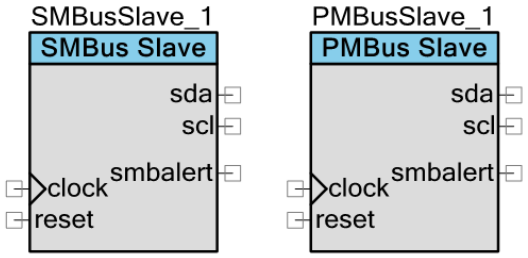
SMBus and PMBus Slave

1.0



Features

- SMBus Slave mode
- PMBus Slave mode
- SMBALERT# pin support
- 25 ms Timeout
- Fixed Function (FF) and UDB implementations
- Configurable SM/PM Bus commands

General Description

The System Management Bus (SMBus) and Power Management Bus (PMBus) Slave component provides a simple way to add an I^2C physical layer interface to a PSoC 3 or PSoC 5 design with either SMBus or PMBus protocol running on top of it.

The SMBus is a two-wire interface with various System Management chips that can communicate with the system host. It uses I^2C as a physical layer. The SMBus Slave component implements most of the SMBus Slave device specifications and provides options for configuring the slave device parameters. The slave device can communicate with the SMBus Master using the provided APIs.

The PMBus protocol is a specific implementation of the more generic SMBus protocol. With the PMBus, the component presents all the possible PMBus commands and allows you to select which commands are relevant to your application.

When to Use a SM Bus and PM Bus Slave

This component can be used in a design that requires an SMBus or PMBus slave communications interface. The component handles much of the physical layer requirements in the hardware. The firmware handles the protocol and memory buffer management; it also manages the data transfers to/from the I^2C.

Input/Output Connections

This section describes the various input and output connections of the SMBus Slave. An asterisk (*) in the list of I/Os states that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

clock – Input

The clock input should be used as the clock source for the I^2C SCL/SDA stuck low timeout timer. When the Implementation parameter is set to UDB, it needs a clock to provide 16 times of oversampling.

The clock input is available when the I^2C **Implementation** parameter is set to **UDB**.

Data rate	Clock
10 kbps	160 kHz
50 kbps	800 kHz
100 kbps	1.6 MHz
400 kbps	6.4 MHz
1000 kbps	16 MHz

reset – Input

Hardware reset for UDB I^2C implementation. If the active-high reset pin is held to logic high, the I^2C block is held in reset and communication over I^2C stops. SDA and SCL should be forced to high. This is a hardware reset only. Software must be independently reset using the Stop() and Start() APIs.

sda (SMBDAT) – Input/Output

Serial data (SDA) is the I^2C data signal. It is a bidirectional data signal used to transmit or receive all bus data. The pin connected to sda should be configured as Open-Drain-Drives-Low. If you select the "External I/O Option" in the customizer, the single sda bidirectional signal is replaced by a separate input and output (sda_o and sda_i). This is necessary to enable the multiplexing of multiple I^2C busses required by certain applications.

scl (SMBCLK) – Input/Output

Serial clock (SCL) is the master-generated I^2C clock. Although the slave never generates the clock signal, it may hold the clock low, stalling the bus until it is ready to send data or ACK/NAK the latest data or address. The pin connected to scl should be configured as Open-Drain-Drives-Low. If you select the "External I/O Option" in the customizer, the single scl bidirectional signal is replaced by a separate input and output (scl_o and scl_i). This is necessary to enable the multiplexing of multiple I^2C busses required by certain applications.

smbalert (SMBALERT#) – Output

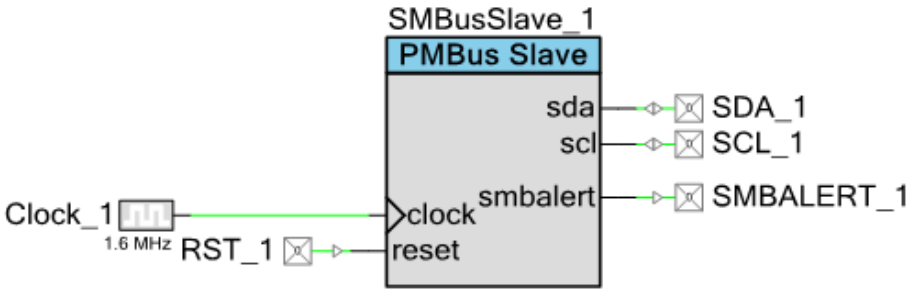
Alert is the optional SMBus defined SMBALERT# pin. This signal may be selectively enabled. The alert pin may be asserted/deasserted via APIs described later. The pin connected to smbalert should be configured as Open-Drain-Drives-Low.

Schematic Macro Information

By default, the PSoC Creator Component Catalog contains two schematic macro implementations for the SM/PM Bus Slave component. These macros contain already connected and configured pins and an internally configured clock value that defines data rate. The schematic macros uses I^2C Slave component configured to use fixed function I^2C block and hardware address decode.

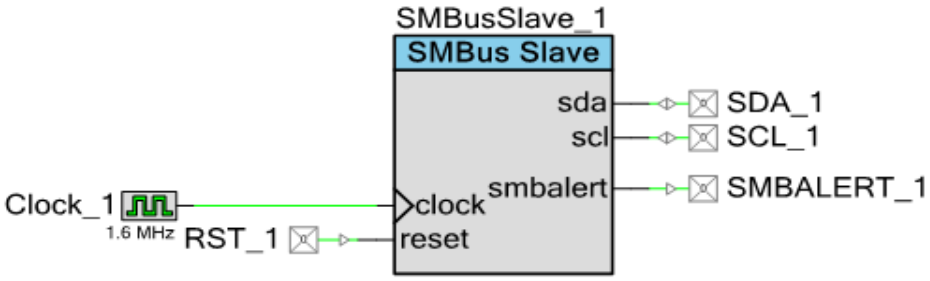
SM Bus Slave Macro

This macro provides the correct settings for the SM/PM Bus Slave component in SM Bus mode. Pins connected to terminals SCL and SDA are configured as bi-directional with Drive Mode set to Open Drain Drives Low. The component defines the data rate as 100 kHz.



PM Bus Slave Macro

This macro provides the correct settings for the SM/PM Bus Slave component in PM Bus mode. Pins connected to terminals SCL and SDA are configured as bi-directional with Drive Mode set to Open Drain Drives Low. The component defines the data rate as 400 kHz.



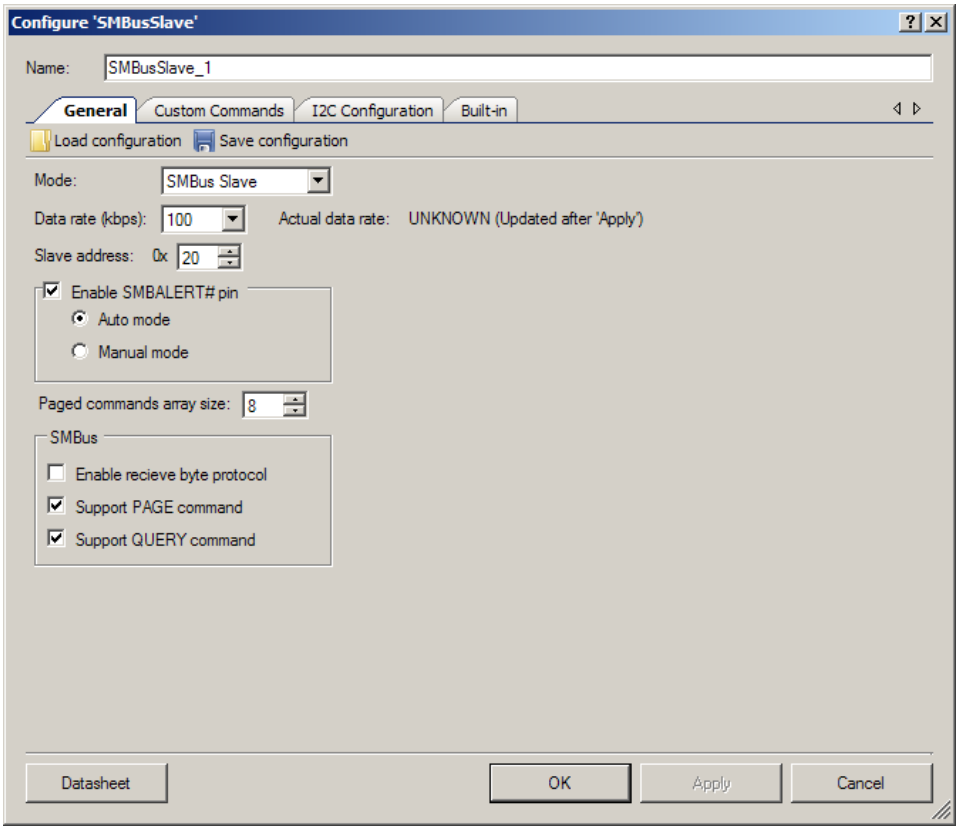
Component Parameters

Drag a SM/PM Bus Slave component onto your design and double click it to open the **Configure** dialog. By default, the **Configure** dialog initially displays the General Tab as shown in Figure 1.

