

TI Designs

Enhanced I²C and SMBus Master Interface Reference Design With PRU-ICSS



Description

TI provides the system solution for Industrial Communication on Sitara™ processors with programmable real-time unit and industrial communication subsystem (PRU-ICSS). PRU-ICSS allows custom firmware applications in the field of real-time applications. The I²C peripheral on many application processors does not support SMBus commands like block read and block write transfers. This TI Design implements those SMBus commands with standard I²C commands into the PRU-ICSS peripheral.

This TI Design supports:

- I²C and SMBus master interface with PRU-ICSS
- Dynamic block mode read and write transfer
- PRU-ICSS source code for customization (already in the design)

Features

- Enhanced I²C and SMBus Master Interface Example Implementation
- Validated With TMSIDK437X Industrial Development Kit (IDK) EVM
- I²C Register Interface Emulation

Applications

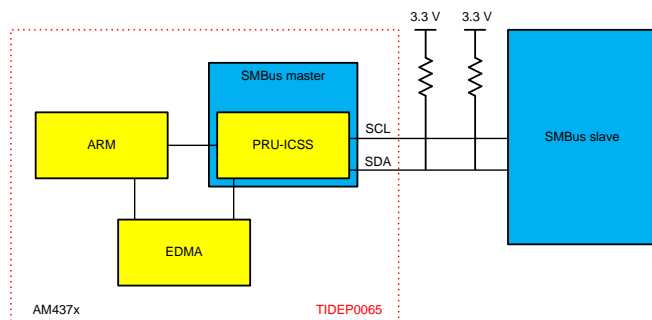
- Programmable Logic Controller (PLC)
- Industrial I/O Modules
- Industrial Sensor and Actuators
- Industrial Ethernet

Resources

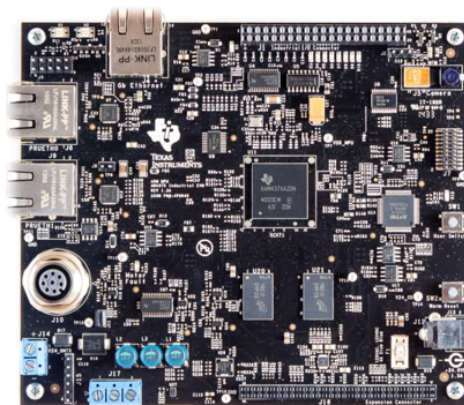
TIDEP0065	Design Folder
AM4379	Product Folder
TMSIDK437X EVM	Tools Folder



[ASK Our E2E Experts](#)



Copyright © 2016, Texas Instruments Incorporated



An IMPORTANT NOTICE at the end of this TI reference design addresses authorized use, intellectual property matters and other important disclaimers and information.

All trademarks are the property of their respective owners.

1 System Overview

1.1 System Description

1.1.1 I²C and SMBus

The system management bus (SMBus) is derived from an inter-integrated circuit (I²C) bus. Various system component chips and devices can communicate through it. The SMBus is a two-wire pair interface developed by Intel in 1995. It carries clock, data, and instructions through this interface, and it supports clock frequencies in the range of 10 to 100 kHz. The SMBus protocol is bidirectional master-slave communication protocol, which communicates half duplex because of a single data wire. The master communicates with the slave by a dedicated 7-bit slave address. The protocol frame formats that the SMBus system supports are Quick Command, Send Byte, Receive Byte, Write Byte/Word, Read Byte/Word, Block Write, and Block Read—all of these protocol commands are discussed in detail in [Section 2.1](#).

1.1.2 Difference Between SMBus and I²C Bus

The differences between the SMBus and I²C bus are:

- Both buses operate in the same way up to 100 kHz. The SMBus cannot operate beyond 100 kHz whereas the I²C bus support versions with 400 kHz and 2 MHz. Complete compatibility of both buses is only ensured if all devices operate at 100 kHz or below.
- The SMBus supports a timeout event whereas I²C has no such event. A timeout event is when the slave device resets its interface whenever clock stays longer than a certain period of time, typically 35 ms. In the I²C, the clock can go static for indefinite amount of time without the occurrence of a timeout.
- Timeout dictates the minimum clock speed specification, so the minimum clock speed for SMBus is 10 kHz. For the I²C, there is no such requirement and the clock speed can be DC.
- The electrical specification for the two buses are given in [Table 1](#):

Table 1. Electrical Specifications for I²C and SMBus

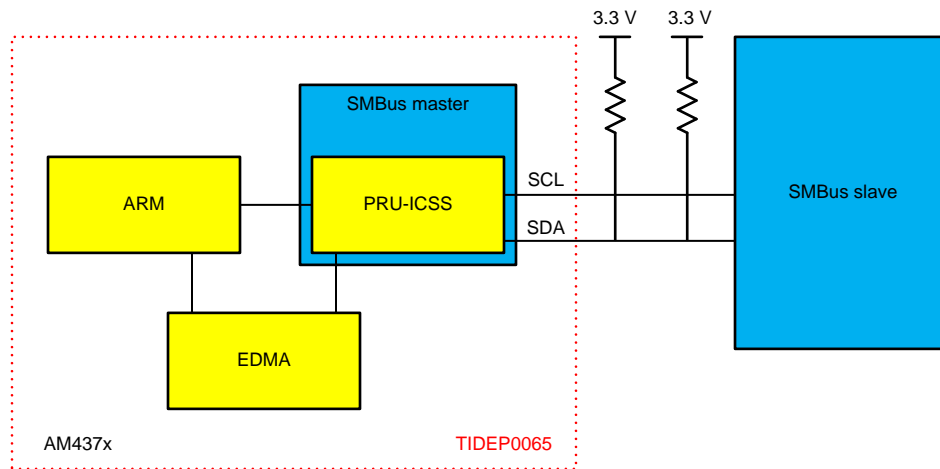
BUS	V-HIGH	V-LOW	I-MAX
I ² C Bus	3 V	1.5 V	3 mA
SMBus	2.1 V	0.8 V	350 μ A

A device is compatible with both buses if its V-high > 3 V and V-low < 0.8 V, and the pullup resistor values are in the range of 2.4 to 3.9 k Ω .

- I²C and SMBus both support the General Call, which is a special slave address (0b0000 000). All slave devices designed for General Call will respond accordingly. The general call is a mechanism by which communication with several slave devices simultaneously is possible.
- SMBus also supports Alert Response, which I²C does not. SMBus provides a line called ALERT Number, which acts as interrupt to the SMBus master. When the master receives this interrupt, it generates an Alert Response, which is sent to special slave address 0b0001100. Any slave device that generated the interrupt after getting this response puts its own address on the bus to identify itself. The master repeats this process until all generated interrupts are cleared.

1.2 Block Diagram

Figure 1 shows the block diagram of the SMBus system:



Copyright © 2016, Texas Instruments Incorporated

Figure 1. SMBus System Block Diagram

1.3 Highlighted Products

1.3.1 AM4379 Processor

Up to 1-GHz SitaraARM® Cortex®-A9 32-Bit RISC Processor

- NEON™ SIMD co-processor and vector floating point (VFPv3) coprocessor
- 32KB of L1 instruction and 32KB of data cache
- 256KB of L2 cache or L3 RAM
- 256KB of on-chip boot ROM
- 64KB of dedicated RAM
- Emulation and debug JTAG
- Interrupt controller

PRU-ICSS

- Supports protocols such as EtherCAT®, PROFIBUS, PROFINET, EtherNet/IP™, EnDat 2.2, and more
- Two PRU subsystems with two PRU cores each
- 32-bit load and store RISC processor capable of running at 200 MHz
- 12KB (PRU-ICSS1), 4KB (PRU-ICSS0) of instruction RAM with single-error detection (parity)
- 8KB (PRU-ICSS1), 4KB (PRU-ICSS0) of data RAM with single-error detection (parity)
- Single-cycle 32-bit multiplier with 64-bit accumulator
- Enhanced GPIO module provides shift-in and shift-out support and parallel latch on external signal
- 12KB (PRU-ICSS1 only) of shared RAM with single-error detection (parity)
- Three 120-byte register banks accessible by each PRU
- Interrupt controller module (INTC) for handling system input events
- Local interconnect bus for connecting internal and external masters to the resources inside the PRU-ICSS

- Peripherals inside the PRU-ICSS:
 - One UART port with flow control pins, supports up to 12 Mbps
 - One enhanced capture (eCAP) module
 - Two MII Ethernet ports that support industrial Ethernet, such as EtherCAT
 - One MDIO port

On-chip memory (shared L3 RAM)

- 256KB of general-purpose on-chip memory controller (OCMC) RAM
- Accessible to all masters

External memory interfaces (EMIF)

- DDR controllers:
 - LPDDR2: 266-MHz clock (LPDDR2-533 data rate)
 - DDR3 and DDR3L: 400-MHz clock (DDR-800 data rate)
 - 32-bit data bus
 - 2GB of total addressable space
 - Supports one x32, two x16, or four x8 memory device configurations
- General-purpose memory controller (GPMC)
 - Flexible 8-bit and 16-bit asynchronous memory interface with up to seven chip selects (NAND, NOR, Muxed-NOR, SRAM)
 - Uses BCH code to support 4-, 8-, or 16-bit ECC
 - Uses hamming code to support 1-bit ECC

See the AM4379 datasheet for a complete list of features ([SPRS851](#)).

1.3.2 AM437X IDK EVM Hardware Specification

- AM4379 ARM Cortex-A9
- 1GB DDR3, QSPI-NOR Flash
- Discrete power solution
- EnDat connectivity for motor feedback control
- 24-V power supply
- USB cable for JTAG interface and serial console

Software and tools

- SYS/BIOS real-time OS
- Starterware base port
- Code Composer Studio™ (CCS) integrated development environment (IDE)
- Application stack for industrial communication protocols
- Sample industrial applications

Connectivity

- PROFIBUS interface
- CANOpen
- EtherCAT
- EtherNet/IP
- PROFINET
- Sercos III
- IEC61850
- PWM
- Motor axis position feedback
- Up to 3-phase motor drive connector
- Sigma-delta decimation filter
- Digital inputs and outputs (I/O)
- SPI
- UART
- JTAG

See the AM437X IDK website for a complete list of features and design resources (<http://www.ti.com/tool/TMDXIDK437X>).

