CHAPTER 22

Serial Communication Protocols

This chapter introduces four important serial communication protocols, including universal asynchronous receiver and transmitter (UART), inter-integrated circuit (I^2C), serial peripheral interface (SPI), universal serial bus (USB). Serial communication transfers a single bit each time and uses either a single wire for each communication direction or a shared wire for both directions. It differs from parallel communications, which use multiple communication wires and can transfer several bits at the same time. Compared with parallel communications, serial communications provide lower speed, but allow longer cable length and are less expensive.

22.1 Universal Asynchronous Receiver and Transmitter

One of the most common usages of universal asynchronous receiver and transmitter (UART) is for exchanging data between a microprocessor and a PC serial port to debug software or monitor systems. UART also has been widely used for various peripherals, such as printers, terminals, and modems. The keyword "universal" means the serial interface is programmable. UART is often configured to communicate synchronously, which is then called USART.

The asynchronous transmission allows bits to be transmitted in a serial fashion without requiring the sender to provide a clock signal to the receiver. However, both senders and receivers must agree on the data transmission rate before the communication starts. The sender and the receiver should use the same baud rate to set up the clock agreement. In digital systems, the baud rate is the bit rate, *i.e.*, the number of bits transmitted per second. Usually, the UART interface can tolerate a clock shift up to 10% during the transmission. In some analog systems, such as modems, the baud rate is larger than the corresponding bit rate when there are more than two voltage levels and a voltage signal transmitted can represent multiple bits.

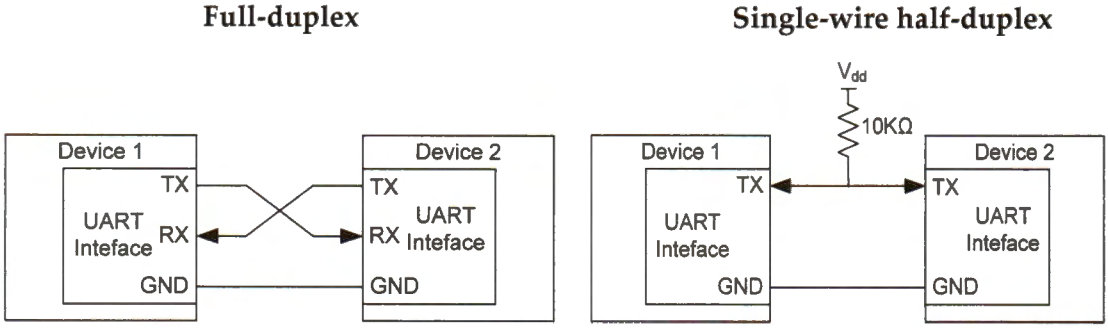

Figure 22-1. Connection between two UART devices in asynchronous mode

The transmission involves two communication lines (TX and RX), as shown in Figure 22-1.

- With full-duplex communication, data is always transmitted out bit by bit from the TX line and is received by the other device on its RX line. The receiver reassembles bits received into bytes.
- With the single-wire half-duplex communication, TX and RX are internally connected, and only one wire is used. TX is used for both sending and receiving data. In this mode, TX pins are pulled up externally because these two pins must be configured as open-drain.

For synchronous serial communication, the clock (CLK) pin of the devices must be connected. Also, the CTS (clear to send) line must connect with the RTS (request to send) line of the other device.

22.1.1 Communication Frame

UART divides data to be transmitted into frames. A frame is the smallest unit of communication. In a frame, the data length (7, 8, or 9 bits), the parity bit (even, odd, or no parity), the number of stop bits (0.5, 1, 1.5, or 2 bits), and the data order (MSB or LSB first) are configurable. Figure 22-2 shows one commonly used data frame: 8-P-1 (8 data bits, parity, 1 stop bit).

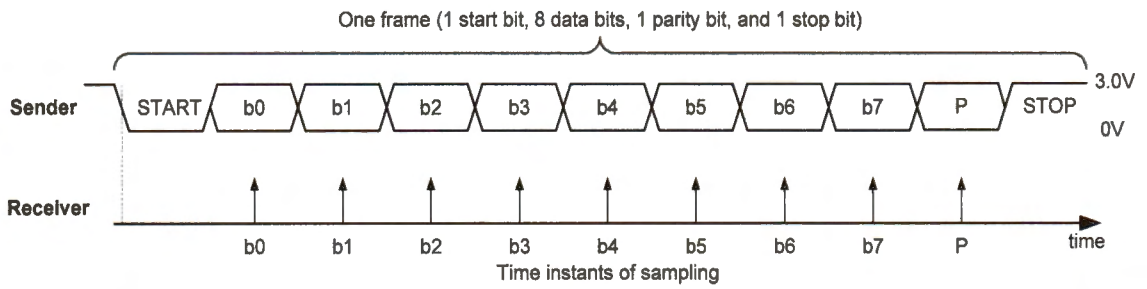

Figure 22-2. 8-P-1 frame (a start bit, eight data bits, one parity bit, and one stop bit). The least significant (LSB) of the data is sent out first in this example.

Each frame begins with a start bit, represented by a low-level voltage. After the start bit, the individual bits of each frame are shifted out of one UART interface and into another. Software can configure the data transmission order. Either the least significant bit (LSB) or the most significant bit (MSB) can be sent first. For example, suppose the LSB is sent first. When UART sends `0xE1`, the bit stream `10001111` (read from left to right) is seen on the transmission line. The number of bits in the data can be programmed to be 7, 8 or 9.

When the sender sends a frame, the sender can optionally calculate the parity of this frame and send the parity bit to the receiver for error checking. The optional parity bit helps improve the data integrity. The parity bit uses a high-level voltage to represent a logic `0` and a low-level voltage to represent a logic `1`. Software can configure logic `1` on the parity bit to represent either an odd or even number of ones in the transmitted data.

- ***Even parity***. The combination of data bits and the parity bit contains an even number of 1s.
- ***Odd parity***. The total number of 1s in the data bits and the parity bit is an odd number.

For example, if the data bits are `00010001` in binary, the parity bit will be 1 if odd parity is used, and 0 if even parity is used.

Each frame ends with a stop bit, represented by a high voltage. If no further data is transmitted, the voltage of the transmission line remains high. If the receiver does not obtain the stop bit, the current frame is considered corrupted and discarded. Additionally, the number of stop bit in each frame is usually one by default. However, it can be programmed to have 0.5, 1, 1.5, or 2 stop bits.

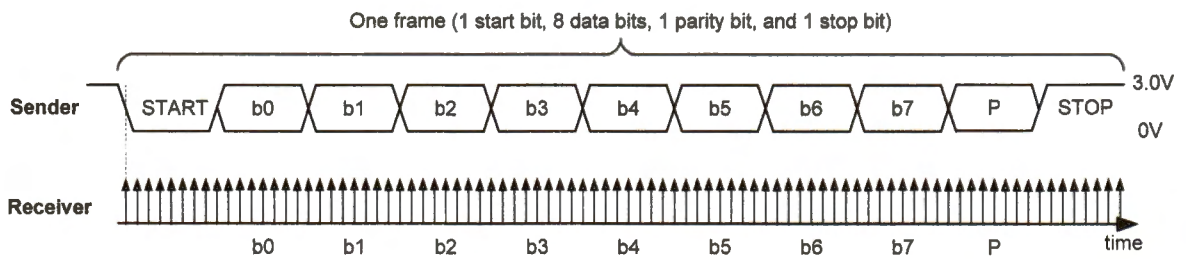

Figure 22-3. Receiver oversamples each bit 8 times

As the transmitter and receiver clocks are independent of each other, oversampling is an effective approach to mitigate the effects of clock deviation and avoid corruption by high-frequency noise. The most commonly used sampling rate is 8 or 16 times the baud rate (introduced in the next section). The receiver samples each bit 8 or 16 times and uses these values to estimate the middle of each bit pulse, resulting in a more reliable and robust transmission link.

22.1.2 Baud Rate

Historically the baud rate was used in telecommunications to represent the number of pulses or transitions physically transferred per second.

Baud Rate ≠ Bit Rate

- By using phase shift and other technologies, a pulse on phone lines can represent multiple binary bits, resulting in a bit rate larger than the baud rate.
- In digital communication systems, because each pulse represents a single bit, the baud rate is the number of bits physically transferred per second, including the actual data content and the protocol overhead, leading to a bit rate lower than the baud rate.

For example, if the baud rate is 9600, and an 8-N-1 frame consists of a start bit, 8 data bits, a stop bit, and no parity bit, then the transmission rate of actual data is not 9600 bits per second/8 = 1200 bytes per second. Instead, it is 9600/(1 + 8 + 1) = 960 bytes per second. The start and stop bits are the protocol overhead.

