I²C Master

1. Introduction

1.1 Core Overview

This I²C master module implements a streamlined, single-master bus controller designed for interfacing with slave devices via a memory-mapped register interface. It operates in I²C Standard-mode (up to 100 kHz) and supports both single and repeated write transactions. The module is controlled by a finite state machine (FSM) and configured through a compact set of control and status registers. It is fully compatible with the Avalon Bus standard, facilitating integration into FPGA-based systems.

The architecture is fully synchronous and omits interrupts, FIFOs, and arbitration mechanisms, making it suitable for low complexity embedded applications. The design features open-drain control for the bidirectional SDA line and uses an output-only SCL signal. Communication strictly follows the I²C specification, including START and STOP conditions, ACK/NACK handling, and proper data framing.

1.2 Supported Features

- Single-Master I²C Operation
- Operates up to 100 kHz SCL .
- Start, Stop: Fully automatic by FSM.
- **7**-Bit Address Support.
- Register-Based Interface**:** Control, data and status via 4 mapped registers.
- Polling Interface**:** DONE and READY flags for host-side flow control.
- Single and Repeated Writes**:** CONTINUE bit enables back-to-back write operations without STOP.
- Read and Write Support: Both directions implemented with ACK checking.
- SDA output delay mechanism.

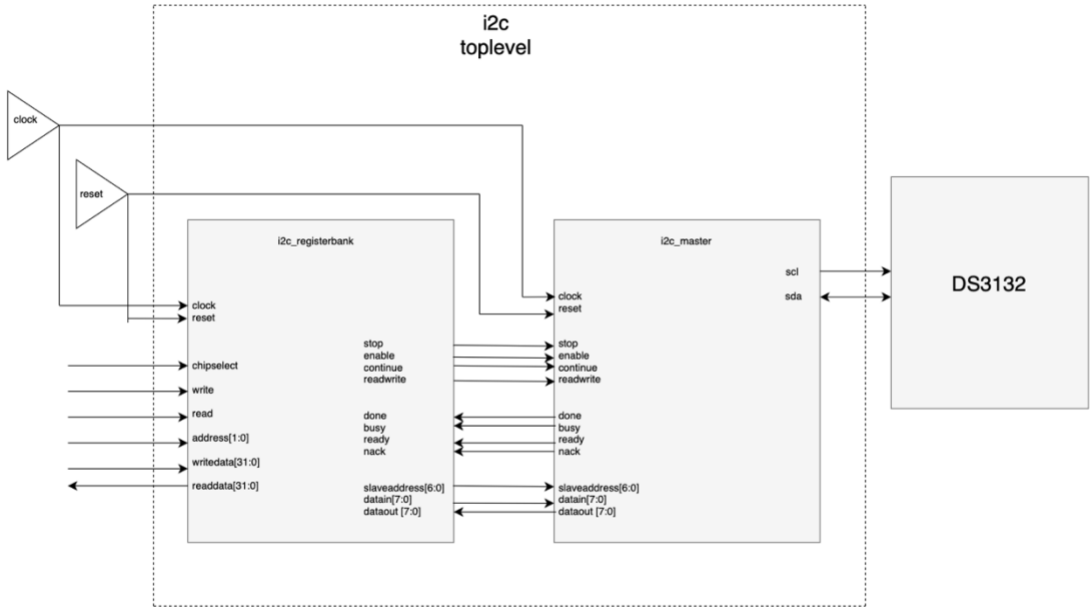
1.3 Limitations

- No Fast-mode (400 kHz) support.
- No FIFO or buffered transfer capability.
- No interrupt output or slave-mode functionality.
- No clock stretching.