General Tab

The **General** tab provides options to configure the general settings of the SM/PM Bus. The following parameters are available.

Figure 1. General SM Bus and PM Bus Slave Tab



Export/Import Configuration

The **Export/Import Configuration** allows you to save and restore the customizer settings to an external file. This allows for easy loading of preset profiles and retention of custom settings.

Mode

The **Mode** parameter selects the mode of the component to eitherSMBus Slave mode or PMBus Slave mode. If SMBus Slave is selected, then the PMBus Commands tab is disabled. The **Mode** also determines the available data rate. In PMBus Slave mode there are only two options: 100

and 400 kHz. In SMBus Slave mode, there are additional 10 and 50 kHz options. The default setting for this parameter is SMBus Slave mode.

Data Rate

The **Data Rate** parameter is used to select the I^2C data rate. The available options are dependent on the SM/PMBus**Mode** selection. The PMBus Slave mode allows **Data Rate** value of 100 kHz and 400 kHz. The SMBus Slave mode provides options for 10 kHz, 50 kHz, 100 kHz, and 400 kHz. The default value for the **Data Rate** parameter is 100 kHz.

Slave address

The **Slave address** parameter determines the I^2C address (7-bit format) of the device. The value is entered either in decimal or in hexadecimal (if preceded by "0x"). The customizer validates the address to ensure that it does not conflict with any of the addresses on the SMBus Slave reserved list. The default value for the **Slave address** is Default: 0x20.

SMBALERT

The **SMBALERT** box allows you to configure the optional SMBALERT# output pin for host notification.If **Enable SMBALERT# pin** is selected, the pin becomes visible as an output on the component symbol.The Auto/Manual buttons determine whether the **SMBALERT# pin** will automatically de-assert after the host queries the device at the Alert Response Address (GUI representation of the SetSmbAlertMode() API). The default for this parameter is set to EnabledAuto.

Paged Commands

This parameter box configures the PMBus PAGE command. The **Maximum page** parameter determines the array size for paged commands. All paged commands share this array size. The default is set to 8 pages. The **valid range is from 1 to 32.**

SMBus Box

The **SMBus** box configures the optional SMBus features. It is present only in SMBus Slave mode.

- The **Enable receives the byte protocol** check box enables/disables support for the SMBus Receive Byte protocol. If unchecked, any Receive Byte transaction is treated as a bus error. If checked, the component calls the SMBus_GetReceiveByteResponse() API to determine the response byte for Receive Byte requests.
- The **Support PAGE Command** check box allows you to have access to the PAGE command while in SMBusSlave mode.
- The **Support QUERY Command** checkbox allows you to have access to the QUERY command while in SMBusSlave mode.

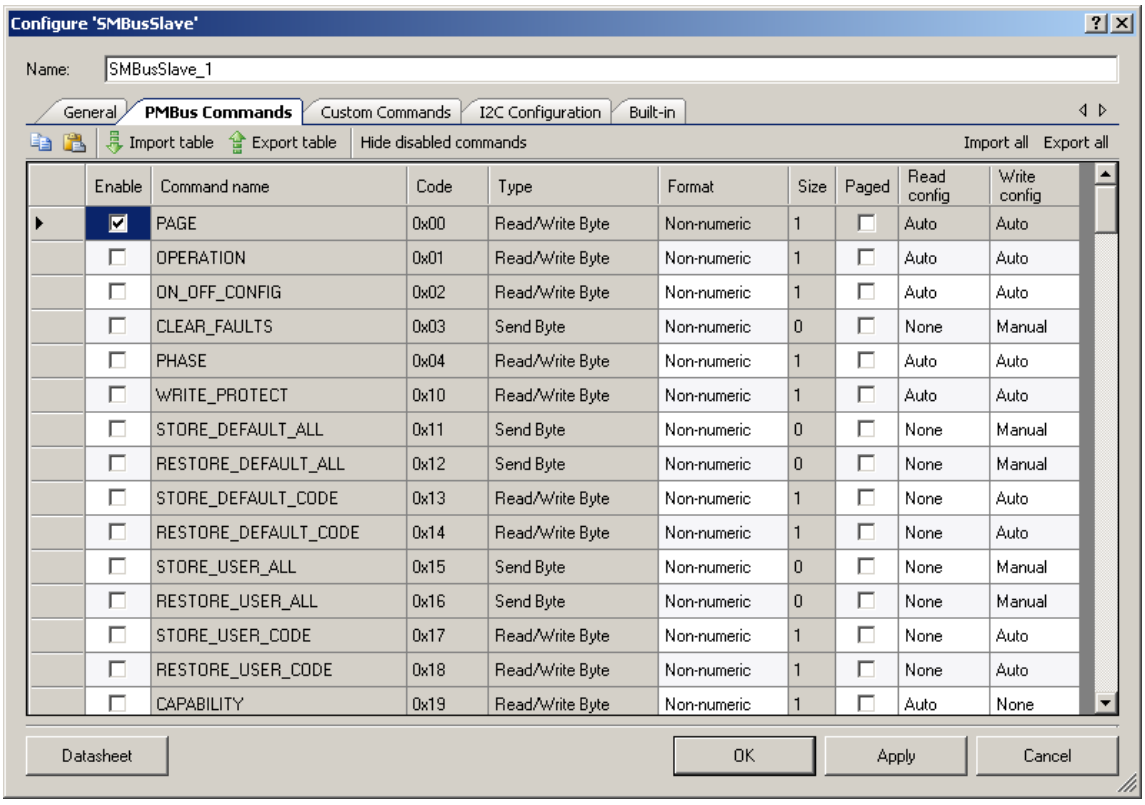
If either the PAGE or QUERY commands are enabled, then these commands are added to the command list on the **Custom Commands** tab. The properties for these commands are based on the PMBus specification, but full customization of the command codes is also possible.

The default values for this box are: **Enable receives byte protocol** unchecked, **Support PAGE Command** enabled, **Command Code**=0x00, **Support QUERY command** enabled, **Command Code**=0x1A.

PM Bus Commands Tab

The **PM Bus Commands** tab is available when the **Mode** of the component has been set to PMBus Slave. The tab presents the entire list of defined commands from the PMBus specification. The pre-filled information includes the command name, its numeric command code, and the type of command (i.e., the SMBus protocol). You may enable/disable the commands that you want the instance of the component to handle. Name, Code, and Type are Read-Only fields. The available parameters in this tab are outlined below.

Figure 2. PM Bus Commands SM Bus and PM Bus Slave Tab



Format

The **Format** parameter specifies the numeric format for this command. This format is used by the component in formulating the response to the QUERY command. The possible format values available in the QUERY command are Linear, Signed, Direct, Unsigned and VID Mode. This field

is only used for purposes of the QUERY command, as the component does not perform any actual numeric conversion.