On STM32L4, the baud rate is calculated as follows:

$$Baud\ Rate = \frac{(1 + OVER8) \times f_{PCLK}}{USARTDIV}$$

where f_{PCLK} is the clock frequency of the processor. The divider *USARTDIV* is stored in the Baud Rate Register (`BRR`) . The value of `OVER8` is defined as follows.

$$OVER8 = \begin{cases} 0, & Signal\ is\ oversampled\ by\ 16 \\ 1, & Signal\ is\ oversampled\ by\ 8 \end{cases}$$

Also, the divider *USARTDIV* can be calculated from `BRR`.

$$USARTDIV = \begin{cases} BRR, & Signal\ is\ oversampled\ by\ 16 \\ BRR[15:4] \times 16 + BRR[2:0] \times 2, & Signal\ is\ oversampled\ by\ 8 \end{cases}$$

Example 1: Oversampling by 16, processor core 80 MHz, baud rate = 9600. Find *BRR*.

$$OVER8 = 0$$

$$USARTDIV = \frac{(1 + OVER8) \times f_{PCLK}}{Baud\ Rate} = \frac{80000000}{9600} = 8333.33 \approx 8333$$

$$BRR = USARTDIV = 8333 = 0x208D$$

Example 2: Oversampling by 8, processor core 80 MHz, baud rate = 9600. Find *BRR*.

$$OVER8 = 1$$

$$USARTDIV = \frac{(1 + OVER8) \times f_{PCLK}}{Baud\ Rate} = \frac{2 \times 80000000}{9600} = 16666.67$$
$$\approx 16667$$

The hex equivalent of 1667 is `0x411B`.

$$BRR[3:0] = USARTDIV[3:0] \gg 1 = 0xB \gg 1 = 0x5$$

$$BRR[15:4] = USARTDIV[15:4] = 0x411$$

$$BRR = BRR[15:4]:BRR[3:0] = 0x4115$$

22.1.3 UART Standards

Voltage signals for UART are defined in different standards, such as RS-232, RS-422, and RS-485. The prefix RS stands for "recommended standard." Table 22-1 compares these three standards. While the voltage of the TX and RX line in RS-422 and RS-485 is differential, with two separate wires for each line, RS-232 uses a single-ended voltage with a shared ground.

Figure 22-4 compares ***single-ended*** and ***differential signaling***. Besides the shared ground, single-ended signaling uses just one wire to transmit signals. Differential signaling uses two twisted wires with equal but opposite signals to transmit digital data. Electrical noise can be inducted into the signal wires or can be generated by the voltage difference between two ground references. Noise is coupled into both wires equally. Therefore, the noise can be canceled out at the receiver. Compared with single-ended signaling, differential signaling can transmit a higher frequency signal over a greater distance.

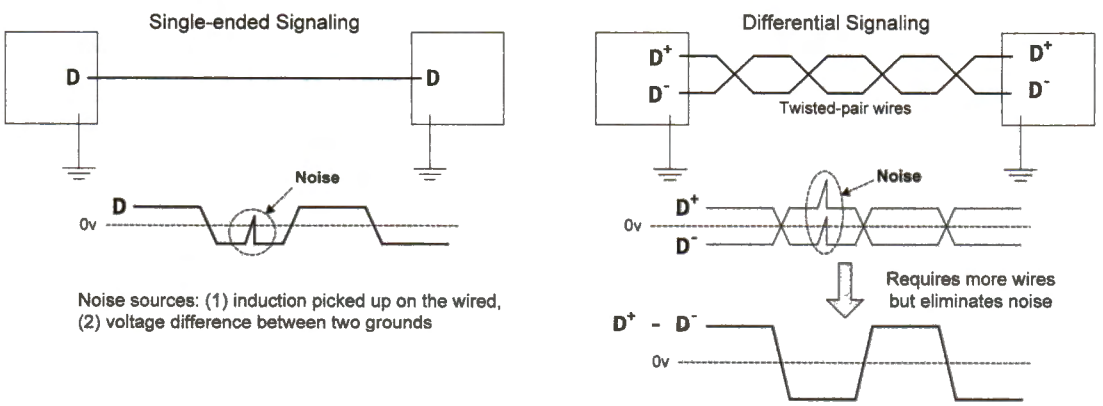

Figure 22-4. Comparison of single-ended and differential signaling

In RS-232, a voltage signal between +5V and +15V represents a logic one being transmitted, whereas a signal between -5V and -15V represents a logic zero. The receiver must interpret a voltage with +3V and +25V as a logic one, and a voltage with -3V to -25V as a logic zero. Any voltage signals between -3V and + 3V are invalid data. When the line is idle, the line must be driven to logic zero.

	RS-232	RS-422	RS-485
Voltage signal	Single-ended (logic 1: +5 to +15V, logic 0: -5 to -15 V)	Differential (-6V to +6V)	Differential (-7V to +12V)
Max distance	50 feet	4000 feet	4000 feet
Max speed	20 Kbit/s	10 Mbit/s	10 Mbit/s
Number of devices	1 master, 1 receiver	1 master, 10 receivers	32 masters, 32 receivers
Mode	Full duplex	Full duplex, half duplex	Full duplex, half duplex

Table 22-1. Comparing popular UART interfaces

Most modern computers only provide USB ports, not UART ports. Some old computers have RS-232 serial ports. However, we cannot directly connect an STM32 processor to the RS-232 port on a computer due to the voltage incompatibility. The STM32 processors can only tolerate voltage signals under 5V. Additionally, STM32 uses 0V to represent logic zero and 3V to represent logic one.

The FT232R chip converts a UART port to a standard USB interface, as shown below.

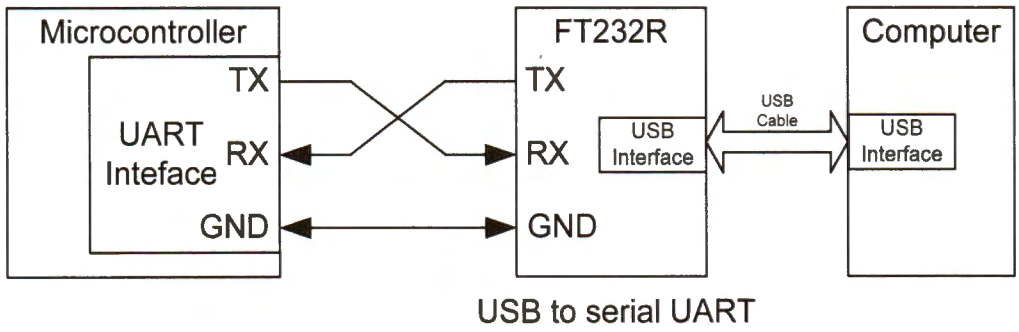

Figure 22-5. Serial communication via a USB-to-UART converter

The following diagram displays the voltage signal of the UART port when transmitting two data bytes, `0x32`, and `0x3C`. Each data frame includes one start bit, 8-bit data, and one stop bit. No parity bit is used in this example. After the start bit, the least significant bit

of the data is transmitted first. For example, the binary value of `0x32` is `0b00110010`, and the bit sequence seen on the transmission (TX) line is `01001100`. The baud rate is set to 9,600, and thus each bit takes approximately 0.104ms. When the TX line is idle, the voltage on it is 3V. The start bit has 0V while the stop bit has 3V.

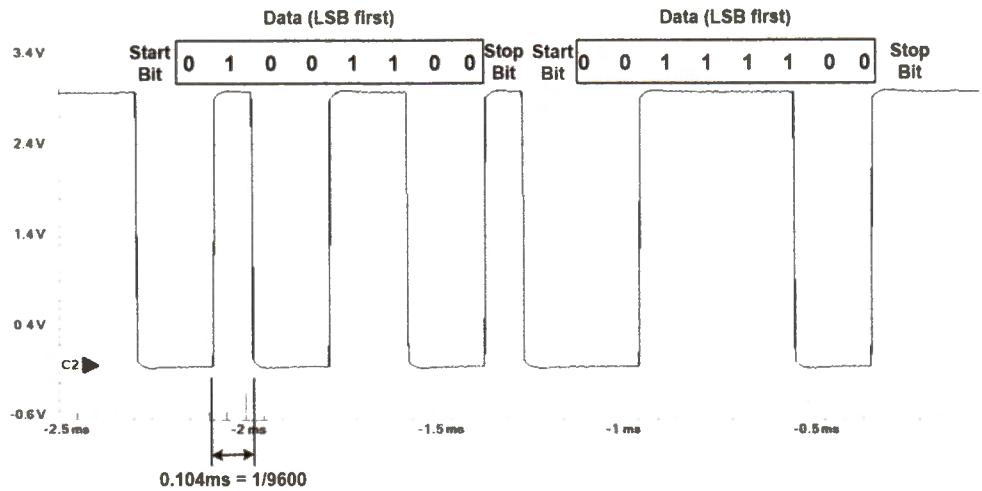

Figure 22-6. Voltage signal when transmitting 0x32 and 0x3C via UART (1 start bit, 1 stop bit, 8 data bits, no parity, baud rate = 9,600)

22.1.4 UART Communication via Polling

The following sections introduce how to send or receive data via UART ports by using three different methods: polling, interrupt, and DMA. Polling is the simplest but most inefficient method. The interrupt approach is more efficient but not suitable for high data transfer rates. The DMA method is complex but the most effective.

Figure 22-7 shows the connection of two UART ports between two processors. The TX pin of one processor is connected to the RX pin of the other processor, and vice versa. Because the polling method blocks the processor from running other tasks, software cannot use polling to send and receive data simultaneously.

