PSOC™ Control C3 MCU: Implementing PMBus® / SMBus Target and Controller application for ModusToolbox™

About this document

Scope and purpose

This application note provides guidance for developing PMBus®/SMBus applications using the Infineon PMBus® middleware library for ModusToolbox™. It covers both Target (device) and Controller (host) implementations for the PSOC™ Control C3 MCU family.

Intended audience

- Embedded software engineers developing power management solutions
- Hardware engineers evaluating PMBus® devices

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 4
1.1 Features . . . 4
1.2 System overview . . . 5
2 Implementation of PMBus® middleware on PSOC™ Control C3 MCU . . . 7
2.1 PMBus® middleware (MW) structure . . . 9
2.2 Software API function . . . 9
2.2.1 PMBus® Target APIs . . . 9
2.2.2 PMBus® Controller APIs . . . 9
2.3 Packet error checking . . . 9
2.4 Fault report mechanism . . . 10
2.5 PMBus® commands and data formats . . . 10
2.6 SMBus protocols . . . 13
2.6.1 BUS protocols . . . 14
2.6.1.1 Quick Command . . . 14
2.6.1.2 Send Byte . . . 15
2.6.1.3 Receive Byte . . . 15
2.6.1.4 Write Byte . . . 16
2.6.1.5 Write Word . . . 16
2.6.1.6 Read Byte . . . 17
2.6.1.7 Read Word . . . 18
2.6.1.8 Write 32-bit and 64-bit . . . 18
2.6.1.9 Read 32-bit and 64-bit . . . 19
2.6.1.10 Block Write . . . 19
2.6.1.11 Block Read . . . 20
2.6.1.12 Process Call . . . 21
2.6.1.13 Block Write-Read Process Call . . . 22
2.6.1.14 SMBus Host Notify Protocol . . . 24
2.6.2 Advanced Protocols . . . 25
2.6.2.1 Extended Command Protocol . . . 25
2.6.2.2 Group Command Protocol . . . 29
2.6.2.3 Zone Protocol . . . 31
3 Getting started with PMBus® middleware . . . 38
3.1 Target Mode using PMBus® Configurator . . . 38
3.2 **Target Mode using PMBus® middleware** . . . 50
3.3 Controller Mode using PMBus® middleware . . . 51
3.4 **Add PMBus® Commands using Device Configurator** . . . 53

Table of contents

References 56
Revision history 57
Trademarks 58
Disclaimer 59

1 Introduction

This application note guides you in developing PMBus®/SMBus applications using the Infineon PMBus® middleware library v1.0 for ModusToolbox™ 3.8 and later.

SMBus is a two-wire interface commonly used to interconnect a variety of system management chips to one or more host systems. It uses I2C with some extensions as the physical layer. There is also a protocol layer, which defines classes of data and how that data is structured. Both the physical layer and protocol layer add a level of robustness not originally embodied in the I2C specification. The SMBus component supports most of the SMBus version 3.2 slave-device specifications with numerous configurable options.

Power Management Bus (PMBus®) is a standard digital power management protocol that extends SMBus with a standardized command language specifically designed for power management applications. The PMBus® component provides all possible PMBus® revision 1.4 commands and allows you to select commands that are relevant to your application.

This middleware library provides a complete and production-ready implementation for both Target Mode (device responds to commands) and Controller Mode (device initiates transactions).

1.1 Features

This section describes the complete feature set of the PMBus® middleware library v1.0.

Target Mode features:

- Built-in page and phase management for complex power systems
- Packet Error Code (PEC) support for data integrity verification
- Complete SMBus 3.2 protocol support:
  - Quick Command, Send Byte, Receive Byte
  - Write Byte/Word, Read Byte/Word
  - Process Call
  - Block Write/Read, Block Write-Read Process Call
  - SMBus Host Notify Protocol
  - Write 32/64, Read 32/64
- PMBus®-specific protocol extensions:
  - Zone Read and Zone Write for multi-device power systems
  - Extended Command Protocol (0xFE prefix)
  - Group Command Protocol for synchronized operations
- Multi-instance support for multiple PMBus® devices
- SMBus timeout detection with configurable handling
- Table-driven command architecture for easy configuration
- Event-driven callback system for application integration

Controller Mode features:

- APIs for all SMBus protocol execution:
  - Quick Command with write direction
  - Send Byte, Receive Byte
  - Write Byte/Word, Read Byte/Word
  - Process Call
  - Block Write/Read, Block Write-Read Process Call
  - SMBus Host Notify Protocol
  - Write 32/64, Read 32/64

- Generic transaction API for custom PMBus®/SMBus sequences
- Packet Error Code (PEC) generation and verification
- Multi-instance support
- Timeout detection and error handling
- Host Mode with Host Notify Protocol support

1.2 System overview

The physical layer consists of a single Controller/Host and multiple Targets. It is a two-wire communication protocol based on the SCB-I2C. The Serial Bus Data and Serial Bus Clock lines perform the I2C functions and are essential lines for the PMBus® functionality.

There are four additional optional signals: SMBALERT, Control, SMBSUS, and WP.

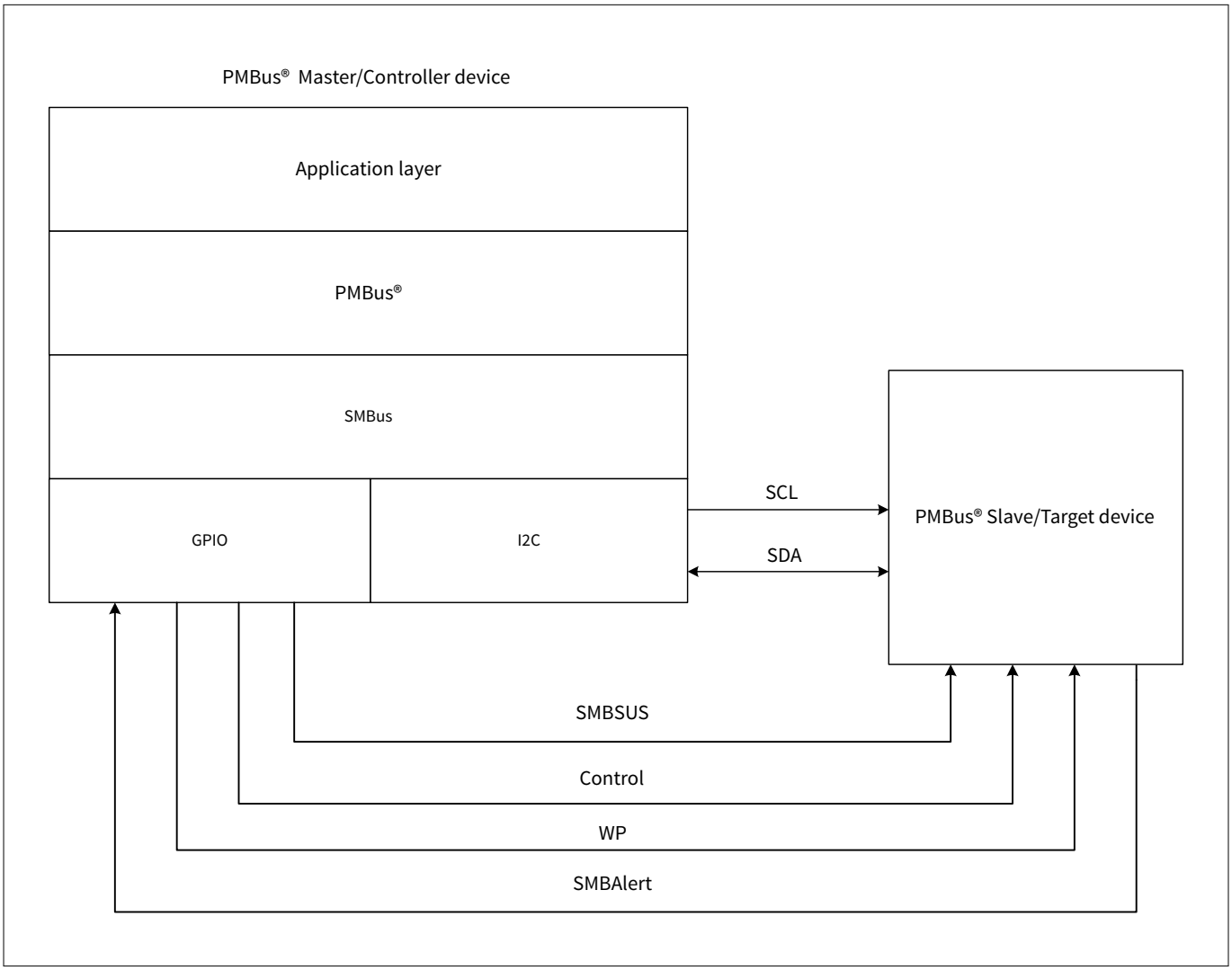

Figure 1 PMBus® communication architecture

- **SMBALERT#:** An optional signal is shared between multiple slaves and master in the SMBus/PMBus® loop. If slave needs to notify the master of any critical error observed, then a PMBus® slave can pull this line $\overline{\text{LOW}}$. The controller then processes the interrupt and simultaneously accesses all SMBALERT# devices through the Alert Response Address and the device(s) which pulled SMBALERT# $\overline{\text{LOW}}$ will acknowledge the Alert Response Address

- **SMBSUS#:** An optional signal, SMBSUS#, goes $\overline{\text{LOW}}$ when the system enters the Suspend Mode. Suspend Mode refers to a low-power mode where most devices are stalled or powered down. After resuming, the SMBSUS# returns $\overline{\text{HIGH}}$. The system then returns all devices to their operational state
- **WP:** An optional signal, WP input is present, then no updates to any internal memory is allowed when the WP input is $\overline{\text{HIGH}}$ or open. Updates are permitted only when the WP signal is $\overline{\text{LOW}}$
- **Control Signal (CONTROL):** The CONTROL signal is an input signal on a PMBus® device. It is used to turn the device on and off in conjunction with commands received via the serial bus

2 Implementation of PMBus® middleware on PSOC™ Control C3 MCU

Architecture

The PMBus® middleware implements both Target Mode (device responding to commands) and Controller Mode (device initiating transactions). The architecture is designed for modularity, portability, and minimal memory footprint.

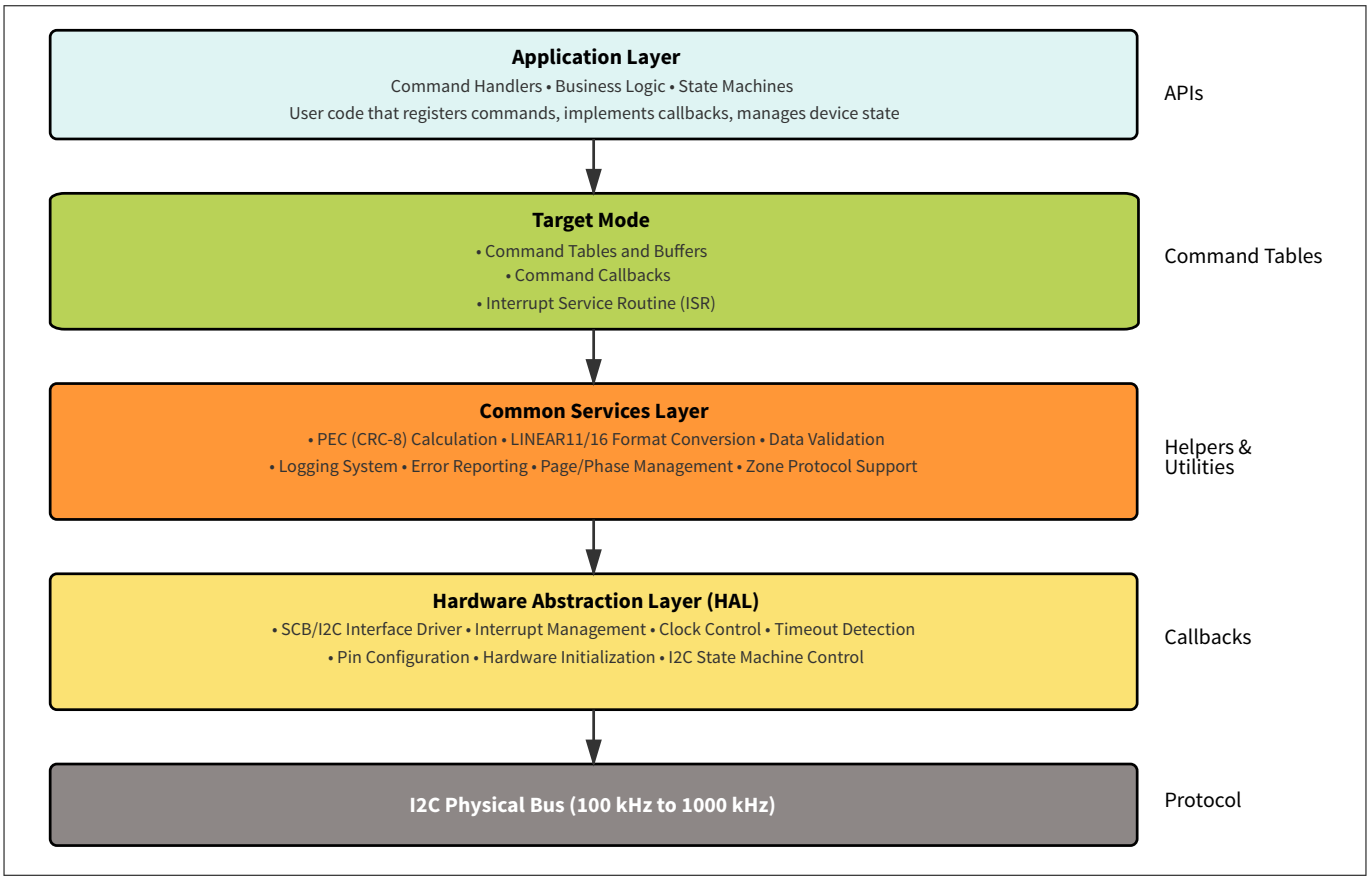

Figure 2 PMBus® middleware architecture

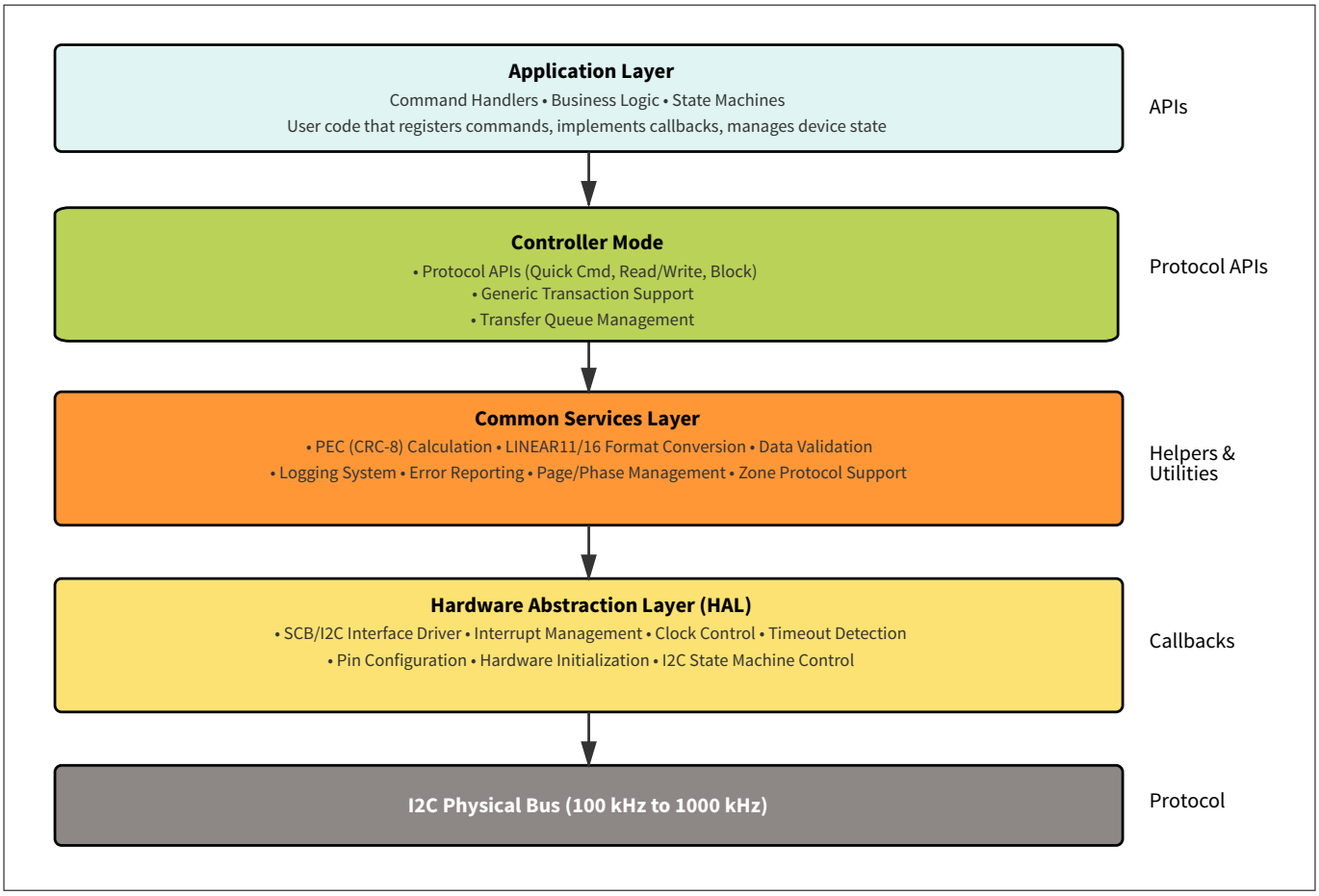

Figure 3 PMBus® middleware controller architecture

Component overview:

Application Layer – Middleware:

- Application registers commands, configures features, and implements callbacks
- Middleware provides APIs for initialization, data updates, and transaction control

Middleware – HAL:

- Hardware-independent middleware calls HAL functions via callback pointers
- HAL abstracts SCB peripheral differences across PSOC™ device families

Common Services Layer:

Shared utilities consumed by both Target Mode and Controller Mode components

- **PEC Calculation (`mtb_pmbus_pec.*`):** Computes CRC-8 (polynomial 0x07) across all transaction bytes; address, command code, and payload. Both Target and Controller call this after every byte to verify or append PEC
- **Data Format Conversion (`mtb_pmbus_helper.c`):** Provides functions, such as `mtb_pmbus_lin11_to_float()`, `mtb_pmbus_float_to_lin11()`, `mtb_pmbus_lin16_to_float()`, and `mtb_pmbus_float_to_lin16()` for translating between PMBus® wire formats (LINEAR11/16, VID, Direct Mode) and IEEE Float32. Called by application callbacks when reading or writing measurement/setpoint registers
- **Logging System (`mtb_pmbus_log.h`):** Macro-based structured logging (`MTB_PMBUS_LOG_ERR`, `MTB_PMBUS_LOG_WRN`, and `MTB_PMBUS_LOG_DBG`). Log level controlled by `MTB_PMBUS_LOG_LEVEL` at compile time; zero overhead when disabled

- **Page/Phase Management:** Internal bookkeeping for multi-page/phase command data. The pre-allocated contiguous buffer layout (page × phases × cmd_size) is managed here; both Target ISR and application APIs (`mtb_pmbus_cmd_update_data_ext()`) use the same indexed accessor logic
- **Error Reporting:** Error bitmask accumulation and forwarding to the application's `callback_events` handler. Individual protocol violations (PEC mismatch, unsupported command, byte-count overflow, invalid page) are OR'd into the error word and reported once per transaction

Interrupt Flow (Target Mode):

- I2C events trigger hardware interrupt
- Middleware executes state machine and processes commands
- Command callbacks invoked for application processing
- ACK/NACK decision returned to hardware

Polling Flow (Controller Mode):

- Application calls protocol API (e.g., `mtb_pmbus_ctrl_ex_read_word()`)
- Middleware builds transaction and programs I2C hardware
- Blocking or non-blocking based on API variant
- Completion callback notifies application

2.1 PMBus® middleware (MW) structure

PMBus® middleware file structure as follows:

- **Target Mode files:** `mtb_pmbus_trgt.*` – Device command handling, interrupt-driven operation
- **Controller Mode files:** `mtb_pmbus_ctrl.*` – Bus controller operations, all 13 SMBus protocols
- **HAL files:** `*_hal.*` – Hardware abstraction for SCB peripheral (PSC3M/P8, PSC3M/P7, PSC3M/P6 support)
- **Helper files:** `mtb_pmbus_helper.c` – Format conversions (LINEAR11/16, VID, Direct Mode)
- **PEC module:** `mtb_pmbus_pec.*` – CRC-8 error checking (0x07 polynomial)
- **Configuration:** `mtb_pmbus_conf_def.h` – Compile-time feature selection

2.2 Software API function

2.2.1 PMBus® Target APIs

For APIs that can be used for the slave implementation over PSOC™ Control C3 MCUs, see PMBus® Target APIs.

2.2.2 PMBus® Controller APIs

For APIs that can be used for the master implementation over PSOC™ Control C3 MCU, see PMBus® Controller APIs .

2.3 Packet error checking

The PEC byte is used to ensure the data reliability of communications. It is calculated using all the data bytes in the communication, including the slave address and the read/write bit. The PEC uses an 8-bit cyclic redundancy check (CRC-8). The polynomial used is $C_8 = x^8 + x^2 + x + 1$.

2.4 Fault report mechanism

When faults occur in the system, the slave handles the condition locally and also reports it to the master using one of two mechanisms:

- SMBALERT#
- Host-notify protocol

SMBALERT# requires an additional signal line connection between the master and slaves. In addition, the master is required to send a read status command with an Alert Response Address (ARA) to determine which slave changed the alert line and the type of fault that occurred.

Host-notify protocol is used to save on hardware connections, and also to ease reporting procedures.

2.5 PMBus® commands and data formats

Commands

Commandsare SMBus/PMBus® protocol operations that a controller initiates and a target device responds to. Each command consists of:

- **Code:** (0x00-0xFF); unique identifier
- **Data:** Command-specific information (e.g., voltage setpoint, current limit)
- **Size:** 1-255 bytes
- **callback:** User-defined command callback registered
- **Capabilities:** Flags indicating Read-only, Write-only, Read/Write (Process Call), page/phase support, block protocol, and Numeric format (LINEAR11/16, Direct, etc.)

Command Table

The Command Table is a configuration array that defines all supported commands for a target device. It maps command codes to their properties and data buffers.

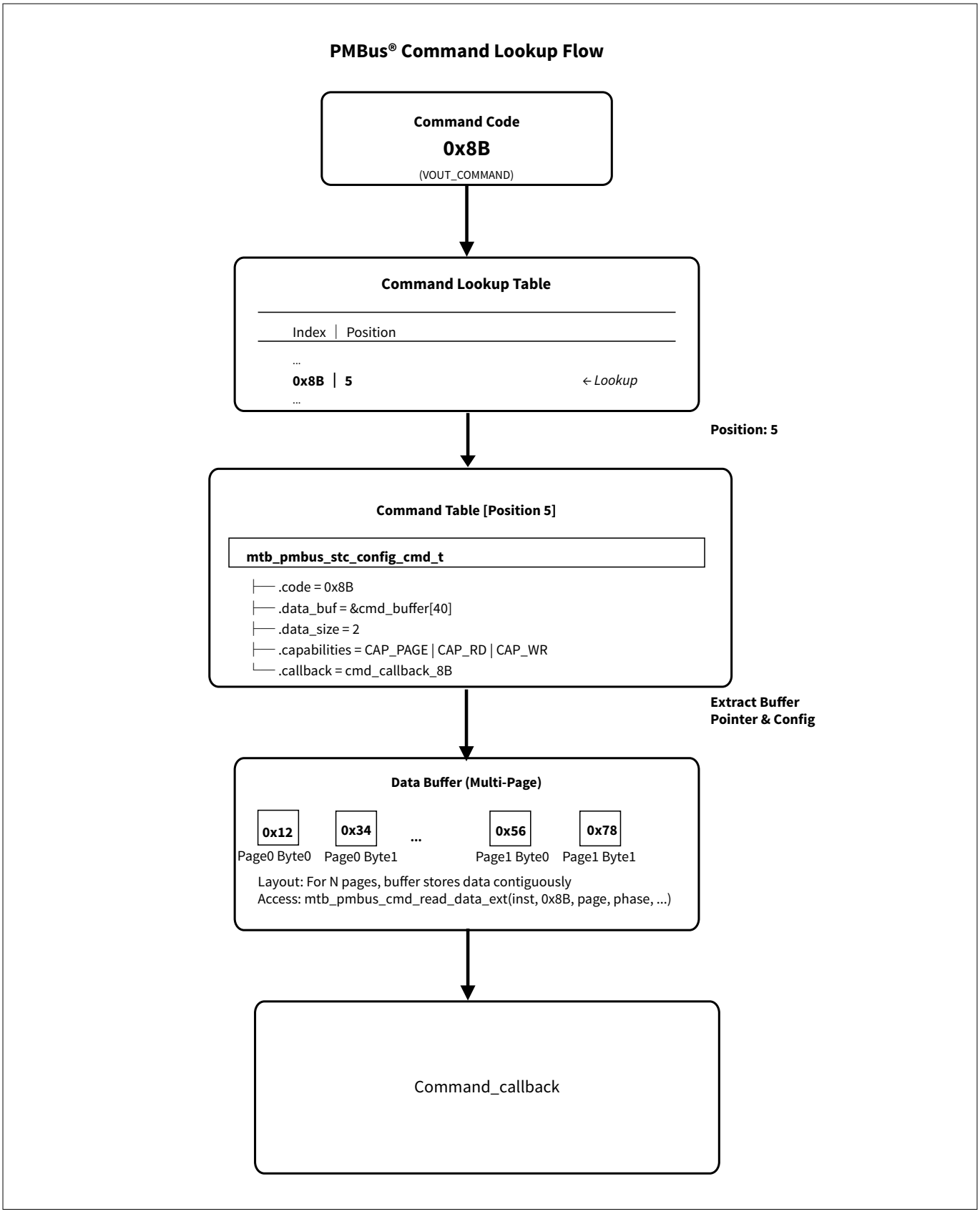

Figure 4 Command flow

This design provides O(1) resolution with minimal overhead while supporting multi-page/phase variants where page/phase indices calculate offsets into the buffer.

Pre-implemented commands

The PMBus® middleware provides a PMBus® Configurator tool that integrates with the Device Configurator. It allows you to configure:

- **Compile-time options** for all PMBus® instances
- **Runtime settings** for each PMBus® instance
- **Command tables** for Target Mode

Pre-implemented commands in PMBus® Configurator

Pre-implemented commands are PMBus® commands that the middleware handles automatically without requiring user code. When you select **Implemented** for a command, the middleware provides a built-in response and it works as follows:

Table 1 Selecting Pre-implemented commands

Command state	Behavior
Enable ✓ + **Implemented** ✓	Middleware handles the command automatically (no custom code needed)
Enable ✓ + **Implemented** ☐	You must provide buffer, size, and callback (customizable)
Enable ☐	Command is disabled and ignored

Table 2 Pre-implemented commands

Command	Command code	Short command description
PAGE	00h	Selects the active page
PHASE	04h	Selects the active phase
CAPABILITY	19h	Provides a way for a controller to determine some key capabilities of a PMBus® device
QUERY	1Ah	Asks a PMBus® device if it supports a given command, and if so, what data formats it supports for that command
PAGE_PLUS_WRITE	05h	Sends a command and its associated data to either a specific PAGE or all PAGES (PAGE = FFh)
PAGE_PLUS_READ	06h	Reads data associated with a command from a specific PAGE
ZONE_CONFIG	07h	Assigns the Zone group to target
ZONE_ACTIVE	08h	Controller to set the Active Write Zone for ZONE_WRITE operations and the Active Read Zone for ZONE_READ operations
P2_PLUS_WRITE	09h	Sends a command and its associated data to: • A specific PAGE and PHASE • All phases (PHASE = FFh) in specific page • A specific phase in all pages (PAGE = FFh) • All phases in all pages (PAGE = FFh and PHASE = FFh)

(table continues...)

Table 2 (continued) Pre-implemented commands

Command	Command code	Short command description
P2_PLUS_READ	0Ah	Reads the data associated with a command from: • A specific PAGE and PHASE • All phases (PHASE = FFh) in specific page • A specific phase in all pages (PAGE = FFh) • All phases in all pages (PAGE = FFh and PHASE = FFh)
PMBUS_REVISION	98h	Stores or reads the revision of the PMBus® to which the device is compliant

Data format support:

- Built-in conversion APIs between PMBus® formats and IEEE Float32
- LINEAR11 format (5-bit signed exponent, 11-bit mantissa)
- LINEAR16 format (external mode with separate exponent)
- VID Mode, Direct Mode, IEEE Half Precision
- **LINEAR:** Receive and transmit values, such as volts, amperes, milliseconds, or degree Celsius. This format provides the least burden on the master at the expense of data manipulation in the PMBus® device
- **DIRECT:** Receive and transmit data as a 2-byte, two's-complement binary integer. This format provides the load on the PMBus® device at the expense of more complex calculations in the master

2.6 SMBus protocols

This section provides a description of the various SMBus protocols with and without a Packet Error Code. Compliant devices do not support all the protocols defined in this section. The results returned by such a device to a protocol it does not support are undefined. Table 3 shows the symbols used to create the protocol formats in this section. Not all protocol elements will be present in every command. For instance, not all packets are required to include the Packet Error Code.

Table 3 SMBus protocols

Symbol	Description
[S]	**START condition:** Controller initiates bus transaction by pulling SDA $\overline{\text{LOW}}$ while SCL is $\overline{\text{HIGH}}$
[P]	**STOP condition:** Controller terminates transaction by releasing SDA $\overline{\text{HIGH}}$ while SCL is $\overline{\text{HIGH}}$
[Sr]	Repeated START - START without preceding STOP (used for direction change in read operations)
[A]	**ACKnowledge (ACK):** Receiver pulls SDA $\overline{\text{LOW}}$ during 9th clock cycle to confirm byte received
[N]	**Not Acknowledge (NACK):** Receiver leaves SDA high during 9th clock; signals end of read or error
[Addr+W]	7-bit target address with R/$\bar{W}$ bit = 0 (Write direction)
[Addr+R]	7-bit target address with R/$\bar{W}$ bit = 1 (Read direction)
[CmdCode]	PMBus® command code byte (0x00-0xFF)

(table continues...)

Table 3 (continued) SMBus protocols

Symbol	Description
[DataByte]	Single data byte
[DataLow]	Low byte of 16-bit word (sent first per SMBus LSB-first convention)
[DataHigh]	High byte of 16-bit word
[ByteCount]	Block protocol byte count (1-255, specifies the number of data bytes following)
[PEC]	**Packet Error Code:** CRC-8 checksum over all transaction bytes (optional)
[B0-B7]	Individual bytes in multi-byte transfers (32/64-bit)

Direction arrows:

- **Controller:** Indicates bytes sent by the controller (master)
- **Target:** Indicates bytes sent by the target (slave)

2.6.1 BUS protocols

The Target-side protocol code snippets provided in the following sections assume that you are familiar with configuring commands using the PMBus® Configurator Command tab. For details, see **Add PMBus® Commands using Device Configurator** section.

2.6.1.1 Quick Command

In the Quick Command, the R/W# bit of the target address denotes the command. The R/W# bit can be used to turn a device function on or off, or enable/disable a low-power standby mode. There is no data sent or received.

Use case: On/off control, LED toggle, or presence detection.

Format: Only address + R/W bit, no command code, or data.

```
Controller -> [S] [Addr+W] [A] [P] (Write direction)
Controller -> [S] [Addr+R] [A] [P] (Read direction)
```

Toggle LED

```
/* Target mode - Detect Quick Command */
void gen_callback(mtb_pmbus_events_t event)
{
if (event == MTB_PMBUS_QUICK_CMD_WR_EVENT) {
toggle_led();
}
}
/* Controller mode - Send Quick Command */
mtb_pmbus_ctrl_ex_quick_cmd(&ctrl_inst, 0x18, true); // Write direction
```

Limitation (Controller Mode)

The controller cannot execute the Quick Command protocol with Read direction due to hardware limitations of the SCB I2C block. Per SMBus/PMBus® specification, the Quick Command with Read direction consists of

sending the target address with the Read bit set, followed immediately by a STOP condition after receiving the ACK bit from the target device, without reading any data bytes.