Size

For Block and Process Call type commands, you may edit the **Size** field to specify the size of the data element. This size does not include the size/count byte that the SMBus protocol appends to the beginning of block transfers. This field can only be edited for Block or Process Call commands. For all other types, the **Size** field comes from the PMBus Specification. The default value is 16.

Read/Write Config

For each command, you select whether that command is readable and/or writeable via the **Read Config** and **Write Config** parameters. For each, you select None, Auto, or Manual. None indicates the action is disabled (i.e. set Write Config to None for a Read-Only command). "Auto" mode commands are handled entirely by the component. They are transferred between the register store and the I^2C transfer buffer without user firmware intervention or notification. "Manual" commands will be added to the Transaction queue and must be handled by user firmware. The default value for these parameters is "Manual".

Note: Because of the possible asymmetry between writes and reads and the complex nature of Process Call protocol, auto mode may not be selected for commands that use that protocol.

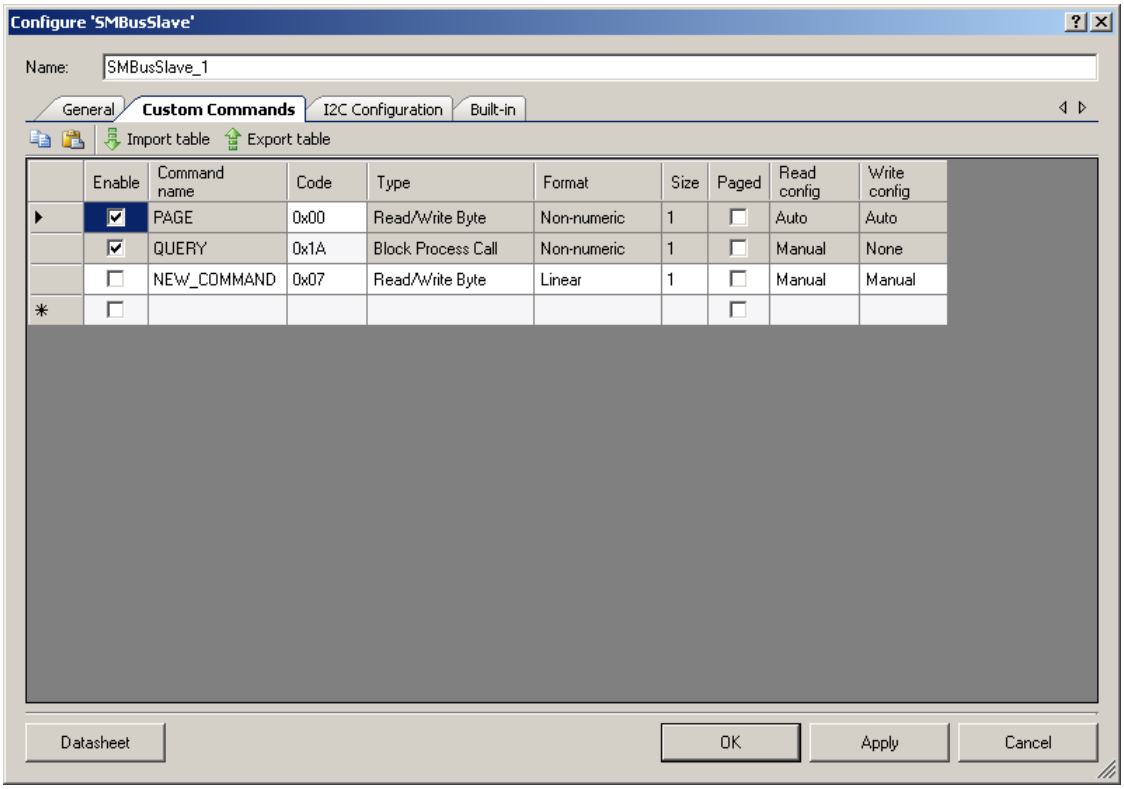
Paged

The **Paged** checkbox indicates whether this command is paged (i.e. indexed) or not. For commands that are specified as paged, the component automatically generates an array for that command in the register store. The size of the array is determined by the **Maximum Page** parameter. For Auto reads and writes, the component automatically indexes to the correct array member of a paged parameter based on the current PMBus page (as selected by the last PAGE command).The default setting for this parameter is unchecked.

Custom Commands Tab

This tab allows you to modify and customize the Command Name, Code, and Type fields.

Figure 3. Custom Commands SM Bus and PM Bus Slave Tab



Command Name

This is the user specified name for the command. Allowed characters are A-Z (all caps), 0-9, and the underscore "_". The maximum length is 24 characters. The first character may not be a number. Command name duplicates are not allowed (including custom command names that duplicate standard PMBus command names). The **Command Name** is blank by default.

Command Code

This is the numeric code for this command. It is a hexadecimal value, limited to two characters (0-9, ssA-F). Duplicate command codes are not allowed. This includes command codes that conflict with enabled PMBus commands. The **Command Code** is left blank by default.

Type

The **Type** specifies the SMBus transfer protocol/data-size used for this command. The possible values are Send Byte, Read/Write Byte, Read/Write Word, Read/Write Block, Process Call, and Block Process Call.It is set to Read/Write Byte as default.

Format

The **Format** parameter specifies the numeric format for this command. This format is used by the component in formulating the response to the QUERY command. The possible format values available in the QUERY command are Linear, Signed, Direct, Unsigned and VID Mode. This field is only used for purposes of the QUERY command, as the component does not perform any actual numeric conversion.

Read/Write Config

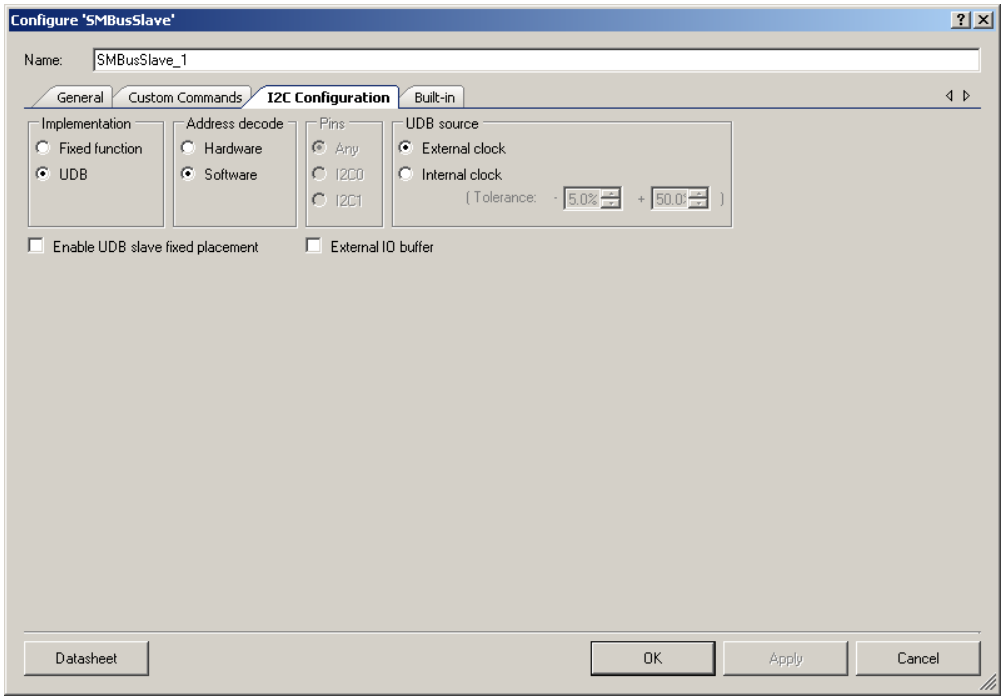
For each command, you select whether that command is readable and/or writeable via the **Read Config** and **Write Config** parameters. For each, you select None, Auto, or Manual. "None" indicates the action is disabled (i.e. set Write Config to None for a Read-Only command). "Auto" mode commands are handled entirely by the component. They are transferred between the register store and the I²C transfer buffer without user firmware intervention or notification. "Manual" commands will be added to the Transaction queue and must be handled by user firmware. The default value for these parameters is "Manual".