2. Interface

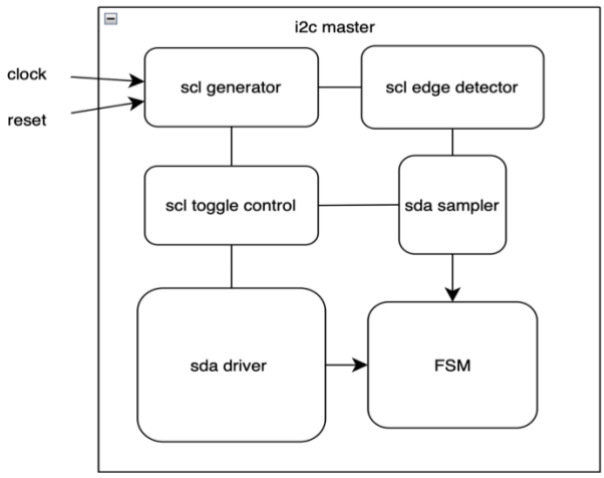
2.1 Top-Level System Integration

The figure below depicts the high-level architecture of the I²C master module in a typical system context. The module is controlled by a host processor or testbench through a simplified memory-mapped interface consisting of four registers. It initiates I²C transactions to a target slave device (e.g., DS3231) using an open-drain SDA line and an output-only SCL signal, adhering to the I²C protocol's electrical and timing specifications.



Note: *Typo in the figure *DS3231**

2.2 Internal Module Breakdown



At a coarse level, the I²C master core consists of the following blocks:

- **FSM**
  Governs all I²C protocol steps: START, address, R/W bit, ACK/NACK, data transfer and STOP.
- **SCL Generator**
  Derives the I²C clock (SCL) from the high-speed system clock by dividing down using a fixed ratio. All timing parameters are met to achieve 100 kHz nominal SCL frequency.
- **SDA Driver**
  Controls the bidirectional data line (SDA) as an open-drain output. When transmitting, it actively drives SDA low; when receiving, it releases SDA to high impedance so the slave may pull the line. All transitions are synchronized to the SCL edges.
- **SDA Sampler**
  Monitors the state of the SDA line at precise instants (on SCL rising edges) to capture incoming bits or ACK/NACK signals.
- **SCL Toggle Control**
  Orchestrates the timing of SCL by enabling or disabling the clock generator output at the correct FSM states. Ensures that SCL only toggles when the bus is active and remains static during START/STOP conditions and “Wait” state.
- **SCL Edge Detector**
  Observes the divided-down SCL signal to generate clean pulses on its rising and falling edges. These pulses drive state transitions in the FSM (e.g. transitions from ack state and into write state).

3. Port Descriptions

The following table summarizes the top-level ports of the I²C master core. These include system inputs (clock, reset), memory-mapped register interface (control/data/status), and serial bus signals (SCL, SDA).

Port	Width	Mode	Data Type	Interface	Description
clk	1	in	std_logic	user logic	System clock
reset	1	in	std_logic	user logic	Active-low synchronous reset
slave_address	7	in	std_logic_vector(6 downto 0)	user logic	7-bit address of target I2C slave device
data_in	8	in	std_logic_vector(7 downto 0)	user logic	Byte to transmit on I2C bus
data_out	8	out	std_logic_vector(7 downto 0)	user logic	Byte received from slave device
enable	1	in	std_logic	user logic	Transaction trigger: 1 = start; 0 = idle. Self-clearing after FSM initiation
continue	1	in	std_logic	user logic	Enables repeated write for continued transfer
read_write	1	in	std_logic	user logic	0 = write to slave; 1 = read from slave
stop_signal	1	in	std_logic	user logic	Assert to request STOP condition after transfer (or in wait state)
busy	1	out	std_logic	user logic	Indicates the controller is actively performing a transaction
ack_error	1	out	std_logic	user logic	Asserted if slave does not acknowledge address or data
done	1	out	std_logic	user logic	Set high when transaction completes (i.e. in stop state)
ready	1	out	std_logic	user logic	Indicates the controller is in wait state and ready for new data byte
scl	1	out	std_logic	slave device	I2C serial clock line (output-only)
sda	1	inout	std_logic	slave device	I2C serial data line (open-drain bidirectional, externally pulled up)