2.6.1.2 Send Byte

A simple device can recognize its own target address and accept up to 256 possible encoded commands in the form of a byte that follows the target address.

Use case: Simple command without additional data (reset, start conversion).

Format: Address + Command code only.

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [P]
```

Trigger conversion

```
/* Target mode - Handle Send Byte */
bool cmd_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_MATCH) {
        if (cmd_code == CMD_START_CONVERSION) {
            adc_start_conversion();
        }
    }
    return true;
}

/* Controller mode - Send command */
mtb_pmbus_ctrl_ex_send_byte(&ctrl_inst, 0x18, CMD_START_CONVERSION,  true);
if (status == MTB_PMBUS_CTRL_STATUS_SUCCESS)
{
     /* Wait for transfer completion (1 second timeout) */
     status = mtb_pmbus_ctrl_wait_cmpl(&qsg_ctrl_inst, PMBUS_CTRL_TIMEOUT_US);
     if (status == MTB_PMBUS_CTRL_STATUS_IS_READY)
     {
         printf(" Command sent successfully\n\r");
     }
     else
    {
         printf("Command failed with status: %d\n\r", status);
    }
}
else
{
    printf("Command API failed with status: %d\n\r", status);
}
```

2.6.1.3 Receive Byte

The Receive Byte is similar to a Send Byte, the only difference being the direction of data transfer.

Use case: Read a single status byte without specifying a command.

Format: Address + Read, target responds with 1 byte.

```
Controller -> [S] [Addr+R] [A]
Target     -> [DataByte] [A/N] [P]
```

Quick status check

```
/* Target mode - Prepare receive byte */
void gen_callback(mtb_pmbus_events_t event) {
    if (event == MTB_PMBUS_RECEIVE_BYTE_EVENT) {
        uint8_t status = get_device_status();
        mtb_pmbus_set_receive_byte(&pmbus_inst, status);
    }
}

/* Controller mode - Receive byte */
uint8_t status;
mtb_pmbus_ctrl_ex_received_byte(&ctrl_inst, 0x18, &status, true);
```

2.6.1.4 Write Byte

Use case: Write a single configuration byte.

Format: Command code + 1 data byte.

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [DataByte] [A] [P]
```

Operation mode

```
/* Target mode - Process write byte */
bool cmd_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_WRITE_DONE) {
        uint8_t mode;
        mtb_pmbus_cmd_read_data_isr(&pmbus_inst, CMD_OPERATION, &mode, 1);
        set_operation_mode(mode);
    }
    return true;
}

/* Controller mode - Write byte */
uint8_t operation_mode = 0x80;  // Turn on
mtb_pmbus_ctrl_ex_write_byte(&ctrl_inst, 0x18, CMD_OPERATION, operation_mode, true);
```

2.6.1.5 Write Word

Use case: Write a 16-bit value (voltage set point, current limit).

Format: Command code + 2 data bytes (LSB first).

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [DataLow] [A] [DataHigh] [A] [P]
```

Set voltage

```
/* Target mode - Process write word */
bool vout_cmd_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte)
{
    if (event == MTB_PMBUS_CMD_WRITE_DONE) {
        uint16_t vout_raw;
        mtb_pmbus_cmd_read_data_isr(&pmbus_inst, CMD_VOUT_COMMAND,
                                    (uint8_t*)&vout_raw, 2);

        float voltage = mtb_pmbus_lin16_to_float(vout_raw, -10);
        set_voltage_regulator(voltage);
    }
    return true;
}

/* Controller mode - Write word */
uint16_t vout = mtb_pmbus_float_to_lin16(3.3f, -10);  // 3.3V
mtb_pmbus_ctrl_ex_write_word(&ctrl_inst, 0x18, CMD_VOUT_COMMAND, vout);
```

2.6.1.6 Read Byte

Use case: Read a single status/configuration byte.

Format: Command code, then read 1 byte.

```
controller -> [S] [Addr+W] [A] [CmdCode] [A] [Sr] [Addr+R] [A]
Target     -> [DataByte] [N] [P]
```

Read status

```
/* Target mode - Prepare read byte */
bool status_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_READ_REQ) {
        uint8_t status = calculate_status_byte();
        mtb_pmbus_cmd_update_data_isr(&pmbus_inst, CMD_STATUS_BYTE, &status, 1);
    }
    return true;
}

/* Controller mode - Read byte */
uint8_t status_byte;
mtb_pmbus_ctrl_ex_read_byte(&ctrl_inst, 0x18, CMD_STATUS_BYTE, &status_byte, true);
```

2.6.1.7 Read Word

Use case: Read 16-bit measurement (voltage, current, and temperature).

Format: Command code, then read 2 bytes (LSB first).

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [Sr] [Addr+R] [A]
Target -> [DataLow] [A] [DataHigh] [N] [P]
```

Read voltage

```
/* Target mode - Prepare voltage reading */
bool vout_read_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t
byte) {
    if (event == MTB_PMBUS_CMD_READ_REQ) {
        float measured_voltage = adc_read_voltage();
        uint16_t vout = mtb_pmbus_float_to_lin16(measured_voltage, -10);
        mtb_pmbus_cmd_update_data_isr(&pmbus_inst, CMD_READ_VOUT,
                                      (uint8_t*)&vout, 2);
    }
    return true;
}

/* Controller mode - Read word */
uint16_t vout_raw;
mtb_pmbus_ctrl_ex_read_word(&ctrl_inst, 0x18, CMD_READ_VOUT, &vout_raw, true);
float voltage = mtb_pmbus_lin16_to_float(vout_raw, -10);
printf("Voltage: %.3f V\n", voltage);
```

2.6.1.8 Write 32-bit and 64-bit

Use case: Extended precision data (high-resolution measurements, and timestamps).

Format: Command code + 4 or 8 data bytes (LSB first).

```
/* Write 32 */
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [B0] [A] [B1] [A] [B2] [A] [B3] [A] [P]

/* Write 64 */
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [B0-B7] [A] ... [P]
```

Write timestamp

```
/* Callback implemenation in the target mode */
/* Target mode - Prepare voltage reading */
bool vout_read_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t
byte) {
    if (event == MTB_PMBUS_CMD_READ_REQ) {
        float measured_voltage = adc_read_voltage();
        uint16_t vout = mtb_pmbus_float_to_lin16(measured_voltage, -10);
        mtb_pmbus_cmd_update_data_isr(&pmbus_inst, CMD_READ_VOUT,
                                      (uint8_t*)&vout, 2);
    }
    return true;
}

/* Controller mode - Write 32 */
uint32_t timestamp = get_unix_time();
mtb_pmbus_ctrl_ex_write_32(&ctrl_inst, 0x18, CMD_MFR_TIMESTAMP, timestamp);
```

2.6.1.9 Read 32-bit and 64-bit

Use case: Read extended data (energy counters, serial numbers).

Read energy counter

```
/* Target mode - Prepare 64-bit counter */
bool energy_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_READ_REQ) {
        uint64_t energy_wh = get_total_energy_wh();
        mtb_pmbus_cmd_update_data_isr(&pmbus_inst, CMD_MFR_ENERGY,
                                      (uint8_t*)&energy_wh, 8, true);
    }
    return true;
}

/* Controller mode - Read 64 */
uint64_t energy;
mtb_pmbus_ctrl_ex_read_64(&ctrl_inst, 0x18, CMD_MFR_ENERGY, &energy, true);
printf("Total Energy: %llu Wh\n", energy);
```

2.6.1.10 Block Write

Use case: Write variable-length data (firmware updates, configuration blocks, and strings).

Format: Command code + byte count + N data bytes (1-255).

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [ByteCount] [A] [Data0] ... [DataN-1] [A] [P]
```

Write device name

```
/* Target mode - Store device name */
static uint8_t device_name_buf[33];  // 1 count + 32 chars maximum
bool name_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_WRITE_DONE) {
        uint8_t count = device_name_buf[0];
        char name[33];
        memcpy(name, &device_name_buf[1], count);
        name[count] = '\0';
        save_device_name(name);
        printf("Device name set: %s\n", name);
    }
    return true;
}

static mtb_pmbus_stc_config_cmd_t cmd_table[] = {
    {
        .cmd_code  = CMD_MFR_ID,
        .cmd_cap   = MTB_PMBUS_CMD_CAP_DIR_WR | MTB_PMBUS_CMD_CAP_BLOCK,
        .data_buf  = device_name_buf,
        .callback  = name_callback,
        .data_size = 32,  // Max data size (excluding count byte)
    }
};

/* Controller mode - Write block */
char device_name[] = "PowerSupply-01";
uint8_t name_len = strlen(device_name);
mtb_pmbus_ctrl_ex_block_write(&ctrl_inst, 0x18, CMD_MFR_ID,
                              (uint8_t*)device_name, name_len, true);
```

2.6.1.11 Block Read

Use case: Read variable-length data (serial numbers, model names, and coefficient arrays).

Format: Command code, then read byte count + N data bytes.

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [Sr] [Addr+R] [A]
Target -> [ByteCount] [A] [Data0] [A] ... [DataN-1] [N] [P]
```

Read model number

```
/* Target mode - Prepare model string */
static uint8_t model_buf[17];  // 1 count + 16 chars

bool model_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_READ_REQ) {
        const char* model = "PSC3M8-EVAL";
        uint8_t len = strlen(model);
        model_buf[0] = len;  // Byte count
        memcpy(&model_buf[1], model, len);
        // Middleware automatically sends count + data
    }
    return true;
}

/* Controller mode - Read block */
uint8_t model_data[32];
uint8_t model_len;
mtb_pmbus_ctrl_ex_block_read(&ctrl_inst, 0x18, CMD_MFR_MODEL,
                             model_data, &model_len, true);
model_data[model_len] = '\0';  // Null terminate
printf("Model: %s\n", (char*)model_data);
```

2.6.1.12 Process Call

Sends data and waits for the target to return a value dependent on that data. The protocol is simply a Write Word followed by a Read Word without the Read-Word command field and the Write-Word STOP bit.

Use case: Write 2 bytes, read 2 bytes in one transaction (calculate, convert).

Format: Write 2 bytes, then immediately read 2 bytes (no STOP between).

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [DataLow] [A] [DataHigh] [A] [Sr] [Addr+R] [A]
Target -> [ResultLow] [A] [ResultHigh] [N] [P]
```

Temperature conversion (°C to °F)

```
/* Target mode - Process call */
static uint8_t temp_convert_buf[2];

bool temp_convert_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t
byte)
{
    if (event == MTB_PMBUS_CMD_WRITE_DONE)
    {
        // Input received - convert °C to °F
        uint16_t temp_c_raw;
        mtb_pmbus_cmd_read_data_isr(&pmbus_inst, CMD_TEMP_CONVERT,
                                    (uint8_t*)&temp_c_raw, 2);

        float temp_c = mtb_pmbus_lin11_to_float(temp_c_raw);
        float temp_f = (temp_c * 9.0f/5.0f) + 32.0f;
        uint16_t temp_f_raw = mtb_pmbus_float_to_lin11(temp_f);

        // Update buffer for read phase
        mtb_pmbus_cmd_update_data_isr(&pmbus_inst, CMD_TEMP_CONVERT,
                                      (uint8_t*)&temp_f_raw, 2);
    }
    return true;
}

/* Command configuration for Process call */
Note: PMBUS configurator will generate this table.
static mtb_pmbus_stc_config_cmd_t cmd_table[] = {
    {
        .cmd_code  = CMD_TEMP_CONVERT,
        .cmd_cap   = MTB_PMBUS_CMD_CAP_DIR_PROCESS_CALL |
                     MTB_PMBUS_CMD_CAP_FORMAT_LIN_11_16,
        .data_buf  = temp_convert_buf,
        .callback  = temp_convert_callback,
        .data_size = 2,
    }
};

/* Controller mode - Process call */
uint16_t temp_c = mtb_pmbus_float_to_lin11(25.0f);  // 25°C
uint16_t temp_f;
mtb_pmbus_ctrl_ex_process_call (&ctrl_inst, 0x18, CMD_TEMP_CONVERT,
                                temp_c, &temp_f, true);
float fahrenheit = mtb_pmbus_lin11_to_float(temp_f);
printf("Temperature: %.1f°F\n", fahrenheit);
```

2.6.1.13 Block Write-Read Process Call

Use case: Complex calculations with variable input/output (coefficient tables, FFT).

Format: Write count + N bytes, read count + M bytes (no STOP between).

***Note**: Ensure that the combined data payload must not exceed 255 bytes.*

The byte length restrictions of this Process Call are as follows:

- M ≥ 0 byte
- N ≥ 0 byte
- M + N ≤ 255 bytes

The Read Byte count does not include the PEC byte.

```
Controller -> [S] [Addr+W] [A] [CmdCode] [A] [WrCount] [A] [WrData...] [A] [Sr] [Addr+R] [A]
Target -> [RdCount] [A] [RdData...] [N] [P]
```

Polynomial evaluation

```
/* Target mode - Block process call */
static uint8_t poly_buf[64];  // Input: count + coefficients, Output: count + result

bool poly_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_WRITE_DONE) {
        uint8_t wr_count = poly_buf[0];
        // Input format: [count] [x] [a0] [a1] [a2]... -> calculate polynomial
        float x, result = 0;
        memcpy(&x, &poly_buf[1], sizeof(float));


        for (int i = 0; i < (wr_count - sizeof(float)) / sizeof(float); i++) {
            float coeff;
            memcpy(&coeff, &poly_buf[1 + sizeof(float) + i*sizeof(float)], sizeof(float));
            result += coeff * pow(x, i);
        }

        // Prepare response
        poly_buf[0] = sizeof(float);  // Read count
        memcpy(&poly_buf[1], &result, sizeof(float));
    }
    return true;
}

/* Controller mode - Block process call */
float input_data[] = {2.0f, 1.0f, 2.0f, 3.0f};  // x=2, poly: 1 + 2x + 3x²
uint8_t wr_count = sizeof(input_data);

uint8_t result_data[32];
uint8_t rd_count;
mtb_pmbus_ctrl_ex_block_wr_rd_proc_call(&ctrl_inst, 0x18, CMD_POLY_CALC,
                                        (uint8_t*)input_data, wr_count,
                                        result_data, &rd_count, true);

float result;
memcpy(&result, result_data, sizeof(float));
printf("Polynomial result: %.2f\n", result);  // Should be 1 + 4 + 12 = 17
```

2.6.1.14 SMBus Host Notify Protocol

Use case: Target-initiated interrupt to alert controller.

Format: Target acts as controller, writes its address to Host Notify address (0x08).

