Inter-Integrated Circuit (I2C)

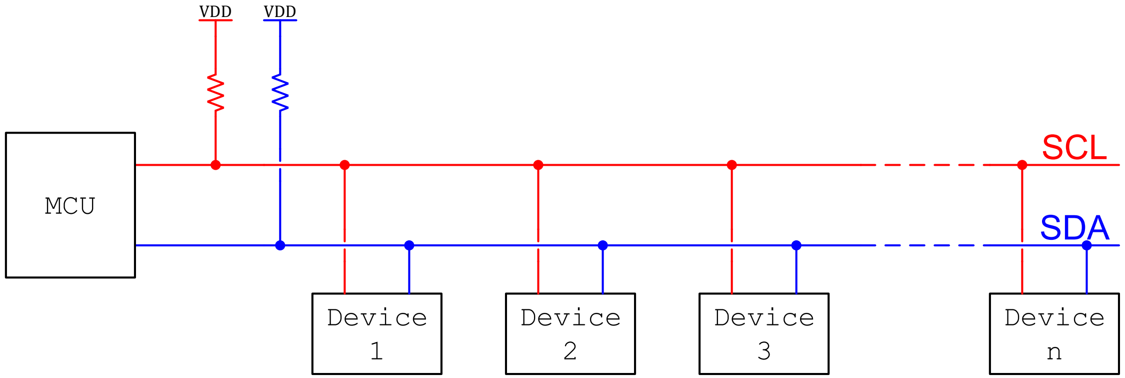
Embedded Systems and Communication Protocols for IoT (252461)

Dr. Witsarut Achariyaviriya

Department of Electrical Engineering, Faculty of Engineering

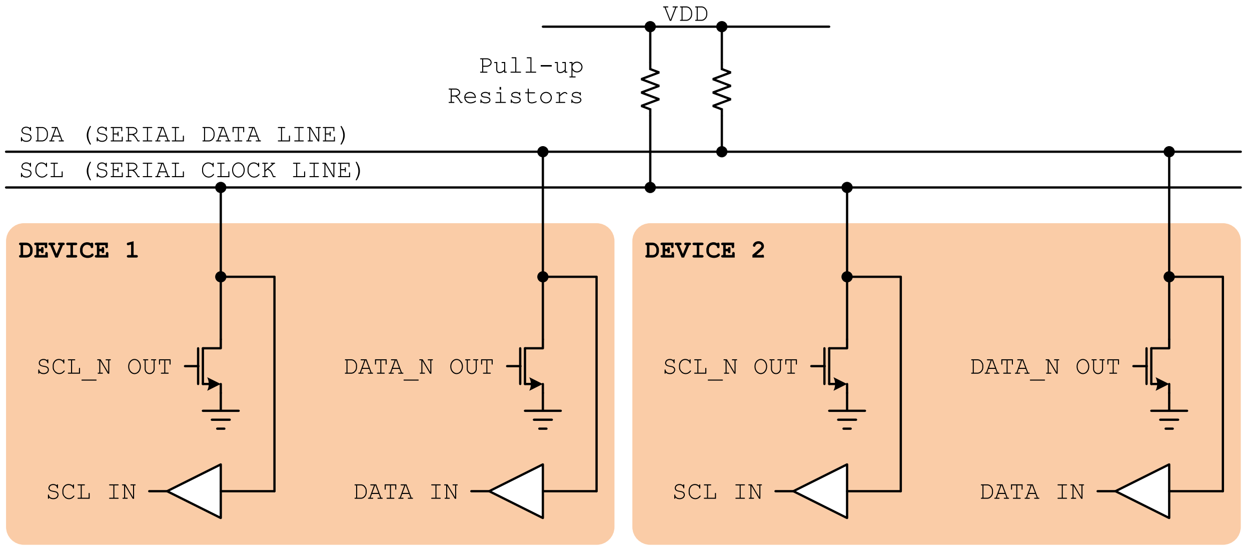
Chiangmai University

I2C



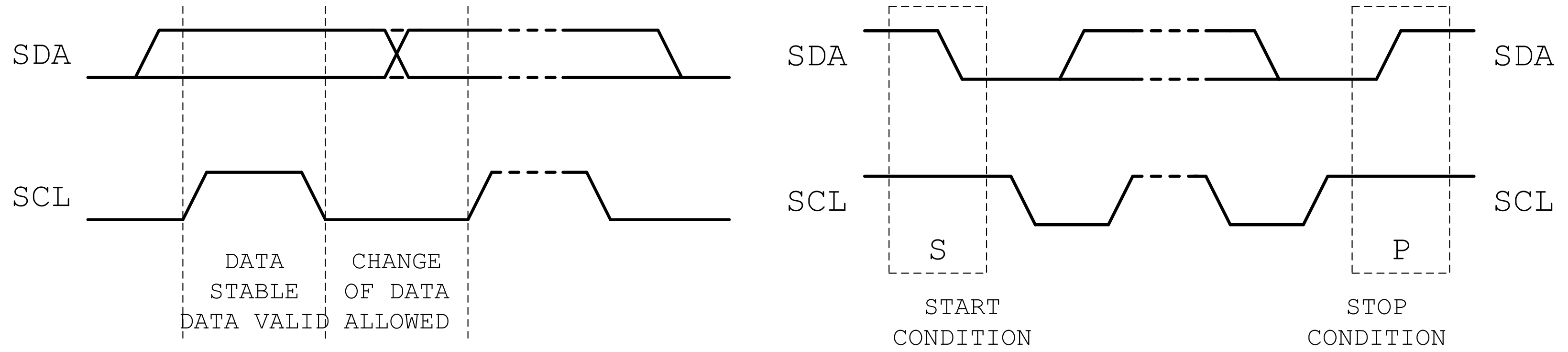
- I²C (Inter-Integrated Circuit) invented by Philips Semiconductor:
    - multi-master
    - multi-slave
    - packet switched
    - single-ended
- It is typically used for attaching lower-speed peripheral ICs to MCUs in short-distance (intra-board).
- I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. Typical voltages used are +5 V or +3.3 V.

I2C: Physical layer



- Pulling a line to ground represents logic '0' and releasing the line high impedance represents logic ‘1’ since the resistors pull the signal up to VDD.
- This wire-ANDing allows multiple nodes to connect to the bus without short circuits from signal contention.
- Multiple nodes may be driving the lines simultaneously.
  - If any node is driving the line low, the signal is low.
  - Nodes that are trying to transmit a logic ‘1’ (i.e. letting the line float high) can detect that another node is active at the same time.

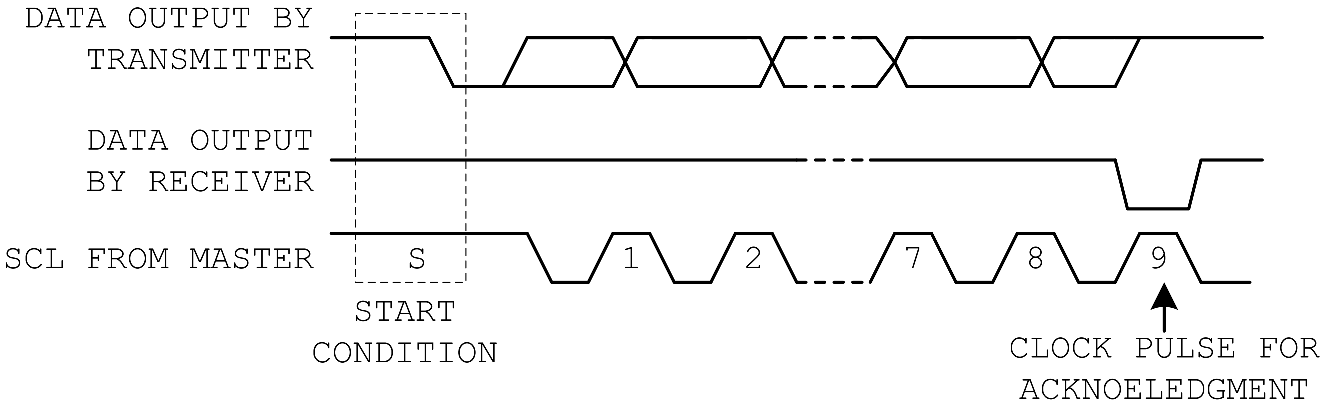
I2C: Data and Conditions



- The SDA line must be stable during the high period of the SCL. The high or low state of the SDA can only change when the SCL is low.
- Within the procedure of the I2C-bus, two situations which are defined as:
  - START: A falling edge of the SDA while SCL is high indicates a START condition.
  - STOP: A rising edge of the SDA while SCL is high defines a STOP condition.
- START and STOP conditions are always generated by the master.

[8]

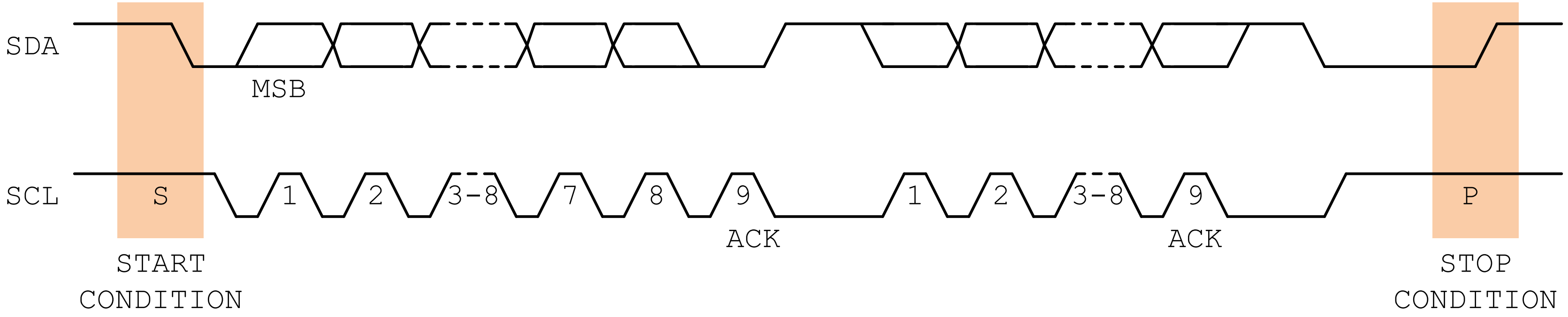
I2C: Acknowledge



- Data transfer with acknowledge is mandatory.
  - The acknowledge clock pulse is generated by the master.
  - The transmitter releases the SDA during the acknowledge clock pulse.
  - The receiver must pull down the SDA during the acknowledge clock pulse so that it remains stable low during the high period of this clock pulse.
  - Consequently, SDA low during the clock pulse is acknowledge (ACK). Otherwise, SDA staying high is not acknowledge (NAK).
- Usually, a receiver is required to generate an acknowledge after each byte has been received.

