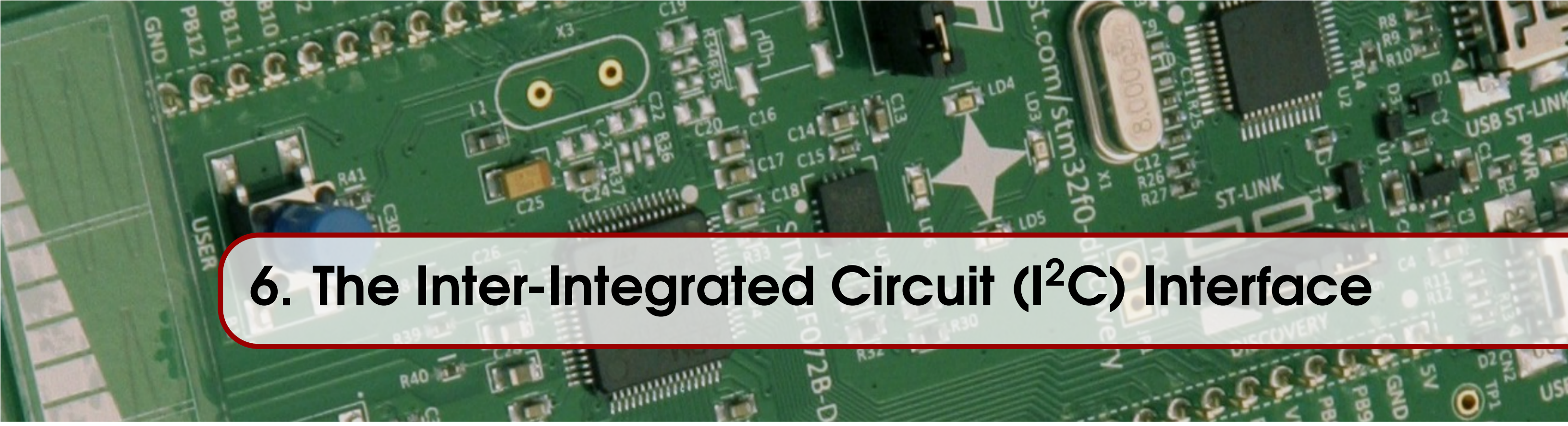

6. The Inter-Integrated Circuit (I^{2}C) Interface

6.1 Introduction to I^{2}C

I^{2}C (Inter-Integrated Circuit), is a synchronous serial communications bus developed by Philips Semiconductor in 1982. It is usually used to connect lower-speed devices such as sensors to a microprocessor. I^{2}C's design enables many devices to share a single data connection and includes addressing so that each device can be individually selected without the requirement for external enable signals. There are multiple speed standards for the I^{2}C interface. The most common of these are the original 100 kHz or the 400 kHz fast-mode.

6.1.1 Design and Topology

I^{2}C has a bus topology where every device directly connects to two bidirectional signal lines. Because all devices share a single connection, only one device can transmit at a time, limiting the bus to half-duplex communications.

Device Modes

Devices on an I^{2}C interface operate in either *master* or *slave* mode.

- Master devices initiate communication with slave devices.
- Each slave device has an unique hardware I^{2}C address.
- A master selects a specific slave by sending its address on the bus.
- Slave devices can respond to a master device when requested, but can't start a new transaction on their own.
- Some devices, such as most processors, can switch between master and slave mode.

Unlike many interfaces, I^{2}C allows multiple master devices to share the same bus. There is an arbitration system in the addressing protocol that resolves conflicts when multiple masters attempt to use the bus at the same time. Figure 6.1 shows an example of a master and slave devices connected via an I^{2}C bus.