```
Target -> [S] [0x08+W] [A] [TargetAddr<<1] [A] [DataLow] [A] [DataHigh] [A] [P]
```

Implementation

```
/* Target mode - Initiate Host Notify */
if (critical_fault_detected()) {
    // Target becomes temporary controller
    uint16_t fault_code = get_fault_code();
    mtb_pmbus_host_notify(&pmbus_inst, fault_code);
}

/* Controller mode - Must implement polling or use GPIO interrupt */
// Controller does not receive Host Notify directly - target initiates
// Controller should monitor SMBALERT# or poll STATUS_WORD regularly

void pmbus_ctrl_hnp_callback(mtb_pmbus_ctrl_host_notify_events_t event)
{
    if (event == MTB_PMBUS_CTRL_HOST_NOTIFY_COMPLETE)
    {
        MTB_PMBUS_LOG_DBG("HOST NOTIFY COMPLETE: Addr: 0x%02X Data: 0x%04X",
                          mtb_pmbus_ctrl_hnp_get_trgt_addr(&pmbus_ctrl_inst),
                          mtb_pmbus_ctrl_hnp_get_trgt_data(&pmbus_ctrl_inst));
    }
    else if (event == MTB_PMBUS_CTRL_HOST_NOTIFY_REQUEST_RECEIVED)
    {
        MTB_PMBUS_LOG_DBG("HOST NOTIFY REQUEST RECEIVED event");
    }
    else if (event == MTB_PMBUS_CTRL_HOST_NOTIFY_ERROR)
    {
        MTB_PMBUS_LOG_ERR("HOST NOTIFY ERROR event");
    }
    else
    {
        MTB_PMBUS_LOG_ERR("Unknown HNP event: %d", event);
    }
}
/* Controller configuration structure */
static mtb_pmbus_stc_config_hw_t myPMBus_0_hw_config =
{
    .hal_config = &myPMBus_0_hal_config,
    .hw_resource_ctrl_callback = myPMBus_0_hw_resource_ctrl,
    .callback_hnp = pmbus_ctrl_hnp_callback;
    .enable_hw_irq_callback = myPMBus_0_hw_irq_enable,
    .disable_hw_irq_callback = myPMBus_0_hw_irq_disable,
};
```

2.6.2 Advanced Protocols

This section provides the protocols used by the PMBus® middleware.

2.6.2.1 Extended Command Protocol

The Extended Command Protocol extends PMBus® capabilities with manufacturer-specific or proprietary commands. All extended commands use the **0xFE** command code as a prefix, followed by the actual extended command code.

Use case: Access manufacturer-specific commands beyond PMBus® specification (firmware updates, calibration, proprietary features).

Format:

```
Standard Extended Command (Write):
Controller -> [S] [Addr+W] [A] [0xFE] [A] [ExtCmdCode] [A] [DataBytes] [A] [PEC] [P]

Standard Extended Command (Read):
Controller -> [S] [Addr+W] [A] [0xFE] [A] [ExtCmdCode] [A] [Sr] [Addr+R] [A]
Target -> [DataBytes...] [A] [PEC] [P]

Variable Length Extended (Block):
Controller -> [S] [Addr+W] [A] [0xFE] [A] [ExtCmdCode] [A] [ByteCount] [A] [Data...] [A] [PEC]
[P]
```

Implementation requirements:

Compile-time Enable: Extended commands must be enabled at compile time:

```
// In mtb_pmbus_conf.h or compile flags:
#define MTB_PMBUS_SUPPORT_EXT_CMD  1U
#define MTB_PMBUS_SUPPORT_PMBUS    1U  // Extended commands require PMBus mode (not pure SMBus)
```

Extended Command Table: Define supported extended commands:

```
static const mtb_pmbus_stc_config_cmd_t ext_cmd_table[] = {
    {
        .cmd_code  = 0x01,  // Extended command code (without 0xFE prefix)
        .cmd_cap   = MTB_PMBUS_CMD_CAP_DIR_RD | MTB_PMBUS_CMD_CAP_DIR_WR,
        .data_buf  = custom_data_buf,
        .callback  = custom_cmd_callback,
        .data_size = 32,
    },
    // Additional extended commands...
};

// Register table during initialization:
mtb_pmbus_stc_config_t config = {
    .enable_pmbus = true,
    .ext_cmd_table = ext_cmd_table,
    .ext_cmd_table_len = sizeof(ext_cmd_table) / sizeof(ext_cmd_table[0]),
    // ... other config
};
```

Firmware Update Extended Command

```
/* Target mode - Handle firmware update command */
static uint8_t fw_buffer[256];
static uint16_t fw_offset = 0;

bool fw_update_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t
byte)
{
    if (event == MTB_PMBUS_CMD_WRITE_DONE) {
        uint8_t count;
        mtb_pmbus_cmd_read_data_isr(&pmbus_inst, 0x01, &count, 1);

        // Store firmware chunk
        mtb_pmbus_cmd_read_data_isr(&pmbus_inst, 0x01, &fw_buffer[fw_offset], count);
        fw_offset += count;

        if (fw_offset >= FW_SIZE) {
            validate_and_program_firmware(fw_buffer);
            fw_offset = 0;
        }
    }
    return true;
}

// Extended command table entry:
{
    .cmd_code  = 0x01,  // FW_UPDATE_CHUNK
    .cmd_cap   = MTB_PMBUS_CMD_CAP_DIR_WR | MTB_PMBUS_CMD_CAP_BLOCK,
    .data_buf  = fw_buffer,
    .callback  = fw_update_callback,
    .data_size = 256,
}

/* Controller mode - Send firmware update */
uint8_t fw_chunk[] = {32, 0xAA, 0xBB, 0xCC, ...};  // count + 32 bytes data
mtb_pmbus_ctrl_ex_ext_cmd_block_write(&ctrl_inst, 0x18, 0x01, fw_chunk, 33);
```

The PMBus® Configurator allows for the enablement of the Extended Command Protocol at compile time, which means instead of manually configuring or enabling extended commands through steps.

Enable Extended Commands

PMBus® Configurator can be used to enable the Extended Command protocol in the **General** tab of the configurator.

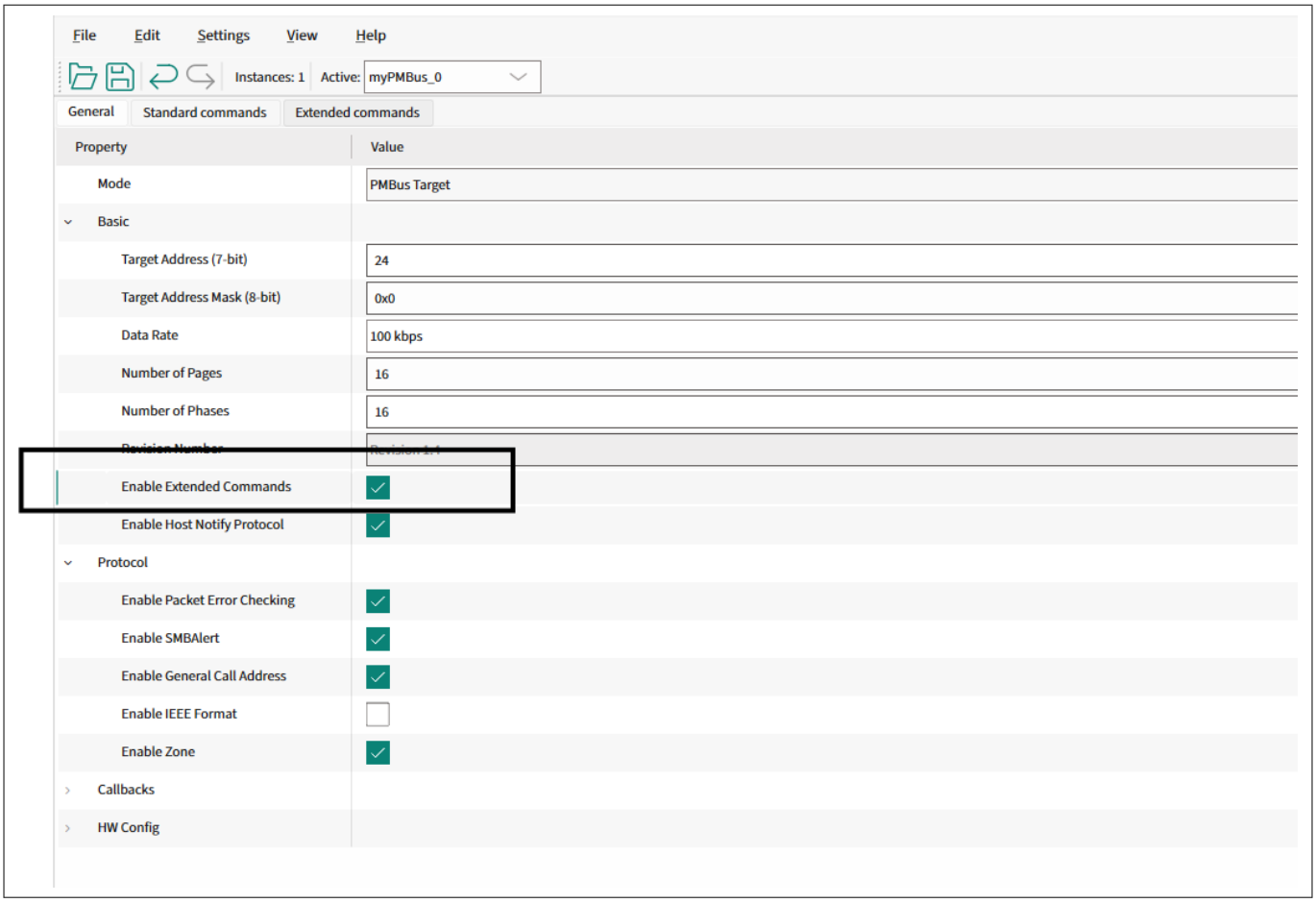

Figure 5 Enable Extended Commands

Extended command table

The PMBus® Configurator has a dedicated tab for extended commands. You can add extended commands and their configurations. Once these commands are added, the configurator automatically updates the PMBus® configuration structure to include the extended command table.

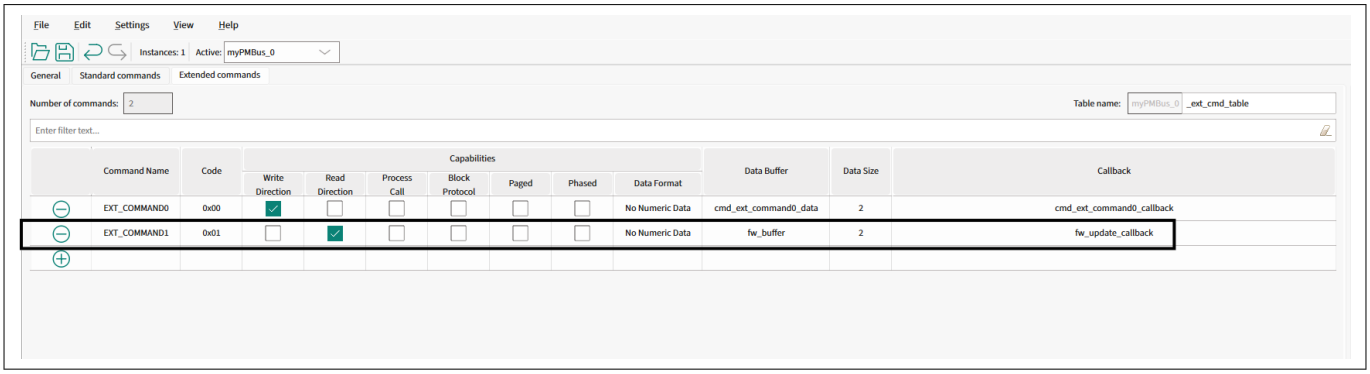

Figure 6 Extended command table in configurator

***Note**: The PMBus® Configurator simplifies the process by automatically generating the command table, registering it to the config structure, and enabling compile-time options. This eliminates the need for manual C code configuration, making the process more efficient and user-friendly.*

2.6.2.2 Group Command Protocol

Enables atomic updates across multiple PMBus® devices on the same bus without requiring individual write operations to each device. All devices matching a group address execute the command simultaneously when they receive the broadcast STOP condition, ensuring synchronized state changes critical for power sequencing.

Use case: Synchronize state changes across multiple devices in a multi-device power system (voltage rail transitions, enable/disable sequences).

Format:

```
Group Write to Multiple Devices:
 | First device transfer || Second device transfer || Third device transfer |
 [S] [ADDR1+W] [A] [0x00] [A] [Page0] [A] [Sr] [ADDR2+W] [A] [0x00] [A] [Page1] [A] [Sr]
[ADDR3+W] [A] [0x00] [A] [Page2] [A] [P]

Example Timeline:
1. Controller sends data frame to each slave address
2. All targets receive the data and buffer command code + data
3. Controller generates STOP condition
4. All targets SIMULTANEOUSLY execute buffered command
5. Updates take effect synchronously across power system
```

Page set sequence

```
void controller_group_command_page_example(void)
{
    mtb_pmbus_ctrl_status_t status;
    uint8_t page_cmd = 0x00U;  /* PAGE command */

    /* Prepare data: [PAGE_CMD, page_number] for each device */
    uint8_t dev1_data[2] = {page_cmd, 0x00U};  /* Device 1: page 0 */
    uint8_t dev2_data[2] = {page_cmd, 0x01U};  /* Device 2: page 1 */
    uint8_t dev3_data[2] = {page_cmd, 0x02U};  /* Device 3: page 2 */

    /* Device 1: Send PAGE command without STOP */
    mtb_pmbus_ctrl_stc_transfer_cfg_t cfg1 =
    {
        .data = dev1_data,
        .addr = 0x18U,
        .wr_size = 2U,
        .rd_size = 0U,
        .execute_stop = false
    };

    status = mtb_pmbus_ctrl_execute_transfer(&pmbus_inst, &cfg1);
    if (status == MTB_PMBUS_CTRL_STATUS_SUCCESS)
    {
        status = mtb_pmbus_ctrl_wait_cmpl(&pmbus_inst, TEST_TIMEOUT_30_MS);
    }

    /* Device 2: Send PAGE command without STOP */
    if (status == MTB_PMBUS_CTRL_STATUS_IS_READY)
    {
        mtb_pmbus_ctrl_stc_transfer_cfg_t cfg2 =
        {
            .data = dev2_data,
            .addr = 0x19U,
            .wr_size = 2U,
            .rd_size = 0U,
            .execute_stop = false
        };

        status = mtb_pmbus_ctrl_execute_transfer(&pmbus_inst, &cfg2);
        if (status == MTB_PMBUS_CTRL_STATUS_SUCCESS)
        {
            status = mtb_pmbus_ctrl_wait_cmpl(&pmbus_inst, TEST_TIMEOUT_30_MS);
        }
    }

    /* Device 3: Send PAGE command WITH STOP (final) */
    if (status == MTB_PMBUS_CTRL_STATUS_IS_READY)
    {
        mtb_pmbus_ctrl_stc_transfer_cfg_t cfg3 =
        {
            .data = dev3_data,
```

```
            .addr = 0x1AU,
            .wr_size = 2U,
            .rd_size = 0U,
            .execute_stop = true
        };

        status = mtb_pmbus_ctrl_execute_transfer(&pmbus_inst, &cfg3);
        if (status == MTB_PMBUS_CTRL_STATUS_SUCCESS)
        {
            status = mtb_pmbus_ctrl_wait_cmpl(&pmbus_inst, TEST_TIMEOUT_30_MS);
        }

        if (status == MTB_PMBUS_CTRL_STATUS_IS_READY)
        {
            /* Group Command completed - all devices have different pages set */
            printf("Group PAGE command completed: Dev1=Page0, Dev2=Page1, Dev3=Page2\n\r");
        }
    }
}
```

2.6.2.3 Zone Protocol

The Zone Protocol extends PMBus® with two specialized communication modes designed for multi-device power systems. Unlike the standard SMBus protocols that address individual devices,

Zone protocols use reserved addresses to enable efficient group communication:

- **Zone Write (Address 0x37):** Broadcast commands to all devices in an active write zone
- **Zone Read (Address 0x28):** Query status from multiple devices with bus arbitration

Table 4 Zone numbering

Zone number	Use case
0x00	Typically unassigned (no operation)
0x01-0xFD	Available for custom grouping
0xFE	Default/fallback zone (all pages initially assigned)
0xFF	Zone ALL - special matching (matches any non-zero zone)

Zone Write Protocol

Use case: Broadcast command to multiple devices simultaneously for synchronized state changes (voltage adjustment, enable/disable).

Format:

```
Zone Write (Broadcast):
Controller -> [S] [0x37+W] [A] [CmdCode] [A] [DataBytes...] [A] [PEC] [P]

All targets in matching write zone execute on STOP condition.
No ACK from individual devices - command is broadcast.