2 System Design Theory

2.1 SMBus Function and Frame Format

All frames of the SMBus protocol start the communication on the bus when the master asserts a start bit condition (S). The start bit is generated when the data line is pulled to a low state from a high state by master while the clock line is at a high state. The start bit is followed by 7 bits of slave address appended with 1 bit for the Read/Write direction. The slave sends an ACK (A) to acknowledge to the master that it was addressed by the slave address. After the ACK, a command code or data transmission of 8 bits can follow from master, which is also acknowledged by the slave. The communication is terminated when the master asserts a stop bit condition (P) on the bus. The stop bit is generated when the data line is pulled to a high state from a low state by master while the clock line is in a high state. Aside from the start and stop bits, all transitions on the data line happen during the duration of the clock when it is low. [Figure 2](#) shows example of a Quick Command frame with each individual bit.

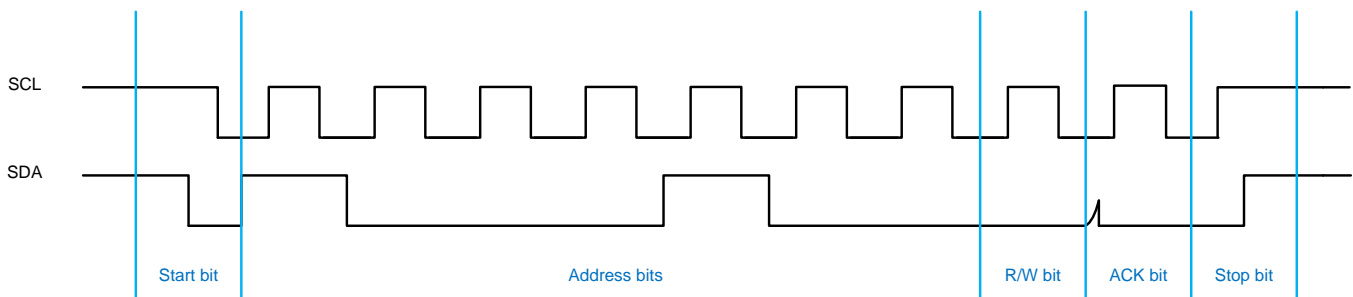


Figure 2. SMBus Frame Format Example Bit by Bit

The following subsections describe the SMBus protocol frames. In each figure, S is the START bit, Sr is the REPEATED START, A is the acknowledge (ACK) bit, N is the no-acknowledge (NACK) bit, and P is the STOP bit. The blue portions represent communication direction by the master to the slave, and the red portions represent communication direction by the slave to the master.

2.1.1 Quick Command

In a Quick Command, there is no data sent or received. Quick Command can be used to turn on or off a device or some other basic feature in the slave device. Quick Command is good for small devices that provide limited support for SMBus. In Quick Command, the Read/Write bit denotes the command, for example, to turn on or off a device. The frame format of a Quick Command is given in [Figure 3](#):

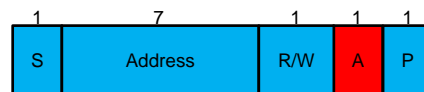


Figure 3. Quick Command Format

2.1.2 Send Byte

With the Send Byte, a slave device can accept a maximum of 256 possible encoded commands, which is sent by the master to the slave in the form of a data byte after the slave address. The frame format of a Send Byte is given in [Figure 4](#):

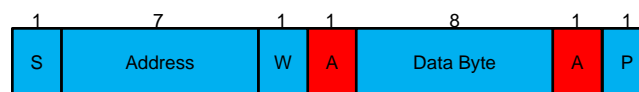


Figure 4. Send Byte Format

2.1.3 Receive Byte

Receive Byte is similar to Send Byte. The difference is in the direction of the data byte and that the NACK from the master precedes the stop bit. With the Receive Byte, a slave device can send information to the master. The frame format of a Receive Byte is given in Figure 5:

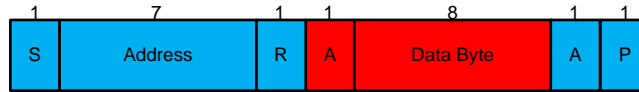


Figure 5. Receive Byte Format

2.1.4 Write Byte

In a Write Byte, the first byte after the slave address is the command code. The next byte contains the data to be written. After every received byte, the slave acknowledges the transfer. The stop bit indicates the end of transfer. The frame format of a Write Byte is given in Figure 6:

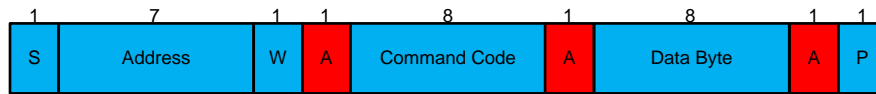


Figure 6. Write Byte Format

2.1.5 Read Byte

Reading data is slightly different than writing data because a change in direction of transfer is required. First, the master writes the command code to slave. After the command code, the master always sends a repeated start condition followed by the slave address with the read bit to indicate that now it wants to read the data byte from the slave. The slave returns a data byte to the master. The master then sends a NACK to indicate the end of transfer followed by the stop bit. The format of a Read Byte is given in Figure 7:



Figure 7. Read Byte Format

2.1.6 Write Word

In Write Word, the first byte after the slave address is the command code. The next two bytes contains the data bytes to be written. After every byte, the slave acknowledges the transfer. The stop bit generated by the master indicates the end of transfer. The frame format of Write Word is given in Figure 8:

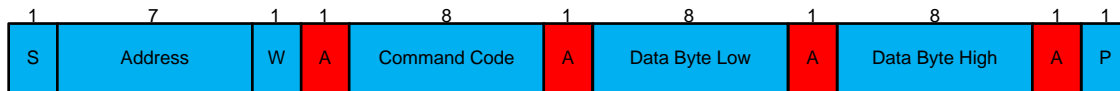


Figure 8. Write Word Format

2.1.7 Read Word

Read Word is same as Read Byte with the difference being instead of one byte, the slave transmits two bytes to the master. The frame format of Write Read is given in [Figure 9](#):

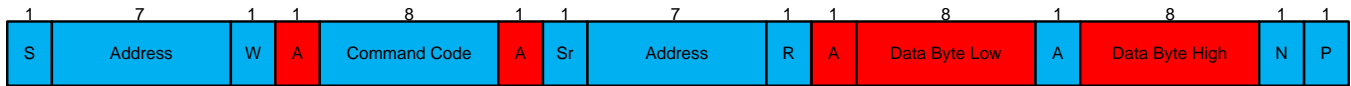


Figure 9. Read Word Format

2.1.8 Block Write

Block Write starts with the start bit followed by slave address with the write bit. Next, the command code is sent followed by block count (N), which specifies the number of bytes that follow in the message. After N data bytes, communication terminates with stop bit. The frame format of Block Write is given in [Figure 10](#):

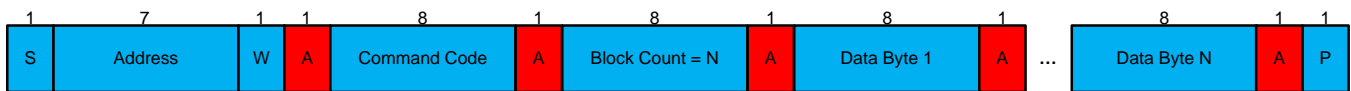


Figure 10. Block Write Format

2.1.9 Block Read

Block Read is different from the Block Write as the direction changes in the middle of communication, which is introduced by the repeated start condition. After all data bytes are sent, the master sends the NACK to end the transfer. The frame format of Block Read is given in [Figure 11](#):

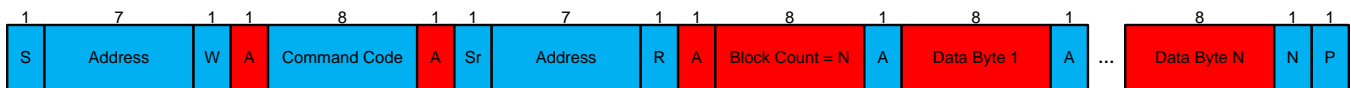


Figure 11. Block Read Format

2.2 PRU Hardware Description for I²C and SMBus Master

- PRU1 of ICSS0 is used as the SMBus master.
- The SCL signal is exposed at J16, pin 19 [cam1_data5]. This uses AM437X pinmux mode 0x4 [pr0_pru1_gpo13].
- The SDA signal is exposed at J16, pin 17 [cam1_data3]. This uses AM437X pinmux mode 0x4 [pr0_pru1_gpo13].
- PRU register R30.b13 is used for SCL output.
- PRU register R30.b11 is used as SDA output.
- PRU register R31.b11 is used as SDA input.
- Bit patterns in SMBus frames are generated using the bit banging technique and the timings of SCL and SDA are generated by PRU delay cycles.
- When the I²C and SMBus is in an idle state, the SMBus master has set the SDA pin multiplex value to input. The external pullup sets the SDA signal state to '1'.
- [Figure 12](#) shows how the PRU hardware interface is connected to the slave device:

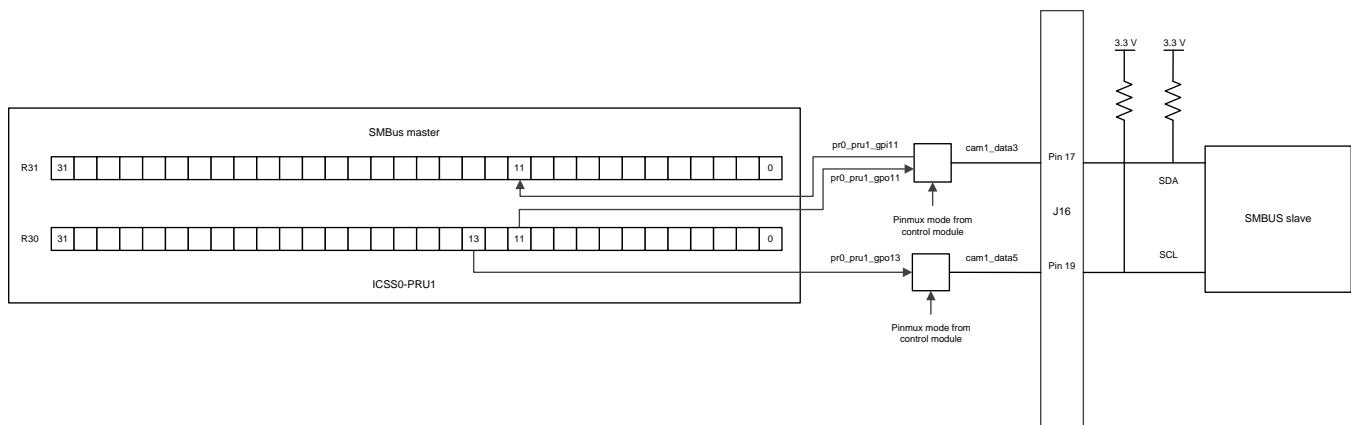


Figure 12. PRU Hardware Connection to Slave

2.3 PRU Firmware Description

The PRU firmware is divided into different modules that are called by the main program. The main program first calls a module for initializing the communication link and setting the correct pin multiplex value for the SDA line. After that, the PRU firmware waits for an interrupt signal from the ARM to execute a specific SMBus operation. When the PRU firmware receives the interrupt signal from the ARM, the firmware clears the access ready bit (ARDY) in the shared register interface so that the ARM cannot access the shared interface while PRU is performing the SMBus operation. After that, the PRU firmware checks the type of SMBus operation that is required to get performed. The PRU firmware then proceeds with executing the requested SMBus operation. After executing the SMBus operation, results are updated by PRU firmware in the control and status registers in shared memory, and the ARDY bit is set to allow ARM access to the shared register interface. Then the PRU firmware generates the interrupt for ARM to let it know that the SMBus operation has been performed, and the PRU firmware goes back to wait for the next interrupt signal from ARM.