Signal Connections

I^{2}C uses two signal lines, these are *SDA* (Serial Data) and *SCL* (Serial Clock). These are shown in the example interface demonstrated in figure 6.1.
When communicating, a master device produces clock transitions on the SCL line. The slave device uses this clock signal for both receiving and transmitting data. A slave can also hold the clock line low to pause the master if it needs more time for processing. This process is called clock-stretching

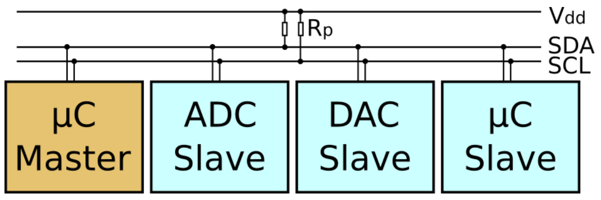

Figure 6.1: Topology and connections of an I^{2}C bus.

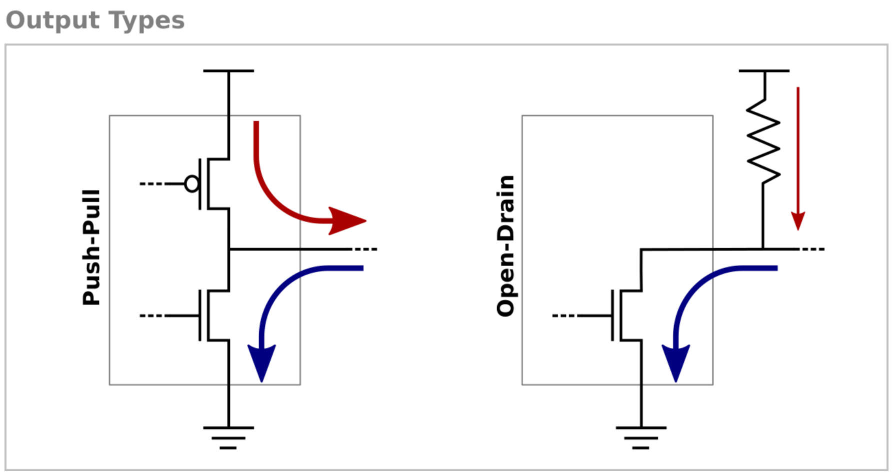

Figure 6.2: Push-Pull & Open-Drain Output Circuitry

Depending on the direction of communication, both the master and slave produce data on the shared SDA line. When receiving data, both slave and master devices acknowledge each communication frame to notify the other that the data was received. Both the clock and data lines use an open-drain I/O structure, this is discussed in section 6.1.2.

6.1.2 Electrical Characteristics

Push-Pull vs Open Drain Outputs

Figure 6.2 shows a simplified representation of output circuitry for *push-pull* and *open-drain* outputs.
Push-Pull Outputs have drive transistors that allow the device to push the output line "high" by connecting to the supply rail of the device, as well as pulling it "low" by connecting to ground. A push-pull output can source or sink current depending on the voltage of the connected circuitry.
Open-Drain Outputs have a single transistor and can only pull the output to a low state. Because of this, open-drain systems require an external influence such as a pull-up resistor to return the line to a high state when no device is pulling it low.

Why Open-Drain for I^{2}C

I^{2}C uses open-drain outputs due to the bi-directional nature of its signal lines. Consider a push-pull connection where two devices are attempting to output different states onto a single wire. One device attempts to drive the line high by connecting to the supply rail, while the other connects to ground to drive the line low. In this event, the two devices generate a high current power-ground short circuit, likely damaging the transistors within both devices.

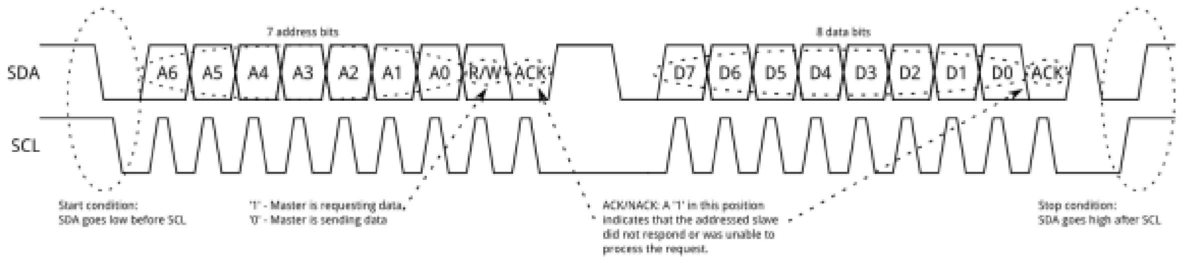
7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition:
SDA goes low before SCL
'1' - Master is requesting data,
'0' - Master is sending data
ACK/NACK: A '1' in this position indicates that the addressed slave did not respond or was unable to process the request.
Stop condition:
SDA goes high after SCL