Example - Zone Write Word:
Controller -> [S] [0x6E] [A] [0x21] [A] [0x00] [A] [0x84] [A] [PEC] [P]
                   |          |          |         |
                0x37+W   |   VOUT_CMD  DataLow   DataHigh
                       ACK (bus pull-down, not device ACK)
```

Implementation requirements:

Compile-time enable

```
// In mtb_pmbus_conf.h or compile flags:
#define MTB_PMBUS_SUPPORT_PMBUS       1U  // Zone protocols require PMBus mode
#define MTB_PMBUS_SUPPORT_ZONE        1U  // Enable zone read/write
#define MTB_PMBUS_IMPL_CMD_ZONE_CONFIG 1U // Enable ZONE_CONFIG (0x07)
#define MTB_PMBUS_IMPL_CMD_ZONE_ACTIVE 1U // Enable ZONE_ACTIVE (0x08)
```

Target configuration - Enable zone support

```
mtb_pmbus_stc_config_t config = {
    .enable_pmbus = true,
    .enable_zone  = true,               // Enable zone protocols
    .zone_callback = zone_callback,     // Zone event handler
    .impl_cmd_mask = MTB_PMBUS_IMPL_CMD_ZONE_CONFIG_EN |
                     MTB_PMBUS_IMPL_CMD_ZONE_ACTIVE_EN,
};
```

Zone assignment after init

```
// After init, all pages default to zone 0xFE (not active)
// Option 1: Controller sends ZONE_CONFIG/ZONE_ACTIVE commands
// Option 2: Set zones programmatically:
mtb_pmbus_set_default_zones(&pmbus_inst,
                            0x01,  // read_zone
                            0x01,  // write_zone
                            0);    // page
```

Synchronized voltage change

```
/* Target mode - Handle zone write */
bool vout_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase, uint8_t byte) {
    if (event == MTB_PMBUS_CMD_WRITE_DONE) {
        uint16_t vout;
        mtb_pmbus_cmd_read_data_ext_isr(&pmbus_inst, CMD_VOUT_COMMAND,
                                        page, MTB_PMBUS_NO_PAGE_PHASE,
                                        (uint8_t*)&vout, 2);
        float voltage = mtb_pmbus_lin16_to_float(vout, -10);
        set_output_voltage(page, voltage);
    }
    return true;
}

/* Controller mode - Broadcast to all devices in zone */
#define ZONE_WR_ADDR  0x37U

mtb_pmbus_ctrl_stc_transfer_cfg_t write_cfg;
uint8_t write_buffer[3U];

// Step 1: Activate zone 0x01
write_buffer[0] = 0x08U;  // ZONE_ACTIVE command
write_buffer[1] = 0x01U;  // Write zone number
write_buffer[2] = 0x01U;  // Read zone number

write_cfg.data = write_buffer;
write_cfg.addr = ZONE_WR_ADDR;
write_cfg.wr_size = 3U;
write_cfg.rd_size = 0U;
write_cfg.execute_stop = true;

mtb_pmbus_ctrl_execute_transfer(&ctrl_inst, &write_cfg);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);

// Step 2: Zone Write voltage command to all devices
uint16_t new_vout = mtb_pmbus_float_to_lin16(3.3f, -10);
write_buffer[0] = 0x21U;                    // VOUT_COMMAND
write_buffer[1] = (uint8_t)(new_vout);      // Data low
write_buffer[2] = (uint8_t)(new_vout >> 8); // Data high

mtb_pmbus_ctrl_execute_transfer(&ctrl_inst, &write_cfg);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);
// All devices in zone 0x01 update simultaneously on STOP
```

Zone Read Protocol

Use case: Query status from multiple devices with automatic bus arbitration (lowest address device wins).

Format

```
Zone Read Phase 1 - Controller Request:
Controller -> [S] [0x28+W] [A] [CtrlCode] [A] [Mask] [A] [PEC] [P]

CtrlCode Format:
  Bit 0 (DS): Data Size (0=Byte, 1=Word)
  Bit 5 (DI): Data Invert (0=normal, 1=inverted)
  Bits 6-7:  Response Type (00=Status, 01=Command data)

Zone Read Phase 2 - Multi-Target Response:
Controller -> [S] [0x28+R] [P]

Targets respond simultaneously (arbitration):
Target     -> [STATUS] [A] [ADDR|RD] [A] [PAGE] [A] [PEC] [P]

Lowest address device wins arbitration, others stop transmitting.
```

Zone read status

```c
/* Target mode - Zone read callback */
uint8_t zone_callback(mtb_pmbus_zone_events_t event, uint8_t mask, int32_t page) {
    uint8_t status = 0;

    switch (event) {
        case MTB_PMBUS_ZONE_READ_STATUS_BYTE_REQ:
            // Return status byte AND'ed with controller's mask
            status = get_device_status(page) & mask;
            break;

        case MTB_PMBUS_ZONE_READ_STATUS_ARB_LOST:
            // Normal - another device won. Controller will retry.
            MTB_PMBUS_LOG_DBG("Lost arbitration to lower address device");
            break;

        case MTB_PMBUS_ZONE_READ_STATUS_CMPLT:
            // Our response was received by controller
            MTB_PMBUS_LOG_DBG("Zone Read completed successfully");
            break;
    }
    return status;
}

/* Controller mode - Query all devices in zone */
#define ZONE_WR_ADDR  0x37U
#define ZONE_RD_ADDR  0x28U

mtb_pmbus_ctrl_stc_transfer_cfg_t write_cfg;
mtb_pmbus_ctrl_stc_transfer_cfg_t read_cfg;
uint8_t write_buffer[3U];
uint8_t read_buffer[4U];

// Activate zone 0x01
write_buffer[0] = 0x08U;  // ZONE_ACTIVE command
write_buffer[1] = 0x01U;  // Write zone number
write_buffer[2] = 0x01U;  // Read zone number

write_cfg.data = write_buffer;
write_cfg.addr = ZONE_WR_ADDR;
write_cfg.wr_size = 3U;
write_cfg.rd_size = 0U;
write_cfg.execute_stop = true;

mtb_pmbus_ctrl_execute_transfer(&ctrl_inst, &write_cfg);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);

// Zone Read - Phase 1: Send control code and mask
write_buffer[0] = 0x00U;  // Control code (status byte request)
write_buffer[1] = 0xFFU;  // Mask byte (read all bits)

write_cfg.data = write_buffer;
```

```
write_cfg.addr = ZONE_RD_ADDR;
write_cfg.wr_size = 2U;
write_cfg.rd_size = 0U;
write_cfg.execute_stop = false;  // No STOP - Zone Read continues

mtb_pmbus_ctrl_execute_transfer(&ctrl_inst, &write_cfg);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);

// Zone Read - Phase 2: Read from first device (lowest address wins)
read_cfg.data = read_buffer;
read_cfg.addr = ZONE_RD_ADDR;
read_cfg.wr_size = 0U;
read_cfg.rd_size = 4U;  // Status + Addr + Page + PEC
read_cfg.execute_stop = false;  // Keep reading for more devices

mtb_pmbus_ctrl_execute_transfer(&ctrl_inst, &read_cfg);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);
// read_buffer[0]=Status, read_buffer[1]=Addr|RD, read_buffer[2]=Page, read_buffer[3]=PEC
```

Zone Control Commands

ZONE_CONFIG (0x07) - Assign Zones to device pages

Format

```
Controller -> [DevAddr+W] [A] [0x07] [A] [WriteZone] [A] [ReadZone] [A] [PEC] [P]
```

Device pages

```
// Controller: Configure device page zones using standard Write Word
uint8_t zone_cfg_data[2U] = { 0x01U, 0x02U };  // WriteZone=0x01, ReadZone=0x02
mtb_pmbus_ctrl_ex_write_word(&ctrl_inst, device_addr, 0x07U, zone_cfg_data, true);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);
// Current page now: write_zone=0x01, read_zone=0x02
```

ZONE_ACTIVE (0x08) - Activate Zones for operations**:**

Format

```
Controller -> [0x37+W] [A] [0x08] [A] [WriteZone] [A] [ReadZone] [A] [PEC] [P](broadcast)
```

Activate zone 0x01 for all devices

```
// Controller: Activate zone 0x01 for all devices using generic transfer
#define ZONE_WR_ADDR  0x37U

mtb_pmbus_ctrl_stc_transfer_cfg_t write_cfg;
uint8_t write_buffer[3U] = { 0x08U, 0x01U, 0x01U };  // ZONE_ACTIVE, WriteZone, ReadZone

write_cfg.data = write_buffer;
write_cfg.addr = ZONE_WR_ADDR;
write_cfg.wr_size = 3U;
write_cfg.rd_size = 0U;
write_cfg.execute_stop = true;