Note: Because of the possible asymmetry between writes and reads and the complex nature of Process Call protocol, auto mode may not be selected for commands that use that protocol.

I²C Configuration Tab

This tab allows you to configure the I²C hardware.

Figure 4. I²C Configuration SM Bus and PM Bus Slave Tab



Implementation

This parameter determines whether the I^2C hardware is implemented using **Fixed Function**[1] or **UDB**. The default mode is set to UDB.

Address decode

This parameter allows you to choose between software and hardware address decoding. For most applications where the provided APIs are sufficient and only one slave address is required, hardware address decoding is preferred. In applications where it is preferred to modify the source code to provide detection of multiple slave addresses or 10-bit addresses, software address detection must be used. Hardware is the default setting for this parameter. If hardware address decode is enabled, the block automatically NAKs addresses that are not its own without CPU intervention. It automatically interrupts the CPU on correct address reception, and holds the SCL line low until CPU intervention.

Pins

This parameter determines which type of pins to use for SDA and SCL signal connections. There are three possible values: Any, I2C0, and I2C1.The default is "Any". "Any" means general-purpose I/O (GPIO or SIO).

UDB clock source

This parameter allows you to choose between an internally configured clock and an externally configured clock for data rate generation. When set to Internal Clock, PSoC Creator calculates and configures the required clock frequency based on the **Data Rate** parameter, taking into account 16 times oversampling. In External Clock mode, the component does not control the data rate but displays the actual data rate based on the user-connected clock source. If this parameter is set to Internal Clock then the clock input is not visible on the symbol. You can enter the desired tolerance values for the internal clock. Clock tolerances are specified as a percentage. The default range is -5% to +50%.

Enable UDB slave fixed placement

The **Enable UDB slave fixed placement** parameter allows you to choose a fixed component placement that improves the component performance over unconstrained placement. If this parameter is set, all of the component resources are fixed in the top right corner of the device. This parameter controls the assignment of pins connected to the component. The choice of pin assignment is not a determining factor for component performance. This option is only valid if **Implementation** is set to UDB. This option is disabled by default. The fixed placement aspect of the component removes the routing variability. It also allows the fixed placement to continue to operate the same as a non-fixed placed design would in a fairly empty design.

---

[1] Fixed function is not applicable for selection when the component is used on PSoC5. PSoC5 I2C Fixed function is not able to recover from bus stock low due bus reset.

External IO Buffer

This parameter allows internal I^2C bus multiplexing. The internal OE buffer is removed and bidirectional scl and sda terminals are replaced with separate inputs (sda_i and scl_i) and outputs (sda_o and scl_o).Application Programming Interface Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name "SMBusSlave_1" to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is "SMBusSlave."

Note Some of component API functions are used in component ISR and, therefore, when building with Keil compiler may provoke a compiler warning prompting. To avoid this, js include these functions to the ".cyre" file.

Function	Description
SMBusSlave_Start()	Initializes and enables the SMBus component. The I^2C interrupt is enabled, and the component can respond to the SMBus traffic.
SMBusSlave_Stop()	Stops responding to the SMBus traffic. Also disables the interrupt.
SMBusSlave_EnableInt()	Enables the I^2C interrupt
SMBusSlave_DisableInt()	Disables the I^2C interrupt.
SMBusSlave_Init()	Initializes the I^2C registers with initial values provided from the customizer.
SMBusSlave_Enable()	Activates the I^2C hardware.Deosn't enable I^2C interrupt required for normal operation.
SMBusSlave_SetAddress()	Sets the I^2C slave address.
SMBusSlave_SetAlertResponseAddress()	Sets the I^2C slave address where the device will respond when in Alert Response Address Mode.
SMBusSlave_GetNextTransaction()	Returns a pointer to the next transaction record in the transaction queue. If the queue is empty, the function returns NULL.
SMBusSlave_GetTransactionCount()	Returns the number of transaction records in the transaction queue.
SMBusSlave_CompleteTransaction()	Causes the component to complete the currently pending transaction at the head of the queue.
SMBusSlave_SetSmbAlert()	Asserts or de-asserts the SMBALERT# smbalert pin.
SMBusSlave_SetSmbAlertMode()	Determines how the component responds to a SMBus master Read at the Alert Response Address.

Function	Description
SMBusSlave_HandleSmbAlertResponse()	Called by the component when the host responds to the Alert Response Address and the SMBALERT Mode is set to FIRMWARE_MODE.
SMBusSlave_GetReceiveByteResponse()	Called by the I^2C ISR to determine the response byte when it detects a "Receive Byte" protocol request.
SMBusSlave_HandleBusError()	Called by the component whenever a bus protocol error occurs.
SMBusSlave_StoreUserAll()	Saves the RAM Register Store to the User Register Store in Flash.
SMBusSlave_RestoreUserAll()	Verifies the CRC field of the User Register Store and then copies the contents of the User Register Store to the RAM Register Store.
SMBusSlave_RestoreDefaultAll()	Verifies the signature field of the Default Register Store and then copies the contents of the Default Register Store to the RAM Register Store.
SMBusSlave_StoreComponentAll()	Calls this function to update the parameters of other components in the system with the current PMBus settings.
SMBusSlave_RestoreComponentAll()	Calls this function to update the PMBus Operating Register Store with the current configuration parameters of other components in the system.
SMBusSlave_Lin11ToFloat()	Converts the argument "linear11" to floating point and returns it.
SMBusSlave_FloatToLin11()	Takes the argument "floatvar" (a floating point number) and converts it to a 16-bit LINEAR11 value (11-bit mantissa + 5-bit exponent), which it returns.
SMBusSlave_Lin16ToFloat()	Converts the argument "linear16" to floating point and returns it.
SMBusSlave_FloatToLin16()	Takes the argument "floatvar" (a floating point number) and converts it to a 16-bit LINEAR16 value (16-bit mantissa), which it returns.

Global Variables

Function	Description
SMBusSlave_initVar (static)	The initVar variable is used to indicate initial configuration of this component. This variable is prepended with the component name, in this case, SMBusSlave. The SMBusSlave_initVar variable is initialized to zero and set to 1 the first time SMBusSlaveStart() is called. This allows for component initialization without reinitialization in all subsequent calls to the SMBusSlave Start() routine. It is necessary to reinitialize the component when the device is going through sleep cycles. Therefore, the variable is set to zero when going into sleep, SMBusSlave Sleep(), and set during the reinitialization done in SMBusSlave Wakeup().

void SMBusSlave_Start(void)

Description: This is the preferred method to begin component operation. SMBusSlave_Start() calls the SMBusSlave_Init() function, and then calls the SMBusSlave_Enable() function. SMBusSlave_Start() must be called before I^2C bus operation. This API enables the I^2C interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SMBusSlave_Stop(void)

Description: This function disables I^2C hardware and interrupt. It releases the I^2C bus if it was locked up by the device and sets it to the idle state.

Parameters: None

Return Value: None

Side Effects: None

void SMBusSlave_EnableInt(void)

Description: This function enables the I^2C interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SMBusSlave_DisableInt(void)

Description: This function disables the I^2C interrupt. This function is not normally required because the I2C_Stop() function disables the i0nterrupt.

Parameters: None

Return Value: None

Side Effects: If the I^2C interrupt is disabled while the I^2C is still running, it can cause the I^2C bus to lock up.

void SMBusSlave_Init(void)

Description: This function initializes or restores the component according to the customizer Configure dialog settings. It is not necessary to call SMBusSlave_Init() because the SMBusSlave_Start() API calls this function, which is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: All registers will be set to values according to the customizer Configure dialog.

void SMBusSlave_Enable(void)

Description: This function activates the hardware and begins component operation. It is not necessary to call SMBusSlave_Enable() because the SMBusSlave_Start() API calls this function, which is the preferred method to begin component operation. If this API is called, SMBusSlave _Start() or SMBusSlave _Init() must be called first.