Figure 6.3: Example I^{2}C transaction

Open-drain systems inherently cannot cause damaging faults because all devices can only pull the signal line low. The pull-up resistor limits the total current flowing through the system, and the most severe error that can occur is corrupted data.

6.2 Structure of an I2C Transaction

I^{2}C uses a strict protocol that defines data transfer as a series of *frames* and conditions. Transmitted data is broken up into two types of frame: an address frame, where the master indicates the slave to which the data is sent and one or more data frames. Data is placed on the SDA line while the clock (SCL) is pulled low, and is sampled during the clock's rising edge. The time between the clock transition and data read/write is determined by the devices on the bus and the configuration of the I2C peripheral. An example I^{2}C transaction is shown in figure 6.3.

6.2.1 Address Frame

Start Condition

A master device initiates an address frame by pulling SDA low while leaving SCL high. This state, known as the *start condition*, notifies all other devices that a transmission is about to begin. If two master devices concurrently attempt to take control of the bus, whichever device pulls SDA low first or transmits a lower slave address wins the arbitration.

Addressing and the Read/Write Bit

Typical I^{2}C addresses are 7-bits. However, some devices support an extended mode allowing for 10-bit addressing. Addresses are transmitted most significant bit (MSB) first, followed by a read/write bit indicating whether the master intends to read (1) or write (0) data to the slave device.

Slave Acknowledgment

The final bit in all frames is the *acknowledge* (ACK) bit. After completing the address or data bits of a frame, the transmitting device allows SDA to return high and waits for the receiver to respond by pulling it low. If this does not occur, indicating a *not-acknowledge* or NACK condition, the transmitter can assume that the data was not received.

6.2.2 Data Frame

Data Byte

An I^{2}C transaction can contain an arbitrary number of data frames, where each frame contains a single data byte. The master device generates clock transitions on the SCL line while either the master or slave device places data on the SDA line according to the direction indicated by the read/write bit.

Stop Condition

After completing the entire transaction, the master generates a *stop condition* and signals the release of the bus. Stop conditions are indicated by a low to high transition of SDA, while the clock (SCL) remains high. During normal data transfer, SDA only changes state while the SCL signal is low.

Restart Condition

Because I^2C is a half-duplex bus, master and slave devices can not transmit simultaneously. If a master wishes to both write and read from a slave, it must begin a new transaction with the read/write bit set correctly.
In the case where chained transactions are desired, ending the current transaction with a stop condition would release the bus and allow other devices to steal control before the master begins again. To prevent this, devices are allowed to issue new start conditions without properly ending the previous transaction with a stop. This event is known as a *restart* condition.

6.3 Using the I2C Peripheral

The I2C peripheral in the STM32F0 has a high-level interface that hides much of the complexity of the bus protocol. The peripheral can automatically generate a complete I2C transaction once the user configures basic parameters such as the slave address and number of bytes to transfer.
While this interface is simpler than the original protocol, the user must be careful when initializing the I2C peripheral. The I2C peripherals in the STM32F0 support a variety of modes and speeds. Due to the strictness of the I2C standard, incorrectly configured settings may prevent slave devices from responding.

6.3.1 Peripheral Registers

Control Register 1 (I2C_CR1)

The control register 1 manages the overall operation of the I2C peripheral. These settings fall into a few different categories:

- **SMBus Configuration** – SMBus is a more restrictive protocol designed for greater reliability than conventional I2C. Configuration bits and registers related to SMBus should be left at default.
- **Slave Mode Configuration** – The I2C peripheral operates as both a slave and master device. Since we won't be using a multi-master network for this lab, these settings can be ignored.
- **Noise Filters** – The I2C peripheral has both analog and digital noise filters. The analog filter is enabled by default and is sufficient for normal operation.
- **Interrupt Enables** – There are many I2C conditions that can generate interrupts. This register controls interrupts for events such as transmission errors, completed transmissions, and when the bus is free.
- **Peripheral Enable** – The I2C peripheral must be enabled in this register after it is initialized. Many initialization settings are protected once this bit is enabled.