2.3.1 PRU Firmware Flowchart

The PRU software functionality as previously described is shown in [Figure 13](#):

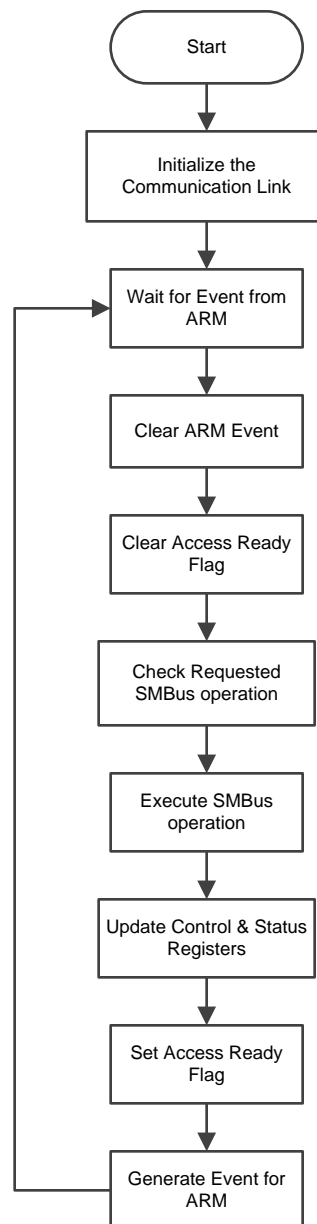


Figure 13. PRU Firmware Flowchart

2.3.2 PRU Firmware Functions

The PRU firmware is divided into the following functions:

1. Init_Comm_Link
2. Check_Set_Pinmux
3. Start_Bit_Generation
4. Write_One_Byte_To_Slave
5. Check_ACK_TX
6. Stop_Bit_Generation
7. Repeated_Start_Bit_Condition
8. Read_One_Byte_From_Slave
9. Check_ACK_RX
10. Stop_Bit_Generation_RX
11. Master_Send_ACK_To_Slave
12. Master_Send_NACK_To_Slave

The SMBus protocols are essentially made up by these building blocks that are called and repeated in this order. For instance, [Figure 14](#) shows the SMBus Quick Command frame calling the following modules:

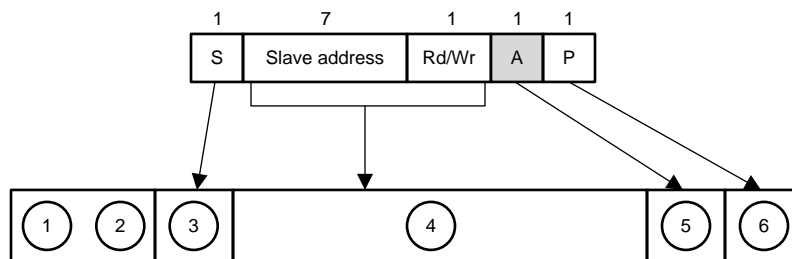


Figure 14. Example of Used Modules in Quick Command

2.3.3 Definition of PRU Action Points

This section describes PRU action points, that is when the PRU firmware functions takes action in an I²C and SMBus clock cycle. Action point examples are the triggering of the EDMA to switch the I/O mode of the SDA signal or take a sample value on SDA signal for determining whether there was an acknowledgment sent by the slave device.

Figure 15 shows these action points with respect to the clock cycle. Action points (1) and (3) show the points where most of the PRU functions trigger the EDMA to change the multiplex value of the SDA pin. Action point (2) is where functions Check_ACK_TX and Check_ACK_RX take a sample from the SDA line to determine whether there was an acknowledgment from the slave device. Action point (4) is used for EDMA triggering to change the multiplex value of SDA pin—this depends on functions Write_One_Byte_To_Slave, Check_ACK_TX, and Check_ACK_RX, all which decide the value based on the last sent bit to the slave device. The details about the functionality of PRU functions are given in the following Section 2.3.4, which uses one of these points.

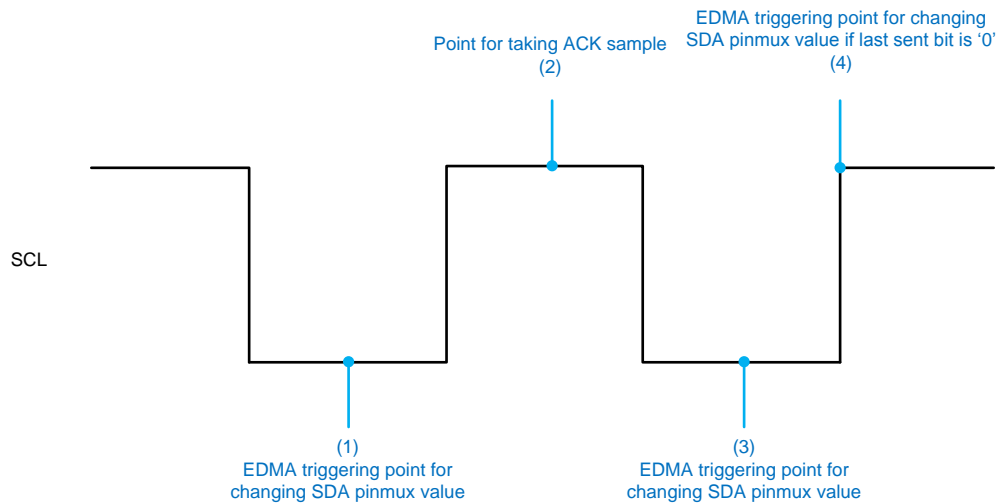


Figure 15. Action Points of Clock Cycle

2.3.4 PRU Firmware Functions Description and Flowchart

The following subsections describe each function of PRU firmware in detail.

2.3.4.1 Init_Comm_Link

This PRU function is responsible for initializing the delay values to generate clock according to the user selected CLK frequency. The module also initializes some of the PRU registers that are used in the PRU firmware. [Figure 16](#) shows the functionality of this module.

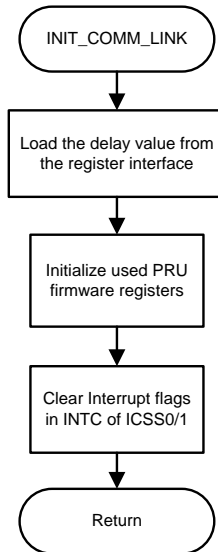


Figure 16. Init_Comm_Link Flowchart

2.3.4.2 Check_Set_Pinmux

This function makes sure that the SDA pin multiplex mode is set to input (mode 5). If the entry multiplex mode is output (mode 4), this module generates the EDMA event to trigger the multiplex value change from output to input. [Figure 17](#) shows the functionality of the module in the form of flowchart.

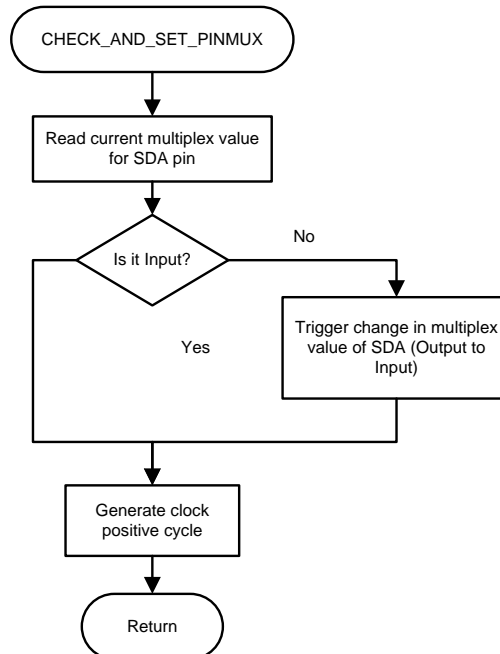


Figure 17. Check_And_Set_Pinmux Flowchart

2.3.4.3 Start_Bit_Generation

This module is used to generate the start bit condition on the I²C and SMBus link. The SDA pin is the input to this function, but on exit it is changed to the output.

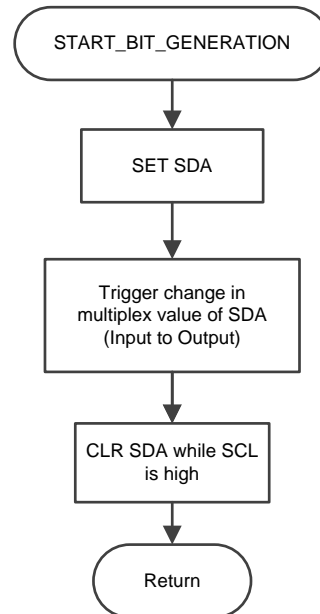


Figure 18. Start_Bit_Generation Flowchart

2.3.4.4 Write_One_Byte_To_Slave

This function is used to write one byte to the slave device. That one byte can be the 7-bit slave address appended with 1 bit for read or write mode, or the byte can be either a command code, block count, or data byte.

This function writes the data byte serially on the SDA signal using a shift register approach. It sets or clears the SDA line whether the MSB of the PRU shift register is '1' or '0'. After the SDA output, the function shifts the register to the left by one bit. This is repeated until all 8 data bits are written onto the SDA line. After the data byte has been written, the function checks whether the last bit on SDA was a '1' or a '0'. If it was a '1', then the function triggers EDMA event to change the multiplex value of the SDA pin from output to input. Setting the SDA pin to input is done to reduce undesired spike generation on the SDA line between the SMBus master and the slave device.

Figure 19 shows the described function.