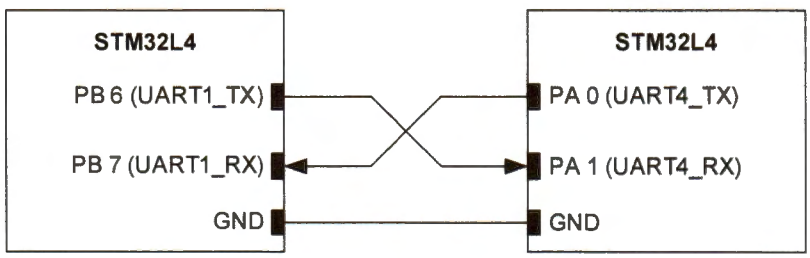

Figure 22-7. Connecting two UART ports

The code in Example 22-1 initializes a UART port in asynchronous mode (no hardware flow control) with oversampling by 16. Assume the UART clock is 80 MHz, and the baud rate is 9600. The data frame consists of 8 data bits, 1 start bit, 1 stop bit, and no parity bit. Software can initialize the UART ports using the following functions.

```
USART_Init(UART4);
USART_Init(USART1);
```

UART4 and USART1 are `struct` variables defined in the device header file (`stm32l476xx.h`).

```
void USART_Init (USART_TypeDef * USARTx) {

  // Disable USART
  USARTx->CR1 &= ~USART_CR1_UE;

  // Set data length to 8 bits
  // 00 = 8 data bits, 01 = 9 data bits, 10 = 7 data bits
  USARTx->CR1 &= ~USART_CR1_M;

  // Select 1 stop bit
  // 00 = 1 Stop bit      01 = 0.5 Stop bit
  // 10 = 2 Stop bits     11 = 1.5 Stop bit
  USARTx->CR2 &= ~USART_CR2_STOP;

  // Set parity control as no parity
  // 0 = no parity,
  // 1 = parity enabled (then, program PS bit to select Even or Odd parity)
  USARTx->CR1 &= ~USART_CR1_PCE;

  // Oversampling by 16
  // 0 = oversampling by 16, 1 = oversampling by 8
  USARTx->CR1 &= ~USART_CR1_OVER8;

  // Set Baud rate to 9600 using APB frequency (80 MHz)
  // See Example 1 in Section 22.1.2
  USARTx->BRR = 0x208D;

  // Enable transmission and reception
  USARTx->CR1 |= (USART_CR1_TE | USART_CR1_RE);

  // Enable USART
  USARTx->CR1 |= USART_CR1_UE;

  // Verify that USART is ready for transmission
  // TEACK: Transmit enable acknowledge flag. Hardware sets or resets it.
  while ((USARTx->ISR & USART_ISR_TEACK) == 0);

  // Verify that USART is ready for reception
  // REACK: Receive enable acknowledge flag. Hardware sets or resets it.
  while ((USARTx->ISR & USART_ISR_REACK) == 0);
}
```

Example 22-1. Initializing a UART port

Example 22-2 selects the system clock to drive USART1 and UART4.

```
int main(void){

  // Enable GPIO clock and configure the Tx pin and the Rx pin as:
  // Alternate function, High Speed, Push-pull, Pull-up

  //------------------  GPIO Initialization for USART 1 -----------------
  // PB.6 = AF7 (USART1_TX), PB.7 = AF7 (USART1_RX), See Appendix I
  RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN; // Enable GPIO port B clock

  // 00 = Input, 01 = Output, 10 = Alternate Function, 11 = Analog
  GPIOB->MODER   &= ~(0xF << (2*6)); // Clear mode bits for pin 6 and 7
  GPIOB->MODER   |=   0xA << (2*6);  // Select Alternate Function mode

  // Alternative function 7 = USART 1
  // Appendix I shows all alternate functions
  GPIOB->AFR[0]  |=   0x77 << (4*6); // Set pin 6 and 7 to AF 7

  // GPIO Speed: 00 = Low speed,  01 = Medium speed,
  //             10 = Fast speed, 11 = High speed
  GPIOB->OSPEEDR |=   0xF<<(2*6);

  // GPIO Push-Pull: 00 = No pull-up/pull-down, 01 = Pull-up (01)
  //                 10 = Pull-down, 11 = Reserved
  GPIOB->PUPDR   &= ~(0xF<<(2*6));
  GPIOB->PUPDR   |=   0x5<<(2*6); // Select pull-up

  // GPIO Output Type: 0 = push-pull, 1 = open drain
  GPIOB->OTYPER  &=  ~(0x3<<6);

  //------------------  GPIO Initialization for USART 4 -----------------
  // PA.0 = AF8 (UART4_TX), PA.1 = AF8 (UART4_RX), See Appendix I
  // The code is very similar to the one given above.
  ...

  RCC->APB2ENR  |= RCC_APB2ENR_USART1EN;   // Enable UART 1 clock
  RCC->APB1ENR1 |= RCC_APB1ENR1_UART4EN;   // Enable UART 4 clock

  // Select system clock (SYSCLK) USART clock source of UART 1 and 4
  // 00 = PCLK,   01 = System clock (SYSCLK),
  // 10 = HSI16,  11 = LSE
  RCC->CCIPR &= ~ (RCC_CCIPR_USART1SEL | RCC_CCIPR_UART4SEL);
  RCC->CCIPR |= (RCC_CCIPR_USART1SEL_0 | RCC_CCIPR_UART4SEL_0);

  USART_Init(USART1);
  USART_Init(UART4);
  ...
}
```

Example 22-2. Enable and select the clock of UART ports

When UART receives a byte, hardware sets the receive register not empty flag (RXNE) in the status register (ISR). In the polling approach, software constantly checks the RXNE flag and reads the receive data register (RDR) once it is set. Reading register RDR clears the RXNE flag automatically.

Example 22-3 shows the implementation of receiving data by polling. This polling method is inefficient, and the `while` loop prevents the processor from running other tasks.

```
void USART_Read (USART_TypeDef *USARTx, uint8_t *buffer, uint32_t nBytes) {

  int i;

  for (i = 0; i < nBytes; i++) {
    while (!(USARTx->ISR & USART_ISR_RXNE)); // Wait until hardware sets RXNE
    buffer[i] = USARTx->RDR;                 // Reading RDR clears RXNE
  }
}
```

Example 22-3. Receive data from a UART port via busy polling

When UART sends a byte, software must wait until the TxE (transmission data register empty) flag is set in the status register (ISR). Hardware sets the TxE flag when the content of the transmission data register (TDR) has been transferred into the shift register. Additionally, writing to the USART data register (DR) clears the TxE flag automatically. After exiting the *for* loop, software must wait for the transmission complete (TC) flag to ensure the last byte has been sent out.

Example 22-4 shows the implementation of sending data by polling. Again, the `while` loop prevents the processor from performing other tasks.

```
void USART_Write (USART_TypeDef *USARTx, uint8_t *buffer, uint32_t nBytes) {

  int i;

  for (i = 0; i < nBytes; i++) {
    while (!(USARTx->ISR & USART_ISR_TXE)); // Wait until hardware sets TXE
    USARTx->TDR = buffer[i] & 0xFF;         // Writing to TDR clears TXE flag
  }

  // Wait until TC bit is set. TC is set by hardware and cleared by software.
  while (!(USARTx->ISR & USART_ISR_TC));    // TC: Transmission complete flag

  // Writing 1 to the TCCF bit in ICR clears the TC bit in ISR
  USARTx->ICR |= USART_ICR_TCCF; // TCCF: Transmission complete clear flag
}
```

Example 22-4. Send data out via a UART port via busy polling

22.1.5 UART Communication via Interrupt

An USART interrupt can be generated upon the occurrence of several events, such as transmission data register empty (TxE), transmission complete (TC), received data register not empty (RXNE), overrun error detected (ORE), idle line detected (IDLE), and parity error (PE).

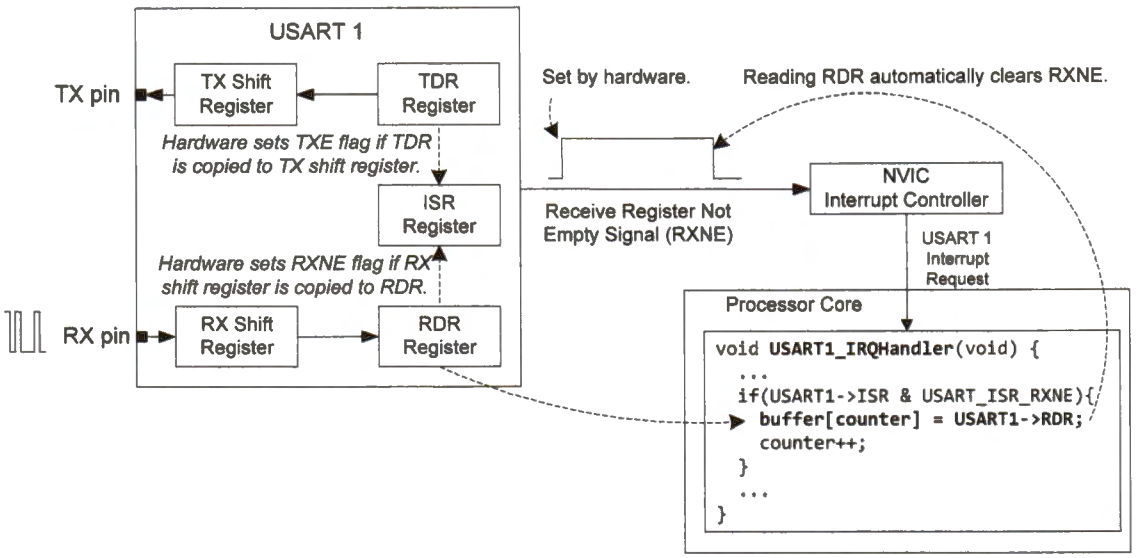

Figure 22-8. Using interrupt to receive data from USART 1

When UART receives a byte, an interrupt request is generated, and the processor responds to the request by executing the corresponding UART interrupt handler. The interrupt handler reads the receive data register (RDR) and copies it to the next empty buffer, as shown in Figure 22-8. Because several UART events can generate interrupts, the interrupt handler must check whether an RXNE event has taken place.