Control Register 2 (I2C_CR2)

The control register 2 contains settings for the current I2C transaction. As such, this register is used whenever communicating with a slave device. Understanding this register is critical to using the peripheral; bits that will be used heavily during the lab exercises are described here.

- **AUTOEND** – When set, the peripheral will automatically generate a stop condition at the end of a transaction. This setting is undesirable when performing chained writes and reads as will be required in the lab assignment.
- **NBYTES[7:0]** – This group of 8-bits set the number of bytes to transmit in the next transaction. These must be modified using bitwise operations to prevent overwriting the rest of the register.
- **STOP & START** – These bits generate start and stop conditions on the bus. The user starts a new transaction by writing to the START bit.
- **RD_WRN** – This bit sets the direction of data transfer for the next transaction. Its state controls the read/write bit sent in the address frame.
- **SADD[9:0]** – This group of 10-bits sets the slave address used in the next transaction. Unfortunately, the default 7-bit addressing mode uses bits [7:1] within the center of the bit field, often leading to an easy user error when learning the peripheral. These must be modified using bitwise operations to prevent overwriting the rest of the register.

Timing Register (I2C_TIMINGR)

The I2C peripheral has a very flexible timing system which allows the user to specify slew-rates and sampling delays. These settings can adjust the I2C peripheral to operate reliably under non-ideal conditions. The configurable timings for the peripheral are:

- **PRESC** – This field sets the prescaler used by internal timers within the I2C peripheral. These timers generate the clock signal and control when data is sampled.
- **SCLL & SCLH** – These fields set the high and low periods of the I2C clock signal (SCL). Because these can be set independently, the I2C peripheral allows the user to specify an asymmetric clock signal.
- **SDADEL & SCLDEL** – These fields determine the data setup and hold timing used when transmitting and receiving.

Interrupt and Status Register (I2C_ISR)

The read-only interrupt and status register indicates the state of every interrupt condition in the peripheral. These flag bits are also used by blocking drivers to determine the state of the bus.
The peripheral hardware clears many of these flags automatically whenever the appropriate action is completed by the user. The documentation should always be examined to determine the specific conditions required for each bit.

Interrupt Clear Register (I2C_ICR)

The interrupt clear register is used to clear status flags that require direct acknowledgment from the user. Many of these involve error condition interrupts, and will not be used in the lab assignment.

Transmit & Receive Data Registers (I2C_TXDR\I2C_RXDR)

These registers are written and read by the user application during use of the peripheral.

Other Peripheral Registers

The I2C peripheral has additional registers used by the slave and SMBus modes. These should be left in their default state.