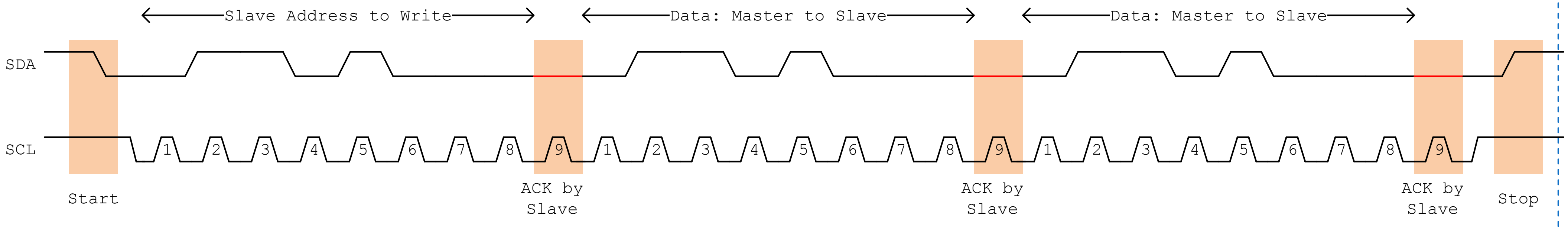
[8]

I2C: Byte Format



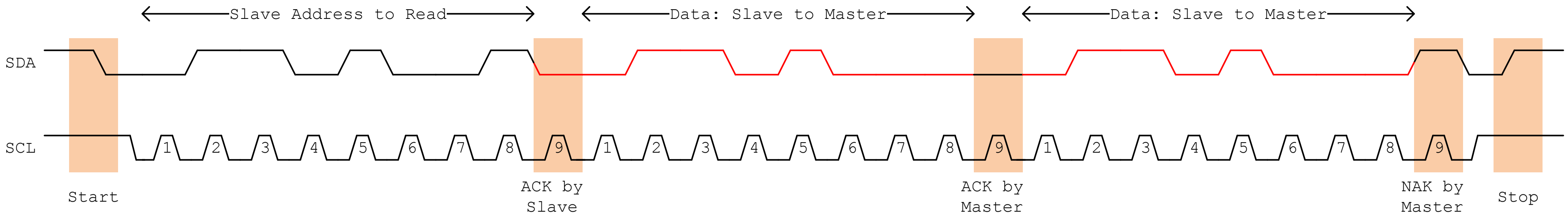
- Every byte put on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first.
- If a receiver can't receive another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the transmitter into a wait state.

I2C: Write Operation



- Devices attached to I2C bus have a unique 7-bit address on the bus.
- The master initializes a transaction by releasing start-condition followed a 8-bit word. The word is 7-bit address of a device appended with an operation bit. For an example, if the address of the device is 0x34, the word transmitted by the master is:
  - 0x68: for write operation; the master want to write data to the device.
  - 0x69: for read operation; the master wants to read data from the device.
- Write operation: When the device address word is transmitted by the master, the addressed device will acknowledge. Then, the master continues to transmit data to the device and the device must acknowledge to every word that it receives. Finally, the master releases a stop-condition when it has no more data.

I2C: Read Operation



- For read operation, the MCU needs data from a device. It performs a sequence.
  - Release a start-condition
  - Transmit a word consisting of 7-bit device address appended with bit-1 indicating read operation
  - When the device acknowledges to the address word, the master continues driving clock in order to synchronize data transmitted by the device. And the MCU is needed to acknowledge to the data word.
  - The MCU acknowledges every word from the device except for the last byte. After the last byte arrives, the MCU does not acknowledge. Then it releases a stop-condition in order to terminate the read operation.

Overview of 2-Wire Interface(TWI) module

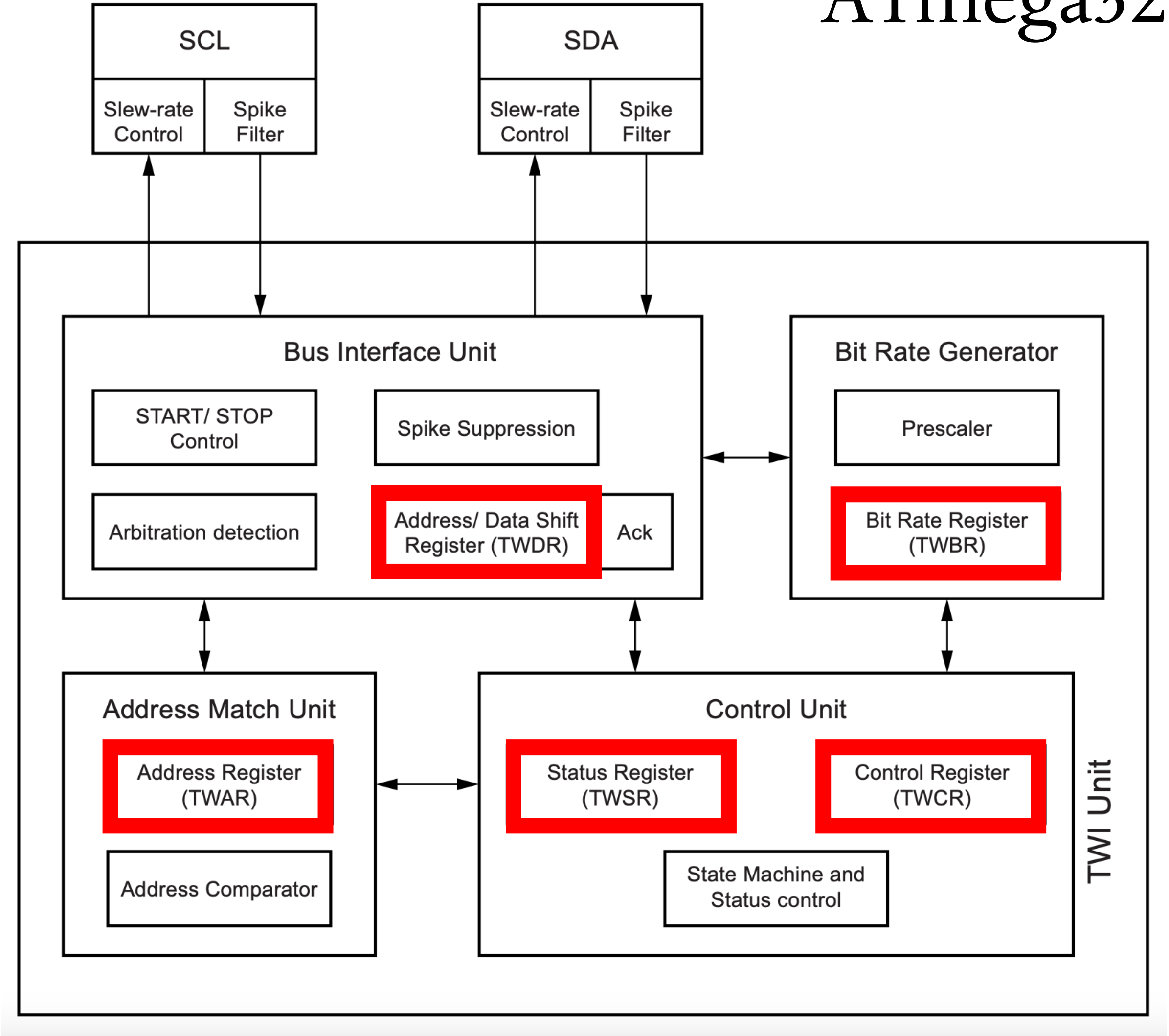
Bit Rate Generator Unit

- This unit controls the period of SCL when operating in a master mode.
- The SCL period is controlled by settings in the TWI bit rate register (TWBR) and the prescaler bits in the TWI status register (TWSR).

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \times (\text{PrescalerValue})}$$

- Slave operation does not depend on bit rate or prescaler settings, but the CPU clock frequency in the slave must be at least 16 times higher than the SCL frequency.