Software must enable UART interrupts to send or receive data, as shown in Example 22-5.

```
USART1->CR1 |= USART_CR1_RXNEIE;    // Receive register not empty interrupt
USART1->CR1 &= ~USART_CR1_TXEIE;    // Transmit register empty interrupt
NVIC_SetPriority(USART1_IRQn, 0);   // Set the highest urgency
NVIC_EnableIRQ(USART1_IRQn);        // Enable NVIC interrupt
```

Example 22-5. Enable UART sending and receiving interrupts

Example 22-6 shows the implementation of receiving data from UART by using interrupts. There are two global counter variables to record the number of bytes that have been received. The *receive()* function is generic, and is called by different UART interrupt handlers. Therefore, it takes three input arguments to differentiate UART ports, the receive buffers, and the byte counters.

```
#define BufferSize 32
uint8_t USART1_Buffer_Rx[BufferSize], USART4_Buffer_Rx[BufferSize];
volatile uint32_t Rx1_Counter = 0, Rx4_Counter = 0;

void USART1_IRQHandler(void) {
  receive(USART1, USART1_Buffer_Rx, &Rx1_Counter);
}

void UART4_IRQHandler(void) {
  receive(UART4, USART4_Buffer_Rx, &Rx4_Counter);
}

void receive(USART_TypeDef *USARTx, uint8_t *buffer, uint32_t *pCounter) {
  if(USARTx->ISR & USART_ISR_RXNE) {   // Check RXNE event
     buffer[*pCounter] = USARTx->RDR;  // Reading RDR clears the RXNE flag
     (*pCounter)++;                    // Dereference and update memory value
     if((*pCounter) >= BufferSize) {   // Check buffer overflow
        (*pCounter) = 0;               // Circular buffer
     }
  }
}
```

Example 22-6. Receiving data from a UART port via interrupt

Example 22-7 gives a generic implementation to transmit data via a UART port by using interrupts. For example, software can send out the data by executing the following statement: `UART_Send(USART1,buffer)`, which writes only the first byte to the transmit data register (`TDR`). This will start the transmission process. This function will immediately return to the caller after enabling `TXE` interrupt and write the first byte to the transmit data register (`TDR`). This allows the caller to continue to execute other tasks while the transmission is being performed in the background, as shown in Figure 22-9.

An interrupt will be generated after each byte has been sent. The interrupt handler writes the next byte to `TDR` to start the next transmission. This process repeats until a total of `BufferSize` bytes have been sent. The interrupt disables the interrupt for the `TXE` events.

```
volatile uint32_t Tx1_Counter = 0, Tx4_Counter = 0;

void UART_Send (USART_TypeDef *USARTx, uint8_t *buffer){
   USARTx->CR1 |= USART_CR1_TXEIE; // Enable TXE Interrupt
   // Write to Transmit Data Register (TDR) to start transmission
   // An interrupt will be initiated after data in TDR has been sent.
   USARTx->TDR = buffer[0];
}

void USART1_IRQHandler(void) {
  send(USART1, USART1_Buffer_Tx, &Tx1_Counter);
}
```

```
void UART4_IRQHandler(void) {
  send(UART4, USART4_Buffer_Rx, &Tx4_Counter);
}

void send(USART_TypeDef *USARTx, uint8_t *buffer, uint32_t *pCounter){
  if(USARTx->ISR & USART_ISR_TXE) {            // Check TXE flag
     (*pCounter)++;                            // Bytes that have been sent
     if(*pCounter <= BufferSize - 1) {         // Transmit the next byte
        USARTx->TDR = buffer[pCounter] & 0xFF; // Writing to TDR clears TXE
     } else {                                  // Transmission completes
        (*pCounter) = 0;                       // Clear the counter
        USARTx->CR1 &= ~USART_CR1_TXEIE;       // Disable TXE interrupt
     }
  }
}
```

Example 22-7. Send data out via a UART port by using interrupt

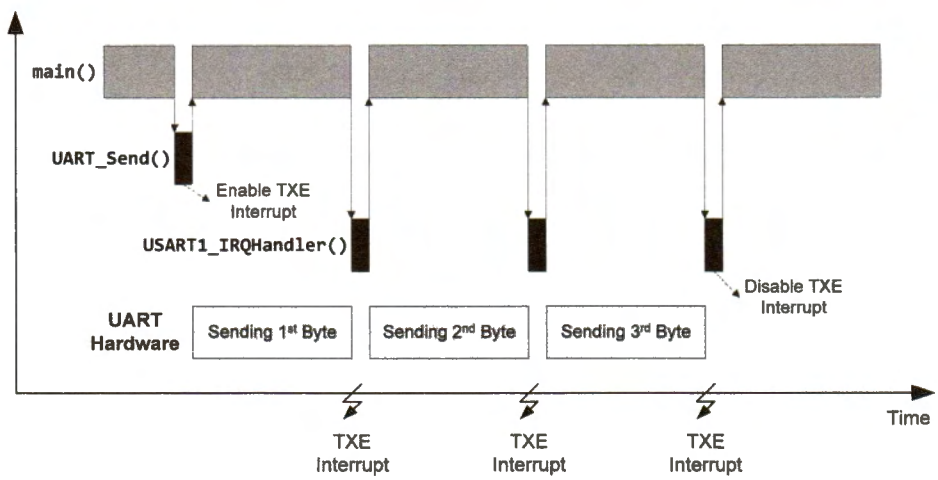

Figure 22-9. Sending three bytes via USART1 by using interrupts

Due to the non-blocking feature of interrupt, we can send and receive data simultaneously between two UART ports on the same processor, as shown in Figure 22-10.

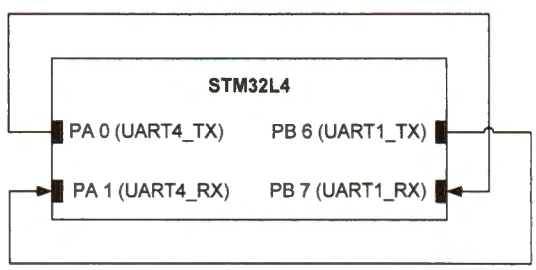

Figure 22-10. Communication between two ports on the same processor

22.1.6 UART Communication via DMA

Using a direct memory access (DMA) controller to move data between a buffer and UART data registers is the most efficient way to perform UART communication. Chapter 19 introduces DMA in detail.

Figure 22-11 shows the basic idea. As shown in Table 19-2, `USART1_TX` and `USART1_RX` can be connected to channels 6 and 7 of DMA controller 2, respectively. A `TXE` or `RXNE` event triggers a DMA request on channel 6 and channel 7, respectively.

- Whenever the `TXE` bit is set in the `ISR` register, DMA controller 2 transfers one byte via channel 6 from the memory buffer (pointed to by the `CMAR` register of DMA 2 channel 6) to the *transmit data register* `TDR` (pointed to by the `CPAR` register of DMA 2 channel 6).
- Similarly, whenever the `RXNE` bit is set in the `ISR` register, DMA controller 2 transfers one byte via channel 7 from the *receive data register* `RDR` to the buffer.