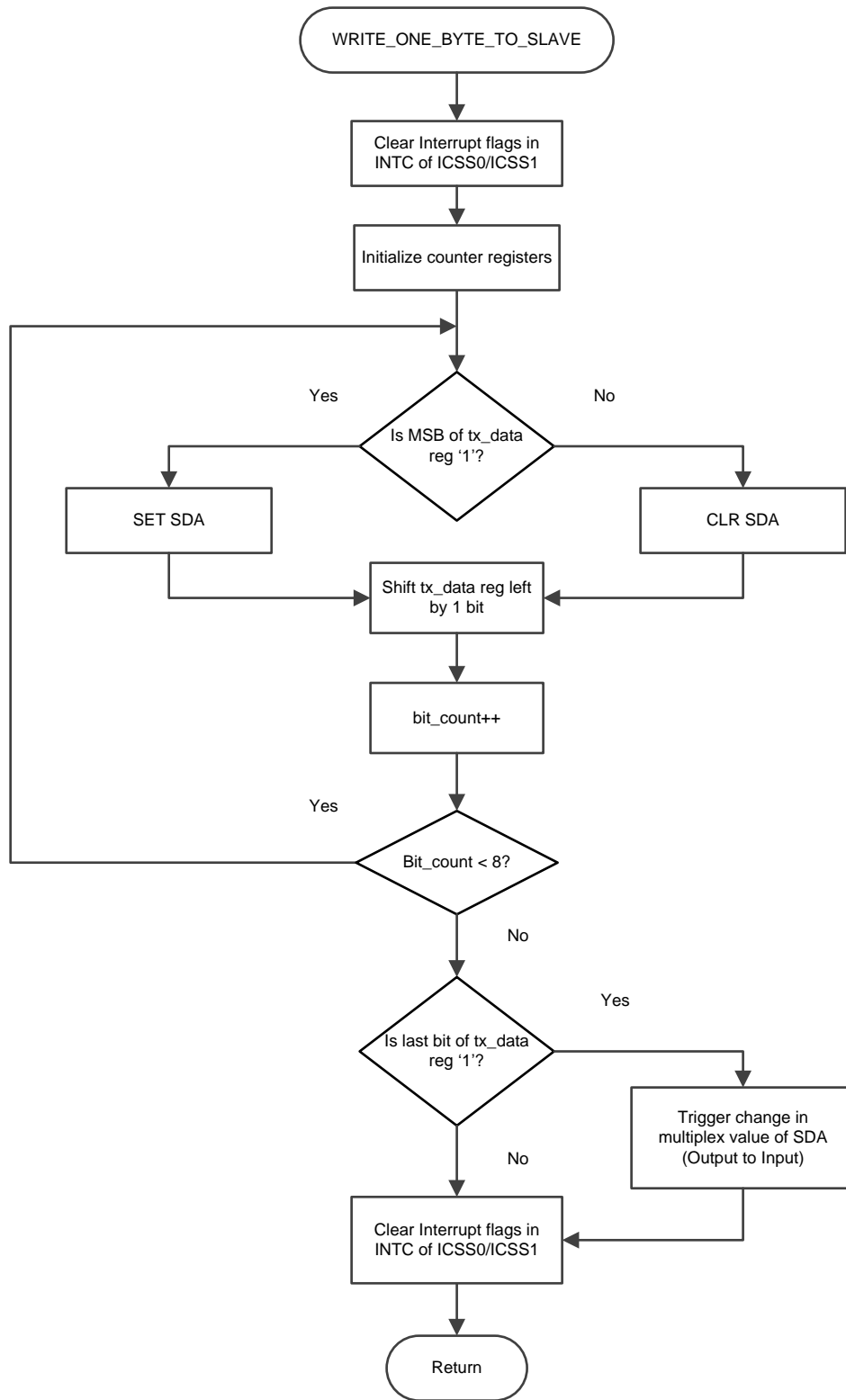


Figure 19. Write_One_Byte_To_Slave Flowchart

2.3.4.5 Check_ACK_TX

This function is used to check if an acknowledgment bit has been generated by the slave device after a byte has been written by the SMBus master. The function takes the acknowledge bit sample at the middle of the positive clock cycle.

The SDA pin multiplex value upon entry to the function can be input or output—this depends whether the EDMA event was triggered at the end of the function Write_One_Byte_To_Slave. This function sets on exit the SDA pin to output. Figure 20 shows the functionality of this function.

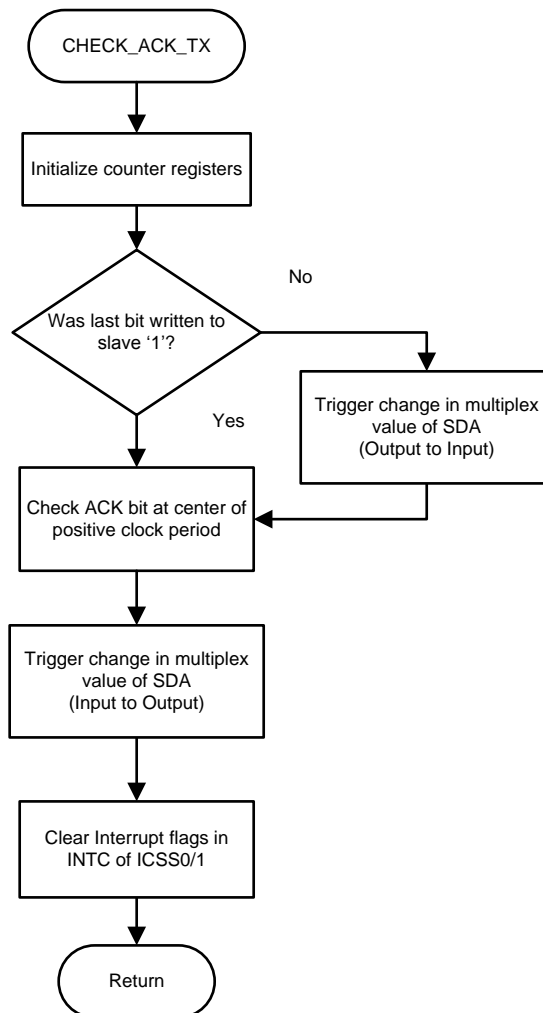


Figure 20. Check_ACK_TX Flowchart

2.3.4.6 Stop_Bit_Generation

This function is used to generate the stop bit condition on the bus. It sets the SDA signal to '1' while the CLK is still at '1'. Upon function entry, the SDA pin multiplex value is output and it is changed to input upon exit. [Figure 21](#) shows how this function behaves.

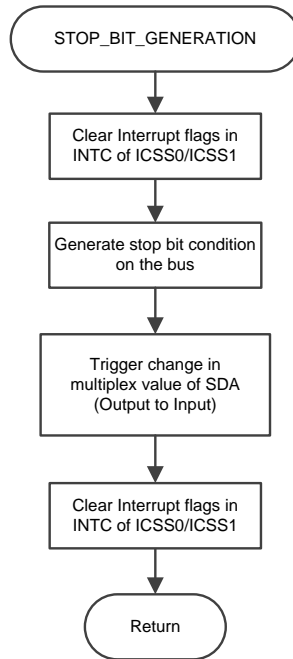


Figure 21. Stop_Bit_Generation Flowchart

2.3.4.7 Repeated_Start_Bit_Condition

This function is used by the SMBus read commands because a change in the direction of data flow is needed in the middle of SMBus frame. Upon function entry the SDA pin multiplex value is input and is changed to output upon exit. [Figure 22](#) shows this functionality.

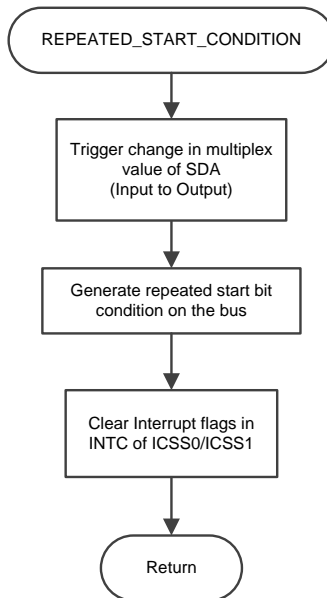


Figure 22. Repeated_Start_Condition Flowchart

2.3.4.8 Read_One_Byte_From_Slave

This function reads bits from the SDA line and saves the bits in the LSB of the rx_data register. This function also uses the shift register to accumulate 8 bits to 1 byte. The value of the SDA pin multiplex is not changed in this function and it remains input upon entry and exit to this function.

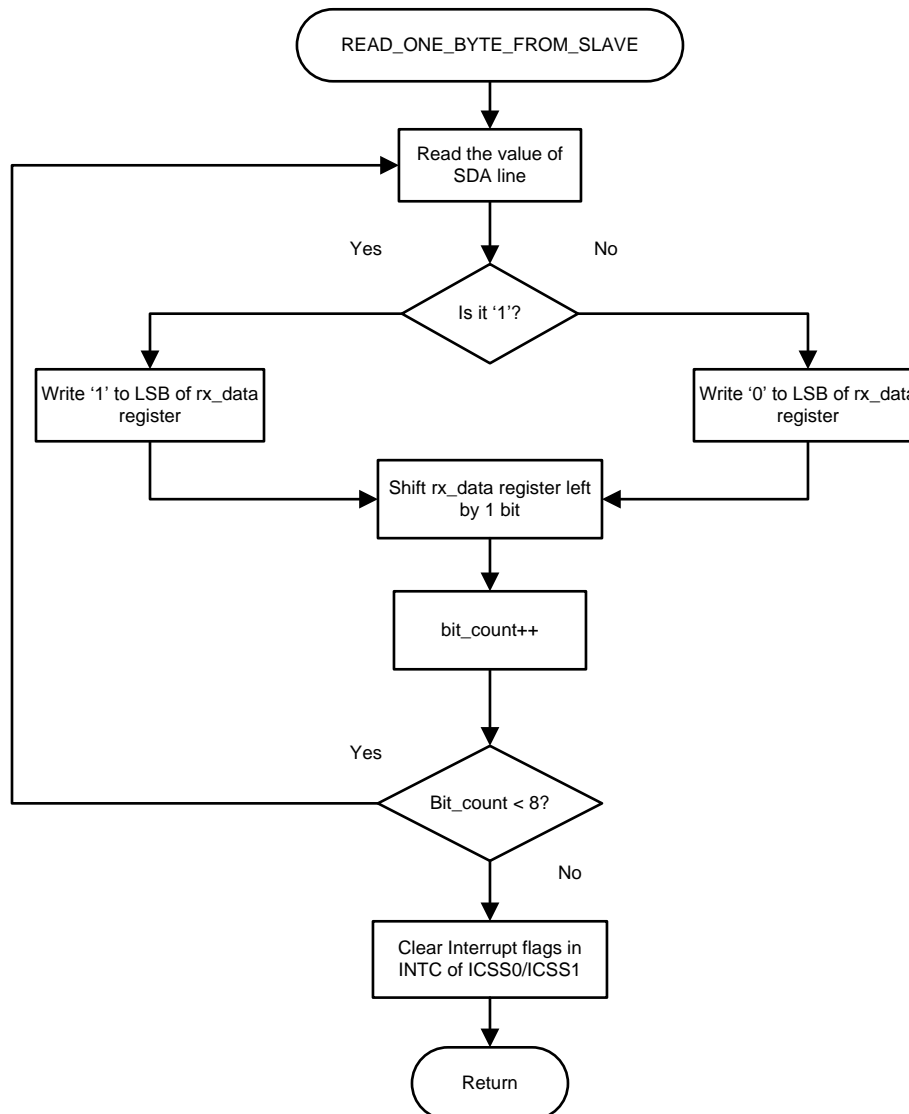


Figure 23. Read_Byte_From_Slave Flowchart

2.3.4.9 Check_ACK_RX

This function is similar to the Check_ACK_TX function. The difference is only in terms of EDMA triggering to change the value of the SDA pin multiplex. This function makes sure that the value of the SDA pin multiplex is input upon exit. Figure 24 shows this behavior.