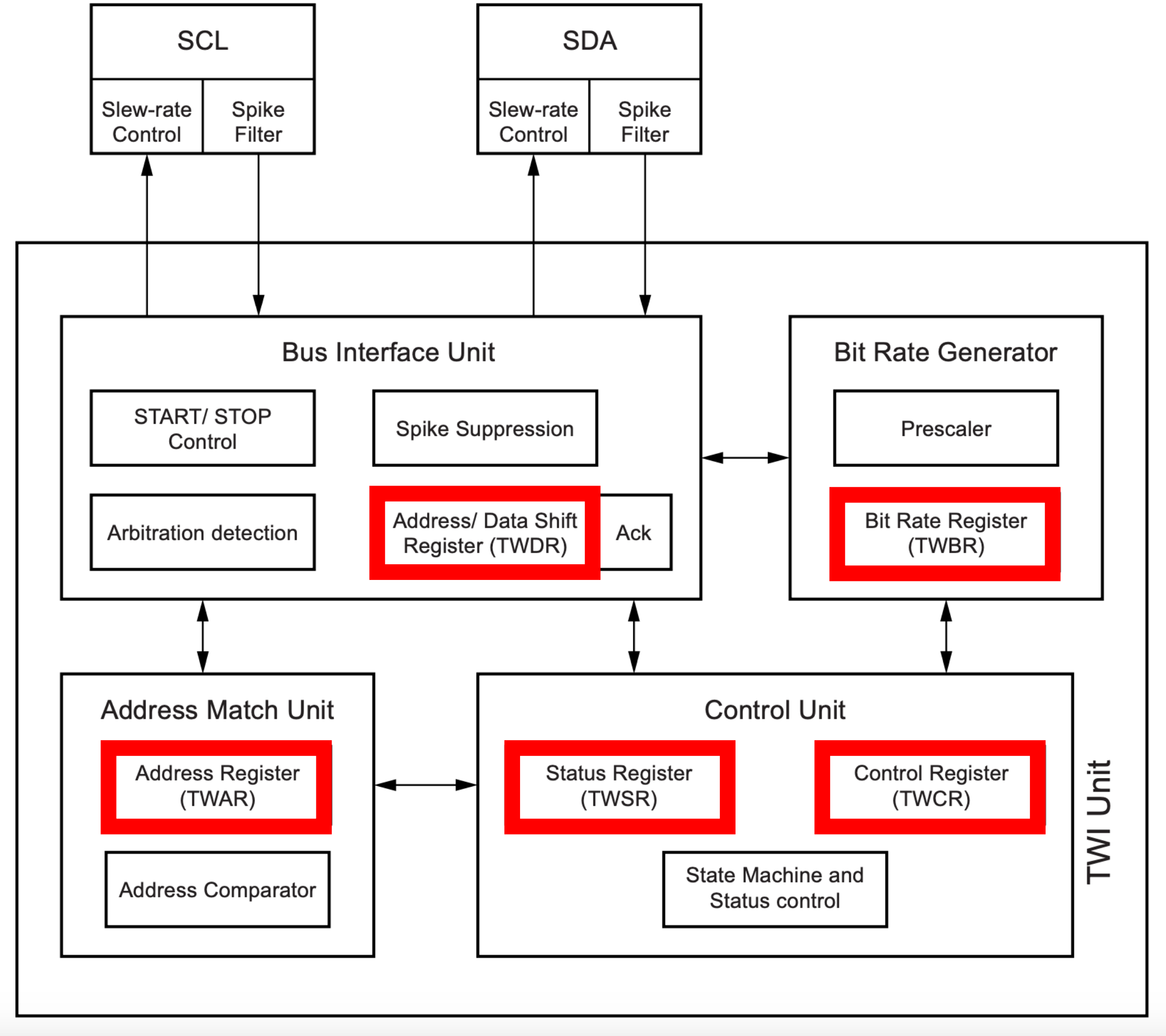
ATmega328P



Overview of 2-Wire Interface(TWI) module

Bus Interface Unit

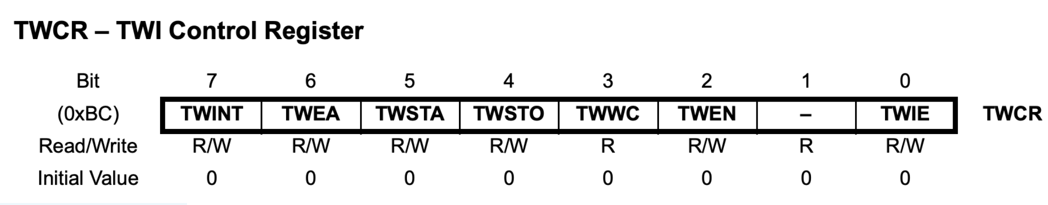
- The TWDR contains the address or data bytes to be transmitted, or the address or data bytes received.
- In addition to the 8-bit TWDR, the bus interface unit also contains a register containing the (N)ACK bit to be transmitted or received.
- (N)ACK register is not directly accessible by the application software, However, when receiving, it can be set or cleared by manipulating the TWI control register (TWCR). When in transmitter mode, the value of the received (N)ACK bit can be determined by the value in the TWSR.



Overview of 2-Wire Interface(TWI) module

Control Unit

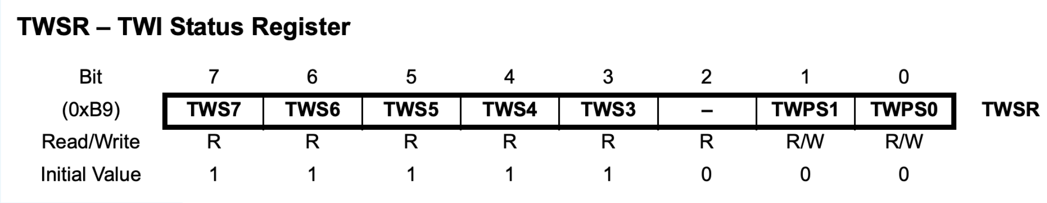
- The control unit monitors the TWI bus and generates responses corresponding to settings in the TWI control register (TWCR).



TWCR – TWI Control Register

Bit	7	6	5	4	3	2	1	0	
(0xBC)	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

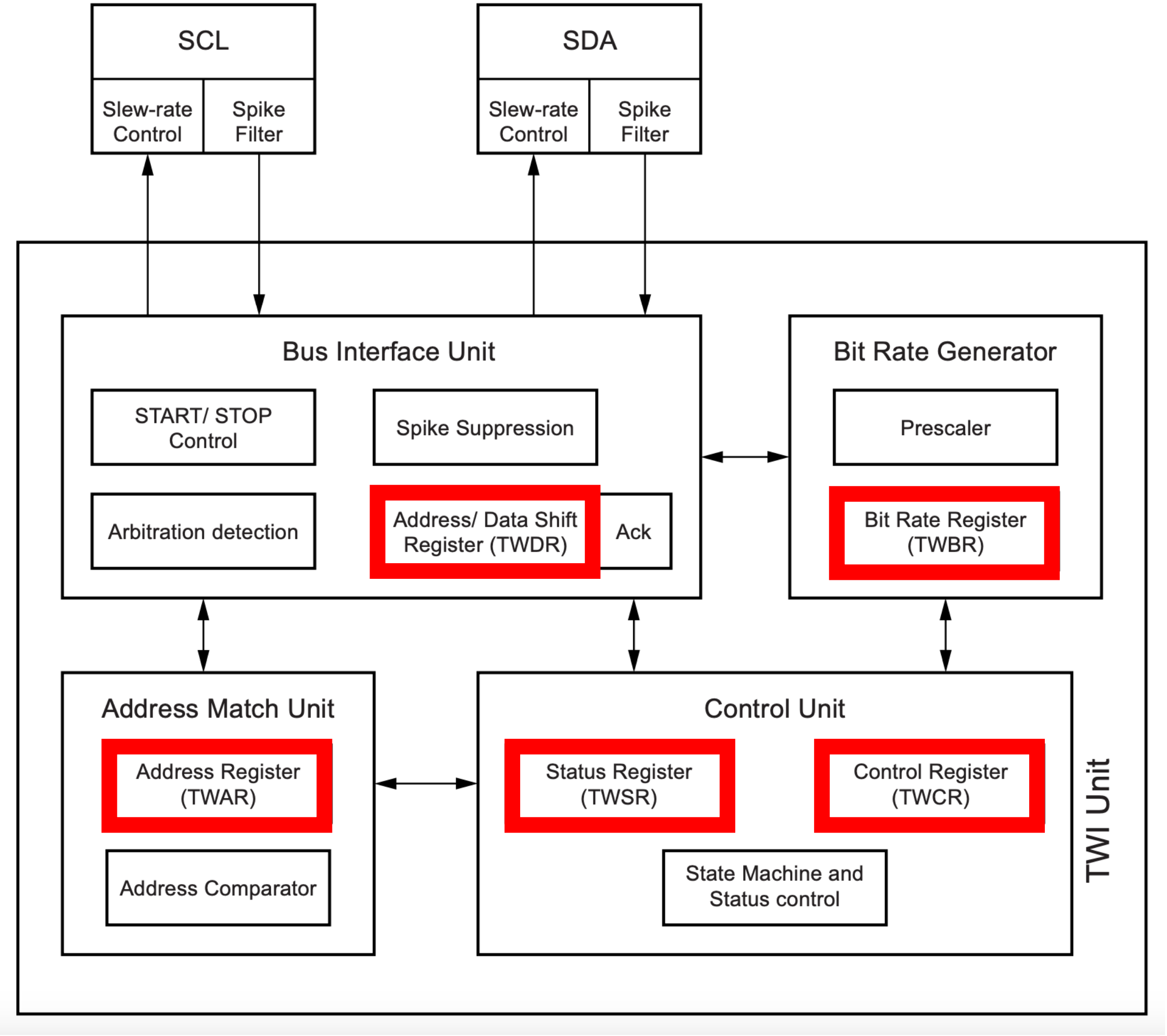
- When an event requiring the attention of the application occurs on the TWI bus, the TWI interrupt flag (TWINT) is asserted.
- In the next clock cycle, the TWI status register (TWSR) is updated with a status code identifying the event.



TWSR – TWI Status Register

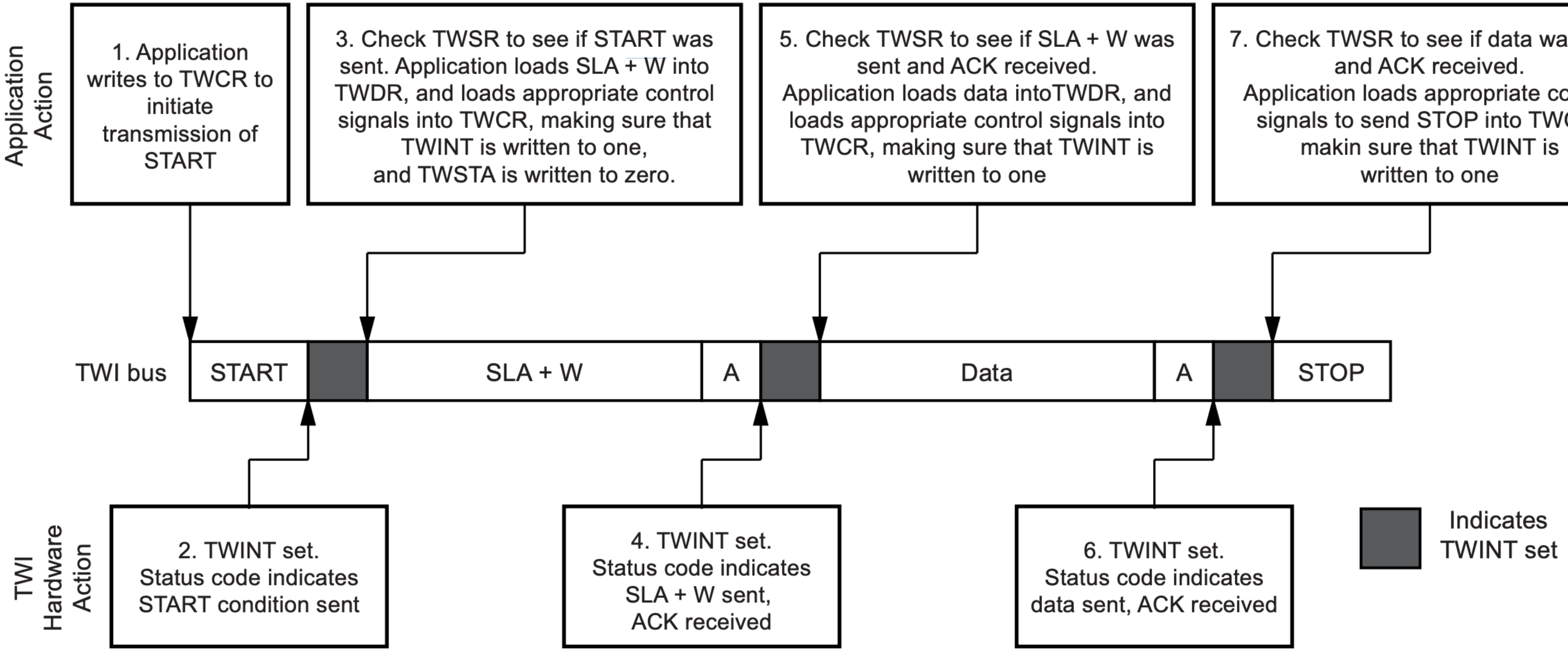
Bit	7	6	5	4	3	2	1	0	
(0xB9)	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	TWSR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	1	1	1	1	1	0	0	0	

- As long as the TWINT flag is set, the SCL line is held low.