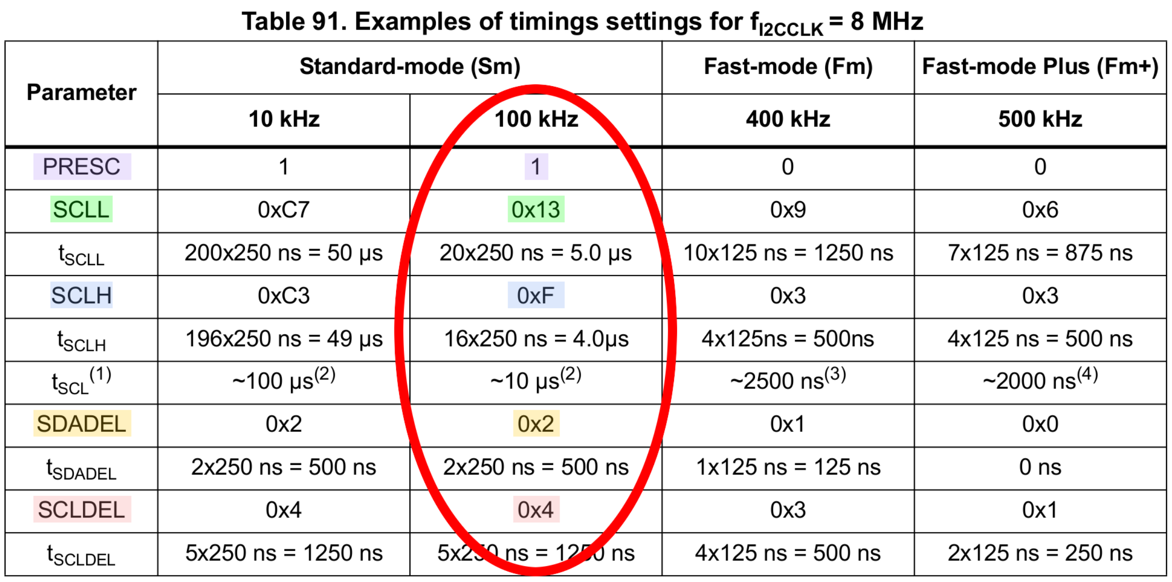

Table 91. Examples of timings settings for f_{I2CCLK} = 8 MHz

Parameter	Standard-mode (Sm)		Fast-mode (Fm)	Fast-mode Plus (Fm+)
	10 kHz	100 kHz	400 kHz	500 kHz
PRESC	1	1	0	0
SCLL	0xC7	0x13	0x9	0x6
t_{SCLL}	200x250 ns = 50 µs	20x250 ns = 5.0 µs	10x125 ns = 1250 ns	7x125 ns = 875 ns
SCLH	0xC3	0xF	0x3	0x3
t_{SCLH}	196x250 ns = 49 µs	16x250 ns = 4.0µs	4x125ns = 500ns	4x125 ns = 500 ns
t_{SCL}(1)	~100 µs(2)	~10 µs(2)	~2500 ns(3)	~2000 ns(4)
SDADEL	0x2	0x2	0x1	0x0
t_{SDADEL}	2x250 ns = 500 ns	2x250 ns = 500 ns	1x125 ns = 125 ns	0 ns
SCLDEL	0x4	0x4	0x3	0x1
t_{SCLDEL}	5x250 ns = 1250 ns	5x250 ns = 1250 ns	4x125 ns = 500 ns	2x125 ns = 250 ns

Figure 6.4: Timing table for the default 8MHz processor speed

6.3.2 Initializing the Peripheral

The I2C peripherals in the STM32F0 offer a variety of operating modes, and have a large set of status flags used for developing interrupt based non-blocking drivers. Depending on the modes and interrupts used, the initialization of the I2C peripheral can be complex or relatively simple. The complete initialization process is documented in section 26.4.5 of the peripheral reference manual.
This lab requires only the I2C master mode using blocking operations; these involve minimal initialization and can be configured by the following steps.

- Enable and configure the GPIO output pins used by the I2C peripheral to alternate function mode.
  - Set the pin's output type to be *open-drain* in the GPIOx_OTYPER register.
  - Set the appropriate alternate function number in the GPIOx_AFR registers.
- Enable the I2C system clock using the RCC peripheral.
- Configure bus timing using the I2Cx_TIMINGR.
- Enable the I2C peripheral using the PE bit in the CR1 register.

Configuring the Bus Timing

The only system-wide initialization step used by the basic modes in this lab is to set the I2C timing register. The values within this register determine the transmission rate of the interface.
The flexibility of the STM32F0's I2C timing system allows for fine-tuned operation. However, this flexibility requires the user to derive timing constants from series of equations documented in the peripheral manual.
Fortunately, section 26.4.10 contains tables of pre-calculated parameters which result in standard behavior suitable for most systems. Figure 6.4 shows the timing table for the default 8MHz processor speed. The columns within this table represent the supported I2C speed modes. Configuring the bit fields in the TIMINGR to these values provides acceptable performance.