Parameters: None

Return Value: None

Side Effects: None

void SMBusSlave_SetAddress(uint8 address)

Description: This function sets the I^2C slave address.

Parameters: uint8 address: I^2C slave address for the primary device. This value can be any address between 0 and 127 (0x00 to 0x7F). This address is the 7-bit right-justified slave address and does not include the R/W bit.

Return Value: None

Side Effects: None

voidSMBusSlave_SetAlertResponseAddress(uint8 address)

Description: This function sets the I^2C slave address where the device will respond when in Alert Response Address Mode.

Parameters: uint8 address: I^2C slave address for Alert Response mode. This value can be any address between 0 and 127 (0x00 to 0x7F). This address is the 7-bit right-justified slave address and does not include the R/W bit.

Return Value: None

Side Effects: None

TRANSACTION_STRUCT* SMBusSlave_GetNextTransaction(void)

Description: This function returns a pointer to the next transaction record in the transaction queue. If the queue is empty, the function returns NULL. Only Manual Reads and Writes will be returned by this function, as the component will handle any Auto transactions on the queue. In the case of Writes, it is the responsibility of the user firmware servicing the Transaction Queue to copy the "payload" to the register store. In the case of Reads, it is the responsibility of user firmware to update the contents of the variable for this command in the register store. For both, call SMBusSlave_CompleteTransaction() to free the transaction record.

Note that for Read transactions, the length and payload fields are not used for most transaction types. The exception to this is Process call, where the Word from the write phase will be stored in the payload field.

Parameters: None

Return Value: Pointer the next transaction record

Side Effects: None

uint8 SMBusSlave_GetTransactionCount(void)

Description: Returns the number of transaction records in the transaction queue.

Parameters: None

Return Value: uint8: Number of records in the transaction queue

Side Effects: None

void SMBusSlave_CompleteTransaction(void)

Description: Causes the component to complete the currently pending transaction at the head of the queue. The user firmware transaction handler calls this function after processing a transaction. This alerts the component code to copy the register variable associated with the pending Read transaction from the register store to the I^2C transfer buffer so that the transfer may complete. It also advances the queue. Must be called for Reads and Writes.

Parameters: None

Return Value: None

Side Effects: None

void SMBusSlave_SetSmbAlert(uint8 assert)

Description: Asserts or de-asserts the SMBALERT# smbalert pin. As long as SMBALERT# is asserted, the component will respond to master READ's to the Alert Response Address. The response will be the device's primary slave address. Depending on the mode setting, the component will automatically de-assert SMBALERT#, call the SMBusSlave_HandleSmbAlertResponse() API, or do nothing.

Parameters: uint8: assert

SPDIF Status Masks	Type
SMBusSlave_SMBALERT_DEASSERT	Deassertsmbalert pin
SMBusSlave_SMBALERT_ASSERT	Assert a smbalert pin

Return Value: None

Side Effects: None

void SMBusSlave_SetSmbAlertMode(uint8 alertMode)

Description: This function determines how the component responds to a SMBus master Read at the Alert Response Address. When SMBALERT# is asserted, the SMBus master may broadcast a Read to the global Alert Response Address to determine which SMBus device on the shared bus has asserted SMBALERT#.

In Auto mode, SMBALERT# is automatically deasserted once the bus master successfully READ's the Alert Response Address.

In Manual mode, the component will call the API SMBusSlave_HandleSmbAlertResponse() where user code (in a merge section) is responsible for deasserting SMBALERT#.

In DO_NOTHING mode, the component will take no action.

Parameters: uint8: alertMode a byte that defines SMBALERT pin mode.

SPDIF Status Masks	Type
SMBusSlave_DO_NOTHING	Do nothing with SMBALERT# pin
SMBusSlave_AUTO_MODE	Automatically deassert SMBALERT# pin
SMBusSlave_FIRMWARE_MODE	User is respinsible for deasserting SMBALERT# pin

Return Value: None

Side Effects: None

void SMBusSlave_HandleSmbAlertResponse(void)

Description: This API is called by the component when the host responds to the Alert Response Address and the SMBALERT Mode is set to FIRMWARE_MODE. This function contains a merge code section where the user inserts code to run after the Master has responded. For example, the user might update a status register and de-assert the SMBALERT# pin.

Parameters: None

Return Value: None

Side Effects: None

uint8 SMBusSlave_GetReceiveByteResponse(void)

Description: This function is called by the I^2C ISR to determine the response byte when it detects a "Receive Byte" protocol request. This function includes a merge code section where the user may insert their code to override the default return value of this function – which is 0xFF. This function will be called in ISR context. Therefore, user merge code must be fast, non-blocking, and may only call re-entrant functions.

Parameters: None

Return Value: uint8: User-Specified status byte

SPDIF Status Masks	Type
SMBusSlave_RET_UNDEFINED	Default return status

Side Effects: None

void SMBusSlave_HandleBusError(uint8 errorCode)

Description: This API is called by the component whenever a bus protocol error occurs. Examples of bus errors would be: invalid command, data underflow, and clock stretch violation. This function is only responsible for the aftermath of an error since the component will already handle errors in a deterministic manner. This function is primarily for the purpose of notifying user firmware that an error has occurred. For example, in a PMBus device this would give user firmware an opportunity to set the appropriate error bit in the STATUS_CML register.

Parameters: uint8 errorCode:

SPDIF Status Masks	Type
SMBusSlave_ERR_READ_FLAG	Read Flag was incorrectly set
SMBusSlave_ERR_RD_TO_MANY_BYTES	Host attempts to read to many bytes
SMBusSlave_ERR_WR_TO_MANY_BYTES	Host attempts to write to many bytes
SMBusSlave_ERR_UNSUPORTED_CMD	Received command is unsupported
SMBusSlave_ERR_INVALID_DATA	Received data is invalid
SMBusSlave_ERR_TIMEOUT	Bus reset timeout occured
SMBusSlave_ERR_WR_TO_FEW_BYTES	Host attempts to write to few bytes

Return Value: None

Side Effects: None

uint8 SMBusSlave_StoreUserAll(char * flashRegs)

Description: This function saves the RAM Register Store to the User Register Store in Flash. The CRC field in the Register Store data structure is recalculated and updated prior to the save. This function does not perform storing anything to Flash by default. Instead it contains a merge region where user can implement an algorithm of storing Operating Memory to flash.

Parameters: flashRegs:

A pointer to a location in Flash where Operating Memory (RAM) should be stored.

Return Value: uint8: One of following standard return statuses

SPDIF Status Masks	Type
CYRET_SUCCESS	Action completed successfully
CYRET_MEMORY	Memory problem occured

Side Effects: None

uint8 SMBusSlave_RestoreUserAll(char * flashRegs)

Description: This function verifies the CRC field of the User Register Store and then copies the contents of the User Register Store to the RAM Register Store. This function does not perform restoring Register Store from Flash by default. Instead it contains a merge region where user can implement an algorithm of restoring data to Operating Memory (RAM).

Parameters: flashRegs:

A pointer to a location in Flash where Operating Memory (RAM) is

stored.

Return Value: uint8: One of following standard return statuses.

SPDIF Status Masks	Type
CYRET_SUCCESS	Action completed successfully
CYRET_BAD_DATA	Data is bad. CRC doesn't match

Side Effects: None

uint8 SMBusSlave_RestoreDefaultAll(void)

Description: This function verifies the signature field of the Default Register Store and then copies the contents of the Default Register Store to the RAM Register Store.

Parameters: None

Return Value: uint8: One of following standard return statuses.

SPDIF Status Masks	Type
CYRET_SUCCESS	Action completed successfully
CYRET_BAD_DATA	Data is bad. CRC does not match

Side Effects: None

uint8 SMBusSlave_StoreComponentAll(void)

Description: The user calls this function to update the parameters of other components in the system with the current PMBus settings. Because this action is very application specific, this function consists almost entirely of a merge section. The only component provided firmware is a return value variable (retval) which is initialized to CYRET_SUCCESS and returned at the end of the function. The rest of the function must be user provided.