4. Internal Memory Map

The I²C master is controlled through four 32-bit memory-mapped registers. These are accessed via a simple bus interface and provide full control over the transaction flow

4.1 Register Overview

Name	31 downto 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Control Register (0x00)	Unused	Unused											Continue	Read/Write	Stop	Enable
Write Register (0x04)	Unused	Slave Addr							Data In							
Status Register (0x08)	Unused	Unused											Ready	Ack Error	Busy	Done
Read Register (0x0C)	Unused	Unused							Data out							

4.2 Bitfield Descriptions

0x00 – Control Register

Used by the host to initiate and configure I²C transactions. Only bits [3:0] are used:

- Bit 3 – CONTINUE
  Instructs the FSM to leave the “wait” state and back into write state. Important to note that the FSM will stay indefinitely in the wait state unless user sets STOP or Continue.
- Bit 2 – Read Write
  0 = Write operation (master → slave)
  1 = Read operation (slave → master)
- Bit 1 – STOP
  Instructs the master to issue a STOP condition at the next safe point (e.g., after ACK sampling). Also used as part of the handshake in the “wait” state, if no further data is to be transferred STOP must be set.
- Bit 0 – ENABLE
  A single-cycle pulse that launches the FSM from IDLE and into the START sate. Note that the signal is latched and cleared by the module after consumption.

All other bits should be written as zero.

0x08 – STATUS Register

Reflects the FSM's current state. Only bits [3:0] are used.

- Bit 3 – READY
  High when the FSM is in the "wait" state. Indicates the module is ready for a CONTINUE or STOP command.
- Bit 2 – ACK_ERROR
  Set if the slave NACKs either the address or data byte. Cleared automatically when a new start condition is asserted.
- Bit 1 – BUSY
  High while the FSM is executing a transaction (from START until the STOP sequence completes). Cleared in the stop state.
- Bit 0 – DONE
  Set high when the transaction fully completes. Cleared when a new start condition is asserted.

0x04 – WRITE Register

Loads data for outgoing transfers. Only bits [14:0] are used.

- Bits 14:8 – SLAVE_ADDRESS
  7-bit I²C address of the target device.
- Bits 7:0 – WRITE_DATA
  Data byte or register pointer to send. Must be written before asserting ENABLE**.**

All other bits should be written as zero.

0x0C – READ Register (Read-Only)

Holds the byte received from the slave. Only bits [7:0] are used.

- Bits 7:0 – READ_DATA
  Valid immediately after the STOP following a read operation. Must be read before the next transaction to avoid overwriting.

Note: Register base addresses (0x00, 0x04, 0x08, 0x0C) assume 32-bit address strides; adjust offsets if your bus is byte-addressed.

5. Functional Description

This section describes how the I²C master operates from a functional standpoint. It includes an overview of the transfer mechanism, the role of control and status flags and the core finite state machine.

5.1 Transaction Flow – Single Write Sequence (Master to Slave)

Step 1: Initiate Write Operation

The host sets the control register as follows:

- RW = 0 (indicating a write operation)
- ENABLE = 1 (trigger transaction start)

The host then polls the status register. Once READY=1, the host sets STOP = 1 to signal completion.

Step 2: I²C Protocol Execution (FSM-Driven)

Upon activation, the finite state machine (FSM) performs the following protocol-compliant operations:

- Issues a START condition
- Transmits the 7-bit slave address with the write bit
- Awaits and checks for ACK from the slave
- Sends the 8-bit data word from the write register
- Awaits and checks for ACK from the slave
- Waits internally for STOP = 1
- On STOP assertion, issues a STOP condition

Step 3: Post-Transaction Status

Once the STOP condition has been transmitted, the following status flags are updated:

- DONE is asserted (set high)
- BUSY is deasserted
- ACK_ERROR is asserted if any NACK was received during the transaction