Enabling the Peripheral

Before the I2C interface can be used the peripheral enable (PE) bit must be set in the CR1 register. Setting this bit locks all of the system-wide configuration bits and registers to prevent accidental modification during transmission. Clearing the PE bit after it has been set performs a peripheral reset and clears all configuration registers.
Transaction specific options found in the CR2 register are not locked by the PE bit. These are modified at the beginning of each I2C transaction to set parameters such as the slave address.

6.3.3 Basic Communication

The processes of writing and reading from a slave device are very similar. Every communication follows a few simple steps: setting transaction parameters, starting the peripheral, waiting on status flags, and looping to previous steps depending on the flags set and length of data. Figure 6.5 contains a flowchart demonstrating the general process of using the master mode with blocking operations.

Setting up the Transaction

Each I2C transaction is initialized by completing the following steps in the CR2 register:

1. Set the slave address in the SADD[7:1] bit field.
2. Set the number of data byte to be transmitted in the NBYTES[7:0] bit field.
3. Configure the RD_WRN to indicate a read/write operation.
4. Do not set the AUTOEND bit, this lab requires software start/stop operation.
5. Setting the START bit to begin the address frame.

! Set the START bit in the CR2 register after configuring the slave address and transaction length. Similar to how the PE bit locks system-wide configurations, setting the START bit locks the transaction parameters until the peripheral has completed the address frame.

■ **Example 6.1 — Setting the SADD and NBYTES Bit Fields.** Since the SADD and NBYTES bit fields are not within separate registers, they cannot be directly assigned without overwriting the remainder of the CR2 register. Because of this, bitwise operations are required to clear and set the bit fields.
Rather than directly assigning each bit, it is helpful to use the desired value as a shifted bitmask directly. The following example demonstrates how to clear and set these bit fields using bitwise operations.

```c
/* Clear the NBYTES and SADD bit fields
 * The NBYTES field begins at bit 16, the SADD at bit 0
 */
I2C2->CR2 &= ~((0x7F << 16) | (0x3FF << 0));

/* Set NBYTES = 42 and SADD = 0x14
 * Can use hex or decimal values directly as bitmasks.
 * Remember that for 7-bit addresses, the lowest SADD bit
 * is not used and the mask must be shifted by one.
 */
I2C2->CR2 &= ~((42 << 16) | (0x14 << 1));
```

■

Transmitting to a Slave Device

Once a transaction has been initiated, the driver must wait until the I2C peripheral has transmitted the address frame and received an acknowledgment from the slave device. This is accomplished by polling on both the *Transmit Interrupt Status* (TXIS) and *Not Acknowledge Received Flag* (NACKF) status bits.
Depending on which flag bit is set determines the following operation:

- **NACKF Flag Set** – This flag indicates that the slave device did not acknowledge the address frame. There is likely a configuration issue; the current transaction has been aborted. Clear the NACKF flag, revise the initialization, and attempt to start a new transaction.
- **TXIS Flag Set** – The address frame completed successfully, and the peripheral is requesting new data to be written into the transmit data (TXDR) register.

Once successfully transmitting data, repeat the polling and writing process with the NACKF and TXIS bits until the number of bytes in the NBYTES bit field has been written. After reaching this limit, poll instead on the *Transfer Complete* (TC) status flag.
The TC flag is set when the peripheral determines that the transaction is complete, and is waiting for the user to perform a restart or stop condition. To restart, return to the top of the process for transmitting or receiving data from the slave. To release the bus by issuing a stop condition, set the STOP bit in the CR2 register.

Receiving from a Slave Device

Receiving from a slave device is nearly identical to transmitting. Often, receive transactions occur as a restart of a transmit. This is because many slave devices operate using a register-based control scheme similar to using peripheral registers. The difference is that to access the registers within a slave device, the user first transmits the register address to select for the follow-up read operation.

Once the transaction is initiated, the driver should poll on the *Receive Data Register Not Empty* (RXNE) and *Not Acknowledge Received Flag* (NACKF) status bits. Assuming that no acknowledgment errors occurred, the RXNE status flag indicates that data has been received and is waiting to be read from the receive data (RXDR) register.
Multi-byte reads are accomplished by repeatedly polling on the status flags. Once the number of bytes has been read as was set in the NBYTES register, the user can issue a restart or stop condition.