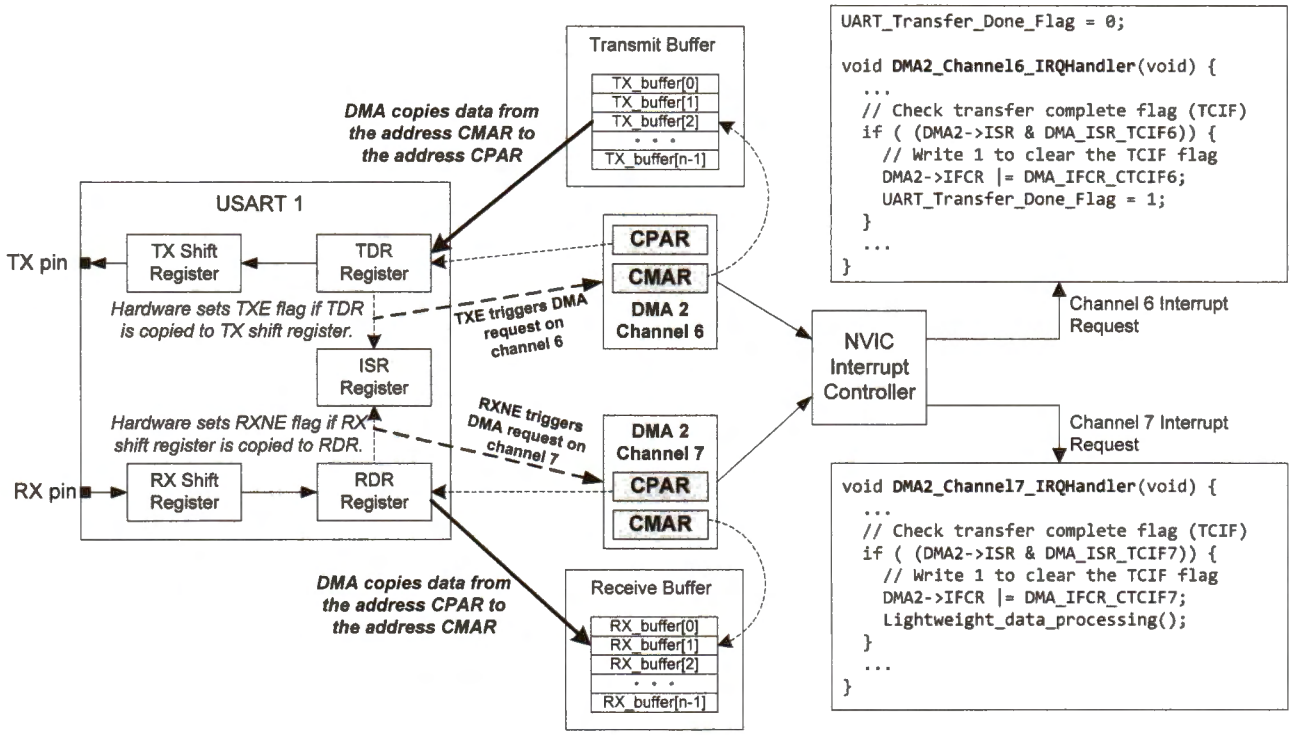

Figure 22-11. Using DMA to receive data from USART 1

To enable DMA for the transmission, software must set the `DMAT` bit in the `CR3` register.

```
USART1->CR3 |= USART_CR3_DMAT;
```

To enable DMA for the reception, software must set the `DMAR` bit in the `CR3` register.

```
USART1->CR3 |= USART_CR3_DMAR;
```

If DMA interrupts are enabled, the DMA controller can generate interrupt requests to execute the corresponding interrupt handler.

Example 22-8 and Example 22-9 give C code that configures channel 6 and 7 of DMA controller 2 to serve TX and RX of UART 1, respectively. Data transmission or reception takes place immediately when *UART1_DMA_Transmit* or *UART1_DMA_Receive* is called.

```
void UART1_DMA_Transmit (uint8_t *pBuffer, uint32_t size) {
  RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;      // Enable DMA2 clock
  DMA2_Channel6->CCR &= ~DMA_CCR_EN;       // Disable DMA channel
  DMA2_Channel6->CCR &= ~DMA_CCR_MEM2MEM; // Disable memory to memory mode
  DMA2_Channel6->CCR &= ~DMA_CCR_PL;       // Channel priority level
  DMA2_Channel6->CCR |=  DMA_CCR_PL_1;     // Set DMA priority to high
  DMA2_Channel6->CCR &= ~DMA_CCR_PSIZE;    // Peripheral data size 00 = 8 bits
  DMA2_Channel6->CCR &= ~DMA_CCR_MSIZE;    // Memory data size: 00 = 8 bits
  DMA2_Channel6->CCR &= ~DMA_CCR_PINC;     // Disable peripheral increment mode
  DMA2_Channel6->CCR |=  DMA_CCR_MINC;     // Enable memory increment mode
  DMA2_Channel6->CCR &= ~DMA_CCR_CIRC;     // Disable circular mode
  DMA2_Channel6->CCR |=  DMA_CCR_DIR;      // Transfer direction: to peripheral
  DMA2_Channel6->CCR |=  DMA_CCR_TCIE;     // Transfer complete interrupt enable
  DMA2_Channel6->CCR &= ~DMA_CCR_HTIE;     // Disable Half transfer interrupt
  DMA2_Channel6->CNDTR = size;             // Number of data to transfer
  DMA2_Channel6->CPAR  = (uint32_t)&(USART1->TDR);  // Peripheral address
  DMA2_Channel6->CMAR  = (uint32_t) pBuffer;        // Transmit buffer address
  DMA2_CSELR->CSELR &= ~DMA_CSELR_C6S;     // See Table 19-2
  DMA2_CSELR->CSELR |= 2<<20;              // Map channel 6 to USART1_TX
  DMA2_Channel6->CCR |= DMA_CCR_EN;        // Enable DMA channel
}
```

Example 22-8. Configure DMA 2 channel 6 for UART 1 transmit

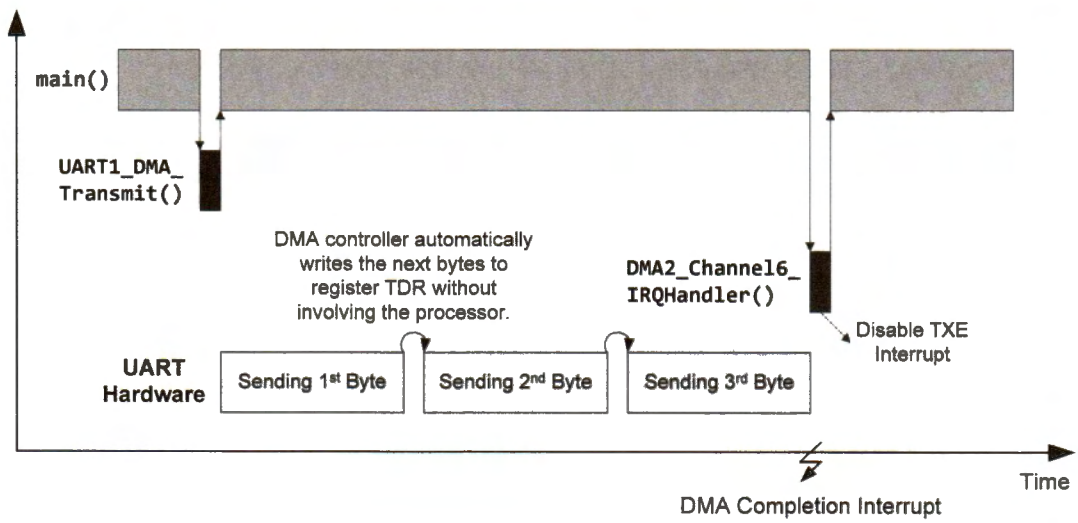

Figure 22-12. Sending three bytes via DMA.

```
void UART1_DMA_Receive (uint8_t *pBuffer, uint32_t size) {
  RCC->AHB1ENR |= RCC_AHB1ENR_DMA2EN;     // Enable DMA2 clock
  DMA2_Channel7->CCR &= ~DMA_CCR_EN;      // Disable DMA channel
  DMA2_Channel7->CCR &= ~DMA_CCR_MEM2MEM;// Disable memory to memory mode
  DMA2_Channel7->CCR &= ~DMA_CCR_PL;      // Channel priority level
  DMA2_Channel7->CCR |=  DMA_CCR_PL_1;    // Set DMA priority to high
  DMA2_Channel7->CCR &= ~DMA_CCR_PSIZE;   // Peripheral data size 00 = 8 bits
  DMA2_Channel7->CCR &= ~DMA_CCR_MSIZE;   // Memory data size: 00 = 8 bits
  DMA2_Channel7->CCR &= ~DMA_CCR_PINC;    // Disable peripheral increment mode
  DMA2_Channel7->CCR |=  DMA_CCR_MINC;    // Enable memory increment mode
  DMA2_Channel7->CCR &= ~DMA_CCR_CIRC;    // Disable circular mode
  DMA2_Channel7->CCR &= ~DMA_CCR_DIR;     // Transfer direction: to memory
  DMA2_Channel7->CCR |=  DMA_CCR_TCIE;    // Transfer complete interrupt enable
  DMA2_Channel7->CCR &= ~DMA_CCR_HTIE;    // Disable Half transfer interrupt
  DMA2_Channel7->CNDTR = size;            // Number of data to transfer
  DMA2_Channel7->CPAR  = (uint32_t)&(USART1->RDR); // Peripheral address
  DMA2_Channel7->CMAR  = (uint32_t) pBuffer;       // Receive buffer address
  DMA2_CSELR->CSELR &= ~DMA_CSELR_C6S;    // See Table 19-2
  DMA2_CSELR->CSELR |= 2<<24;             // Map channel 7 to USART1_RX
  DMA2_Channel7->CCR |= DMA_CCR_EN;       // Enable DMA channel
}
```

Example 22-9. Configure DMA 2 channel 7 for UART 1 receive

Comparing Figure 22-9 and Figure 22-12, we can see that DMA is more efficient than the interrupt approach. When DMA completes, a DMA interrupt request will be generated. The DMA interrupt handler can change the completion flag, as shown below.

```
volatile uint8_t TransmissionCompleteFlag = 0;

void DMA2_Channel7_IRQHandler(void) { // USART1_RX

  if ( (DMA2->ISR & DMA_ISR_TCIF7) == DMA_ISR_TCIF7 ) {
    // Write 1 to clear the corresponding TCIF flag
    DMA2->IFCR |= DMA_IFCR_CTCIF7;
    TransmissionCompleteFlag = 1;
  }

  if ( (DMA2->ISR & DMA_ISR_HTIF7) == DMA_ISR_HTIF7 ) // half transfer
    DMA2->IFCR |= DMA_IFCR_CHTIF7;

  if ( (DMA2->ISR & DMA_ISR_GIF7) == DMA_ISR_GIF7 )   // global interrupt
    DMA2->IFCR |= DMA_IFCR_CGIF7;

  if ( (DMA2->ISR & DMA_ISR_TEIF7) == DMA_ISR_TEIF7 ) // transfer error
    DMA2->IFCR |= DMA_IFCR_CTEIF7;
}
```

Example 22-10. DMA interrupt handler

22.3 Serial Peripheral Interface Bus (SPI)

Serial peripheral interface (SPI) is a synchronous serial communication interface widely used to exchange data between a microprocessor and peripheral devices using four wires. For example, a digital camera often uses SPI to control its lens and save photos to a MMC or SD media.