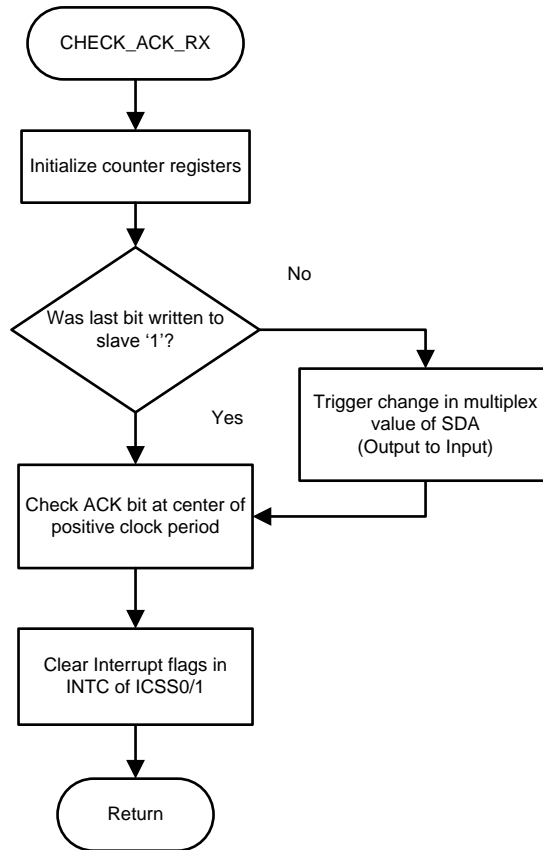


Figure 24. Check_ACK_RX Flowchart

2.3.4.10 Stop_Bit_Generation_RX

This function is similar to the Stop_Bit_Generation function. The difference is in terms of the value of the SDA pin multiplex upon entry to the function. The SDA pin is always input upon entry to this function whereas the Stop_bit_Generation function is always output upon entry. Figure 25 shows the behavior of the function.

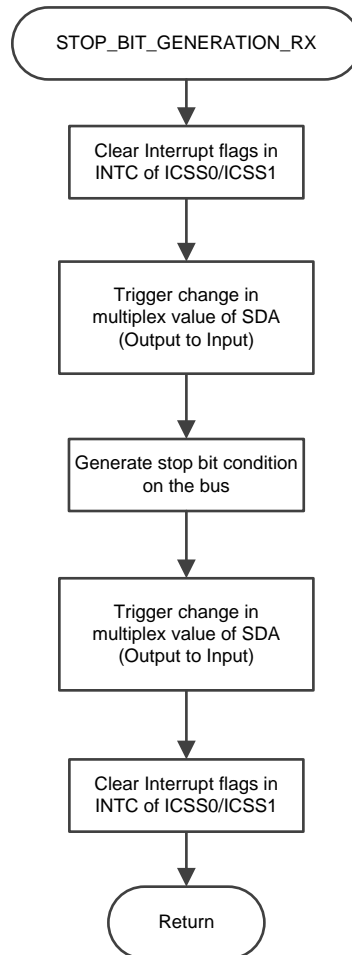


Figure 25. Stop_Bit_Generation_RX Flowchart

2.3.4.11 Master_Send_ACK_To_Slave

This function is used by the master to send the acknowledgment (ACK) bit to the slave. The master pulls the SDA line to '0' during the negative period of the clock. The slave registers this bit value during the positive period of the clock. The slave interprets the ACK bit as an acknowledgment of the correct data byte reception the slave sent to the master. The SDA pin multiplex value is input upon entry of this function and changed to output for asserting SDA line to '0'. The function changes back the pin multiplex value to input upon exit. [Figure 26](#) shows the functionality of this function.

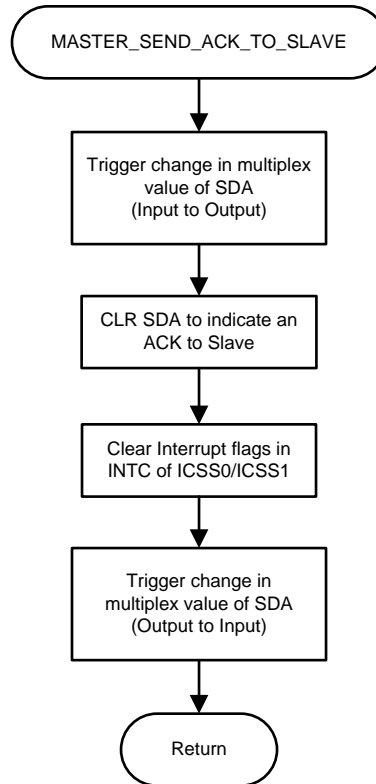


Figure 26. Master_Send_ACK_To_Slave Flowchart

2.3.4.12 Master_Send_NACK_To_Slave

The structure of this function is similar to the function Master_Send_ACK_To_Slave. This function is used at the end of a SMBus read operation – it sends a no-acknowledgment (NACK) bit to the slave from the master. The master pulls the SDA line to '1' during the negative period of the clock. The slave registers this value during the positive period of the clock and interprets it as no-acknowledgment signaling the end of SMBus read operation. The value of the SDA pin multiplex value is input upon entry to this function and output upon exit. [Figure 27](#) shows the behavior of this function.

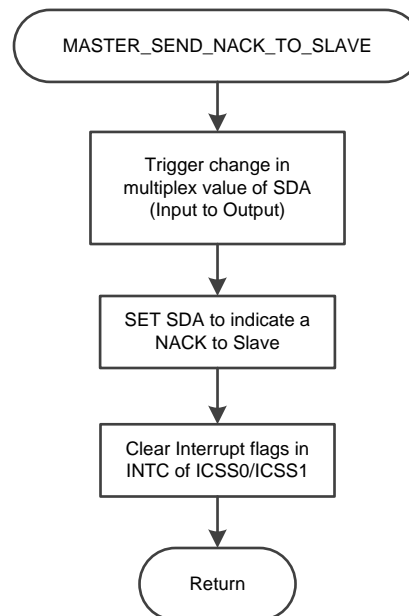


Figure 27. Master_Send_NACK_To_Slave Flowchart

2.3.5 EDMA Configuration

- This TI Design needs to change the data direction of SDA dynamically. This is performed by PRU firmware through use of register mapped GPI (R31) and GPO (R30) registers.
- The direction of the pin is changed through the Control Module register, selecting pr0_pru1_gpi11 for input and pr0_pru1_gpo11 for output.
- The PRU firmware cannot change the direction of the pin as it cannot write to the Control Module. This TI Design uses EDMA to achieve this.
- Because an EDMA transfer can only be triggered by ICSS1, a mapping of INTC event from ICSS0 to ICSS1 has to be set up. See [Section 2.3.6](#) for more details.
- The EDMA PaRAM set is set up to write the pin multiplex value for SDA from the ICSS0-PRU1-DRAM0 into the Control Module register (shown in [Figure 28](#)):

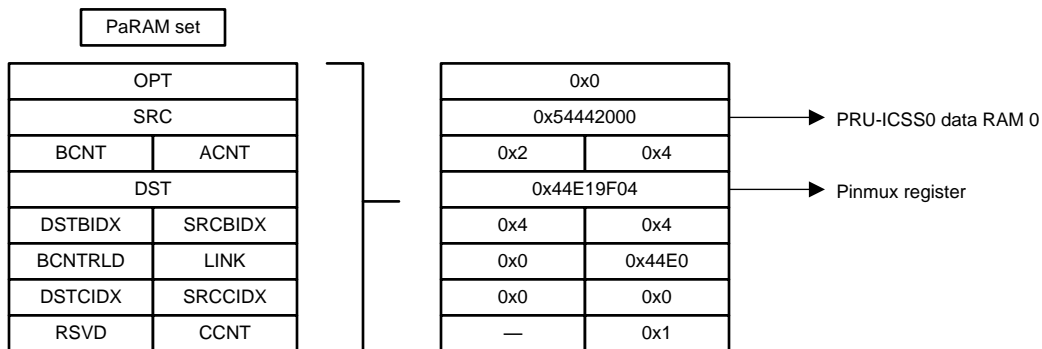


Figure 28. EDMA Configuration

2.3.6 INTC Configuration

- ICSS0-PRU1 cannot generate an event for the EDMA module directly. Therefore, the interrupt controller (INTC) of ICSS0 and ICSS 1 are configured in the following way:
 - ICSS0-PRU1 issues an event (16) to ICSS0-INTC
 - INTC-ICSS0 interrupt maps an event (56) to ICSS1-INTC
 - INTC-ICSS1 event triggers the EDMA to make the transfer
- [Figure 29](#) shows the INTC configuration:

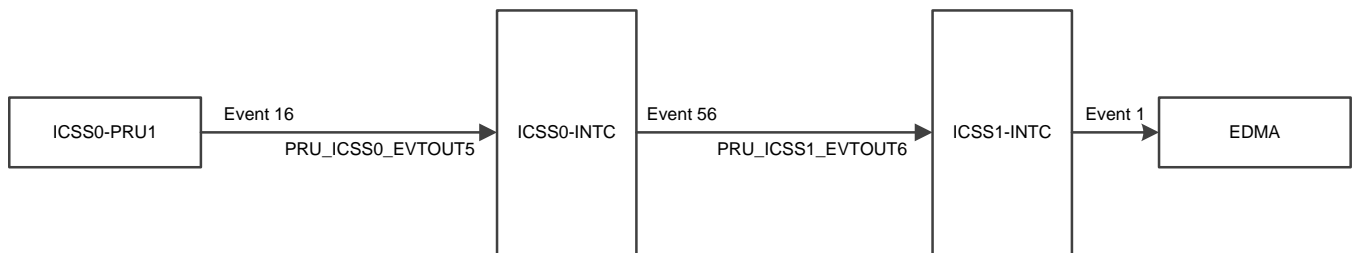


Figure 29. INTC Configuration

- ICSS0-PRU1 issues an event (17), which maps to PRU_ICSS0_EVTOUT0 at the ARM side to let the ARM know that SMBus transaction is completed.
- ARM issues an event (18), which is mapped to Host 0 through Channel 0 and shows up in R31 register of ICSS0-PRU1 so that PRU can begin execution of an SMBus operation.

