.

Revision history

Revision number	Release date	Description
1	26 July 2023	Initial revision

7 Note about the source code in the document

8 Legal information

8.1 Definitions

8.2 Disclaimers

8.3 Trademarks

Contents

Date of release: 26 July 2023
Document identifier: AN13952