5.2 Transaction Flow – Read Sequence (Slave to Master)

Step 1: Initiate Read Operation

The host sets the control register as follows:

- RW = 1 (indicating a read operation)
- ENABLE = 1 (trigger transaction start)

Step 2: I²C Protocol Execution (FSM-Driven)

Upon activation, the finite state machine (FSM) performs the following operations:

- Issues a START condition
- Transmits the 7-bit slave address with the read bit
- Waits for ACK from the slave
- Receives 8-bit data from the slave
- Sends a NACK to indicate end of reception
- Issues a STOP condition.

Step 3: Post-Transaction Status

Once the read operation completes:

- Received data is latched into the READ register
- DONE is asserted (set high)
- ACK_ERROR is asserted if the slave did not acknowledge the address

5.3 Transaction Flow – Continued Write Sequence (Master to Slave)

Step 1: Prepare for Continuation

After the initial transaction completes and the READY flag is detected, the host performs the following:

- Loads a new 8-bit data word into the write register
- Sets CONTINUE = 1 in the control register

Step 2: FSM Execution – Continued Write

The FSM re-enters the write state and performs a transaction sequence that is functionally identical to the standard write operation:

- Transmits the 7-bit slave address with the write bit
- Waits for ACK from the slave
- Sends the new 8-bit data word from the write register
- Waits for ACK from the slave
- Waits internally for STOP = 1 before issuing STOP condition

Step 3: Post-Transaction Status

After STOP is issued:

- DONE is set high
- BUSY is cleared
- ACK_ERROR is asserted if any NACK occurred during address or data phase

5.4 Finite State Machine (FSM)

The I²C master is governed by a deterministic FSM that progresses through the following states:

- IDLE

  The FSM remains in this state until a transaction is initiated by setting ENABLE = 1. All control signals are inactive.

- START

  Generates the I²C START condition by pulling SDA low while SCL is high, marking the beginning of a transaction.

- SLAVE ADDRESS

  Sends the 7-bit slave address along with the R/W bit on the bus.

- ADDRESS ACK

  Waits for acknowledgment (ACK/NACK) from the slave.
  - On ACK: Proceeds to either WRITE DATA or READ DATA depending on the R/W bit.
  - On NACK: Proceeds directly to PREP STOP.

- WRITE DATA

  Transmits an 8-bit data byte to the slave, sourced from the internal write register.

- SLAVE ACK

  Waits for an ACK from the slave after data transmission.
  - On ACK: Proceeds to WAIT (WRITE) to check whether to continue or stop.
  - On NACK: Proceeds to PREP STOP and sets ACK_ERROR.

- WAIT (WRITE)

  Waits for host input:
  - If CONTINUE = 1, returns to WRITE DATA with new data.
  - If STOP = 1, proceeds to PREP STOP.

- READ DATA

  Receives 8 bits from the slave and latches them into the read register.

- MASTER ACK

  Master sends a NACK to signal the end of the read operation, then proceeds to PREP STOP.

- PREP STOP

  Final preparations for terminating the transaction. Incorporated to ensure a proper STOP condition.

- STOP

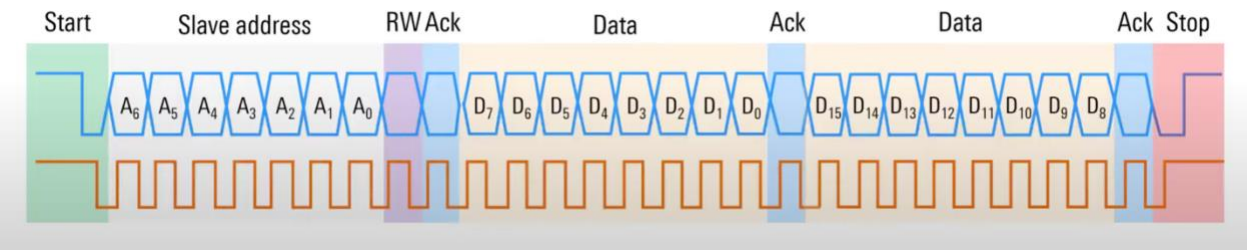
  Generates a valid I²C STOP condition by transitioning SDA high while SCL is high.
  Clears BUSY, sets DONE, and returns to IDLE.