2.3.7 PRU Registers Used

The PRU registers used by firmware are shown in [Table 2](#):

Table 2. PRU Registers Used

REGISTER	REGISTER NAME IN CODE	REGISTER USAGE
R3.w0	delay_count_reg_quater_cycle	Delay value for quarter cycle
R3.w2	delay_count_reg_half_cycle	Delay value for half cycle
R4.b0	SMBus_frequency_sel	Stores the selected frequency
R4.b1	pnmx_val	Stores the pinmux value of SDA
R4.b2	SMBus_protocol_sel_reg	Stores the selected SMBus Command
R4.b3	slave_addr	Stores the slave address
R5	temp_reg	Temporary register
R6	pru1_intc	Stores address of PRU1 INTC
R7.b0	tx_data	Data to be written to slave
R7.b1	bit_count	Number of bits (counter)
R7.b2	byte_count	Number of bytes (counter)
R7.b3	total_byte_count_N	Stores Block Count (N) for Block Write
R8.b0	rx_data	Data received from slave
R8.b2	rx_total_byte_count_N	Stores Block Count (N) for Block Read
R9	sda_pin_reg	Stores address of CTRL_CONF_CAM1_DATA3, GPIO pin used as SDA
R10.b0	rd_wr_bit	Stores Read/Write direction bit for Quick Command
R10.b2	last_bit_sent	Stores value of the last bit sent
R10.b3	bit_count_and_last_bit_sent	Stores sum of the bit_count and last_bit_sent
R11.w0	ack_reg	Stores ACK bit information from slave
R12.w0	ack_reg	Stores the data sent in testing mode
R13	mem_idx_tx_data	Stores address of PRU Memory area where TX data is located
R14	mem_idx_rx_data	Stores address of PRU Memory area where RX data is stored

2.3.8 PRU Memory Map

- ICSS0-PRU1 Data RAM 0 is used as register interface between ARM and PRU.
- [Table 3](#) shows the address and corresponding register information.
- The register interface emulates the I²C interface of an I²C peripheral (here AM437x) in order to reuse an existing I²C ARM driver. As this is a shared memory interface, the ARM driver cannot start any SMBus command by writing into the register interface. Instead, the ARM has to issue a PRU event to start an SMBus operation.

Table 3. Addresses and Corresponding Registers

ICSS0-PRU1 DATA RAM (OFFSET)	LOCATION NAME IN CODE	CONTAINED INFORMATION
0x24	I2C_IRQSTS_RAW	I ² C status raw register; bit ARDY = 2, bit NACK = 1
0x28	I2C_IRQSTS	I ² C status register; bit ARDY = 2, bit NACK = 1
0x2C	I2C_IRQEN_SET	I ² C interrupt enable set register; bit ARDY = 2, bit NACK = 1
0x98	I2C_CNT	Data count for Write_block and Read_block
0xA4	I2C_CON	I ² C configuration register, 9th bit (TRX) used as Rd/Wr direction
0xAC	I2C_SA	Slave address
0xE0	SDA_PINMUX_VALUE_OUTPUT	Pinmux value for setting SDA to output
0xE4	SDA_PINMUX_VALUE_INPUT	Pinmux value for setting SDA to input
0xE8	CLOCK_HALF_CYCLE_DELAY_VALUE	Delay value for half clock cycle
0xEC	COMMAND_CODE	Command code
0xF0	SELECTED_SMBUS_COMMAND	Selected SMBus command
0x100	TX_DATA_LOCATION	Data to be transmitted to slave by master (max 256 bytes)
0x200	RX_DATA_LOCATION	Data read from the slave by master (max 256 bytes)

2.4 ARM Driver Description

The ARM driver calls different functions to achieve desired functionality. The functions are:

- I2c_init()
- I2c_write()
- I2c_read()

The detail about these functions is given in the following subsections. [Section 2.4.1](#) describes in detail how the ARM driver functions. Furthermore, [Section 2.4.2](#), [Section 2.4.3](#), and [Section 2.4.4](#) describe in detail the role of their respective functions.

2.4.1 ARM Driver Functionality

The ARM driver first sets up the PRU and the pin multiplex values and then configures the EDMA. It then initializes the I²C and SMBus with the required frequency by calling the `I2c_init()` function. Then it calls either the `I2c_write()` or `I2c_read()` function depending on the respective command received from user application. After executing the function, it sends the status back to the user application. Figure 30 shows how different components of the design interact on the system level:

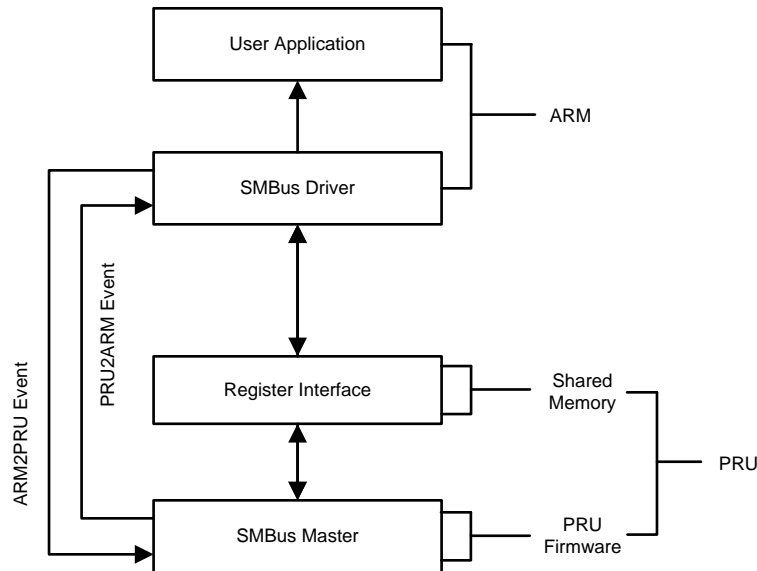


Figure 30. Interaction Between SMBus Design Components

Figure 31 shows the ARM driver's functionality as previously described:

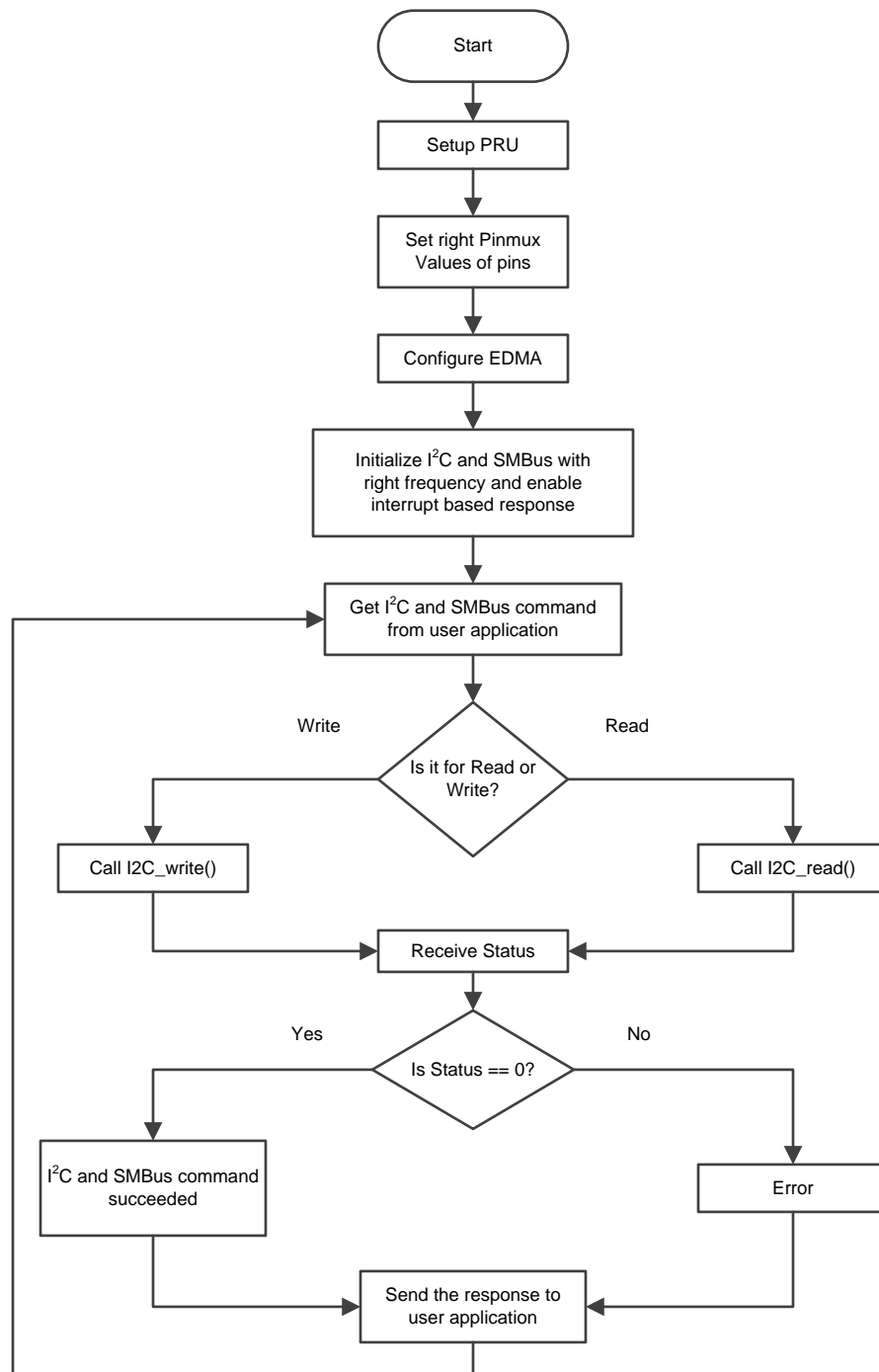


Figure 31. ARM Driver Functionality

2.4.2 I2c_init() Function

The I2c_init() function is a parameter-less function that initializes the frequency that will be used for communication between the SMBus master and slave. It also sets interrupt enable flags in case the design is interrupt based instead of polling based.