Interfacing the Application to the TWI in a Typical Transmission

Figure 21-10. Interfacing the Application to the TWI in a Typical Transmission



1. Application writes to TWCR to initiate transmission of START

2. TWINT set. Status code indicates START condition sent

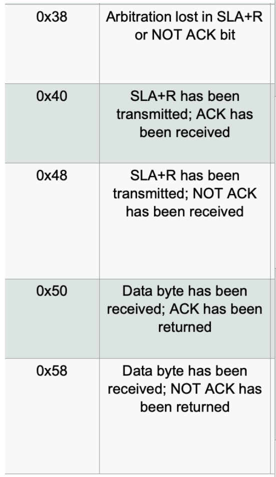
3. Check TWSR to see if START was sent. Application loads SLA + W into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one, and TWSTA is written to zero.

4. TWINT set. Status code indicates SLA + W sent, ACK received

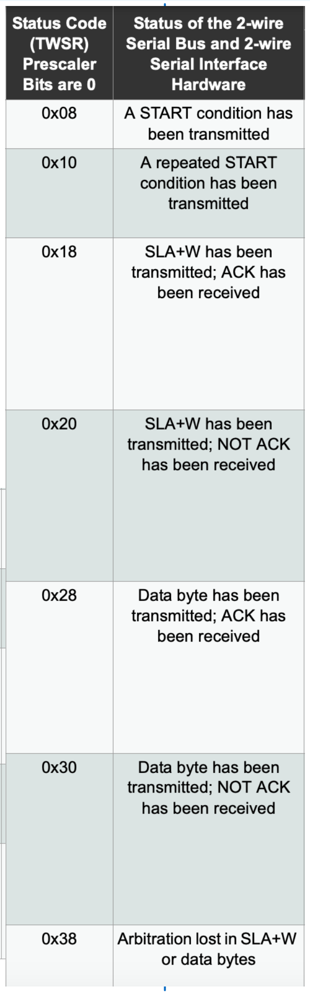
5. Check TWSR to see if SLA + W was sent and ACK received. Application loads data intoTWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one

6. TWINT set. Status code indicates data sent, ACK received

7. Check TWSR to see if data was sent and ACK received. Application loads appropriate control signals to send STOP into TWCR, makin sure that TWINT is written to one



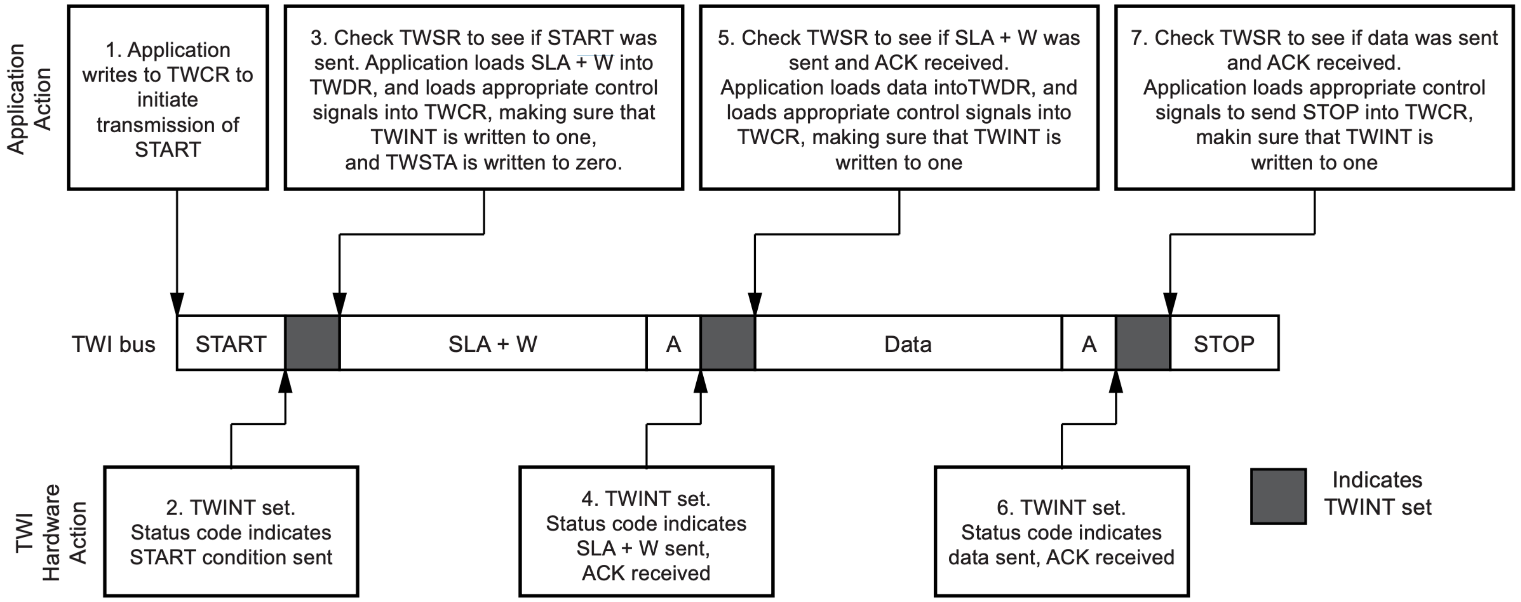
0x38	Arbitration lost in SLA+R or NOT ACK bit
0x40	SLA+R has been transmitted; ACK has been received
0x48	SLA+R has been transmitted; NOT ACK has been received
0x50	Data byte has been received; ACK has been returned
0x58	Data byte has been received; NOT ACK has been returned



Status Code (TWSR) Prescaler Bits are 0	Status of the 2-wire Serial Bus and 2-wire Serial Interface Hardware
0x08	A START condition has been transmitted
0x10	A repeated START condition has been transmitted
0x18	SLA+W has been transmitted; ACK has been received
0x20	SLA+W has been transmitted; NOT ACK has been received
0x28	Data byte has been transmitted; ACK has been received
0x30	Data byte has been transmitted; NOT ACK has been received
0x38	Arbitration lost in SLA+W or data bytes

Interfacing the Application to the TWI in a Typical Transmission

Figure 21-10. Interfacing the Application to the TWI in a Typical Transmission



Application Action

1. Application writes to TWCR to initiate transmission of START

3. Check TWSR to see if START was sent. Application loads SLA + W into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one, and TWSTA is written to zero.

5. Check TWSR to see if SLA + W was sent and ACK received. Application loads data intoTWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one

7. Check TWSR to see if data was sent and ACK received. Application loads appropriate control signals to send STOP into TWCR, makin sure that TWINT is written to one