Parameters: None

Return Value: uint8: One of following standard return statuses.

SPDIF Status Masks	Type
CYRET_SUCCESS	Action completed successfully

Or other user-determined non-SUCCESS status.

Side Effects: None

uint8 SMBusSlave_RestoreComponentAll(void)

Description: The user calls this function to update the PMBus Operating Register Store with the current configuration parameters of other components in the system. Because this action is very application specific, this function consists almost entirely of a merge section. The only component provided firmware is a return value variable (retval) which is initialized to CYRET_SUCCESS and returned at the end of the function. The rest of the function must be user provided.

Parameters: None

Return Value: uint8: One of following standard return statuses.

SPDIF Status Masks	Type
CYRET_SUCCESS	Action completed successfully

Or other user-determined non-SUCCESS status.

Side Effects: None

float SMBusSlave_Lin11ToFloat (uint16 linear11)

Description: This function converts the argument "linear11" to floating point and returns it.

Parameters: uint16 linear11: A number in LINEAR11 format.

Return Value: float: The linear11 parameter converted to floating point

Side Effects: None

uint16 SMBusSlave_FloatToLin11 (float floatvar)

Description: This function takes the argument "floatvar" (a floating point number) and converts it to a 16-bit LINEAR11 value (11-bit mantissa + 5-bit exponent), which it returns.

Parameters: float floatvar: A floating point number

Return Value: uint16: floatvar converted to LINEAR11

Side Effects: None

float SMBusSlave_Lin16ToFloat(uint16 linear16, int8 inExponent)

Description: This function converts the argument "linear16" to floating point and returns it. The argument Linear16 contains the mantissa. The argument inExponent is the 5-bit 2's complement exponent to use in the conversion.

Parameters: uint16 linear16: The 16-bit mantissa of a LINEAR16 number.

int8inExponent: The 5-bit exponent of a LINEAR16 number. Packed in the lower 5 bits. 2's Complement.

Return Value: float: The parameters converted to floating point

Side Effects: None

uint16 SMBus_FloatToLin16(float floatvar, int8 outExponent)

Description: This function takes the argument "floatvar" (a floating point number) and converts it to a 16-bit LINEAR16 value (16-bit mantissa), which it returns. The argument outExponent is the 5-bit 2's complement exponent to use in the conversion.

Parameters: floatfloatvar: A floating point number to be converted to LINEAR16.

int8outExponent: User provided 5-bit exponent to use in the conversion.

Return Value: uint16: The parameters converted to LINEAR16.

Side Effects: None

Bootloader Support

The SMBus and PMBus Slave component can be used as a communication component for the Bootloader. For more information about the Bootloader, refer to the "Bootloader System" section of the *System Reference Guide*.

The SMBus and PMBus Slave component provides a set of API functions for Bootloader use.

Function	Description
SMBusSlave_CyBtldrCommStart	Starts the SMBus and PMBus Slave component and enables its interrupt.

Function	Description
SMBusSlave_CyBtldrCommStop	Disables the SMBus and PMBus Slave component and disables its interrupt.
SMBusSlave_CyBtldrCommReset	Sets read and write I^2C buffers to the initial state and resets the slave status.
SMBusSlave_CyBtldrCommWrite	Allows the caller to write data to the bootloader host. This function manages polling to allow a block of data to be completely sent to the host device.
SMBusSlave_CyBtldrCommRead	Allows the caller to read data from the bootloader host. This function manages polling to allow a block of data to be completely received from the host device.

void SMBusSlave_CyBtldrCommStart(void)

Description: Starts the communication component and enables the interrupt. The read buffer initial state is full and the read always is 0xFFu. The write buffer is clear and ready to receive a command.

Parameters: None

Return Value: None

Side Effects: This function enables component interrupt. If I^2C is enabled without the interrupt enabled, it could lock up the bus.

void SMBusSlave_CyBtldrCommStop(void)

Description: Disables the communication component and disables the interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SMBusSlave_CyBtldrCommReset(void)

Description: Sets buffers to the initial state and reset the statuses. The read buffer initial state is full and the read always is 0xFFu. The write buffer is clear and ready to receive a command.

Parameters: None

Return Value: None

Side Effects: None

cystatus SMBusSlave_CyBtldrCommRead(uint8 * Data, uint16 size, uint16 * count, uint8 timeOut)

Description: Receives the command from the Host. All bytes are received by the I^2C ISR and stored in internal I^2C buffer. The function checks status with timeout to detemine the end of transfer and then copy data to bootloader buffer. After exist this function the I^2C ISR is able to receive more data.

Parameters: uint8 *Data: Pointer to storage for the block of data to be read from the bootloader host

uint16 size: Number of bytes to be read

uint16 *count: Pointer to the variable to write the number of bytes actually read

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information, see the "Return Codes" section of the *System Reference Guide*.

Side Effects: None

cystatus SMBusSlave_CyBtldrCommWrite(uint8 * Data, uint16 size, uint16 * count, uint8 timeOut)

Description: Transmits the status of executed command to the Host. The function updates the I^2C read buffer with response and releases it to the host. All reads return 0xFF till the buffer will be released. All bytes are transferred by the I^2C ISR. The function waits with timeout till all bytes will be read. After exit this function the reads return 0xFF.

Parameters: uint8 *Data: Pointer to the block of data to be written to the bootloader host

uint16 size: Number of bytes to be written

uint16 *count: Pointer to the variable to write the number of bytes actually written

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information see the "Return Codes" section of the *System Reference Guide*.

Side Effects: Temporary enables component interrupt when called on a start of clock stretching during "Bootloader Write" transaction but disables the interrupt before return from this function (to meet manual command handling behavior). When called not using "Bootloader Write" - always leaves component interrupt enabled.

Macros

- SMBusSlave_FL_ADDR_TO_ROW(addr) – Extracts Flash row number from specified address.
- SMBusSlave_FL_ADDR_TO_ARRAYID(addr) - Extracts Flash array ID from specified address.

- SMBusSlave_SIZE_TO_ROW(size) – Calculates and returns the number of Flash rows required to store the number of data defined by ***size***.
- SMBusSlave_MAX_PAGES – Specifies the maximum number of pages used by paged commands.
- SMBusSlave_NUM_COMMANDS – Defines the number of pages for paged commands.

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the "Find Example Project" topic in the PSoC Creator Help for more information.

Interrupt Service Routine

SMBus and PMBus Slave component uses two ISR for operation. First, ISR handles command decoding and data transactions over the SM/PM Bus. Second, is a 25 ms timeout interrupt that is designed to reset the bus when a stuck low condition occurs.

Functional Description

The theory of operation of this component is very similar to an I^2C Slave component. All references the SMBus specification refers to the SMBus specification version 2.0. Key functionalities of the component are highlighted in this section.

I^2C Physical Layer

The physical layer of the SM/Pm Bus slave component is based on the I^2C protocol. The three main differences that impact this component are:

The SMBus specification mandates that the component must reset and release the SCL and SDA lines if the SCL signal is detected stuck low for 25 ms. This is described in more detail in the AC & DC Electrical Performance Requirements section. This component also monitors the SDA line and resets if it is also stuck low for 25 ms as an extra precaution.

The SMBus Specification mandates that the component must not stretch the clock more than 25 ms (cumulative) in any given transfer. The slave is allowed to delay transfers when it is busy by pulling SCL low (clock stretching) provided that the cumulative stretch time in any transaction does not exceed 25 ms

Addition of an SMBALERT# pin to notify the host that the device needs attention

SMBus/PMBus Addressing

Every SMBus/PMBus slave device has an I^2C address. The following addresses are reserved for specific SMBus usage and must not be used as the generic slave address for a SMBus/PMBus slave.