mtb_pmbus_ctrl_execute_transfer(&ctrl_inst, &write_cfg);
mtb_pmbus_ctrl_wait_cmpl(&ctrl_inst, PMBUS_CTRL_TIMEOUT_US);
// Only devices with matching zones participate in Zone Read/Write
```

3 Getting started with PMBus® middleware

This section provides a step-by-step guide to quickly get started with the PMBus® middleware and how to set up the PMBus® middleware in your project, configure the necessary hardware, and implement the basic SMBus/ PMBus® functionality.

3.1 Target Mode using PMBus® Configurator

This section details PMBus® Target Mode initialization using the PMBus® Configurator.

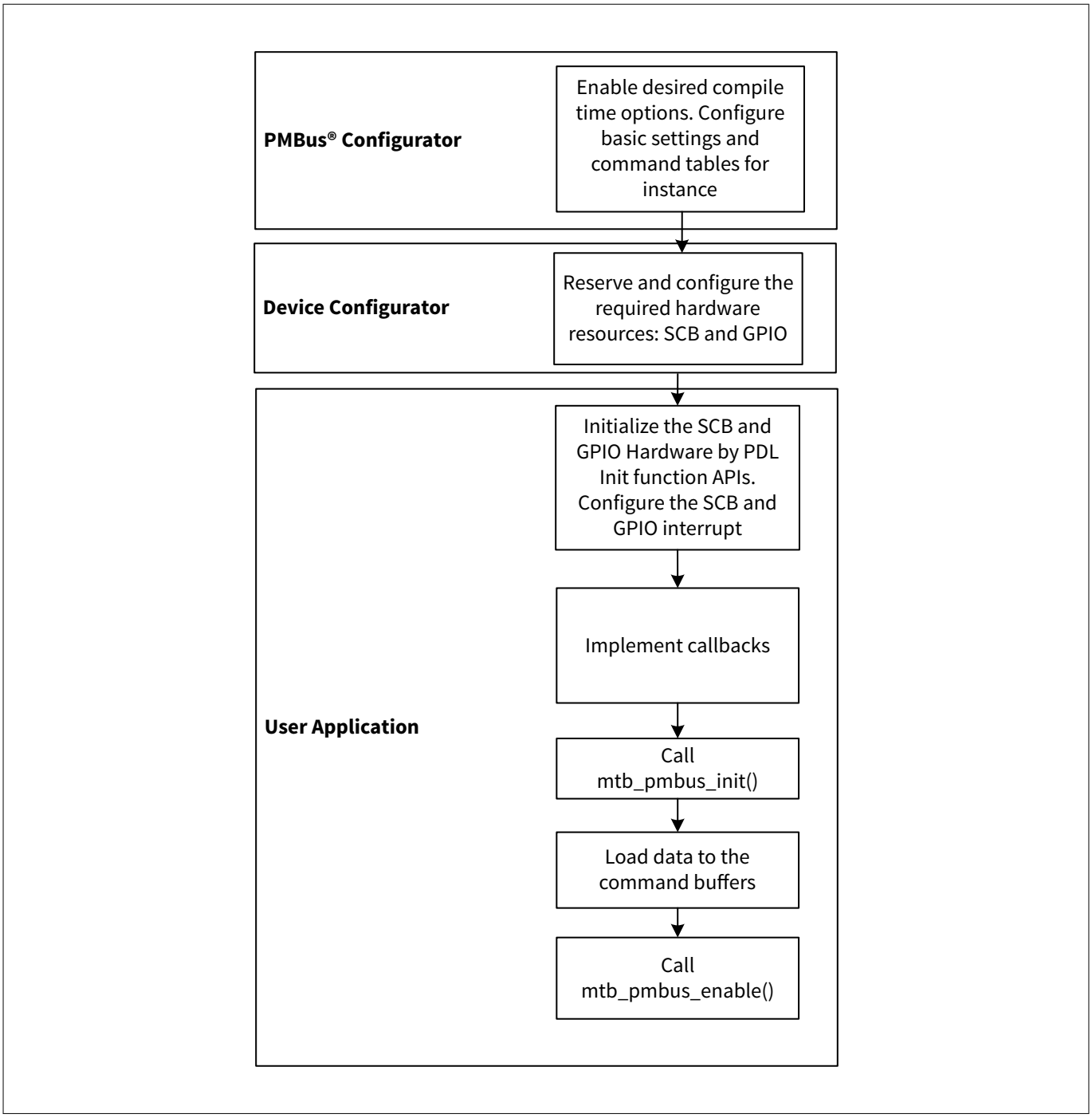

Figure 7 PMBus® Target initialization sequence with solution personality

1. Add `mtb-pmbus` middleware to your project

If you work in the ModusToolbox™ IDE, use the ModusToolbox™ Library Manager to add the `mtb-pmbus` middleware to your project. Otherwise, ensure that `mtb-pmbus` middleware is included in your project.

***Note**: The PMBus® middleware uses the `printf()` function for logging purposes. To use the `printf()` function for the terminal output, add retarget-io middleware from the Library Manager or in any other way.*

2. Configure PMBus® using Device Configurator

1. **Add PMBus® Solution instance**
   - **a.** Open the **Device Configurator** and navigate to the **Solutions** tab
   - **b.** Add a new **PMBus** instance under the PMBus category
   - **c.** Enter the desired instance name (e.g., `PMBUS`)

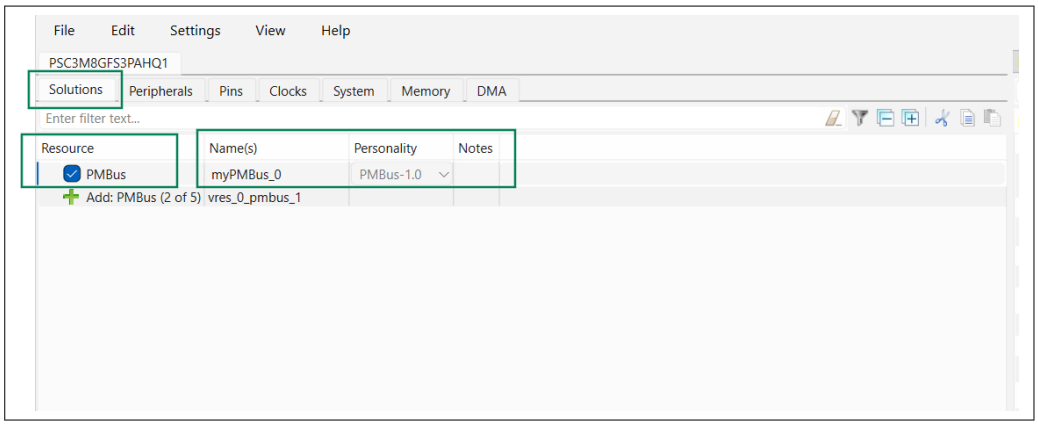

Figure 8 PMBus® solution instance

2. **Configure I2C hardware**

   In the PMBus® solution instance configuration:

   - **a.** **Data Rate (kbps)**: Set to the desired value (e.g., 100 kbps)
   - **b.** **Clock Signal**: Select the desired clock for the SCB block
   - **c.** **SCL/SDA Pins**: Select the desired pins (auto-configured to Open Drain mode)
   - **d.** **I2C Placement**: Select the SCB block to use

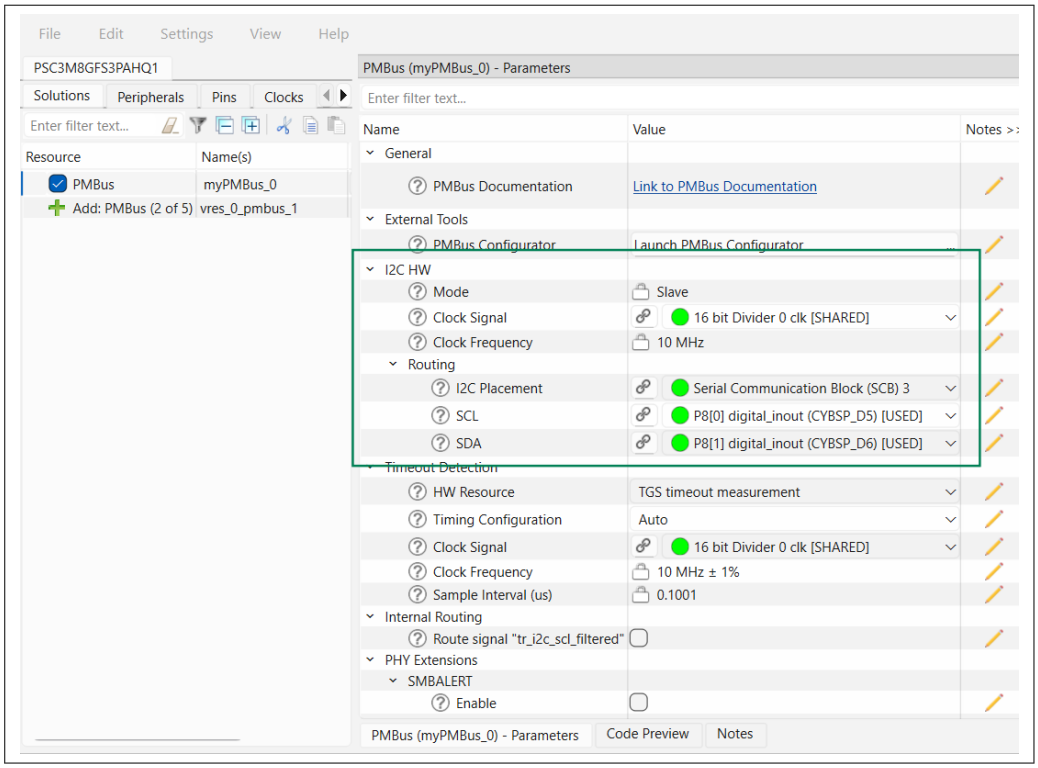

Figure 9 Configure the I2C hardware section

3. **(Optional): Configure Timeout Detection**
   a. **Hardware Resource**: Select TGS or TCPWM for timeout detection
   b. **Timing Configuration**: Set to Auto or Manual
   c. **Clock Signal**: Assign the timeout timer clock

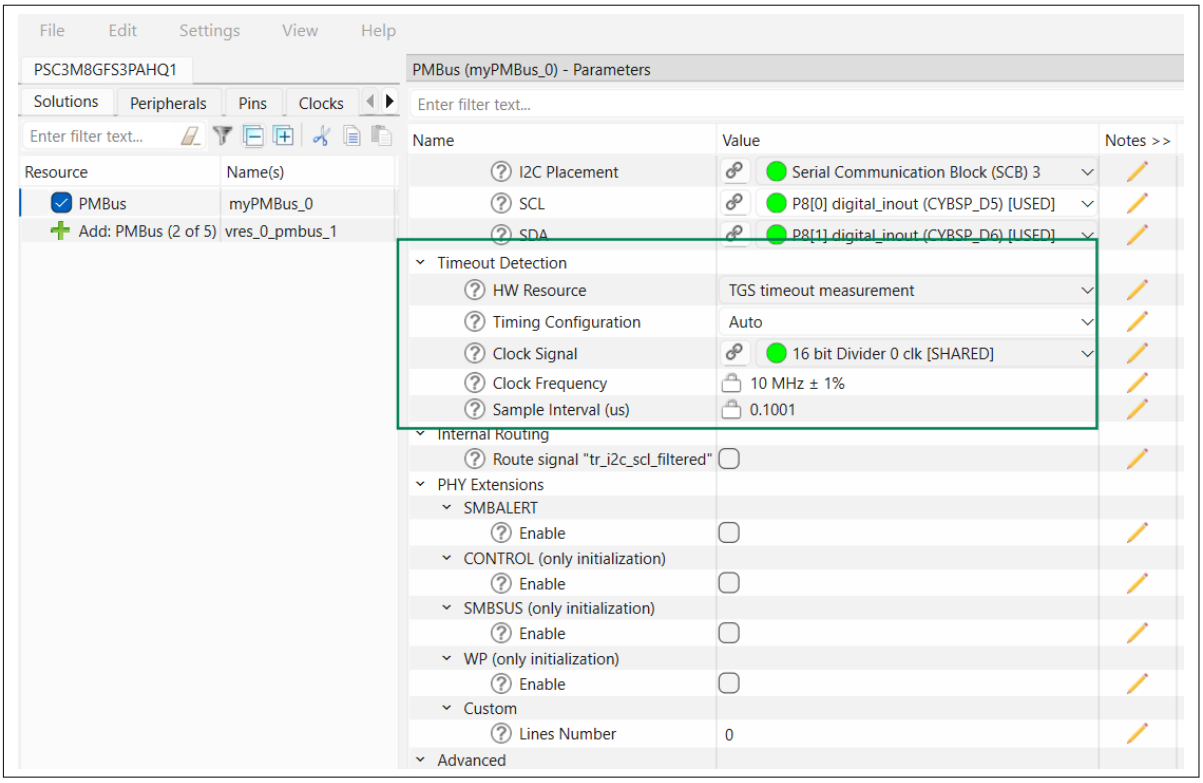

Figure 10 Configure Timeout Detection

4. **(Optional): Configure PHY Extensions**

a. **SMBALERT**: Enable if SMBus Alert signal is required
b. **Custom Lines**: Add any additional GPIO lines as needed

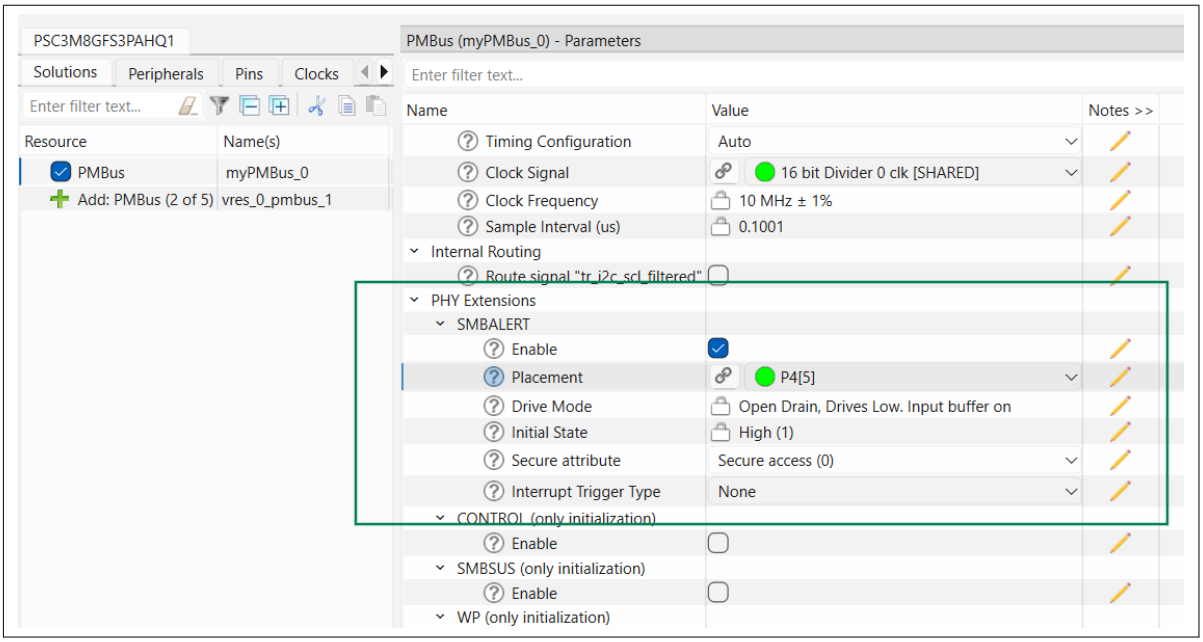

Figure 11 Configure PHY Extensions

5. **Configure Debug UART**
   a. Enable another SCB block in **Peripherals** > **Communication** and select **UART Personality**
   b. Select the desired name for the SCB (e.g., `DEBUG_UART`)
   c. Configure pins and clock for the UART
6. **Configure User LED**
   a. Switch to the **Pins** tab and enable any User LED GPIO pin
   b. In the **General** section, select **Strong Drive, input buffer off** Drive mode
7. **Save configuration**
   Select **File** > **Save** to generate initialization code

3. Configure PMBus® Commands using PMBus® Configurator

1. **Launch PMBus Configurator**
   a. In the Device Configurator, select your PMBus instance
   b. Click **PMBus Configurator** in the **External Tools**

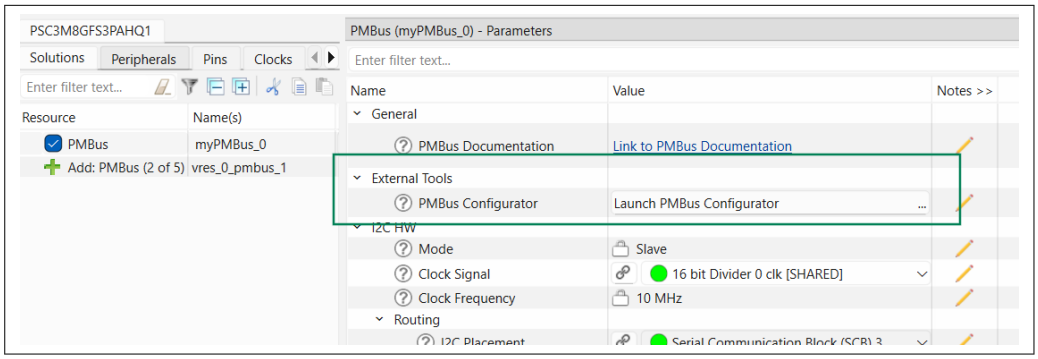

Figure 12 Launch PMBus Configurator

2. **Configure General tab**
   The **General** tab contains the fundamental PMBus® instance settings:
   Mode

Select **PMBus Target** to configure the device as a PMBus® slave that responds to controller commands. Other options include SMBus Target, PMBus® Controller, and SMBus Controller.

Data Rate

Set to **100 kbps** for standard SMBus/PMBus® communication. Higher speeds (400 kbps, 1000 kbps) are available but require compatible bus hardware and pull-up resistors.

Target Address

Enter **0x18** as the 7-bit I2C slave address. This is the address that the controller uses to communicate with this device. Avoid reserved SMBus addresses (0x00-0x07, 0x78-0x7F).

Number of Pages

Set to **2** to enable page-indexed command data. Pages allow a single PMBus® device to manage multiple outputs or channels. Each paged command maintains separate data buffers for each page (0 and 1 in this example).

Number of Phases

Set to 0 to disable phase support. Phases are used for multi-phase power converters where each phase can be monitored/controlled independently. Not needed for this quick start example.

Enable PMBus

Check this option to enable PMBus®-specific protocol features including the PAGE command, QUERY command, CAPABILITY command, and other PMBus® extensions beyond basic SMBus functionality.

Enable PEC

Check this option to enable Packet Error Checking (PEC). When enabled, a CRC-8 byte is added to each transaction for error detection per SMBus specification. The middleware automatically calculates and validates PEC bytes, rejecting transactions with checksum errors.

Enable SMBAlert

Check this option to enable the SMBus Alert signal. When enabled, the target can assert the SMBALERT# line to notify the controller of alert conditions, such as faults, warnings, or status changes. The Alert Response Address (0x0C) protocol is automatically supported.

Enable Zone

Check this option to enable Zone Read and Zone Write protocols. Zone protocols allow a controller to broadcast commands to multiple devices simultaneously using special zone addresses (Zone Read: 0x28, Zone Write: 0x37). When enabled, the middleware handles ZONE_CONFIG (0x07) and ZONE_ACTIVE (0x08) commands automatically.

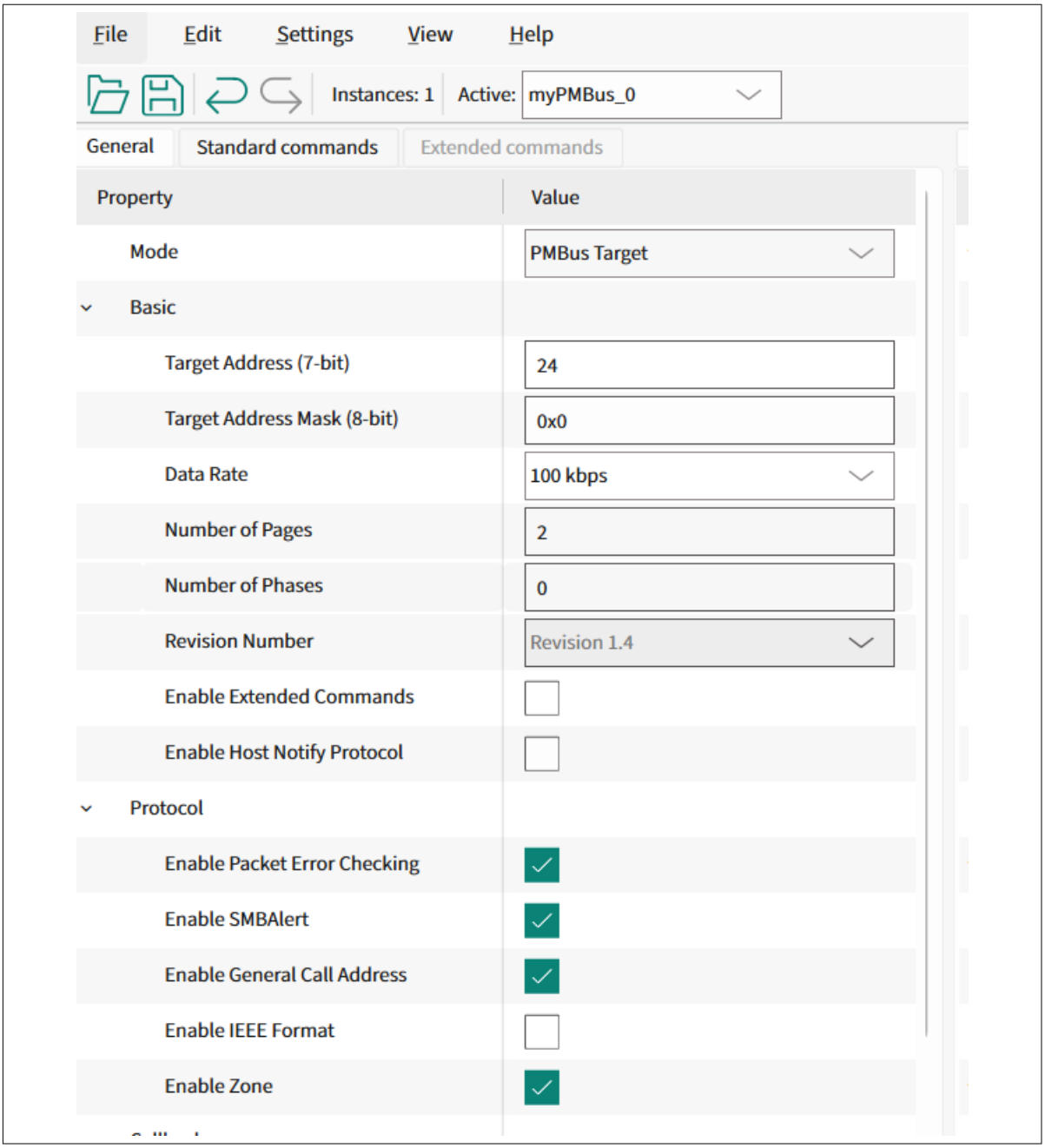

Figure 13 General configuration

3. Configure PMBus Commands tab

For more information about configuring commands using the PMBus® Configurator Command tab, see the **Add PMBus® Commands using Device Configurator** section.

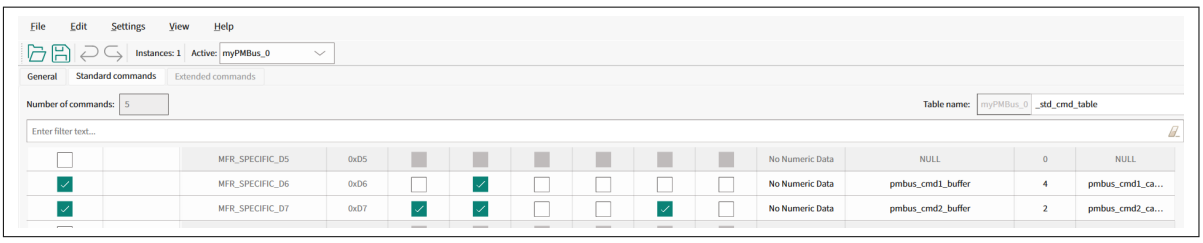

Figure 14 Command configuration

4. Configure Compile time options tab

The Compile time options tab sets global configuration macros that affect all PMBus® instances and are used for conditional compilation to optimize code size.

Log Level

Select Debug to enable all logging messages including debug-level output. This is helpful during development but increases code size. For production, select Warning or Error.

Enable Timeout

Check this option to enable the 25 ms SMBus timeout detection. When enabled, the middleware monitors the SCL line and generates a timeout error if the clock is held low for more than 25 ms, indicating a stuck bus condition per SMBus specification. The timeout detection uses either the Timeout Generation Support (TGS) counter built into the SCB block or an external TCPWM timer, depending on device capabilities.

Enable PEC

Check this option to include Packet Error Checking (PEC) calculation code in the build. This enables CRC-8 error detection for all PMBus® transactions across all instances. The middleware automatically calculates PEC for outgoing data and validates PEC for incoming data, issuing a NACK if the checksum is incorrect.

Pages Number

Set to 2 to define the maximum number of pages supported across all instances. This value must be >= the highest `num_pages` value used by any instance.

Phases Number

Set to 0 since phase support is not used in this example.

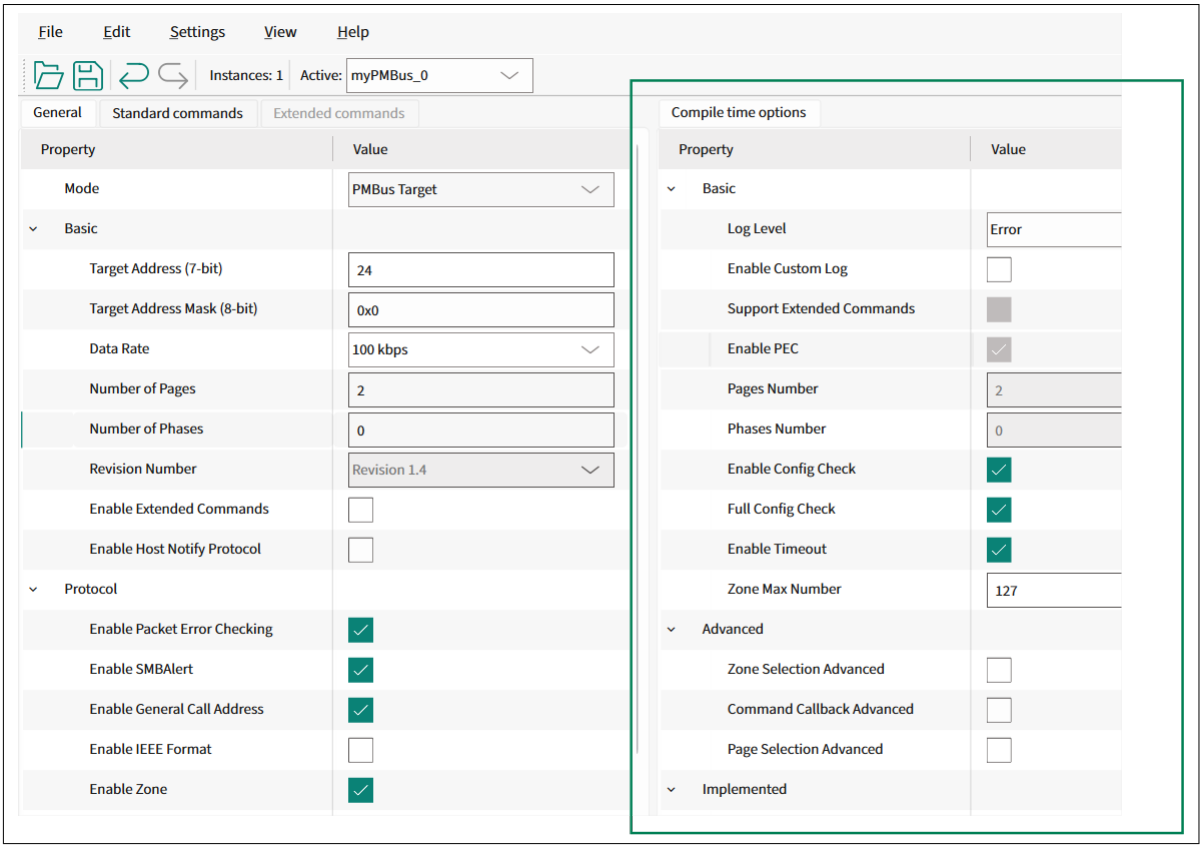

Figure 15 Compile time options

5. Save and Close

Save the configuration and close the PMBus® Configurator. The Device Configurator will generate:

- `cycfg_pmbus.h/c` file: Configuration structures and command tables
- mtb_pmbus_conf.h: Compile-time options

4. Add PMBus Code to your project

With the Configurator, most configuration code is auto-generated. You only need to implement:

- Command data buffers (declared as extern)
- Callback functions (declared as extern)
- I2C ISR handler
- Minimal `main()` initialization

1. Include header files