6. Timing Diagrams

This section defines the timing behavior of the I²C master interface, including both bus-level waveforms (SCL/SDA) and internal control sequences.

6.1 I²C Write Transaction Timing

Below is a simplified diagram showing a two byte write with a STOP condition:



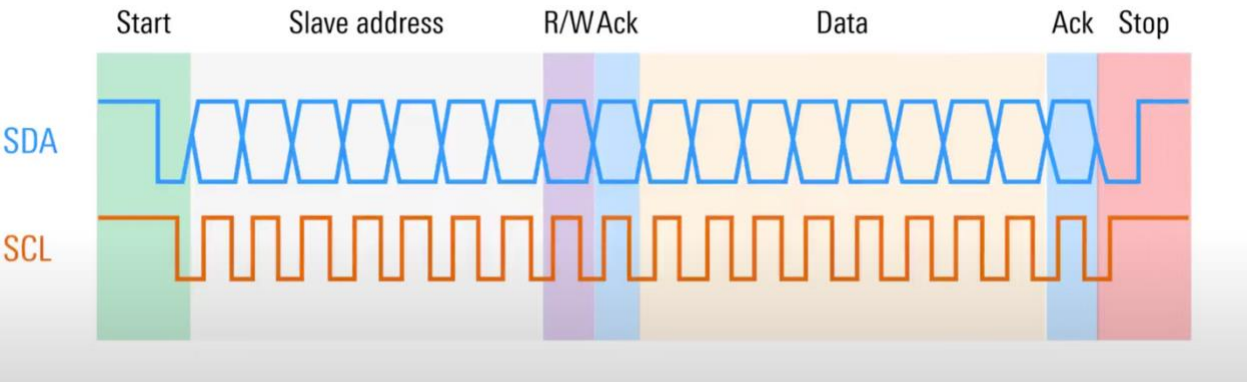
Note:

This timing diagram is adapted from a YouTube video. While it closely reflects the actual I²C sequence, it is important to clarify that in the implemented design, the FSM transitions into a wait state following the second ACK. It remains in this state until receiving explicit confirmation from the software, either to proceed with a STOP condition or to initiate a subsequent write transaction. Actual Waveforms from the module can be found in the report.[1]

Key timing phases:

- START**:** SDA falls while SCL is high.
- ADDRESS**:** 7 bits + R/W; sent MSB first on SDA, sampled on rising SCL.
- ACK/NACK**:** Slave pulls SDA low for ACK, high for NACK.
- DATA BYTE**:** Transmitted on SDA; each bit latched on rising SCL.
- STOP**:** SDA rises while SCL is high.

6.2 I²C Read Transaction Timing



Note:

This timing diagram is adapted from a YouTube video. While it closely reflects the actual I²C sequence, it is important to clarify that the actual waveforms differ. Actual Waveforms from the module can be found in the report.[1]

Key timing phases:

- The FSM sets SDA to high-Z after the address phase.
- The slave drives the SDA line with data.
- Master sends NACK after the last byte to signal end-of-read.