SPI is simple, has low power requirements, and supports high throughput. Disadvantages of SPI include that it does not support multiple masters, and slaves cannot start the communication or control data transfer speed. The master initiates and controls all communications.

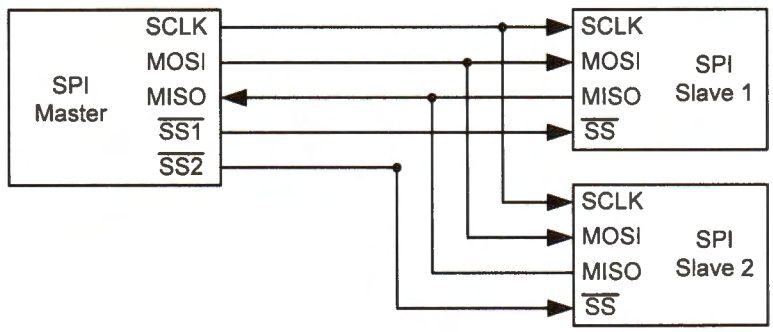

Figure 22-31. A SPI master device connecting to multiple SPI slave devices

A SPI interface consists of four lines: a master-in-slave-out data line (`MISO`), a master-out-slave-in data line (`MOSI`), a serial clock line (`SCLK`), and an active-low slave select line ($\overline{SS}$), as shown in Figure 22-31. SPI is also called four-wire serial interface.

SPI only supports a single master communicating with multiple slave devices. As shown in Figure 22-33, when the master wishes to exchange data with a slave, it pulls down the corresponding select line (SS_N). The master then generates clock pulses to coordinate the data transmission on the `MOSI` and `MISO` lines.

Data exchange can take place in both directions simultaneously, and this two-way serial channel is often called ***full duplex***. Data bits are transmitted on both the `MOSI` line and the `MISO` line synchronously, with the flow directions opposite to each other. Note the `SCLK` line has only one direction, and only the master can generate the clock signal. The slave devices cannot control the clock line.

When there are multiple slave devices, the master decides which slave device it wants to communicate. There is a dedicated Slave Select (`SS`) line for each slave device. The master

selects the target slave device by pulling the corresponding SS line to a low voltage prior to data transfer. The selected slave device then listens for the clock and MOSI signals. When there is only one slave device, the SS line can be directly connected to ground physically, or the program can make the slave continuously selected.

22.3.1 Data Exchange

SPI is a synchronous protocol, and the slave devices must send and receive data based on the clock provided by the master. It differs from an asynchronous protocol in which no clock signal is provided physically. SPI devices must exchange data at the same speed.

The master and a slave perform data exchange at synchronized time steps based on the clock signal generated by the master.

SPI master provides clock signal (SCLK) to SPI slaves.

- When a bit is shifted out on the MISO line from the slave's data register during a clock period, a new data bit is shifted into this register from the MOSI line in the same clock period, as shown in Figure 22-32.
- When one device writes a bit to the data line at the rising or falling edge of the clock, the other device then reads the bit at the opposite edge of the same clock period.
- The data transfer size is usually a byte or halfword (16 bits).

Communication from the master to a slave and communication from a slave to the master are always taking place concurrently. In each communication link (either MISO or MOSI), each device sends out a data item and at the same time receives a new data item. No devices can just be a transmitter or a receiver.

Therefore, when a slave wants to send data to the master via the MISO line, the slave must wait for the clock signal. At the same time, the master must send some dummy data out via the MOSI line to generate the clock signal to initiate the data transfer, as shown in Figure 22-32.

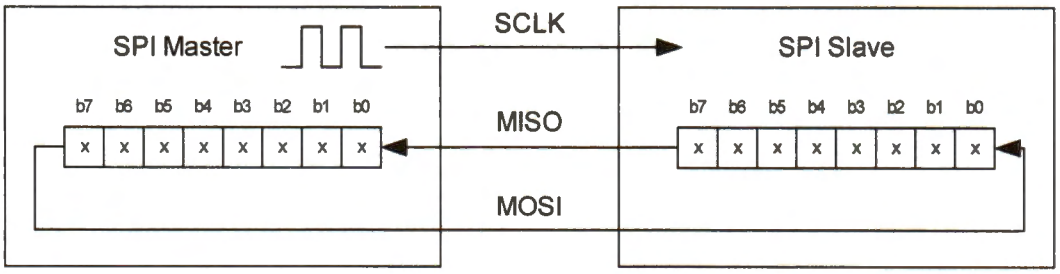

Figure 22-32. A byte is shifted out and in simultaneously via MOSI and MISO.

When a master exchanges data with slave *n*, the master must set $\overline{SS_n}$ low to select slave *n*, as shown in Figure 22-33. During the communication, the most significant bit of both data registers is sent out first.

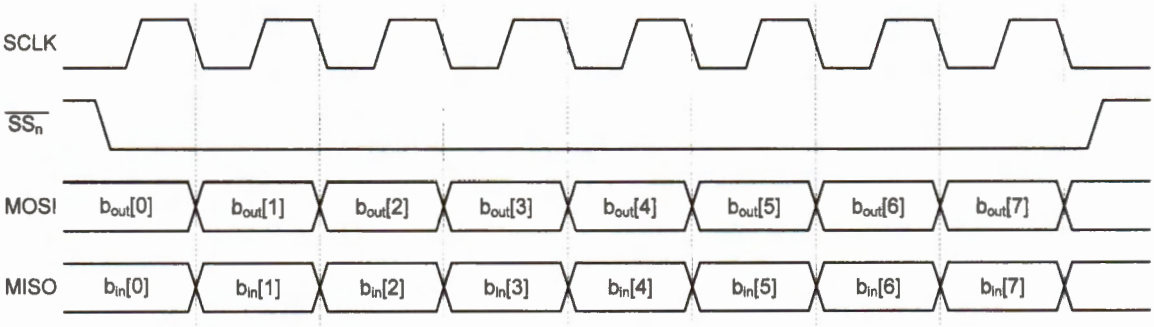

Figure 22-33. Communication signals between a master and slave *n*. In this example, the most significant bit is transferred first.

Figure 22-34 shows the signal of `SCLK` and `MOSI` when two bytes (`0xAA` and `0x3C`) are sent out. In this example, the least significant bit (LSB) of each data is sent out first.

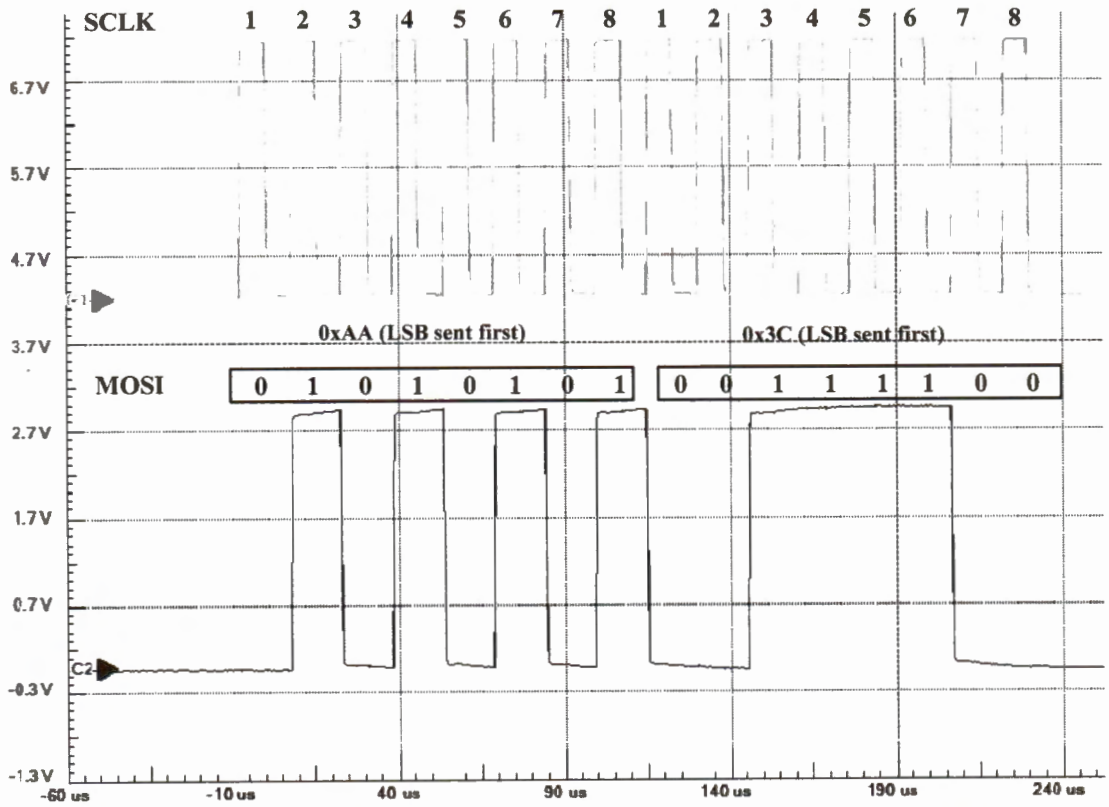

Figure 22-34. `SCLK` and `MOSI` signals when the master sends two bytes: `0xAA` and `0x3C`