TWI bus

START	■	SLA + W	A	■	Data	A	■	STOP

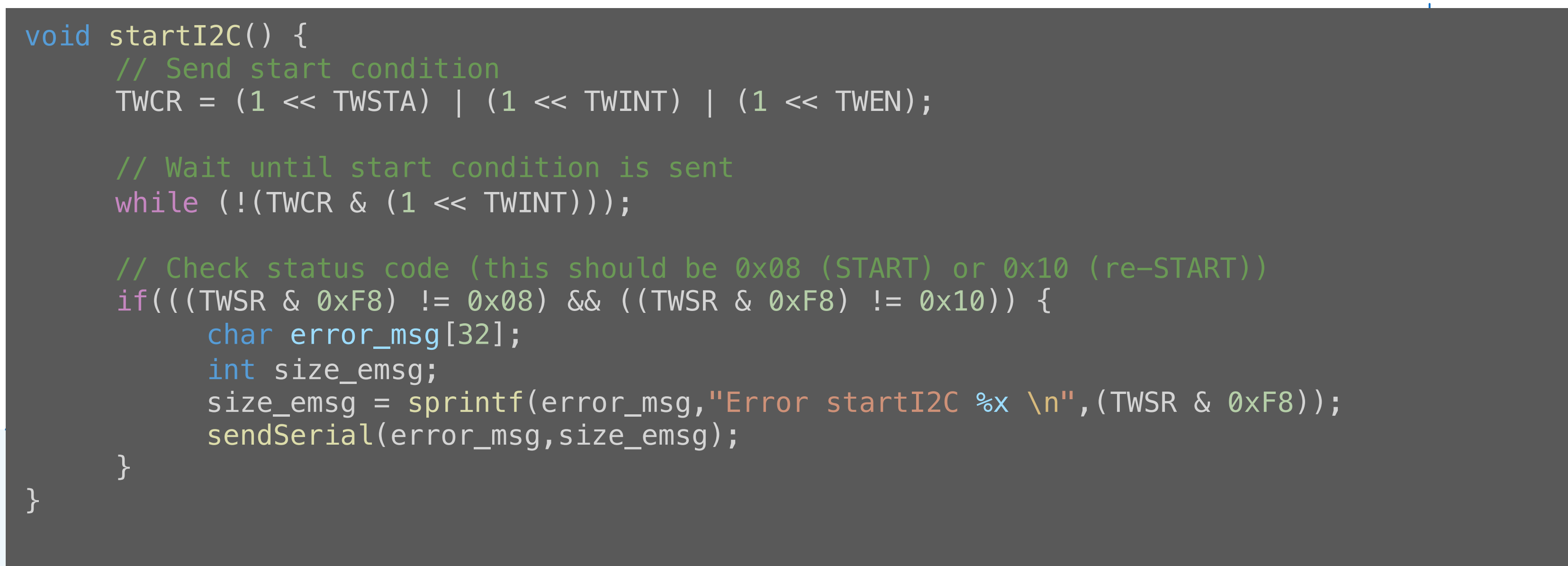
TWI Hardware Action

2. TWINT set. Status code indicates START condition sent

4. TWINT set. Status code indicates SLA + W sent, ACK received

6. TWINT set. Status code indicates data sent, ACK received

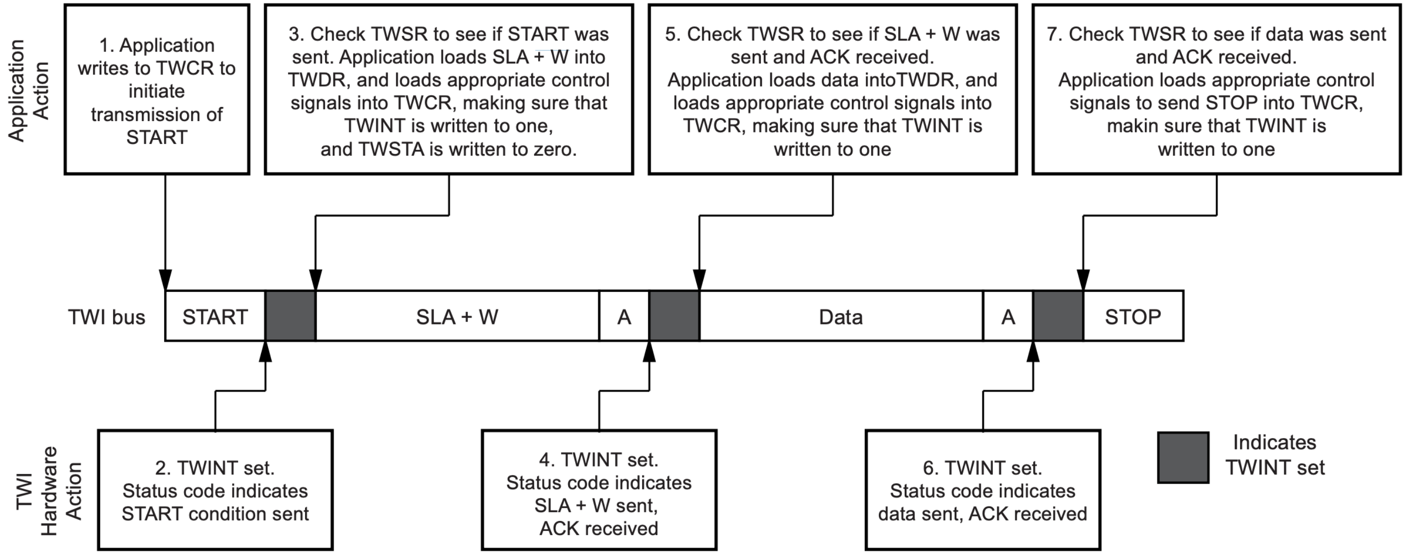
■ Indicates TWINT set



```
void startI2C() {
      // Send start condition
      TWCR = (1 << TWSTA) | (1 << TWINT) | (1 << TWEN);

      // Wait until start condition is sent
      while (!(TWCR & (1 << TWINT)));

      // Check status code (this should be 0x08 (START) or 0x10 (re-START))
      if(((TWSR & 0xF8) != 0x08) && ((TWSR & 0xF8) != 0x10)) {
            char error_msg[32];
            int size_emsg;
            size_emsg = sprintf(error_msg,"Error startI2C %x \n",(TWSR & 0xF8));
            sendSerial(error_msg,size_emsg);
      }
}
```

Interfacing the Application to the TWI in a Typical Transmission

Figure 21-10. Interfacing the Application to the TWI in a Typical Transmission



Application Action

1. Application writes to TWCR to initiate transmission of START

3. Check TWSR to see if START was sent. Application loads SLA + W into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one, and TWSTA is written to zero.

5. Check TWSR to see if SLA + W was sent and ACK received. Application loads data intoTWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one

7. Check TWSR to see if data was sent and ACK received. Application loads appropriate control signals to send STOP into TWCR, makin sure that TWINT is written to one

TWI bus: START | SLA + W | A | Data | A | STOP

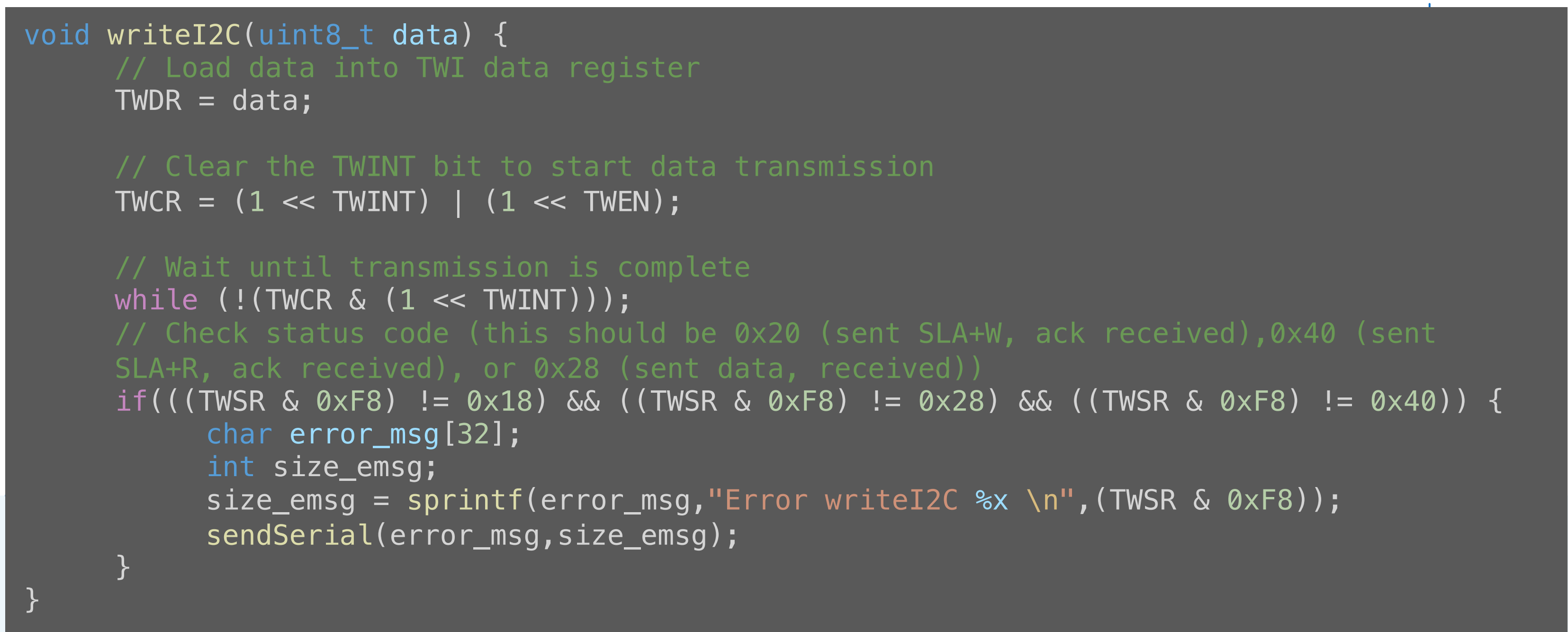
TWI Hardware Action

2. TWINT set. Status code indicates START condition sent

4. TWINT set. Status code indicates SLA + W sent, ACK received

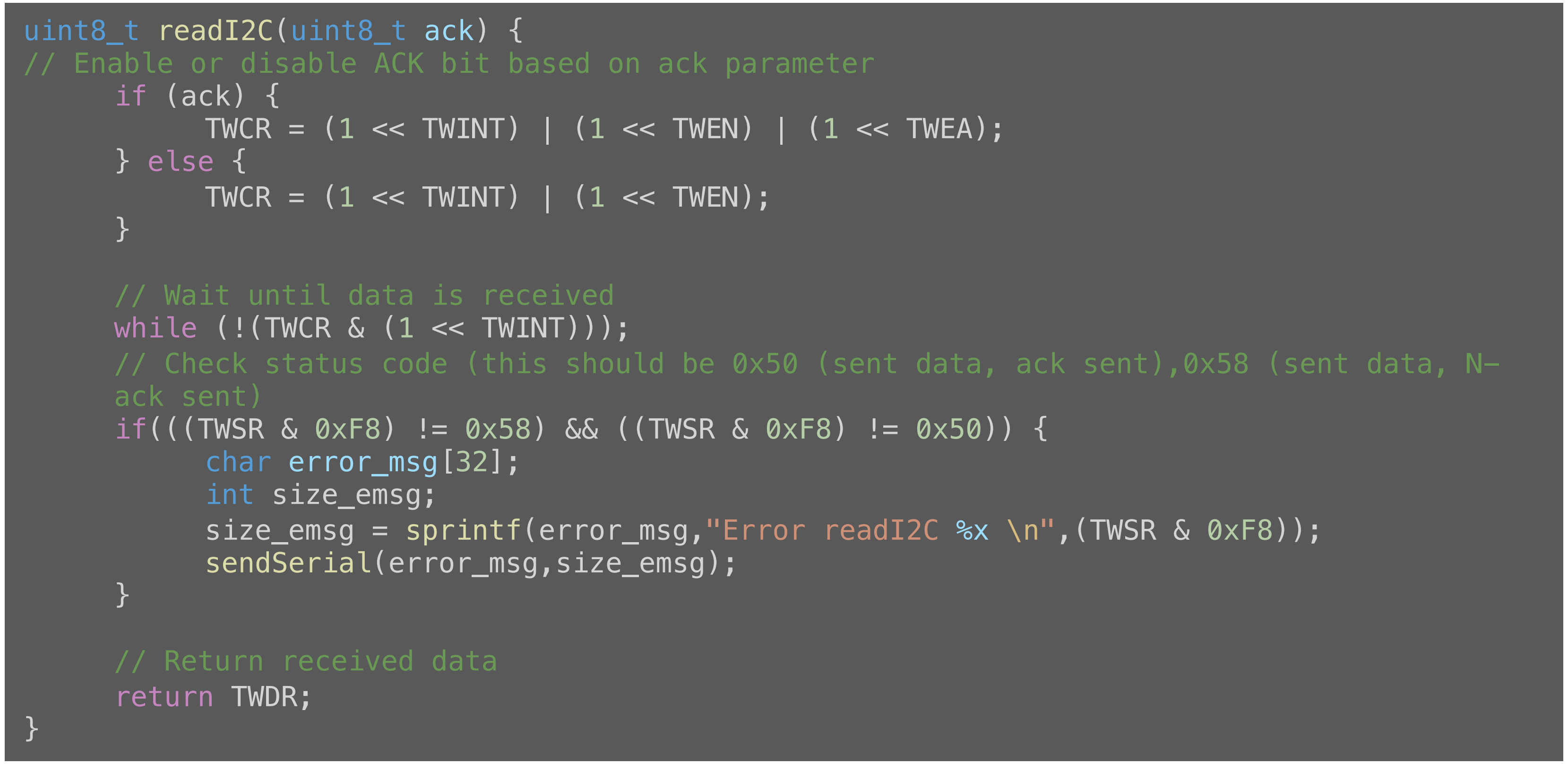
6. TWINT set. Status code indicates data sent, ACK received