6.3 Timing Constraints and Parameters

All timing measurements presented were obtained through simulation using Questa, based on a nominal I²C bus frequency of 100 kHz. No additional parasitic loads or external pull-up delays were modeled. While waveform results are available, they exhibit signal clutter typical of digital simulation environments and are therefore included only in the full report where they are explained in detail. These values represent typical observed behavior and should be validated against actual post-synthesis or post-layout timing for deployment

Parameter	Symbol	Measured Value (ns)	Description
SCL Low Period	t_LOW	5004	Duration SCL remains low
SCL High Period	t_HIGH	5004	Duration SCL remains high
SDA Setup Time	t_SU	4984	Time SDA is stable before SCL rising edge
SDA Hold Time	t_HD	20	Time SDA is held stable after SCL falling edge
START Hold to SCL High	t_HDSTA	4406	SDA low hold after SCL goes high (START condition)
STOP Setup to SCL High	t_SUSTO	4053	SDA high setup before SCL goes high (STOP condition)

[2]

Parameter	Symbol	Standard-Mode Min [ns]	Measured Value [ns]
SCL Low Period	tLOW = tHIGH	4700	5004
SCL High Period	tHIGH	4000	5004
SDA Setup Time	tSU;DAT	250	4984
SDA Hold Time	tHD;DAT	0	20
START Hold to SCL High	tHD;STA	4000	4406
STOP Setup to SCL High	tSU;STO	4000	4053

The following tables summarizes key timing parameters measured during operation of the I²C master interface. All values are recorded in nanoseconds based on simulation or hardware test conditions. These represent typical observed values and are not guaranteed limits. Users integrating this IP should validate timing against their specific system clock and bus configuration.

Timing margins must be accounted for in any deployment to ensure reliable communication under worse-case operating conditions.

6.3.1 Input Signal Timing Requirements

Signal	Condition	Setup Time (ns)	Hold Time (ns)	Notes
ENABLE	Rising edge	3	1	Sampled on rising edge of 50 MHz system clock
RW	Level sampled	3	1	Must be valid in same cycle as ENABLE
STOP	Level sampled	3	1	Synchronous condition check before STOP condition is generated
CONTINUE	Rising edge	3	1	Drives FSM transition out of WAIT state

[2]

The control signals used to initiate and manage transactions (ENABLE, RW, STOP, and CONTINUE) are sampled synchronously on the rising edge of a 50 MHz system clock. The required setup and hold times are defined to ensure correct latching into the FSM's state registers. All inputs must remain stable for at least 3 ns before and 1 ns after the clock edge to guarantee proper operation under nominal conditions. These constraints assume single-cycle signal assertions without asynchronous edges or metastability risks.

7. Software Integration and Example Driver

This section describes how the I²C master module is accessed from C software running on a host processor. A memory-mapped interface is used to initiate I²C transactions, check status, and read or write data. The following examples show how the module is used to communicate with a DS3231 real-time clock (RTC) over I²C.

The example driver provides a minimal abstraction layer that hides low-level control logic and register handling, simplifying integration in embedded systems. All operations are blocking (polling-based), using READY and DONE flags for synchronization.