22.3.2 Clock Configuration

The clock speed determines the data transfer rate. The data rate ranges from 1 to 20 megabits per second. The master can change the clock speed by programming the clock prescaler register. For STM32L processors, the baud rate control factor is stored in the `BR[2:0]` bits of the SPI control register (`CR1`). The `SCLK` clock frequency is programmed by setting the baud rate control factor.

$$f_{SCLK} = \frac{f_{SYSCLK}}{2^{1+BR[2:0]}}$$

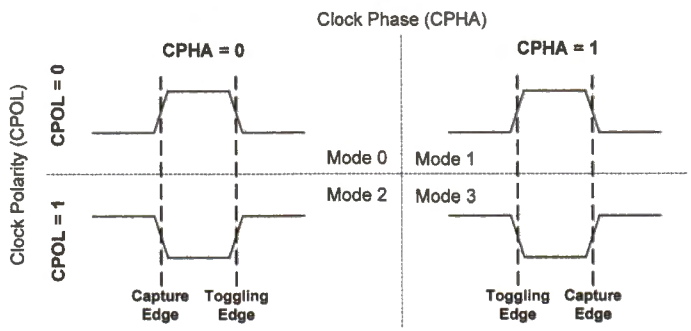

Figure 22-35. Configuration of clock phase and clock polarity

Four possible clock modes are available to program the clock edge used for data sampling and data toggling, as shown in Figure 22-35. The clock modes depend on two parameters: clock phase (`CPHA`) and clock polarity (`CPOL`). When `CPOL` is `0`, the `SCLK` line is pulled low during idle time. When `CPOL` is `1`, the `SCLK` line is pulled high during idle. When `CPHA` is `0`, the first clock transition (either rising or falling) is the first data capture edge. When `CPHA` is `1`, the second clock transition is the first capture edge.

The combination of `CPOL` and `CPHA` selects the clock edge for transmitting and capturing data. The capture of the first bit is delayed a half cycle in mode `0` and `2`.

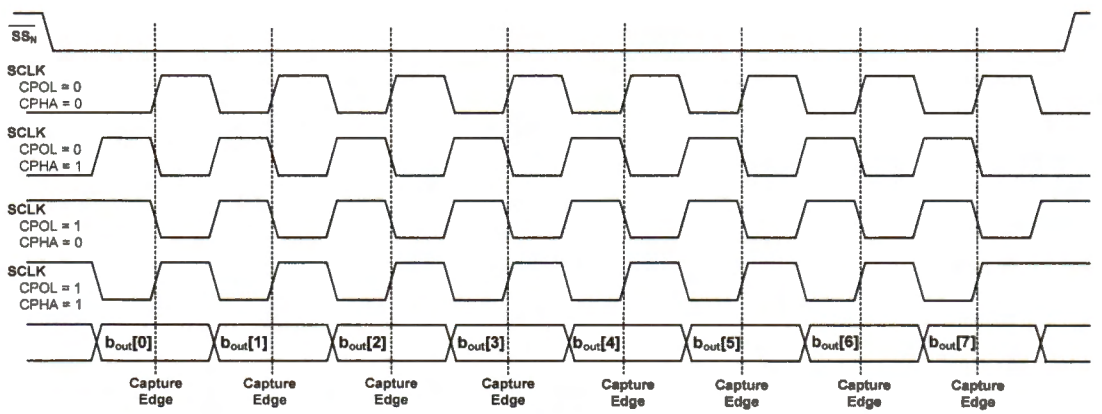

Figure 22-36. Receiving 8 bits under different settings of `CPOL` and `CPHA`

22.3.3 Using SPI to Interface a Gyroscope

This section shows how to interface the 3-axis gyro sensor L3GD20 which provides the angular velocity in three axes (yaw, pitch, and roll). The L3GD20 sensor supports two digital interfaces: I²C and SPI. When the voltage on the GYRO_CS pin is low, the SPI interface is selected. Otherwise, the I²C interface is selected.

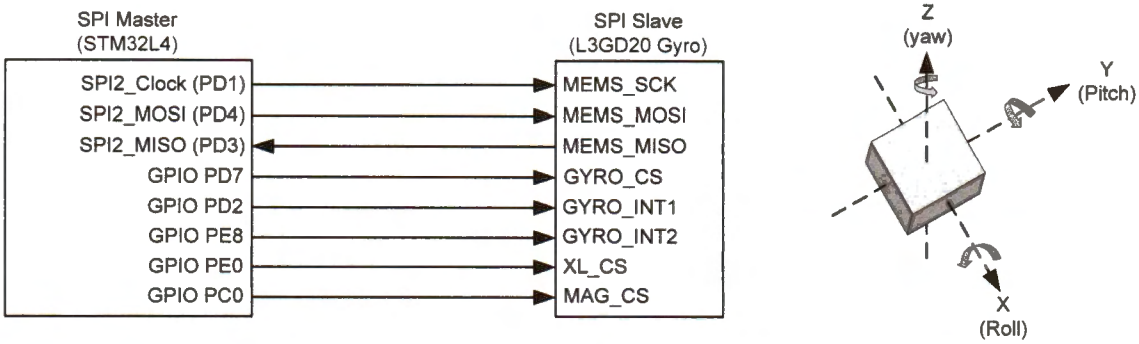

Figure 22-37. Connection between STM32L4 and L3GD20 gyroscope

The L3GD20 gyroscope internally has a set of 8-bit registers. The following program first reads the status register and checks whether angular velocity data are ready to read. If yes, the program reads 6 bytes of raw data and converts them into angular velocities.

```
#define L3GD20_STATUS_REG_ADDR          0x27  // Status register
#define L3GD20_OUT_X_L_ADDR             0x28  // Output Register
struct {
  float x;  // X axis rotation rate, degrees per second
  float y;  // Y axis rotation rate, degrees per second
  float z;  // Z axis rotation rate, degrees per second
} gyro;

int16_t gyro_x, gyro_y, gyro_z;
uint8_t gyr[6], status;

GYRO_IO_Read(&status, L3GD20_STATUS_REG_ADDR, 1); // Read status register
if ( (status & 0x08) == 0x08 ) {  // ZYXDA ready bit set
  // Read 6 bytes from gyro starting at L3GD20_OUT_X_L_ADDR
  GYRO_IO_Read(gyr, L3GD20_OUT_X_L_ADDR, 6);
  // Assume little endian (check the control register 4 of gyro)
  gyro_x = (int16_t) ((uint16_t) (gyr[1]<<8) + gyr[0]);
  gyro_y = (int16_t) ((uint16_t) (gyr[3]<<8) + gyr[2]);
  gyro_z = (int16_t) ((uint16_t) (gyr[5]<<8) + gyr[4]);
  // For +/-2000dps, 1 unit equals to 70 millidegrees per second
  gyro.x = (float) gyro_x * 0.070f;  // X angular velocity
  gyro.y = (float) gyro_y * 0.070f;  // Y angular velocity
  gyro.z = (float) gyro_z * 0.070f;  // Z angular velocity
}
```

Example 22-23. Reading x, y, and z rotation rates from the gyro sensor

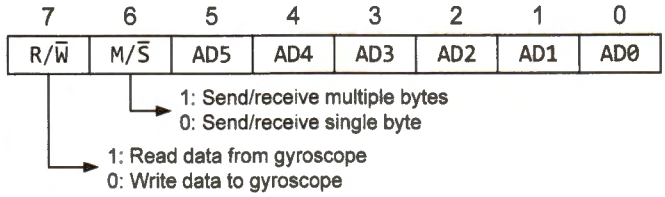

Figure 22-38. Command and address byte of internal registers in L3GD20 gyroscope

Example 22-24 and Example 22-25 show the procedure of writing data to and reading data from the gyro sensor by using the SPI interface. Figure 22-38 illustrates the bit definitions of the command and address byte. Bit 7 indicates the data transmission direction. Bit 6 shows whether single or multiple bytes will be transmitted. When the $M/\bar{S}$ bar is set, the address will be automatically incremented by 1 after each byte is transmitted.

```
// PD7: GYRO_CS (High = I2C, Low = SPI)
#define L3GD20_CS_LOW      GPIOD->ODR &= ~(1U << 7);
#define L3GD20_CS_HIGH     GPIOD->ODR |=  (1U << 7);

void GYRO_IO_Write (uint8_t *pBuffer, uint8_t WriteAddr, uint8_t size) {

  uint8_t rxBuffer[32];

  if (NumByteToWrite > 0x01) {
    WriteAddr |= 1U << 6; // Select the mode of writing multiple-byte
  }

  // Set SPI interface
  L3GD20_CS_LOW;  // 0 = SPI, 1 = I2C
  Delay(10);      // Short delay

  // Send the address of the indexed register
  SPI_Write(SPI2, &WriteAddr, rxBuffer, 1);

  // Send the data that will be written into the device
  // Bit transfer order: Most significant bit first
  SPI_Write(SPI2, pBuffer, rxBuffer, size);

  // Set chip select High at the end of the transmission
  Delay(10);      // Short delay
  L3GD20_CS_HIGH; // 0 = SPI, 1 = I2C
}
```