2.4.3 I2c_write() Function

The I2c_write() function takes the following parameters:

- `smbus_format`: An enumeration type; the values it can contain are shown in [Table 4](#):

Table 4. smbus_format Values

VALUE	INTERPRETATION
0	no_command
1	quick_command
2	Send_byte
3	receive_byte
4	write_byte
5	write_word
6	read_byte
7	read_word
8	write_block
9	read_block

- `slave_addr`: 8-bit unsigned integer that contains 7 bits of slave address appended with 1 bit for the read or write direction
- `command_code`: 8-bit unsigned integer for the command code byte
- `byte_count`: 8-bit unsigned integer that is used with the Block_Write SMBus frame to let the slave device know how many data bytes will be transmitted in this transaction
- `tx_data_ptr`: A pointer to the type of 8 bits unsigned integer for the `tx_data_array`; contains the data bytes that are to be transmitted to the slave

The I2c_write() function first checks whether it is allowed to access the I²C and SMBus register interface by checking the second bit, which is the access ready bit (ARDY) in register PRU_I2C_IRQSTS_RAW at offset 0x24. If it is not allowed to access the I²C and SMBus register, the function error code is returned by the function. Otherwise the function first configures the I²C and SMBus register interface in PRU shared memory. After that the function copies the data bytes from the transmit data array into the TX_DATA_LOCATION in PRU memory, clears the interrupt flag, and sends an event to the PRU. Then the function waits for an event from PRU, which indicates the completion of the SMBus command. After receiving the event, the I2c_write() function reads the status register and returns a success code (0) or error code (-1) to the function caller. [Figure 32](#) shows the functionality of the I2c_write() function:

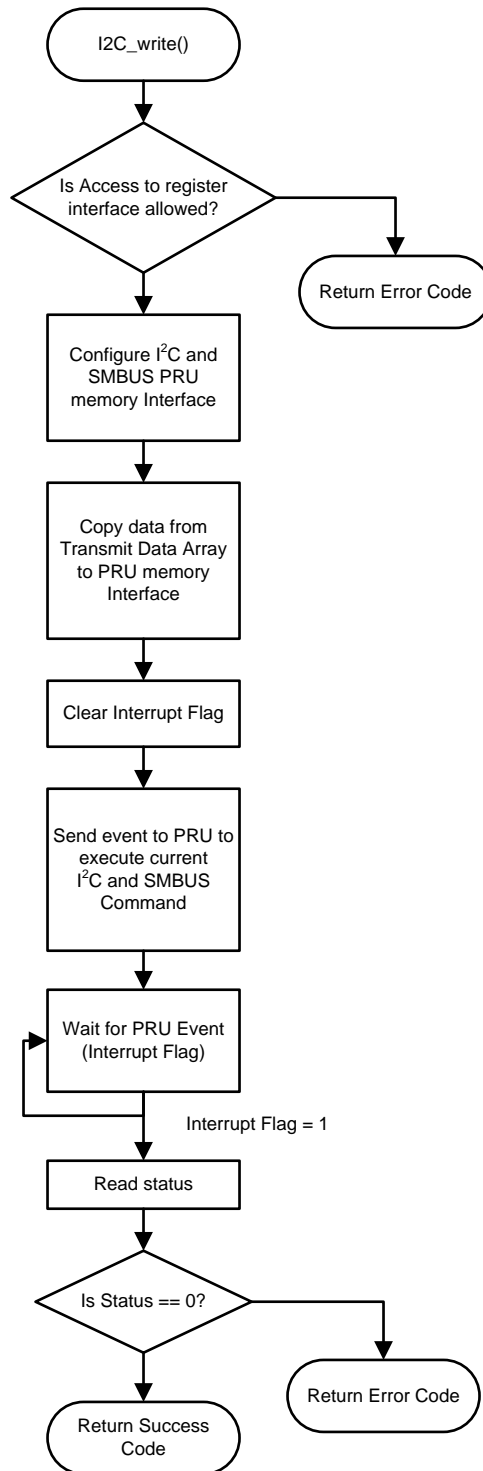


Figure 32. I2c_write() Function

2.4.4 I2c_read() Function

The I2c_read() function has the same parameters as in the I2c_write() function like smbus_format, slave_addr, and command_code, but there is no byte_count parameter; instead of tx_data_ptr, there is rx_data_ptr, which points to rx_data_array that contains the data bytes received from the slave.

The I2c_read() function first checks whether it is allowed to access the I²C and SMBus register interface by checking the second bit, which is access ready bit (ARDY) in register PRU_I2C_IRQSTS_RAW at offset 0x24. If it is not allowed to access the I²C and SMBus register, the function error code is returned by the function. Otherwise the function first configures the I²C and SMBus register interface in PRU shared memory. After that, the function clears the interrupt flag and sends an event to the PRU. Then the function waits for an event from the PRU, which indicates the completion of the SMBus command. After receiving the event, the I2c_write() function reads the status register, copies the data bytes from the TX_DATA_LOCATION in PRU memory into the receive data array if transaction with slave is successful, and then returns a success code (0) or error code (-1) to the function caller. [Figure 33](#) shows the functionality of I2c_write() function:

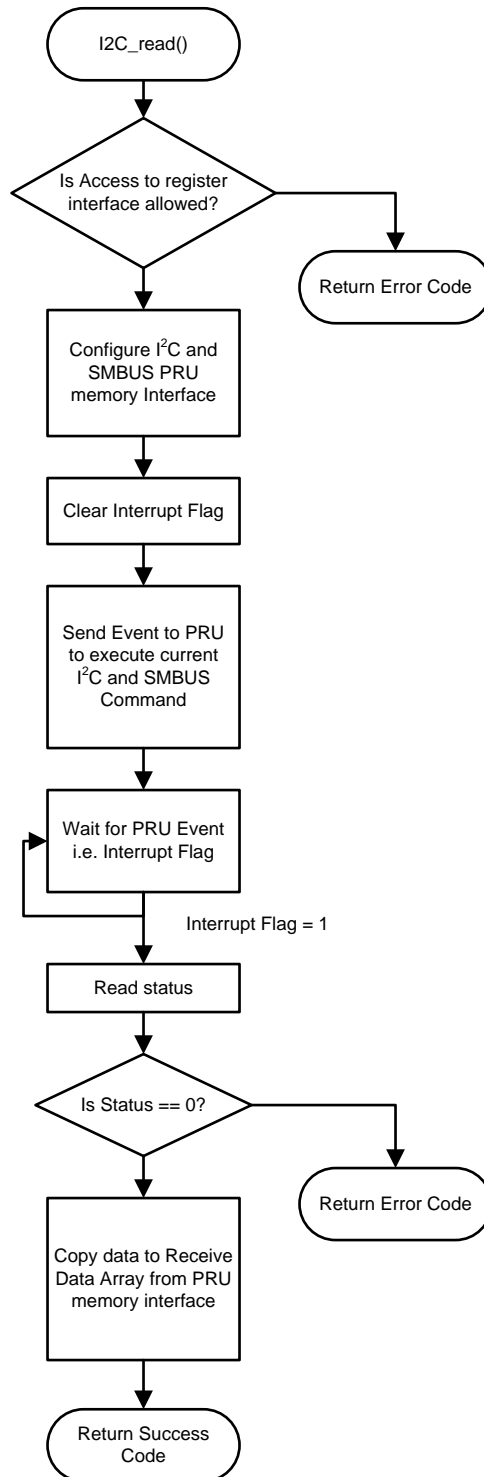


Figure 33. I2c_read() Function

3 Getting Started Hardware and Software

3.1 Hardware

3.1.1 TMSIDK437X IDK Board

Figure 34 shows the TMSIDK437X board.

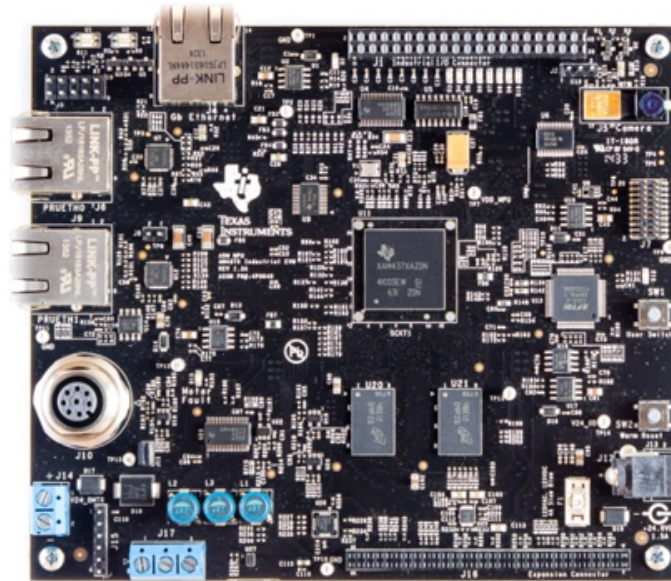


Figure 34. TMSIDK437X Board

The I²C and SMBus master interface can be accessed through the following signals:

- J16, pin 17 → SDA
- J16, pin 19 → SCL
- J16, pin 60 → GND

3.1.2 I²C and SMBus Slave Board

Connect any I²C slave board with the I²C and SMBus interface to the I²C signals on J16. The slave board needs to get powered separately.

3.2 Software

3.2.1 CCS

The following hardware and software are required:

- TMSIDK437X board
- I²C or SMBus slave board
- CCSv6 or higher
- PRU Compiler for CCSv6 (install through CCS add-on)
- Industrial SDK 2.1.0.1
 - SYS-BIOS (refer to Industrial SDK release notes)
 - XDC-Tools (refer to Industrial SDK release notes)
- TI Design TIDEP0065 software download files

After installing the development tools, extract the TIDEP0065 project in the C:/TI folder. Import the PRU and ARM project into CCS from the downloaded source code file folder.

3.2.2 PRU Firmware

Once the project has been imported the PRU firmware can be compiled. The outcome of the PRU compiler is an .out file. The .out file must get converted into a C-Header file, which is then included by the ARM application. The ARM application loads the PRU firmware header at run time into the PRU core.