Slave Address (Bits 7:1)	R/W# bit (Bit 0)	Comment
0000 000	0	General Call Address
0000 000	1	START byte
0000 001	X	CBUS address
0000 010	X	Reserved for different bus format
0000 011	X	Reserved for future use
0000 1XX	X	Reserved for future use
0101 000	X	Reserved for ACCESS bus host
0110 111	X	Reserved for ACCESS bus default address
1111 0XX	X	Reserved for 10-bit slave addressing
1111 1XX	X	Reserved for future use
0001 000	X	Reserved for SMBus Host
0001 100	X	SMBus Alert Response Address
1100 001	X	SMBus Device Default Address

If the user enables the SMBALERT# pin option, the component responds to its primary address and the Alert Response Address.

SMBus/PMBus Protocols

Nine different protocols are defined in the SMBus Specification. A summary of these protocols and their support statuses are shown below.

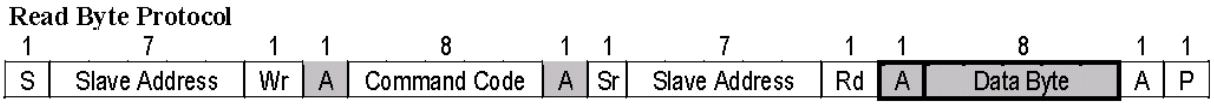
Protocol	Support Status	Comment
Quick Command	Not Supported	SMBus only
Send Byte	Supported	Both
Receive Byte	Supported	SMBus only
Write Byte/Word	Supported	Both
Read Byte/Word	Supported	Both
Process Call	Supported	Both
Block Write/Read	Supported	Both

Protocol	Support Status	Comment
Block Write/Block Read Process Call	Supported	Both
SMBus Host Notify Protocol	Not Supported	Both

Key concepts that characterize the SM/PM Bus component are as follows.

Neither SMBus nor PMBus use the concept of a sub-address or an I^2C register map. All transfers are command based. Immediately following the I^2C slave address is a command code. Commands can be write-only, read-only or read/write. In SMBus, the definition of the commands and their read/write restrictions are almost entirely up to the user. In PMBus, the commands are largely pre-defined but there are a range of command codes that are unassigned and available for "manufacturer-specific" implementations.

For read transactions, the host switches between writing the read command code to reading the requested data by issuing a REPEATED START bus condition. The STOP bit is only generated once the entire transaction is completed. An example is shown below:



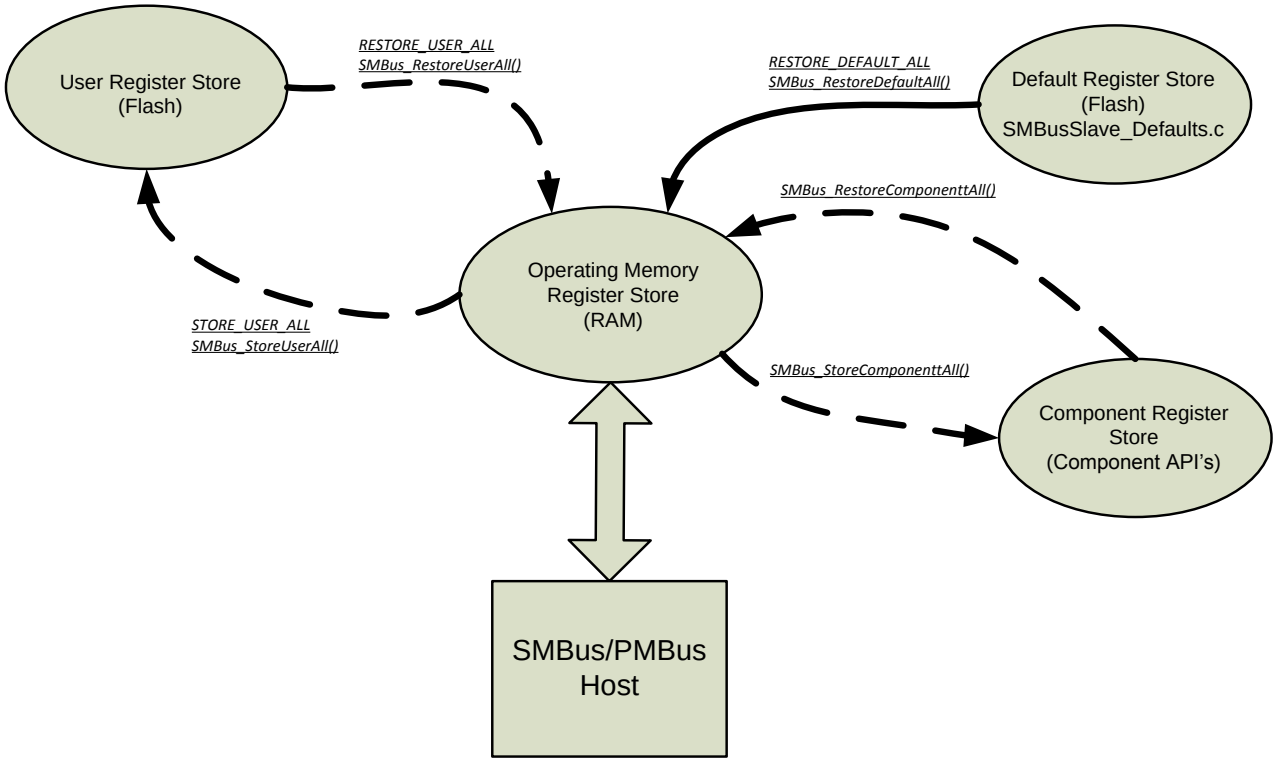
SMBus and PMBus are little-endian protocols (i.e. the least significant byte of a multi-byte data-type is transmitted first on the bus). It is a requirement that from the PSoC perspective, all registers be stored in the PSoC native endian format. The component is responsible for correctly translating endianess between PSoC and the I^2C buffer. This only applies to 16-bit word transfers.

The SMBus Specification does not define any of the command codes;they are up to the user to define. However, the PMBus Specification Part II, Appendix I does define all 255 possible command codes; 46 of these are up to the user to define (called manufacturer specific commands).

Register Stores Concept

According to the PMBus Specification II 5.4.2 – "Every Parameter That Can Be Written Must Be Readable. In general, any command that accepts a value for writing must also return that value when read. For this purpose, a concept of Register Stores is used.

Figure 5. Register Stores Concept



Operating Memory Register Store

This is the RAM version of the register store. Runtime SMBus/PMBus commands modify this version of the register store. Since this register store is located in RAM, its contents are assumed to be invalid at reset.

Default Register Store

A Flash version of the register store containing default values for all of the SMBus/PMBus parameters. At startup, the Default Register Store can be copied to the Operating Memory Register Store to initialize the Operating Register Store. The parameter values in the default register store are fixed at compile/link time. User is responsible to provide a default values for parameters in register store. To do open the SMBusSlave_Defaults.c, copy the parameters from commend block and paste them to the merge region below. If you do that they will be stored in SMBusSlave_regsDefault and every time on startup Operating Memory will be initialized with

these parameters. Another way to initialize Operating Memory with default values is to call SMBus_RestoreDefaultAll(). This function can be used in user handler for RESTORE_DEFAULT_ALL (0x12) PM Bus command.

User Register Store

This is another Flash version of the register store. Unlike the Default Register Store, which is essentially read-only, the User Register Store can be updated based on the current contents of the Operating Register Store. As storing data into Flash is very implementation specific, the component doesn't perform storing anything into Flash; it provides two API functions with merge regions that can be used for that purpose. SMBusSlave_StoreUserAll() does perform calculations of CRC required for checking data validness at restore.

To design your own method of storing data into a Flash, please refer to System Reference Guide that comes with PSoC Creator. Section 10 of System Reference Guide has basic information and also a description of functions that work with Flash. There are also a set of macros provided in the component that may be useful when working with Flash.

Component Register Store

The primary motivation for the component register store is to allow PMBus to extract parameters from and configure standard PSoC Creator components (thus, the name – Component Register Store). Creator components have their own configuration parameters, which are usually set up with the component customizer. These parameters are accessed at runtime via component specific SMBusSlave_StoreComponentAll()/SMBusSlave_RestoreComponentAll() API's. The component parameters accessible via the API's comprise the component register store. At startup, user PMBus firmware may want to:

- Update the other component settings based on PMBus parameters stored in User or Default register stores, or
- Update the PMBus register store based on the component settings.