Example 22-24. Writing data to the gyro sensor via the SPI interface

```
void GYRO_IO_Read (uint8_t *pBuffer, uint8_t ReadAddr, uint8_t size) {

  uint8_t rxBuffer[32];

  // Select read & multiple-byte mode
  uint8_t AddrByte = ReadAddr | 1U << 7 | 1U << 6;

  // Set chip select low at the start of the transmission
  L3GD20_CS_LOW; // 0 = SPI, 1 = I2C
  Delay(10);     // Short delay

  // Send the address of the indexed register
  SPI_Write(SPI2, &AddrByte, rxBuffer, 1);

  // Receive the data that will be read from the device (MSB First)
  SPI_Read(SPI2, pBuffer, size);

  // Set chip select High at the end of the transmission
  Delay(10);      // Short delay
  L3GD20_CS_HIGH; // 0 = SPI, 1 = I2C
}
```

Example 22-25. Receiving data from the gyro sensor via the SPI interface

The following shows the initialization of SPI, which sets SPI as the master.

```
void SPI_Init(SPI_TypeDef * SPIx){

  // Enable SPI clock
  if(SPIx == SPI1){
    RCC->APB2ENR   |= RCC_APB2ENR_SPI1EN;      // Enable SPI1 Clock
    RCC->APB2RSTR  |= RCC_APB2RSTR_SPI1RST;    // Reset SPI1
    RCC->APB2RSTR  &= ~RCC_APB2RSTR_SPI1RST;   // Clear the reset of SPI1
  } else if(SPIx == SPI2){
    RCC->APB1ENR1  |= RCC_APB1ENR1_SPI2EN;     // Enable SPI2 Clock
    RCC->APB1RSTR1 |= RCC_APB1RSTR1_SPI2RST;   // Reset SPI2
    RCC->APB1RSTR1 &= ~RCC_APB1RSTR1_SPI2RST;  // Clear the reset of SPI2
  } else if(SPIx == SPI3){
    RCC->APB1ENR1  |= RCC_APB1ENR1_SPI3EN;     // Enable SPI3 Clock
    RCC->APB1RSTR1 |= RCC_APB1RSTR1_SPI3RST;   // Reset SPI3
    RCC->APB1RSTR1 &= ~RCC_APB1RSTR1_SPI3RST;  // Clear the reset of SPI3
  }

  SPIx->CR1 &= ~SPI_CR1_SPE;  // Disable SPI

  // Configure duplex or receive-only
  // 0 = Full duplex (transmit and receive), 1 = Receive-only
  SPIx->CR1 &= ~SPI_CR1_RXONLY;
```

```
// Bidirectional data mode enable: This bit enables half-duplex
// communication using common single bidirectional data line.
// 0 = 2-line unidirectional data mode selected
// 1 = 1-line bidirectional data mode selected
SPIx->CR1 &= ~SPI_CR1_BIDIMODE;

// Output enable in bidirectional mode
// 0 = Output disabled (receive-only mode)
// 1 = Output enabled (transmit-only mode)
SPIx->CR1 &= ~SPI_CR1_BIDIOE;

// Data Frame Format
SPIx->CR2 &= ~SPI_CR2_DS;
SPIx->CR2  =  SPI_CR2_DS_0 | SPI_CR2_DS_1 | SPI_CR2_DS_2;  // 0111: 8-bit

// Bit order
// 0 = MSB transmitted/received first
// 1 = LSB transmitted/received first
SPIx->CR1 &= ~SPI_CR1_LSBFIRST;  // Most significant bit first

// Clock phase
// 0 = The first clock transition is the first data capture edge
// 1 = The second clock transition is the first data capture edge
SPIx->CR1 &= ~SPI_CR1_CPHA; // 1st edge

// Clock polarity
// 0 = Set CK to 0 when idle
// 1 = Set CK to 1 when idle
SPIx->CR1 &= ~SPI_CR1_CPOL; // Polarity low

// Baud rate control:
// 000 = f_PCLK/2    001 = f_PCLK/4    010 = f_PCLK/8    011 = f_PCLK/16
// 100 = f_PCLK/32   101 = f_PCLK/64   110 = f_PCLK/128  111 = f_PCLK/256
// SPI baud rate is set to 5 MHz
SPIx->CR1 |= 3U<<3;          // Set SPI clock to 80MHz/16 = 5 MHz

// CRC Polynomial
SPIx->CRCPR = 10;

// Hardware CRC calculation disabled
SPIx->CR1 &= ~SPI_CR1_CRCEN;

// Frame format: 0 = SPI Motorola mode, 1 = SPI TI mode
SPIx->CR2 &= ~SPI_CR2_FRF;

// NSSGPIO: The value of SSI is forced onto the NSS pin and the IO value
// of the NSS pin is ignored.
// 1 = Software slave management enabled
// 0 = Hardware NSS management enabled
SPIx->CR1 |= SPI_CR1_SSM;
```

```
  // Set as Master: 0 = slave, 1 = master
  SPIx->CR1 |= SPI_CR1_MSTR;

  // Manage NSS (slave selection) by using Software
  SPIx->CR1 |= SPI_CR1_SSI;

  // Enable NSS pulse management
  SPIx->CR2 |= SPI_CR2_NSSP;

  // Receive buffer not empty (RXNE)
  // The RXNE flag is set depending on the FRXTH bit value in the SPIx_CR2 register:
  // (1) If FRXTH is set, RXNE goes high and stays high until the RXFIFO level is
  //     greater or equal to 1/4 (8-bit).
  // (2) If FRXTH is cleared, RXNE goes high and stays high until the RXFIFO level is
  //     higher than or equal to 1/2 (16-bit).
  SPIx->CR2 |= SPI_CR2_FRXTH;

  // Enable SPI
  SPIx->CR1 |= SPI_CR1_SPE;
}
```

Example 22-26. Initializing SPI

The following subroutine is for the SPI master to send the data to an SPI slave.

- It checks the transmission buffer empty flag (TXE) and waits until hardware sets TXE. If TXE is set, the transmission register is ready to accept the next data to be transmitted.
- Writing to the SPI data register (DR) automatically clears the TXE flag.
- The subroutine also waits until the busy flag is cleared to ensure the last data has been successfully sent.

```
void SPI_Write(SPI_TypeDef * SPIx, uint8_t *txBuffer, uint8_t * rxBuffer, int
size) {
  int i = 0;

  for (i = 0; i < size; i++) {

    // Wait for TXE (Transmit buffer empty)
    while( (SPIx->SR & SPI_SR_TXE ) != SPI_SR_TXE );
    SPIx->DR = txBuffer[i];

    // Wait for RXNE (Receive buffer not empty)
    while( (SPIx->SR & SPI_SR_RXNE ) != SPI_SR_RXNE );
    rxBuffer[i] = SPIx->DR;

  }

  // Wait for BSY flag cleared
  while( (SPIx->SR & SPI_SR_BSY) == SPI_SR_BSY );
}
```

Example 22-27. Send data to an SPI slave by using polling

The following subroutine allows an SPI master to receive data from an SPI slave. Only the master can initiate the data transfer and controls the communication clock (SCLK). Therefore, the master must send a dummy byte data to the slave to start the clock.

```
void SPI_Read(SPI_TypeDef * SPIx, uint8_t *rxBuffer, int size) {
  int i = 0;
  for (i = 0; i < size; i++) {
    // Wait for TXE (Transmit buffer empty)
    while( (SPIx->SR & SPI_SR_TXE ) != SPI_SR_TXE );
    // The clock is controlled by master.
    // Thus, the master must send a byte
    SPIx->DR = 0xFF;  // A dummy byte

    // data to the slave to start the clock.
    while( (SPIx->SR & SPI_SR_RXNE ) != SPI_SR_RXNE );
    rxBuffer[i] = SPIx->DR;
  }

  // Wait for BSY flag cleared
  while( (SPIx->SR & SPI_SR_BSY) == SPI_SR_BSY );
}
```

Example 22-28. Receive data from an SPI slave by using polling

In Example 22-27 and Example 22-28, the SPI master uses a polling approach to send and receive data from an SPI slave. A more efficient approach is to use SPI interrupt or SPI DMA. Section 22.1.5 and 22.1.6 shows how to use interrupt and DMA for UART communication. Similarly, SPI can also use interrupt and DMA.

If enabled, a wide range of SPI events can generate interrupt requests. These events include:

1. transmit TXFIFO ready to accept new data,
2. data received in receive RXFIFO,
3. master mode fault when a bus conflict has been detected in multi-bus communication,
4. overrun error when RXFIFO is full and cannot accept new data,
5. TI frame format error when NSS signal does not follow the data format, and
6. CRC protocol error when the received CRC value does not match the CRC value calculated based on the received data.

SPI communication handled via DMA is the most efficient. Software can enable DMA by setting the RXDMAEN and TXDMAEN bit in the CR2 register. If enabled, hardware automatically generates a DMA request each time when the TXE or RXNE enable bit in the CR2 register is set. SPI also supports a special DMA mode in which hardware generates DMA requests when the receive or transmit FIFO reaches a pre-defined threshold.