7.1 Header File

This file defines register addresses, bit masks, and the public interface for time and temperature access via the RTC:

```
#ifndef RTC_DRIVER_H          // Prevents multiple inclusions of this header file
#define RTC_DRIVER_H          // Defines the macro to indicate the header has been included

#include <system.h>           // Includes system-level definitions
#include <io.h>               // Includes I/O access macros/functions for hardware registers
#include <stdio.h>            // Includes standard I/O functions

typedef unsigned char byte;   // Defines 'byte' as an alias for 'unsigned char'

#define I2C_0_BASE        0x81000    // Base memory address for the I2C controller
#define CONTROL_REGISTER  0x00       // control register
#define WRITE_REGISTER    0x04       // write register (data to be sent)
#define STATUS_REGISTER   0x08       //status register (to check I2C state)
#define READ_REGISTER     0x0C       // read register (to receive data)

#define ENABLE_BIT        0x01       // Bit mask to enable I2C transaction
#define STOP_BIT          0x02       // Bit mask to indicate a STOP condition
#define RW_BIT            0x04       // Bit mask to specify read (1) or write (0) mode
#define CONTINUE_BIT      0x08   // Bit mask to continue an I2C transaction

#define READY_BIT         0x08       // Status bit indicating readiness for operation
#define ACKERROR_BIT      0x04       // Status bit indicating acknowledgment error
#define DONE_BIT          0x01       // Status bit indicating transaction completion

#define RTC_ADDRESS       0x68       // I2C address of the RTC device
#define REG_SECONDS       0x00       // Register address for seconds
#define REG_MINUTES       0x01       // Register address for minutes
#define REG_HOURS         0x02       // Register address for hours
#define REG_WEEKDAY       0x03       // Register address for weekday
#define REG_DAY           0x04       // Register address for day of month
#define REG_MONTH         0x05       // Register address for month
#define REG_YEAR          0x06       // Register address for year
#define REG_TEMP_HIGH     0x11       // Register address for high byte of temperature
#define REG_TEMP_LOW      0x12       // Register address for low byte of temperature

void set_rtc_time(byte second, byte minute, byte hour,       // Function prototype to set RTC time
                  byte week_day, byte day, byte month, byte year);

void get_rtc_time(byte *second, byte *minute, byte *hour,    // Function prototype to retrieve RTC time
                  byte *week_day, byte *day, byte *month, byte *year);

float get_rtc_temp(void);                                    // Function prototype to retrieve temperature from RTC

#endif // RTC_DRIVER_H        // End of include guard
```