▨ Indicates TWINT set



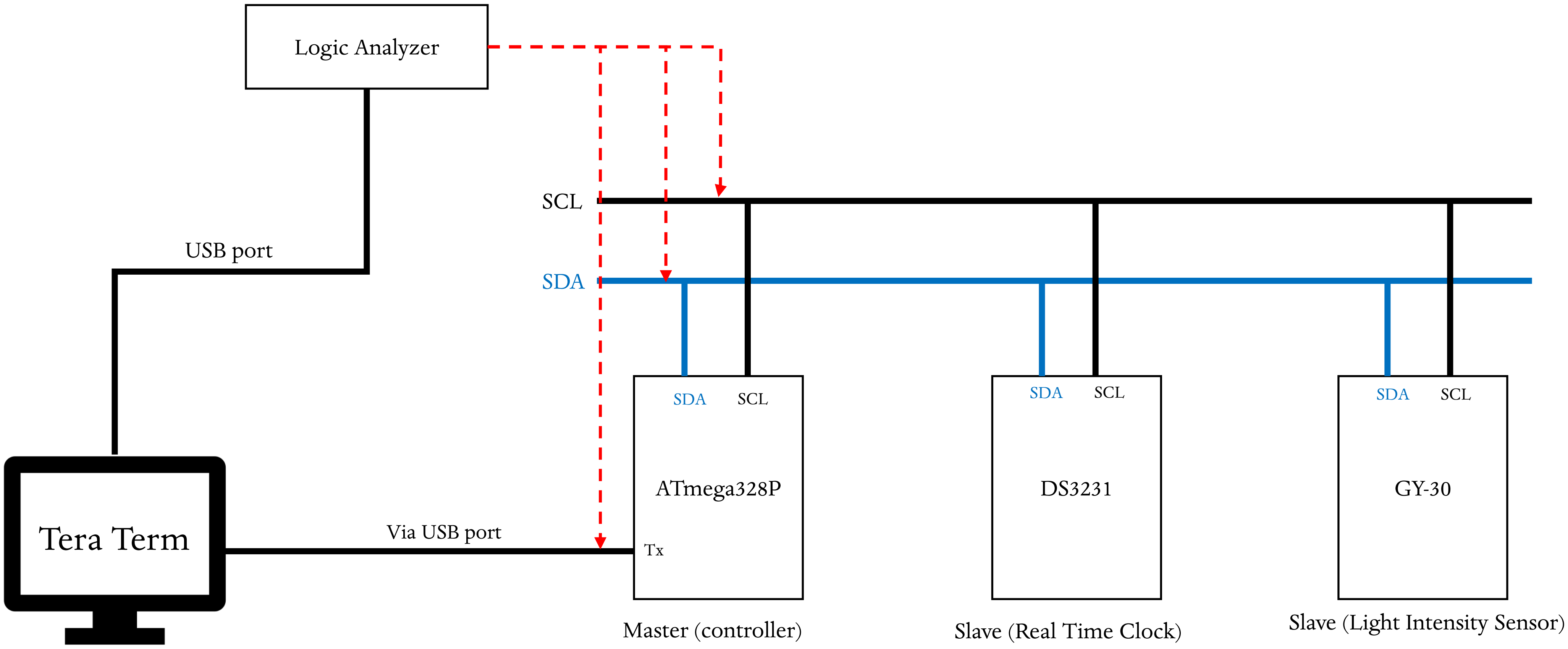
```
void writeI2C(uint8_t data) {
	// Load data into TWI data register
	TWDR = data;

	// Clear the TWINT bit to start data transmission
	TWCR = (1 << TWINT) | (1 << TWEN);

	// Wait until transmission is complete
	while (!(TWCR & (1 << TWINT)));
	// Check status code (this should be 0x20 (sent SLA+W, ack received),0x40 (sent
	SLA+R, ack received), or 0x28 (sent data, received))
	if(((TWSR & 0xF8) != 0x18) && ((TWSR & 0xF8) != 0x28) && ((TWSR & 0xF8) != 0x40)) {
		char error_msg[32];
		int size_emsg;
		size_emsg = sprintf(error_msg,"Error writeI2C %x \n",(TWSR & 0xF8));
		sendSerial(error_msg,size_emsg);
	}
}
```

Interfacing the Application to the TWI in a Typical Transmission

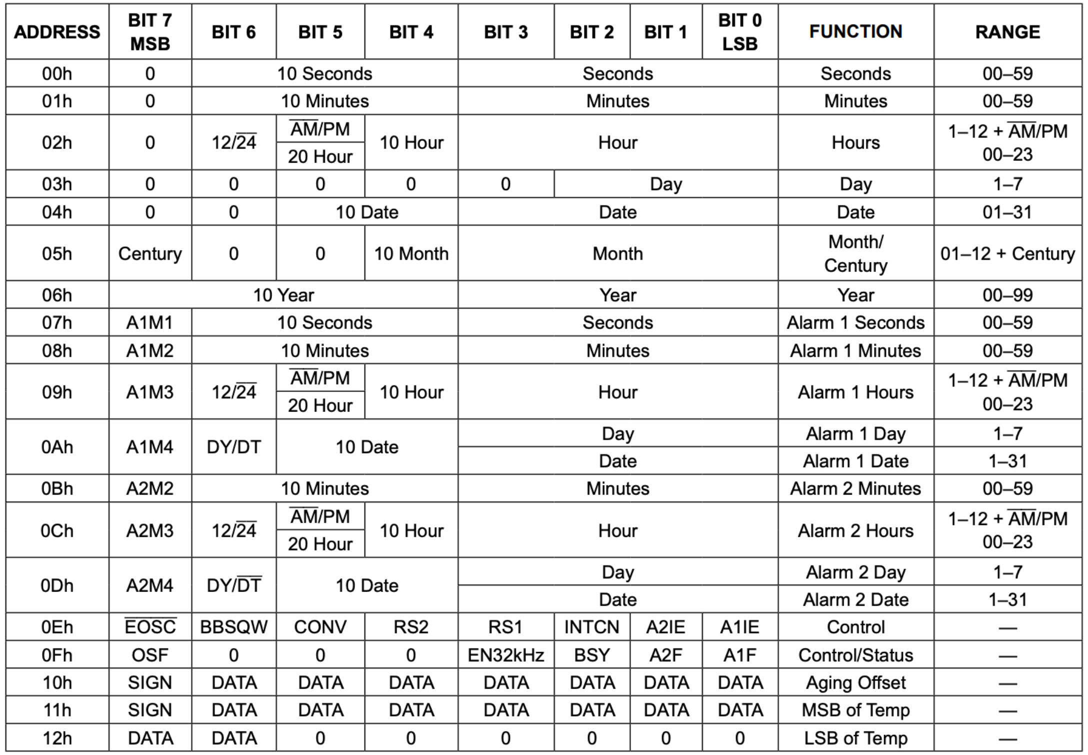


```
uint8_t readI2C(uint8_t ack) {
// Enable or disable ACK bit based on ack parameter
      if (ack) {
            TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWEA);
      } else {
            TWCR = (1 << TWINT) | (1 << TWEN);
      }

      // Wait until data is received
      while (!(TWCR & (1 << TWINT)));
      // Check status code (this should be 0x50 (sent data, ack sent),0x58 (sent data, N-
      ack sent)
      if(((TWSR & 0xF8) != 0x58) && ((TWSR & 0xF8) != 0x50)) {
            char error_msg[32];
            int size_emsg;
            size_emsg = sprintf(error_msg,"Error readI2C %x \n",(TWSR & 0xF8));
            sendSerial(error_msg,size_emsg);
      }

      // Return received data
      return TWDR;
}
```

Experiment with GY-30 and DS3231



DS3231 Communication



ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/$\overline{\text{24}}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{\text{24}}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Ah	A1M4	DY/DT	10 Date		Day Date				Alarm 1 Day Alarm 1 Date	1–7 1–31
0Bh	A2M2	10 Minutes			Minutes				Alarm 2 Minutes	00–59
0Ch	A2M3	12/$\overline{\text{24}}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Dh	A2M4	DY/$\overline{\text{DT}}$	10 Date		Day Date				Alarm 2 Day Alarm 2 Date	1–7 1–31
0Eh	$\overline{\text{EOSC}}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