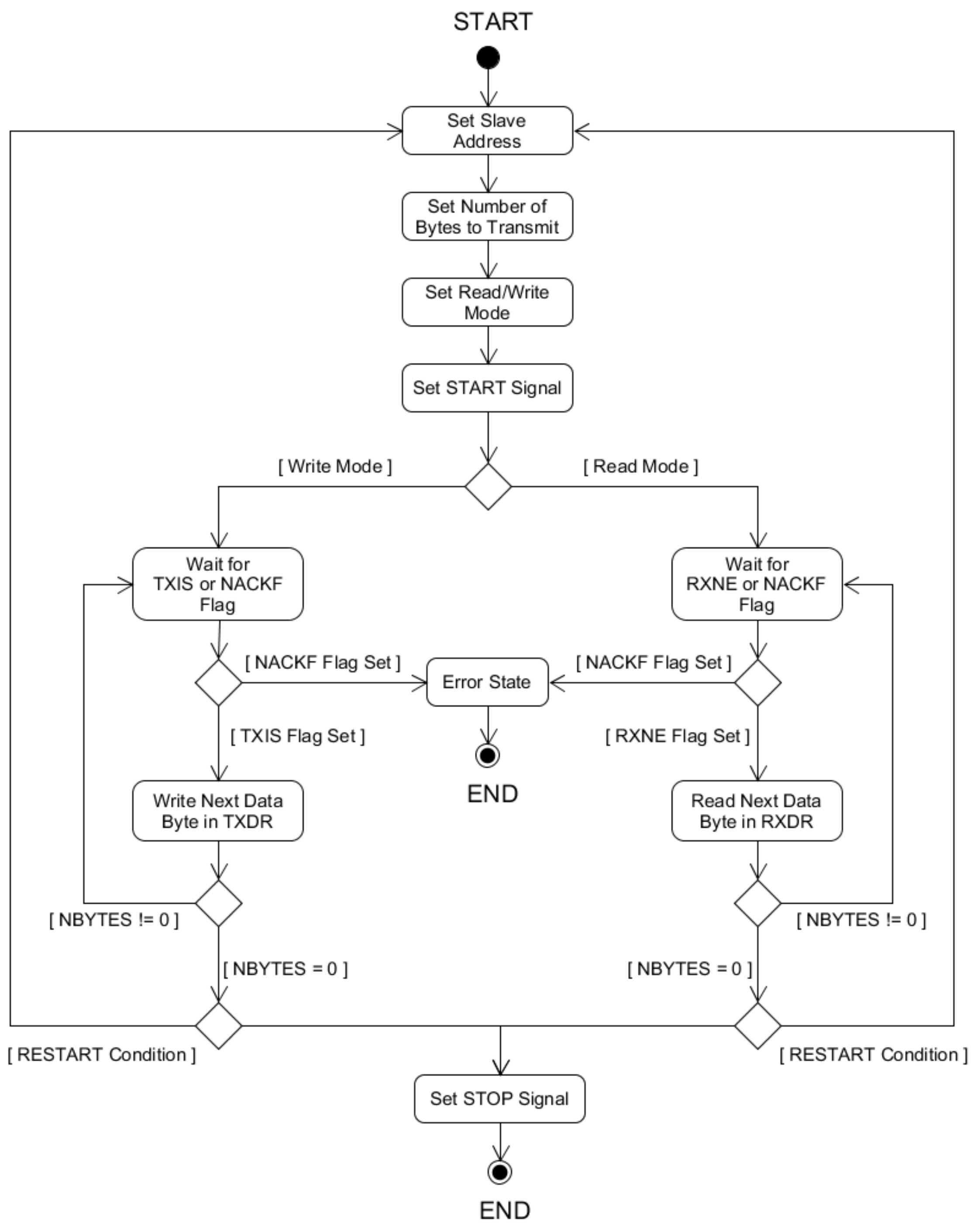

Figure 6.5: Blocking transmit and receive flowchart