```
#include "mtb_pmbus.h"
#include "cybsp.h"
#include "cy_retarget_io.h"
#include "cycfg_pmbus.h"  /* Auto-generated PMBus configuration */
```

2. Create project defines

```
#define PMBUS_TEST_CMD_1_SIZE       (4U)
#define PMBUS_TEST_CMD_2_SIZE       (2U)
#define PMBUS_TEST_CMD_2_PAGES      (2U)
```

3. Create variables

```
/* Debug UART variables */
static cy_stc_scb_uart_context_t    DEBUG_UART_context;
static mtb_hal_uart_t               DEBUG_UART_hal_obj;

/* I2C context (referenced by generated HAL config) */
cy_stc_scb_i2c_context_t  PMBUS_I2C_context;

/* PMBus instance */
static mtb_pmbus_stc_t qsg_pmbus_inst;

/* Data storage */
volatile uint32_t user_led_toggled_cnt = 0U;
volatile uint16_t user_data = 0U;
```

4. Implement general event callback (extern declaration in `cycfg_pmbus.h`)

```
void PMBUS_gen_callback(mtb_pmbus_events_t event)
{
    if (event == MTB_PMBUS_QUICK_CMD_WR_EVENT)
    {
        printf("Gen: MTB_PMBUS_QUICK_CMD_WR_EVENT\n\r");
        Cy_GPIO_Inv(CYBSP_USER_LED_PORT, CYBSP_USER_LED_PIN);
        user_led_toggled_cnt++;
        printf("User LED toggled\n\r");
    }
}
```

5. Implement command callbacks (extern declarations in `cycfg_pmbus.h`)

```
bool pmbus_cmd1_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase,
uint8_t byte)
{
    (void) page;
    (void) phase;
    (void) byte;

    if (event == MTB_PMBUS_CMD_MATCH)
    {
        printf("\n\rCMD1 match\n\r");
    }
    else if (event == MTB_PMBUS_CMD_READ_REQ)
    {
        printf("\n\rCMD1 read request\n\r");
        uint32_t tmp = user_led_toggled_cnt;
        mtb_pmbus_cmd_update_data_isr(&qsg_pmbus_inst, 0x10U, (uint8_t*)&tmp,
PMBUS_TEST_CMD_1_SIZE);
        printf("Data buffer for read: 0x%02X\n\r", pmbus_cmd1_buffer[0U]);
    }
    return true;
}

bool pmbus_cmd2_callback(mtb_pmbus_cmd_events_t event, int32_t page, int32_t phase,
uint8_t byte)
{
    (void) byte;
    (void) phase;

    if (event == MTB_PMBUS_CMD_MATCH)
    {
        printf("\n\rCMD2 match\n\r");
        uint16_t tmp = user_data;
        mtb_pmbus_cmd_update_data_ext_isr(&qsg_pmbus_inst, 0x11U, page, phase,
(uint8_t*)&tmp, PMBUS_TEST_CMD_2_SIZE);
        printf("Data buffer for page %ld: 0x%02X 0x%02X\n\r", page,
pmbus_cmd2_buffer[page][0U], pmbus_cmd2_buffer[page][1U]);
    }
    else if (event == MTB_PMBUS_CMD_WRITE_DONE)
    {
        printf("\n\rCMD2 write done\n\r");
        printf("Data buffer after write for page %ld: 0x%02X 0x%02X\n\r", page,
pmbus_cmd2_buffer[page][0U], pmbus_cmd2_buffer[page][1U]);
        uint16_t tmp;
        mtb_pmbus_cmd_read_data_ext_isr(&qsg_pmbus_inst, 0x11U, page, phase,
(uint8_t*)&tmp, PMBUS_TEST_CMD_2_SIZE);
        user_data = tmp;
    }
    return true;
}
```

6. Implement I2C ISR handler

```
void i2c_isr(void)
{
    mtb_pmbus_i2c_isr(&qsg_pmbus_inst);
}
```

7. Implement hardware callbacks (extern declarations in `cycfg_pmbus.h`)

```
void myPMBus_0_hw_resource_ctrl(mtb_pmbus_hw_resources_ctrl_action_t action)
{
    if (action == MTB_PMBUS_HW_RESOURCES_INIT)
    {
        /* Initialize I2C */
        Cy_SCB_I2C_Init(PMBUS_I2C_HW, &PMBUS_I2C_config, &PMBUS_I2C_context);

        /* Configure and initialize I2C interrupt */
        cy_stc_sysint_t i2c_isr_cfg =
        {
            .intrSrc = PMBUS_I2C_IRQ,
            .intrPriority = 3U
        };
        Cy_SysInt_Init(&i2c_isr_cfg, i2c_isr);

        printf("I2C hardware initialized\n\r");
    }
    else if (action == MTB_PMBUS_HW_RESOURCES_ENABLE)
    {
        Cy_SCB_I2C_Enable(PMBUS_I2C_HW);
        printf("I2C hardware enabled\n\r");
    }
    else if (action == MTB_PMBUS_HW_RESOURCES_DISABLE)
    {
        Cy_SCB_I2C_Disable(PMBUS_I2C_HW, &PMBUS_I2C_context);
        printf("I2C hardware disabled\n\r");
    }
}

void myPMBus_0_hw_irq_enable(void)
{
    NVIC_EnableIRQ((IRQn_Type) PMBUS_I2C_IRQ);
}

void myPMBus_0_hw_irq_disable(void)
{
    NVIC_DisableIRQ((IRQn_Type) PMBUS_I2C_IRQ);
}
```

8. Implement `main()` function

```
int main(void)
{
    cy_rslt_t result;
    cy_en_scb_uart_status_t init_status;
    mtb_pmbus_status_t status;

    /* Initialize the device and board peripherals */
    result = cybsp_init();
    if (result != CY_RSLT_SUCCESS)
```

```
    {
        CY_ASSERT(0);
    }

    /* Initialize Debug UART */
    init_status = Cy_SCB_UART_Init(DEBUG_UART_HW, &DEBUG_UART_config,
&DEBUG_UART_context);
    if (init_status != CY_SCB_UART_SUCCESS)
    {
        CY_ASSERT(0);
    }
    Cy_SCB_UART_Enable(DEBUG_UART_HW);

    result = mtb_hal_uart_setup(&DEBUG_UART_hal_obj, &DEBUG_UART_hal_config,
&DEBUG_UART_context, NULL);
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    result = cy_retarget_io_init(&DEBUG_UART_hal_obj);
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Terminal header */
    printf("\x1b[2J\x1b[;H");
    printf("*************************************************************\r\n");
    printf("PMBus Quick Start Guide (with Configurator)\r\n");
    printf("*************************************************************\r\n\n");

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize PMBus middleware using auto-generated configuration */
    status = mtb_pmbus_init(&qsg_pmbus_inst, &myPMBus_0_config);

    if (MTB_PMBUS_STATUS_SUCCESS != status)
    {
        printf("PMBus initialization failed! Status: %d\n\r", status);
        CY_ASSERT(0);
    }
    else
    {
        printf("PMBus initialized successfully\n\r");

        status = mtb_pmbus_enable(&qsg_pmbus_inst);
        if (MTB_PMBUS_STATUS_SUCCESS != status)
        {
            printf("PMBus enable failed! Status: %d\n\r", status);
            CY_ASSERT(0);
        }
```

```
        else
        {
            printf("PMBus enabled - Ready for communication\n\r");
        }
    }