7.2 Driver Implementation

Implements the transaction sequences required to write and read bytes to/from the DS3231 over I²C:

```
#include "rtc_driver.h"  // Includes definitions and constants from the RTC driver header

// Reads the status register from the I2C controller
static byte read_status(void) {
    return IORD_32DIRECT(I2C_0_BASE, STATUS_REGISTER);  // Read status register at base address + STATUS_REGISTER
}

// Clears specific status bits by writing them back to the status register
static void clear_status_bits(byte bits) {
    IOWR_32DIRECT(I2C_0_BASE, STATUS_REGISTER, bits);
}

// Waits until the I2C controller is ready or reports an acknowledgment error
static void wait_ready(void) {
    byte status;
    do {
        status = read_status();  // Continuously read the status register
    } while (!(status & (READY_BIT | ACKERROR_BIT)));  // Wait until READY or ACK error occurs
    if (status & ACKERROR_BIT) {
        clear_status_bits(ACKERROR_BIT);  // Clear ACK error if it occurred
    }
}

// Waits until the I2C controller signals transaction completion or an acknowledgment error
static void wait_done(void) {
    byte status;
    do {
        status = read_status();  // Continuously read the status register
    } while (!(status & (DONE_BIT | ACKERROR_BIT)));  // Wait until DONE or ACK error occurs
    if (status & ACKERROR_BIT) {
        clear_status_bits(ACKERROR_BIT);  // Clear ACK error if it occurred
    }
}

// Sends a byte to a specific register of the RTC over I2C
static void write_byte(byte reg, byte data) {
    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, 0);  // Reset control register before transaction
    IOWR_32DIRECT(I2C_0_BASE, WRITE_REGISTER, (RTC_ADDRESS << 8) | reg);  // Load slave address and register
    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, ENABLE_BIT);  // Start write transaction
    wait_ready();  // Wait for transaction to complete

    IOWR_32DIRECT(I2C_0_BASE, WRITE_REGISTER, (RTC_ADDRESS << 8) | data);  // Load slave address and data
    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, ENABLE_BIT | STOP_BIT);  // Send data with STOP condition
    wait_done();  // Wait for transaction to complete
}

// Reads a byte from a specific register of the RTC over I2C
static byte read_byte(byte reg) {
    byte result;

    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, 0);  // Reset control register before transaction
    IOWR_32DIRECT(I2C_0_BASE, WRITE_REGISTER, (RTC_ADDRESS << 8) | reg);  // Load slave address and register
    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, ENABLE_BIT);  // Start write transaction
    wait_ready();  // Wait for transaction to complete

    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, ENABLE_BIT | STOP_BIT);  // Send STOP condition
    wait_done();  // Wait for transaction to complete

    IOWR_32DIRECT(I2C_0_BASE, CONTROL_REGISTER, ENABLE_BIT | RW_BIT);  // Start read transaction
    wait_done();  // Wait for transaction to complete
    result = IORD_32DIRECT(I2C_0_BASE, READ_REGISTER);  // Read received data
    return result;  // Return result to caller
}

// Writes the provided time and date values to the RTC
void set_rtc_time(byte second, byte minute, byte hour,
                  byte week_day, byte day, byte month, byte year) {
    byte bcd;
    bcd = ((second / 10) << 4) | (second % 10);      // Convert to BCD format
    write_byte(REG_SECONDS, bcd);                    // Write to seconds register
    bcd = ((minute / 10) << 4) | (minute % 10);      // Convert to BCD format
    write_byte(REG_MINUTES, bcd);                    // Write to minutes register
    bcd = ((hour   / 10) << 4) | (hour   % 10);      // Convert to BCD format
    write_byte(REG_HOURS,   bcd);                    // Write to hours register
    bcd = ((week_day/ 10) << 4) | (week_day% 10);       // Convert to BCD format
    write_byte(REG_WEEKDAY, bcd);                    // Write to weekday register
    bcd = ((day    / 10) << 4) | (day    % 10);      // Convert to BCD format
    write_byte(REG_DAY,     bcd);                    // Write to day register
    bcd = ((month  / 10) << 4) | (month  % 10);      // Convert to BCD format
    write_byte(REG_MONTH,   bcd);                    // Write to month register
    bcd = ((year   / 10) << 4) | (year   % 10);      // Convert to BCD format
    write_byte(REG_YEAR,    bcd);                    // Write to year register
}

// Reads the time and date from the RTC and converts them from BCD format
void get_rtc_time(byte *second, byte *minute, byte *hour,
                  byte *week_day, byte *day, byte *month, byte *year) {
    byte raw;
    raw = read_byte(REG_SECONDS);                    // Read seconds register
    *second   = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
    raw = read_byte(REG_MINUTES);                    // Read minutes register
    *minute   = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
    raw = read_byte(REG_HOURS);                      // Read hours register
    *hour     = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
    raw = read_byte(REG_WEEKDAY);                    // Read weekday register
    *week_day = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
    raw = read_byte(REG_DAY);                        // Read day register
    *day      = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
    raw = read_byte(REG_MONTH);                      // Read month register
    *month    = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
    raw = read_byte(REG_YEAR);                       // Read year register
    *year     = ((raw >> 4) * 10) + (raw & 0x0F);    // Convert from BCD to binary
}
```

7.3 Application Example

This application demonstrates the full write-read-verify flow using the I²C master interface. It writes a known time, reads it back, and prints both the time and temperature returned from the RTC.

```
#include "rtc_driver.h"
#include "rtc_driver.c"

int main(void) {
    byte s, m, h, wd, d, mo, yr;
    float temp;

    // Set example time: 12:00:00 on Monday 1st January 2025
    set_rtc_time(0, 0, 12, 3, 1, 1, 25);

    // Read back time and temperature
    get_rtc_time(&s, &m, &h, &wd, &d, &mo, &yr);
    temp = get_rtc_temp();

    // Display results
    printf("Time: %02d:%02d:%02d\n", h, m, s);
    printf("Date: %02d/%02d/20%02d\n", d, mo, yr);
    printf("Temp: %.2f°C\n", temp);

    return 0;
}
```

8. References

[1] “Understanding I2C - YouTube.” Accessed: May 08, 2025. [Online]. Available: https://www.youtube.com/
[2] “UM10204 I2C-Bus Specification and User Manual,” DigiKey. Accessed: May 08, 2025. [Online]. Available: https://www.digikey.com/en/pdf/n/nxp-semiconductors/um10204-i2c-bus-specification-and-user-manual