Figure 1. Timekeeping Registers

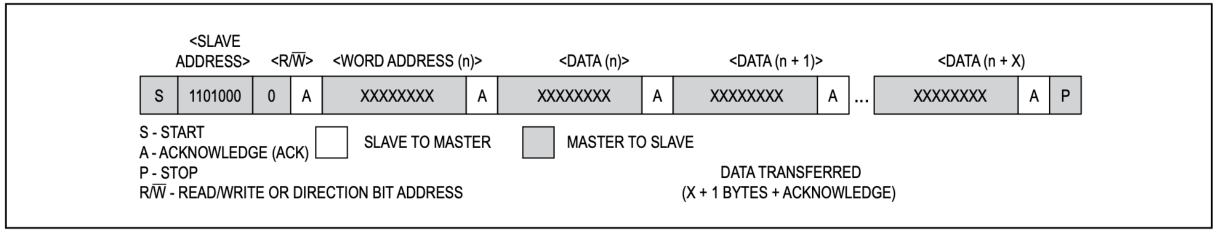


Figure 3. Data Write—Slave Receiver Mode

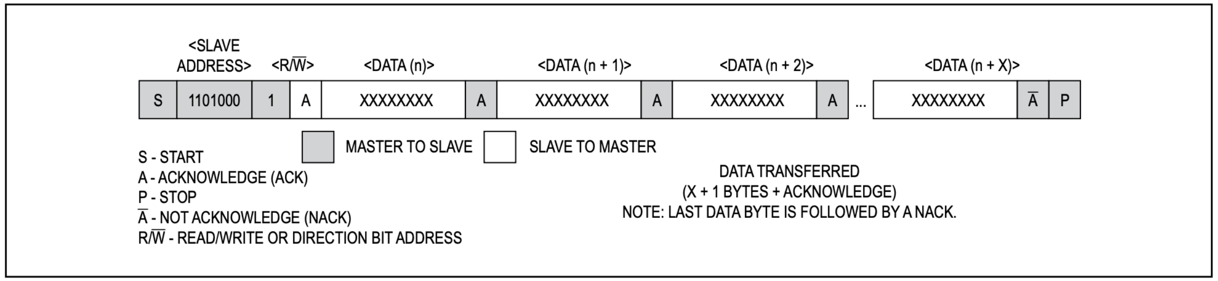


Figure 4. Data Read—Slave Transmitter Mode

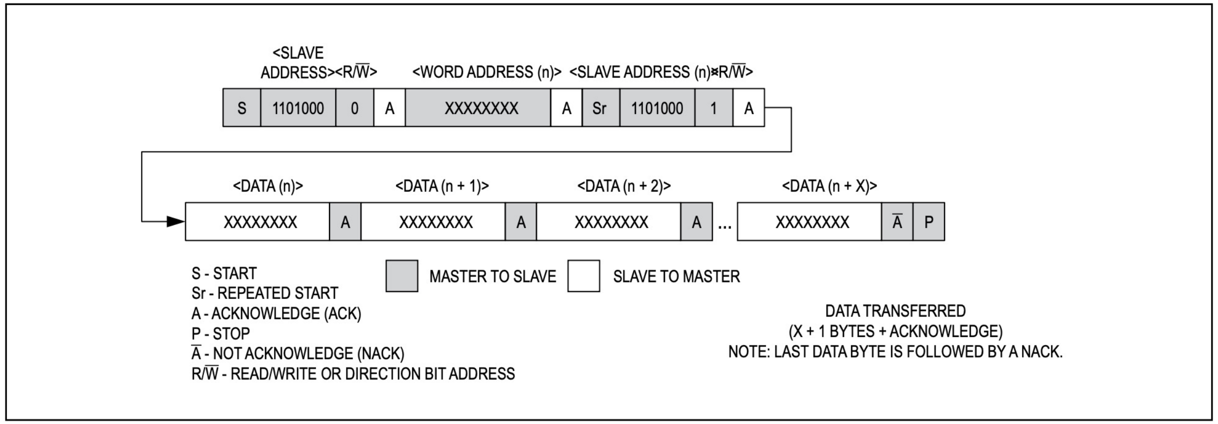
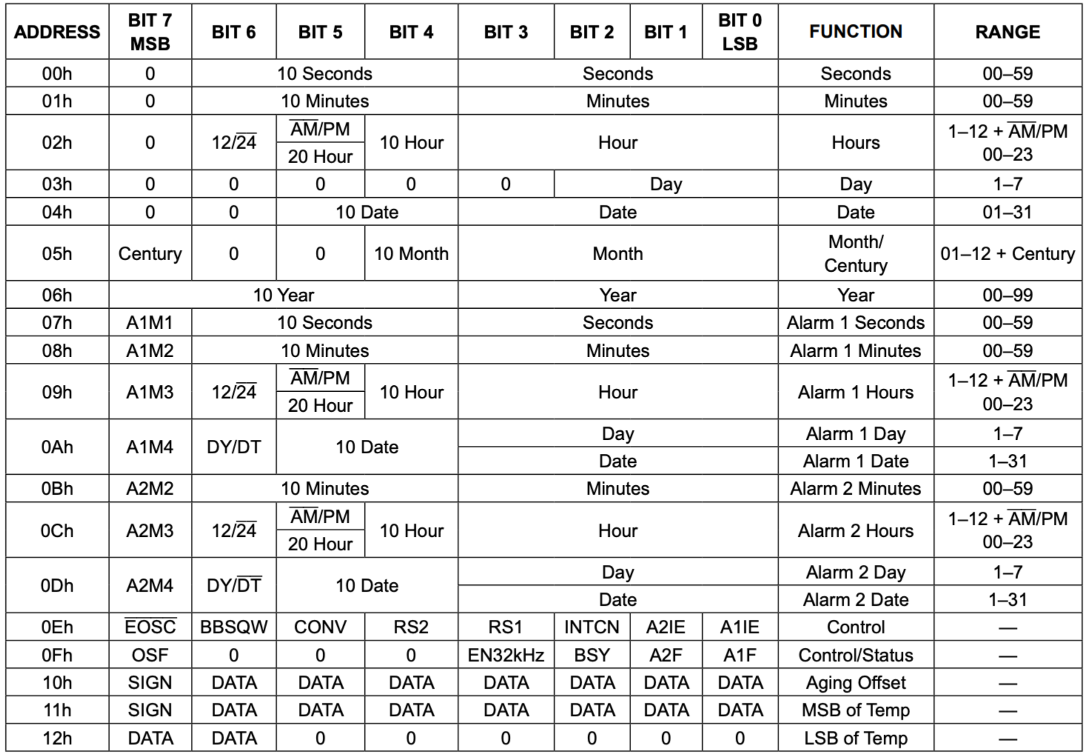


Figure 5. Data Write/Read (Write Pointer, Then Read)—Slave Receive and Transmit

DS3231 Communication



ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Ah	A1M4	DY/$\overline{\text{DT}}$	10 Date		Day				Alarm 1 Day	1–7
					Date				Alarm 1 Date	1–31
0Bh	A2M2	10 Minutes			Minutes				Alarm 2 Minutes	00–59
0Ch	A2M3	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Dh	A2M4	DY/$\overline{\text{DT}}$	10 Date		Day				Alarm 2 Day	1–7
					Date				Alarm 2 Date	1–31
0Eh	$\overline{\text{EOSC}}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

Figure 1. Timekeeping Registers

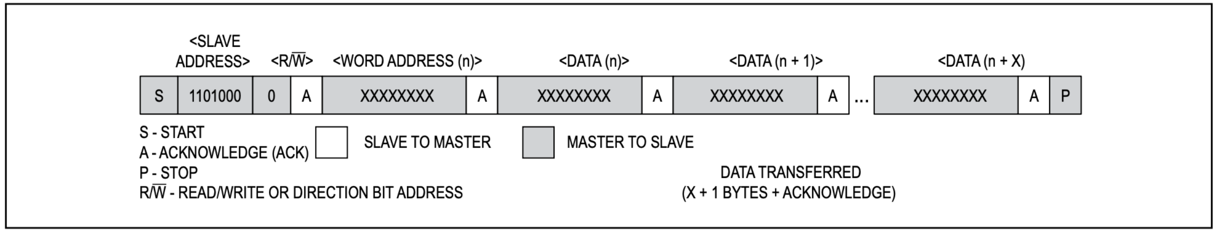
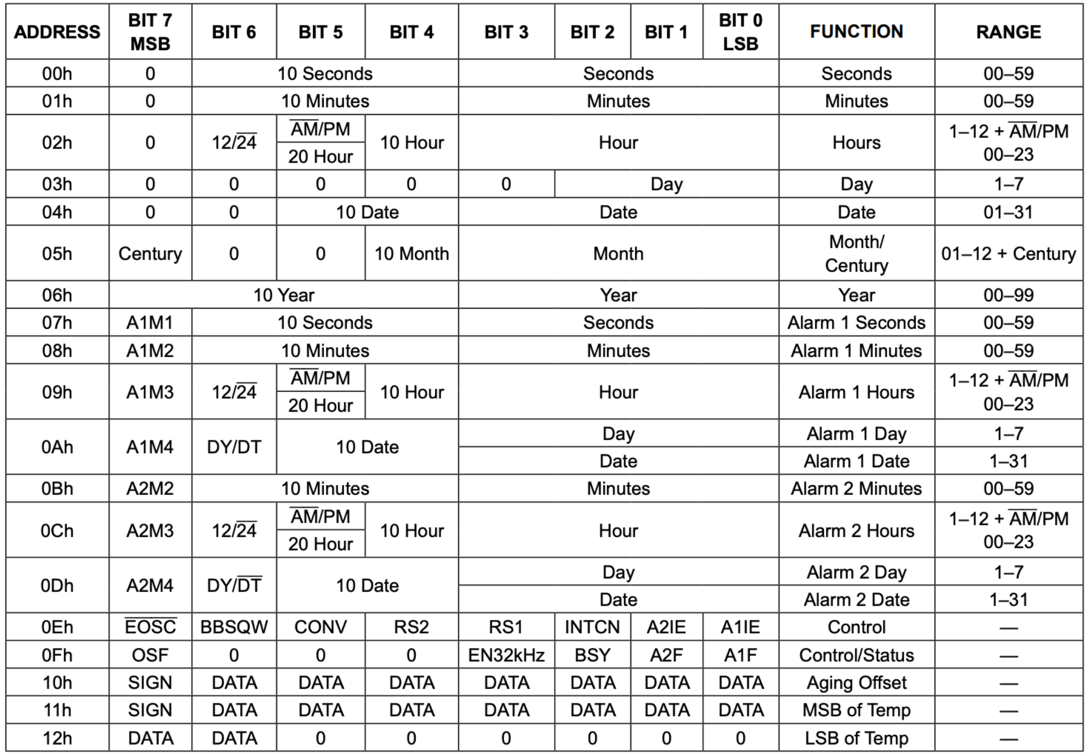


Figure 3. Data Write—Slave Receiver Mode

```
void initDS3231() {
// Assuming you want to set the initial time to 12:00:00
startI2C();
writeI2C(DS3231_ADDRESS << 1); // Write address
writeI2C(0x00); // Set register pointer to 0x00 (Seconds)
writeI2C(0x00); // Seconds
writeI2C(0x00); // Minutes
writeI2C(0x12); // Hours (12-hour format)
// Set other relevant registers as needed
stopI2C();
}
```

DS3231 Communication



ADDRESS	BIT 7 MSB	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0 LSB	FUNCTION	RANGE
00h	0	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
03h	0	0	0	0	0	Day			Day	1–7
04h	0	0	10 Date		Date				Date	01–31
05h	Century	0	0	10 Month	Month				Month/ Century	01–12 + Century
06h	10 Year				Year				Year	00–99
07h	A1M1	10 Seconds			Seconds				Alarm 1 Seconds	00–59
08h	A1M2	10 Minutes			Minutes				Alarm 1 Minutes	00–59
09h	A1M3	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Ah	A1M4	DY/$\overline{\text{DT}}$	10 Date		Day				Alarm 1 Day	1–7
					Date				Alarm 1 Date	1–31
0Bh	A2M2	10 Minutes			Minutes				Alarm 2 Minutes	00–59
0Ch	A2M3	12/$\overline{24}$	$\overline{\text{AM}}$/PM 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{\text{AM}}$/PM 00–23
0Dh	A2M4	DY/$\overline{\text{DT}}$	10 Date		Day				Alarm 2 Day	1–7
					Date				Alarm 2 Date	1–31
0Eh	$\overline{\text{EOSC}}$	BBSQW	CONV	RS2	RS1	INTCN	A2IE	A1IE	Control	—
0Fh	OSF	0	0	0	EN32kHz	BSY	A2F	A1F	Control/Status	—
10h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	Aging Offset	—
11h	SIGN	DATA	DATA	DATA	DATA	DATA	DATA	DATA	MSB of Temp	—
12h	DATA	DATA	0	0	0	0	0	0	LSB of Temp	—