    /* Main loop */
    for (;;)
    {
        /* Application code here */
    }
}
```

3.2 Target Mode using PMBus® middleware

This section provides steps to quickly get started with the PMBus® middleware for implementing SMBus/PMBus® target functionality.

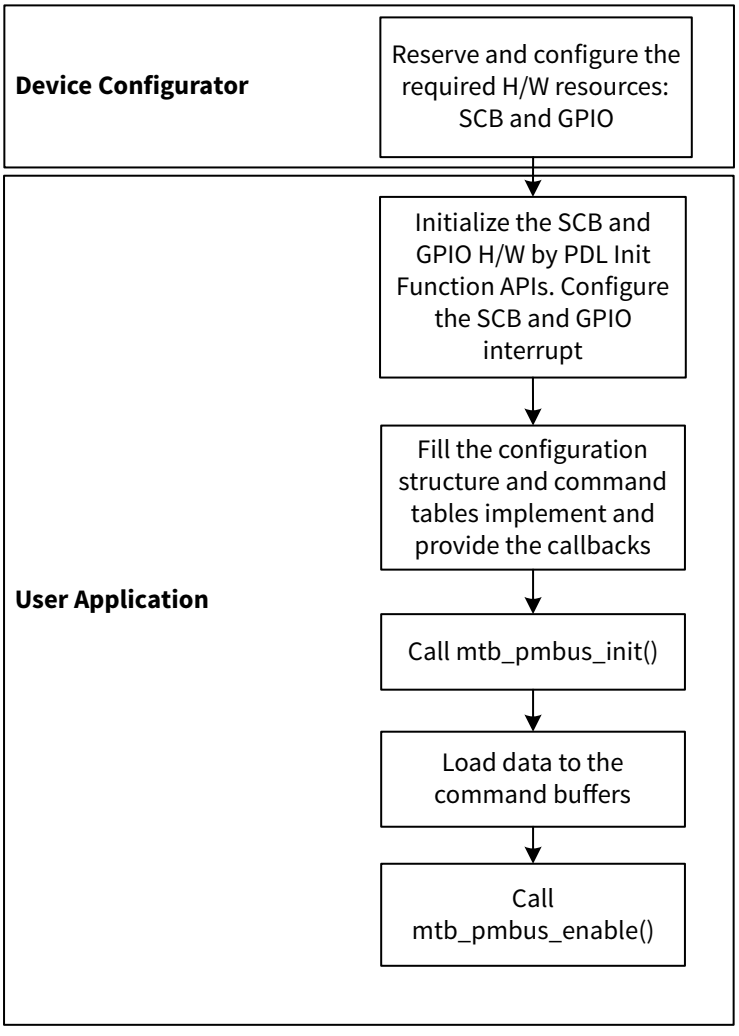

Figure 16 PMBus® target initialization sequence without solution personality

1. **Add mtb-pmbus middleware**

   Add the `mtb-pmbus` middleware to your project using the ModusToolbox™ Library Manager. Include the retarget-io middleware for terminal output logging

2. **Configure hardware**

   Configure the following in the Device Configurator:

   - **I2C SCB block**: Set to Slave mode with the desired data rate (e.g., 100 kHz), disable TX FIFO, enable "Accept Matching Address in RX FIFO", and set the slave address
   - **UART SCB block**: For debug output
   - **GPIO pin**: For User LED indicator

3. **Implement SMBus/PMBus® code**

   a. Configure `mtb_pmbus_conf.h` with compile-time options
   b. Create I2C context and PMBus® instance variables
   c. Implement callback functions for events and commands
   d. Define a command table with command codes, capabilities, and buffers
   e. Implement I2C interrupt handler and hardware control callbacks
   f. Initialize peripherals and call `mtb_pmbus_init()` followed by `mtb_pmbus_enable()`

4. **Verify operation**

   Build, program, and test using Quick Command and Read Byte protocols. For PMBus® mode, test PAGE command and Write/Read Word protocols with page support

For complete implementation details including code snippets, see Target mode quick start guide.

3.3 Controller Mode using PMBus® middleware

This section describes how to set up the PMBus® middleware for Controller Mode operation.

***Note**: Testing Controller Mode requires a second PMBus® device operating in Target Mode. See the Target Mode using PMBus® Configurator section to set up the PMBus® Target device.*

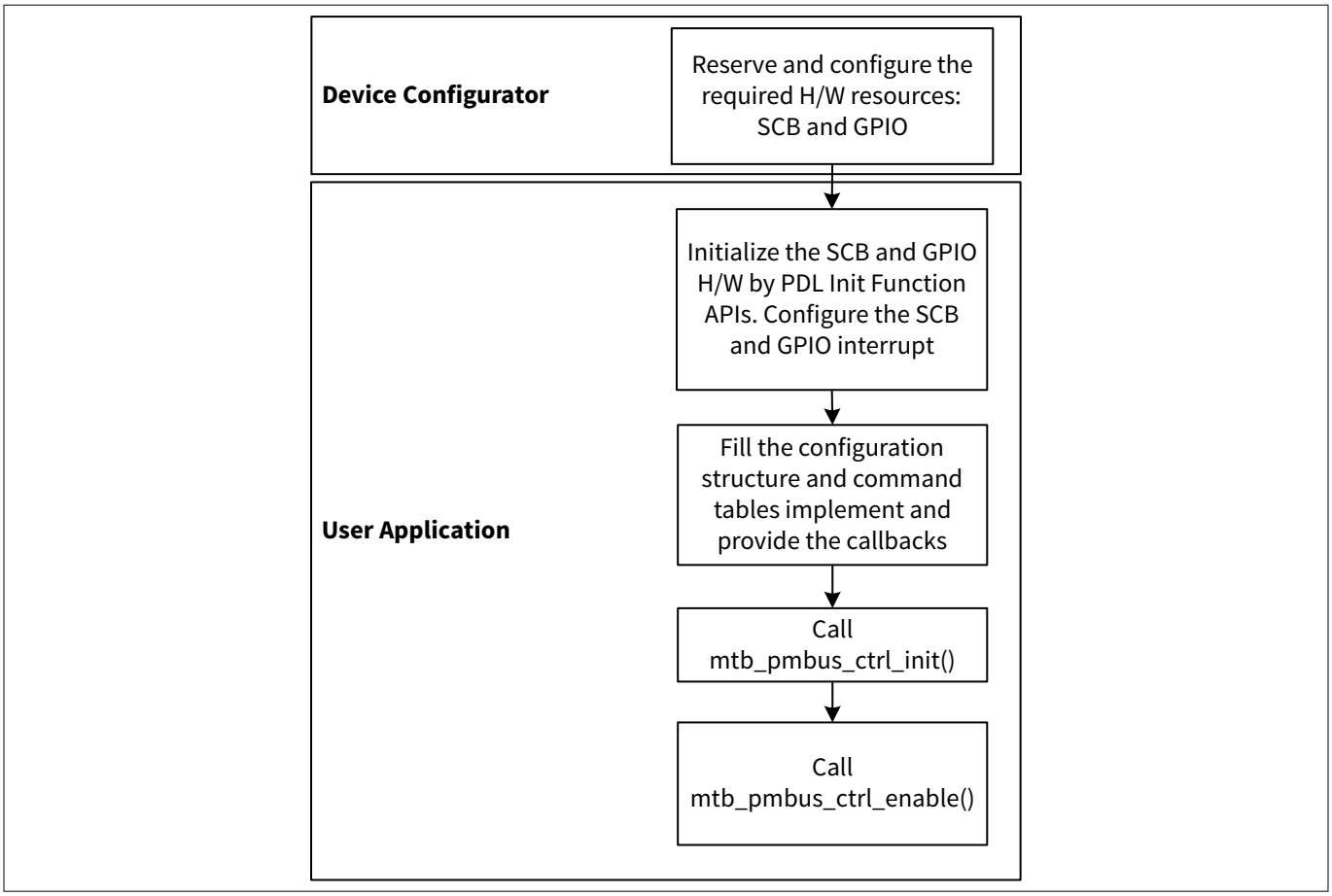

Figure 17 PMBus® controller initialization without configurator

Step 1: Add `mtb-pmbus` middleware

Add the `mtb-pmbus` middleware to your project using the ModusToolbox™ Library Manager. Optionally add retarget-io middleware for terminal logging via `printf()`.

Step 2: Configure SCB blocks

In Device Configurator:

- **I2C SCB**: Enable an SCB block. Select I2C Personality in **Master** mode with the desired data rate (e.g., 100 kHz). Disable "Use RX FIFO". Configure SDA/SCL pins
- **UART SCB**: Enable a second SCB for debug output with UART Personality

Save to generate initialization code.

Step 3: Add PMBus Controller Code

1. **Configure `mtb_pmbus_conf.h`**: Set compile-time configuration macros. Add `MTB_PMBUS_MANUAL_CONFIG` to the project Makefile if not using Device Configurator personality
2. **Create variables**: Define I2C context, PMBus® controller instance, and debug UART objects
3. **Implement callbacks**:
   - I2C interrupt handler calling `mtb_pmbus_ctrl_isr()`
   - Hardware resource control callback (init/enable/disable I2C)
   - ISR enable/disable callbacks
4. **Configure structures**: Create `mtb_pmbus_ctrl_stc_config_hal_t` and `mtb_pmbus_ctrl_cfg_t` structures
5. **Initialize in `main()`**:
   - Initialize board peripherals (`cybsp_init()`)

- Initialize debug UART and retarget-io
- Enable global interrupts
- Initialize and enable PMBus® controller using `mtb_pmbus_ctrl_init()` and `mtb_pmbus_ctrl_enable()`

6. **Execute transfers** using dedicated protocol APIs or generic transfer API:
   - **Quick Command**: `mtb_pmbus_ctrl_ex_quick_cmd()`; toggles target LED
   - **Read 32**: `mtb_pmbus_ctrl_ex_read_32()`; reads 4-byte counter value
   - **Generic Transfer**: `mtb_pmbus_ctrl_execute_transfer()`; for PAGE command, Write Word, Read Word protocols

Step 4: Verify operation

1. Build and program the project
2. Connect terminal application to device COM port
3. Verify I2C initialization and controller ready status
4. Connect PMBus® target device (address 0x18) to I2C bus
5. Test protocols:
   - **Quick Command**: Observe target LED toggle
   - **Read 32**: Verify LED toggle counter readback
   - **Generic Transfer**: Confirm PAGE, Write Word, and Read Word operations

For complete implementation details including code snippets, see Controller mode quick start guide.

3.4 Add PMBus® Commands using Device Configurator

The **Command** tab in PMBus Configurator provides the complete list of PMBus® specification commands. This section provides how to add and configure commands:

Step 1: Access the Command tab

1. Open **PMBus Configurator** from Device Configurator
2. Select **Mode = PMBus Target**
3. Navigate to the **Command Tab**

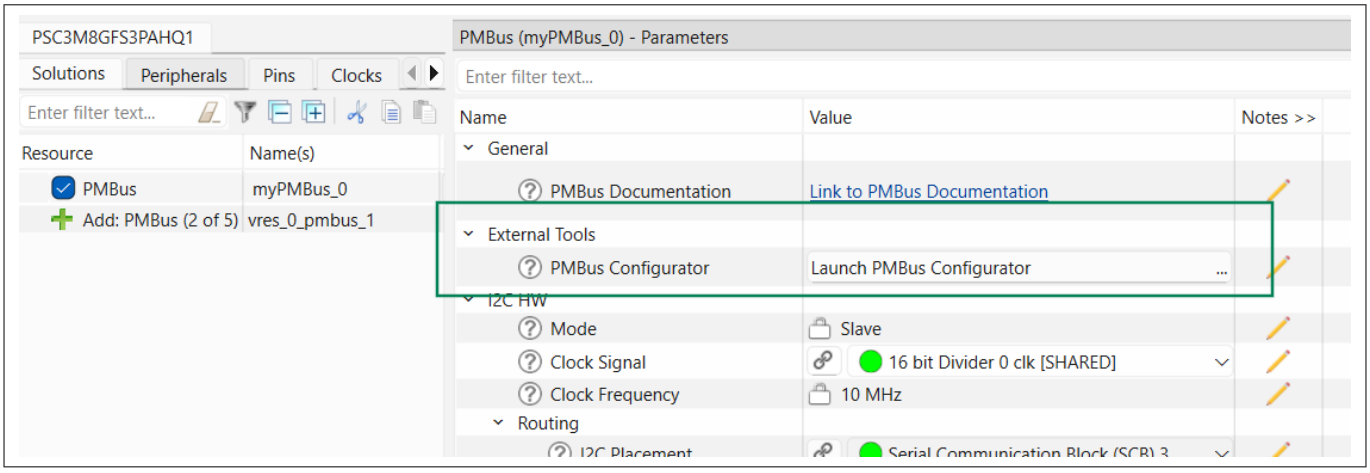

Figure 18 PMBus configurator launch

Step 2: Enable a command

Find the command that you want to support and select the Enable option:

Step 3: Choose Implementation type

Option A: Use Pre-Implemented (Automatic Response)

Select both **Enable** and **Implemented** options.

- No further configuration needed
- Middleware handles everything automatically
- Available only for: PAGE, PHASE, CAPABILITY, QUERY, PMBUS_REVISION, and ZONE commands

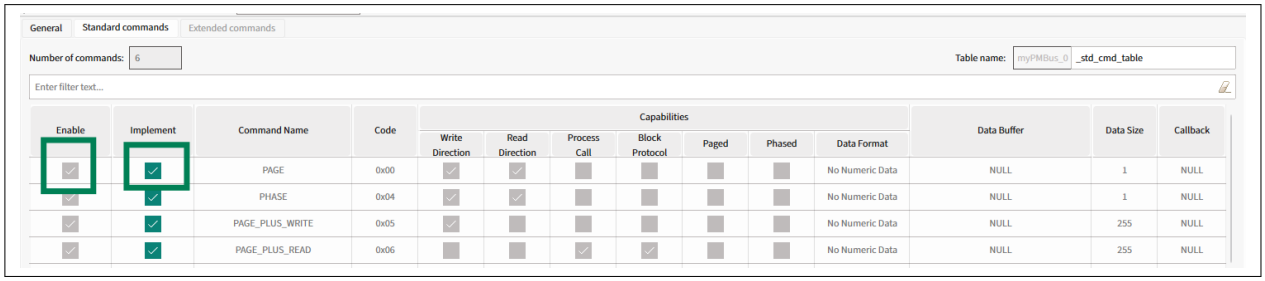

Figure 19 Pre-Implemented commands

Option B: Custom Implementation

Select **Enable** but leave **Implemented** as unselected

- Configure the fields manually

Step 4: Configure Custom Command Fields

When **Implemented** is unchecked, configure these fields:

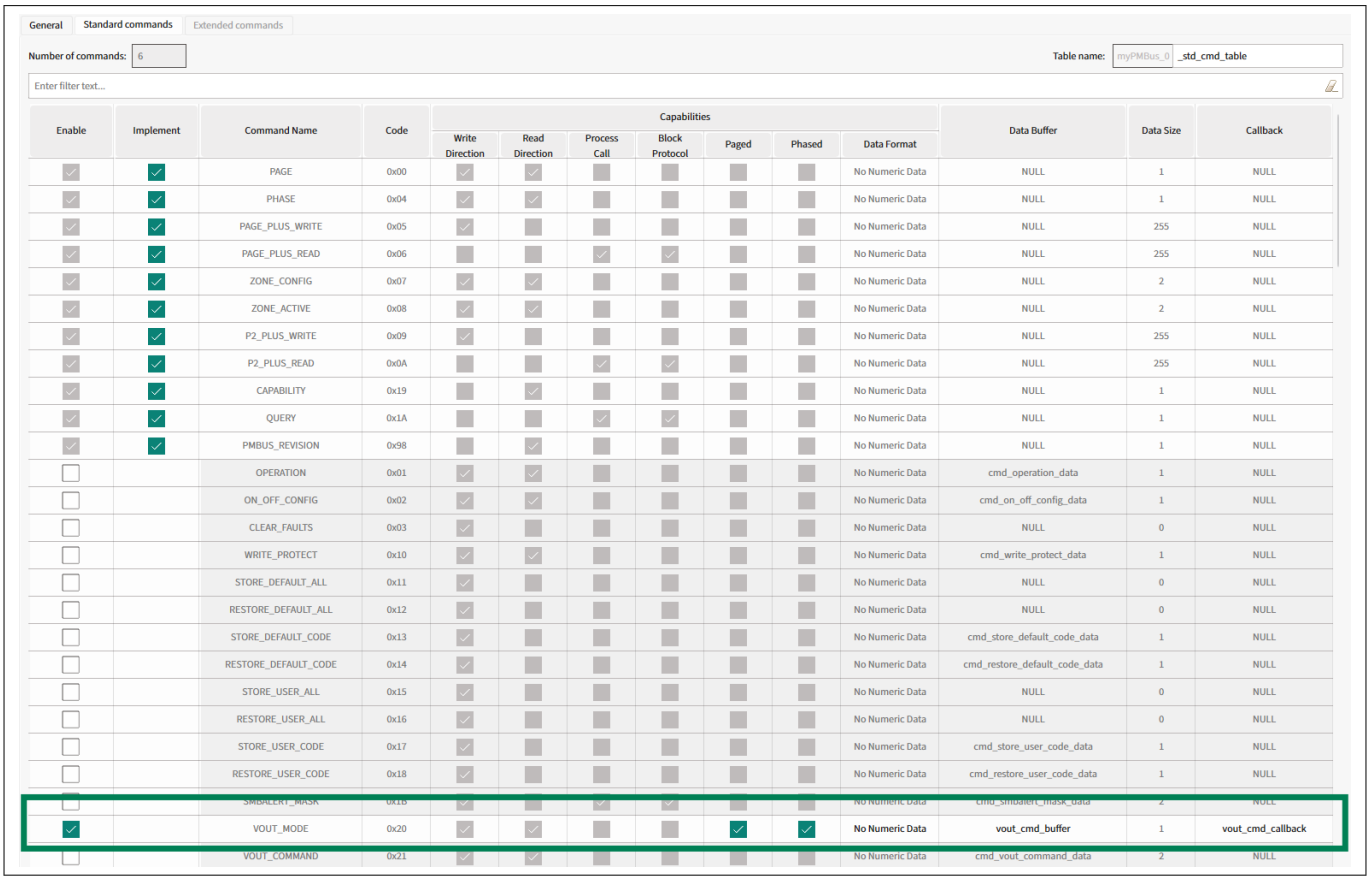

Figure 20 Custom command

Step 6: Implement callback function

```
bool vout_mode_callback(mtb_pmbus_cmd_events_t event,
                        int32_t page,
                        int32_t phase,
                        uint8_t byte)
{
    switch (event)
    {
        case MTB_PMBUS_CMD_MATCH:
            /* Command code received */
            printf("VOUT_MODE command received\n");
            break;

        case MTB_PMBUS_CMD_WRITE_BYTE:
            /* Data byte received - return true to ACK, false to NACK */
            printf("Received byte: 0x%02X\n", byte);
            return true;  /* Accept the byte */

        case MTB_PMBUS_CMD_READ_BYTE:
            /* Middleware will send data from buffer */
            break;

        case MTB_PMBUS_CMD_DONE:
            /* Transaction complete */
            printf("Transaction done on page %d\n", page);
            break;

        default:
            break;
    }
    return true;
}
```

References

[1] Infineon Technologies AG: *PSOC™ Control C3 MCU webpage*; Available online
[2] PMBus®: *PMBus® website*; Available online
[3] SMBus: *SMBus website*; Available online
[4] Infineon Technologies AG: *PMBus® API Reference Guide*; Available online

Revision history

Document revision	Date	Description of changes
**	2026-03-26	Initial release
*A	2026-09-11	Removed EAP instructions and updated with PMBus® GitHub links

Trademarks

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2026-09-11

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-izy1763397522698

Important notice

Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.

For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability. Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.

If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.

The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.

Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury.

However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer. Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG). If the Product includes security features:

Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.

If this document includes or references software:

The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved. Therefore, you may use the software only as provided in the software license agreement accompanying the software.

If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.

For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit www.infineon.com.