Special case commands

PAGE command

The PMBus PAGE (Code 0x00) command (PMBus Part II – Section 11.10) allows the access of multiple logical PMBus devices at the same PMBus I^2C address. For example, a PMBus power supervisor that controls multiple power rails could provide access to the commands/parameters for each rail on its own page. The page can be thought of as an index into an array of commands/registers. Once the page is set via the PAGE command, the page setting is persistent until set again by another PAGE command.

In PMBus mode, the component has built-in support for the PAGE command. In SMBus mode, the user has the option to enable the PAGE command and specify the command code to use for PAGE.

If the PAGE command is enabled, the valid range of the PAGE command values are between 1 and 32, and must be within the **Max Page** setting determined by the user. The exception is the "All Pages" wild card setting, which is 0xFF. The "All Pages" wild card setting is only valid for write transactions, and must always be handled in "Manual" mode. If the PAGE is set to 0xFF, the following transactions are treated as errors.

An attempt to Read from a Paged command

An attempt to Write to a Paged command that is configured as Manual

Even in a PMBus device with multiple pages, some commands are not-dependent on the current page and are always handled in the same way regardless of the PAGE setting. For example, the PMBUS_REVISION command, which returns the PMBus version that the device supports, is common to all pages. Thus, there is a concept of Page commands and Common commands. For each SMBus/PMBus command, the customizer allows the user to specify whether the command is Paged or Common. When a command is marked as Paged, the component defines an array in the Register Store for that command.

QUERY Command

The PMBus QUERY (Code 0x1A) command is used to ask a PMBus device if it supports a given command, and if so, what data formats it supports for that command. In PMBus mode, the component has built-in support for the QUERY command. In SMBus mode, the user has the option to enable the QUERY command and specify the command code to use for QUERY.

This command uses the Block Write-Block Read Process Call described in the SMBus specification. For the write portion of the process call, the one data byte is an unsigned binary integer, the value of which is equal to the command code of the command being investigated. For the read portion of the process call, the one data byte is an unsigned binary integer with values as follows:

Bits	Value	Meaning
7	1	Command is supported
	0	Command is not supported
6	1	Command is supported for write
	0	Command is not supported for write
5	1	Command is supported for read
	0	Command is not supported for read
4:2	000	Linear Data Format used
	001	16 bit signed number
	010	Reserved
	011	Direct Mode Format used
	100	8 bit unsigned number

Bits	Value	Meaning
	101	VID Mode Format used
	110	Manufacturer specific format used
	111	Command does not return numeric data. This is also used for commands that return blocks of data.
1:0	XX	Reserved for future use

All of the information in the above table can be generated based on user selections in the Customizer. If the command code investigated with QUERY isn't supported then "0x00" is returned in this case.

Bootloader Commands

When the component is placed in a "Bootloader" project, two additional commands become available in the Configure dialog on the **Custom Commands** tab. These commands are BOOTLOAD_READ and BOOTLOAD_WRITE with default command codes of 0xFD and 0xFC, respectively. They add a capability to the component to act as a communication component for the Bootloader component.

These two commands interact with the bootloader component placed on a design schematic. After placing the bootloader component, select "Custom interface" in the bootloader component Configure dialog.

Resources

The resource figures for the fixed function implementation SM/PM Bus Slave component are listed below. The data was collected with data rate set to 400 kbps.

PSoC 3

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	I^2C Fixed Blocks	Interrupts
SM Bus / PM Bus(UDB)	3	32	2	6	-	2
SM Bus / PM Bus(Fixed Function)	2	9	1	4	1	2

PSoC 5

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	I²C Fixed Blocks	Interrupts
SM Bus / PM Bus(UDB)	3	32	2	6	-	2
SM Bus / PM Bus(Fixed Function)	N/A	N/A	N/A	N/A	N/A	N/A

PSoC 5LP

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	I²C Fixed Blocks	Interrupts
SM Bus / PM Bus(UDB)	3	32	2	6	–	2
SM Bus / PM Bus(Fixed Function)	1	5	2	2	1	2

API Memory Usage

The component memory usage varies significantly depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with an associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 5 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
SM Bus Slave (Fixed Function)	5501	200	N/A	N/A	3540	168
SM Bus Slave (UDB)	5587	196	3832	156	3824	156
PM Bus Slave (Fixed Function)	8166	1788	N/A	N/A	4632	4068
PM Bus Slave (UDB)	8080	1792	4924	4056	4916	4056

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

SMBus and PMBus Slave DC Specifications

Parameter	Description		Min	Typ[2]	Max	Units
I_{DD}	Component current consumption					
	SMBus (UDB)	DataRate = 50kbps	–	380	–	µA
	SMBus (Fixed Function)		–	670	–	µA
	PMBus (UDB)	DataRate = 100kbps	–	440	–	µA
	PMBus (Fixed Function)		–	620	–	µA
	SMBus (UDB)	DataRate = 400kbps	–	720	–	µA
	SMBus (Fixed Function)		–	860	–	µA
	PMBus (UDB)	DataRate = 400kbps	–	750	–	µA
	PMBus (Fixed Function)		–	980	–	µA

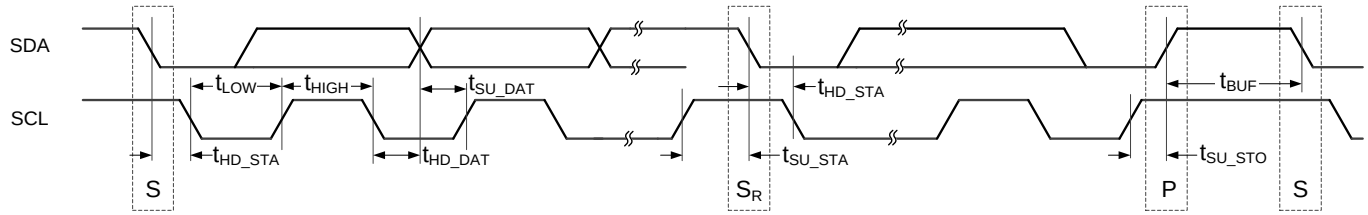
SMBus and PMBus Slave AC Specifications

Parameter	Description	Min	Typ	Max	Unit
f_{SCL}	SCL clock frequency	– – –	– – –	100 400 1000	kHz
f_{CLOCK}	Component input clock frequency	–	16 × f_{SCL}	–	kHz
t_{RESET}	Reset pulse width	–	2	–	t_{CY_clock}[3]
t_{LOW}	Low period of the SCL clock	4.7 1.3 0.5	– – –	– – –	µs
t_{HIGH}	High period of the SCL clock	4.0 0.6 0.26	– – –	– – –	µs
t_{HD_STA}	Hold time (repeated) start condition	4.0	–	–	µs

2. Device IO and clock distribution current are not included. The values are at 25 °C. Data was measured at BUS_CLK set to 24 MHz.
[3] $t_{CY_clock} = 1/f_{CLOCK}$. This is the cycle time of one clock period

Parameter	Description	Min	Typ	Max	Unit
		0.6 0.26	– –	– –	
t_{SU_STA}	Setup time for a repeated start condition	4.7 0.6 0.26	– – –	– – –	µs
t_{HD_DAT}	Data hold time	5.0 – –	– – –	– – –	µs
t_{SU_DAT}	Data setup time	250 100 50	– – –	– – –	ns
t_{SU_STO}	Setup time for stop condition	4.0 0.6 0.26	– – –	– – –	µs
t_{BUF}	Bus free time between a stop and start condition	4.7 1.3 0.5	– – –	– – –	µs
t_{FLASH_WRITE}		-	40.6[4]	-	ms

Figure 6. Data Transition Timing Diagram



[4] Value was obtained by storing a Register Store configured for PM Bus mode with all PM Bus commands enabled with their default configuration at BUS_CLK of 24 MHz.

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.0	First release	

© Cypress Semiconductor Corporation, 2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control, or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.