Figure 1. Timekeeping Registers

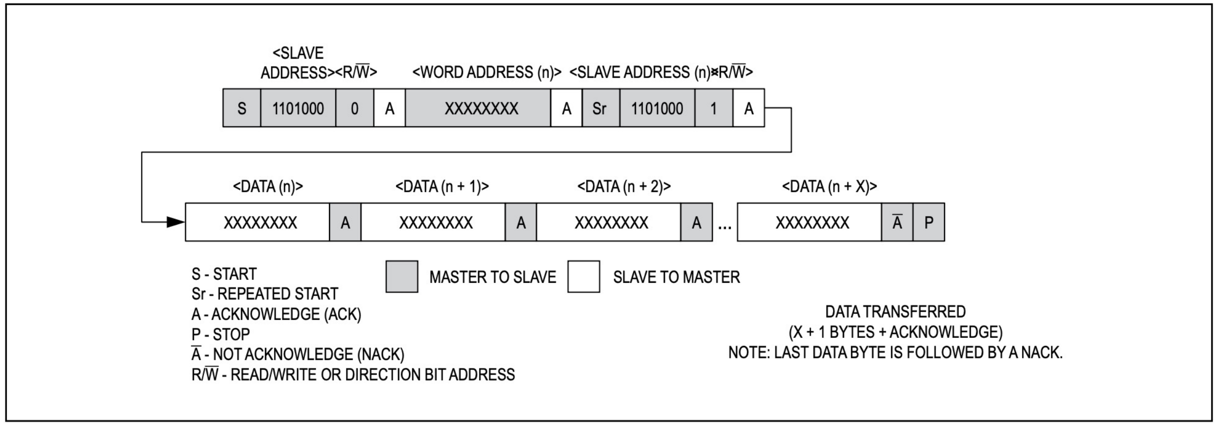
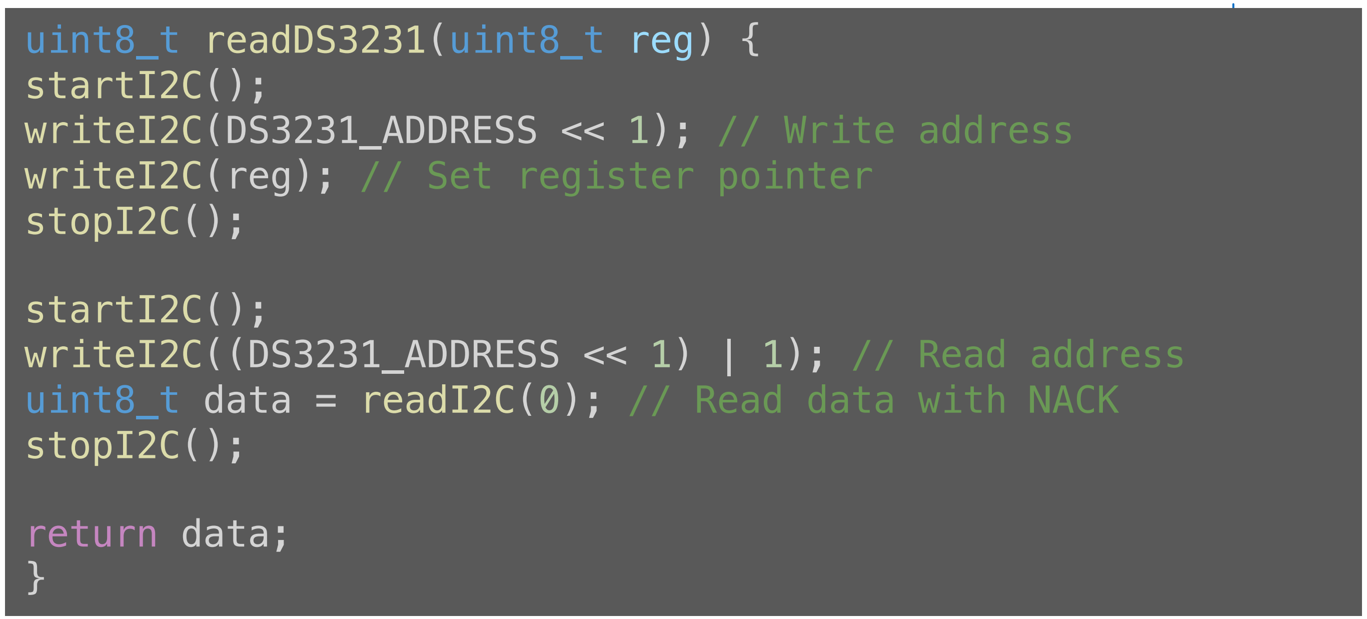
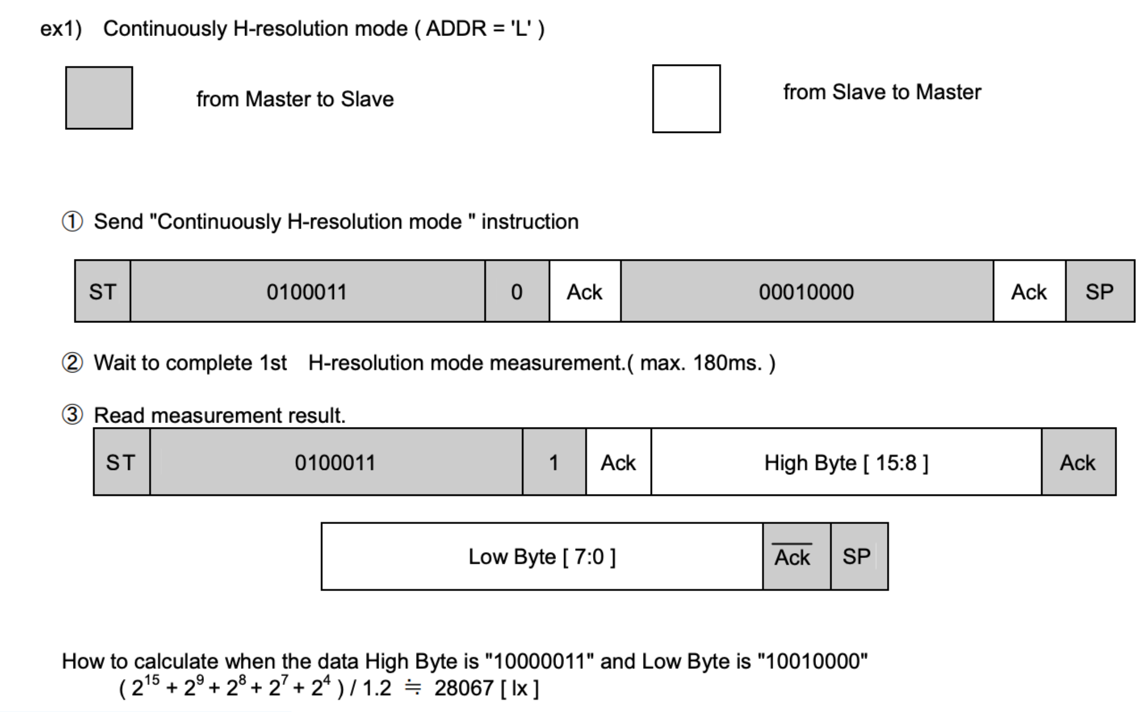


Figure 5. Data Write/Read (Write Pointer, Then Read)—Slave Receive and Transmit



```c
uint8_t readDS3231(uint8_t reg) {
startI2C();
writeI2C(DS3231_ADDRESS << 1); // Write address
writeI2C(reg); // Set register pointer
stopI2C();

startI2C();
writeI2C((DS3231_ADDRESS << 1) | 1); // Read address
uint8_t data = readI2C(0); // Read data with NACK
stopI2C();

return data;
}
```

GY-30 Communication



ex1) Continuously H-resolution mode (ADDR = 'L')

▨ from Master to Slave

□ from Slave to Master

① Send "Continuously H-resolution mode " instruction

ST	0100011	0	Ack	00010000	Ack	SP

② Wait to complete 1st H-resolution mode measurement.(max. 180ms.)

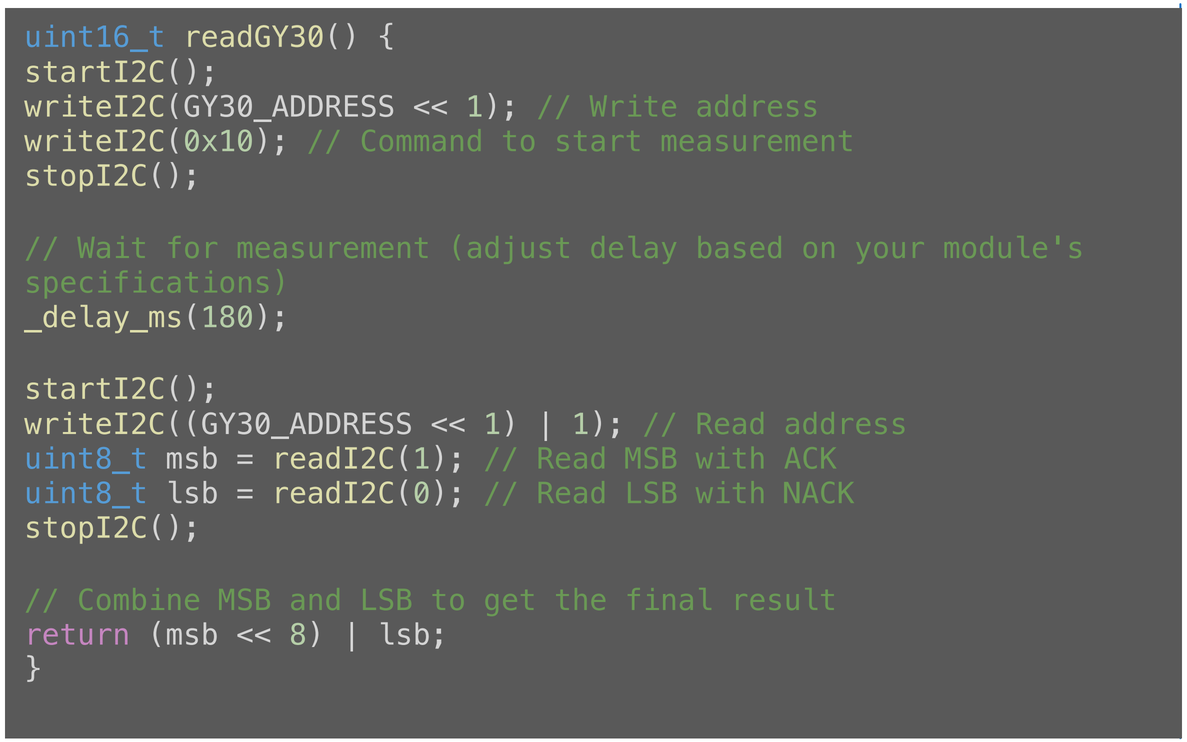
③ Read measurement result.

ST	0100011	1	Ack	High Byte [15:8]	Ack

Low Byte [7:0]	$\overline{\text{Ack}}$	SP

How to calculate when the data High Byte is "10000011" and Low Byte is "10010000"

$(2^{15} + 2^{9} + 2^{8} + 2^{7} + 2^{4}) / 1.2 \fallingdotseq 28067$ [lx]



```
uint16_t readGY30() {
startI2C();
writeI2C(GY30_ADDRESS << 1); // Write address
writeI2C(0x10); // Command to start measurement
stopI2C();

// Wait for measurement (adjust delay based on your module's
specifications)
_delay_ms(180);

startI2C();
writeI2C((GY30_ADDRESS << 1) | 1); // Read address
uint8_t msb = readI2C(1); // Read MSB with ACK
uint8_t lsb = readI2C(0); // Read LSB with NACK
stopI2C();

// Combine MSB and LSB to get the final result
return (msb << 8) | lsb;
}
```