Follow these steps to convert the PRU .out file into a C-Header file.

1. Go to the Smbus_PRU folder.
2. Read the readme.txt in header_gen folder.
 - If required, adopt file names and paths in build_header_pru1.bat.
3. Copy the following files from header_gen to the debug folder.
 - build_header_pru1.bat
 - pru_header.cmd
4. Open a CMD DOS box and navigate to the debug folder.
5. Execute the build_header_pru1.bat file: This generates the C-Header file and copies this into the ARM project include folder.

For development and testing purposes, download the .out file through JTAG to the PRU core. Note that if the ARM application is reloaded and executed, it will in general rewrite the PRU firmware into the PRU code.

3.2.3 ARM Application

The ARM application initializes the PRU-ICSS subsystem, sets the external pinmux for SMBus, and loads the PRU firmware. The example application uses the I²C function API to make access to the I²C peripheral. In addition, the example application configures the register I²C and SMBus interface in PRU-ICSS memory.

Note that I²C slave address must match with the actual slave address to successfully communicate with the I²C slave board.

Compile the ARM project and load the .out file into the ARM code. Execute the application to start I²C and SMBus communication. Use an oscilloscope to validate bus communication. Use breakpoints in the ARM software and read out the I²C function return variables to validate the expected results.

4 Testing and Results

4.1 Test Setup

The following test setup is used (as shown in [Figure 35](#)):

- TMSIDK437X IDK Board
- TUSB8041RGCEVM Board Rev D (used as slave)
- Digital Logic Analyzer LogicStudio 16 from Teledyne LeCroy
- Windows® 7 PC with LogicStudio

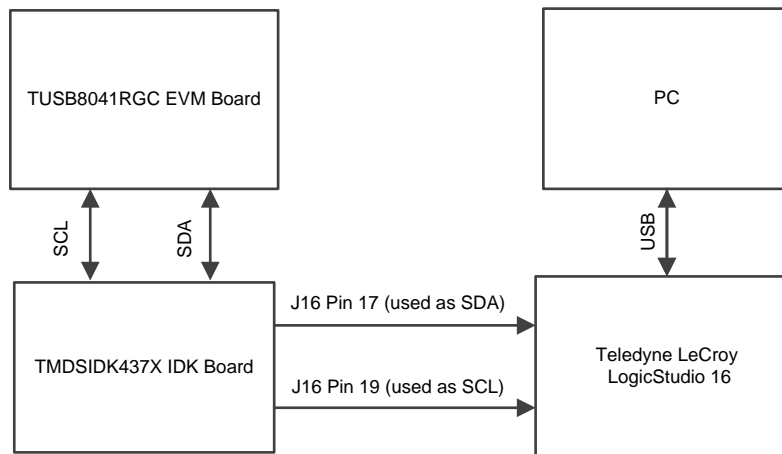


Figure 35. Test Setup

4.2 Test Data

Two types of testing and validation were performed. Firstly, validation of all individual SMBus frames was performed using LogicStudio; the results are shown in [Section 4.2.1](#). Secondly, long-term testing was performed where the SMBus master and slave communicated with each other for almost two days and results were analyzed (as seen in [Section 4.2.2](#)).

4.2.1 Validation with Logic Studio

A variety of measurements have been performed with all the SMBus frame formats at 100 kHz to validate the SMBus master PRU firmware implementation. The I²C signals have been measured with a digital logic analyzer (LogicStudio from Teledyne LeCroy). The measurement results are shown in the following subsections.

4.2.1.1 Quick Command

Figure 36 and Figure 37 shows the results obtained on LogicStudio when Quick Command is executed with a Write bit and Read bit, respectively.

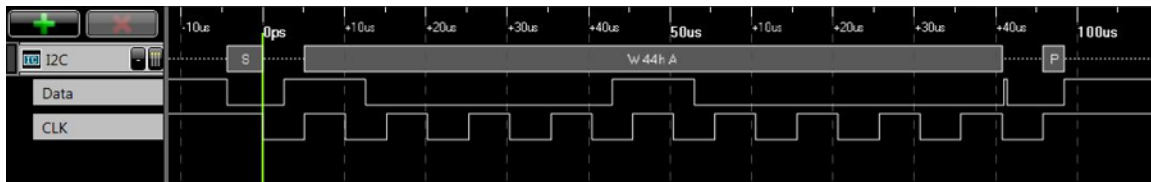


Figure 36. Validation of Quick Command Frame Format With Write Bit



Figure 37. Validation of Quick Command Frame Format With Read Bit

4.2.1.2 Send Byte

Figure 38 shows the results obtained on LogicStudio when the Send Byte command is executed.

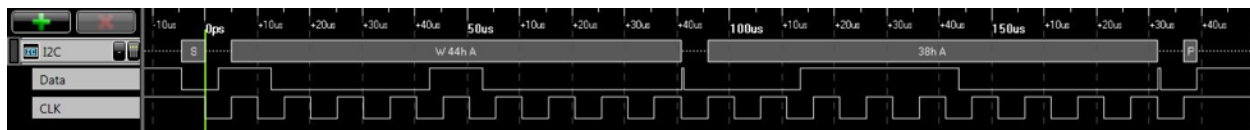


Figure 38. Validation of Send Byte Frame Format

4.2.1.3 Receive Byte

Figure 39 shows the results obtained on LogicStudio when the Receive Byte command is executed.



Figure 39. Validation of Receive Byte Frame Format

4.2.1.4 Write Byte

Figure 40 shows the results obtained on LogicStudio when the Write Byte command is executed.

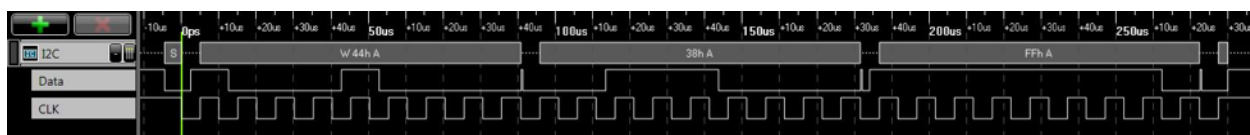


Figure 40. Validation of Write Byte Frame Format

4.2.1.5 Read Byte

Figure 41 shows the results obtained on LogicStudio when Read Byte command is executed.

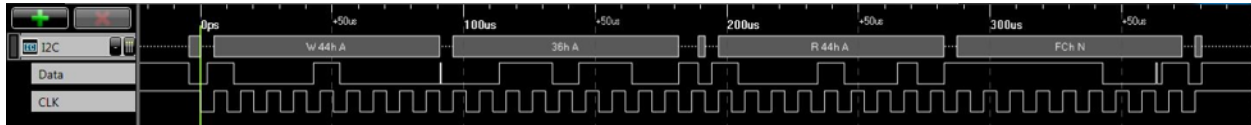


Figure 41. Validation of Read Byte Frame Format

4.2.1.6 Write Word

Figure 42 shows the results obtained on LogicStudio when the Write Word command is executed.

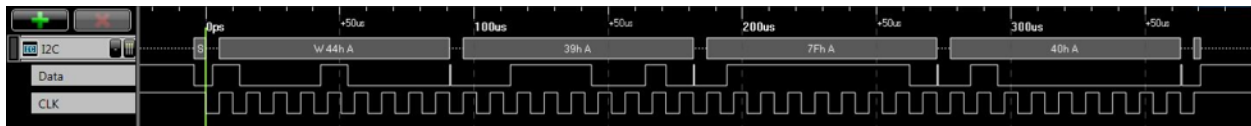


Figure 42. Validation of Write Word Frame Format

4.2.1.7 Read Word

Figure 43 shows the results obtained on LogicStudio when the Read Word command is executed.

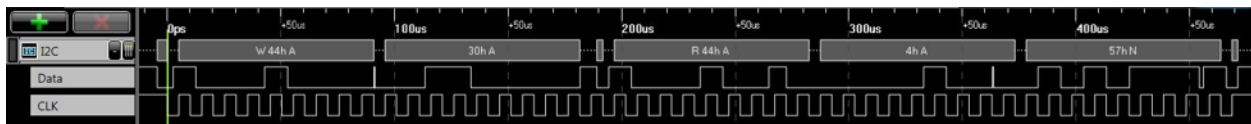


Figure 43. Validation of Read Word Frame Format

4.2.1.8 Block Write

Figure 44 shows the results obtained on LogicStudio when the Block Write command with Block Count (N) = 8 is executed.

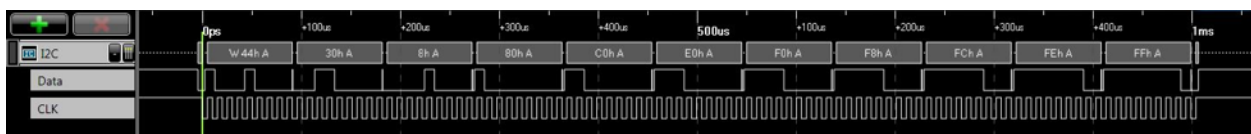


Figure 44. Validation of Block Write Frame Format

4.2.1.9 Block Read

Figure 45 shows the results obtained on LogicStudio when the Block Read command is executed and received Block Count (N) = 4 from the slave.

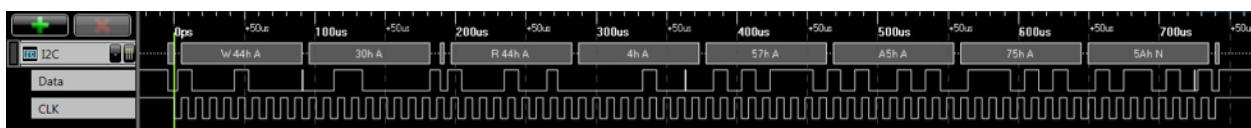


Figure 45. Validation of Block Read Frame Format

4.2.2 Long-Term Test

Figure 46 shows the results for a long-term test (two days) between the PRU master and slave. The user application is calling `I2c_write()` and `I2c_read()` for SMBus Quick_Command, Write/Read_Byte, Write/Read_Word, and Write/Read_Block in a repeated manner with different data sets. After the command execution the results are compared from the counter values, which shows how many times a specific SMBus format was sent and how many ACKs and NACKs were received. It also shows how many data bytes were sent and how many of them were correctly or incorrectly received back.

For example, the values of the three variables `quick_cmd_sent`, `quick_cmd_ACKs` and `quick_cmd_NACKs` sent a total of 46,768,238 Quick_Command frames by the master to the slave where slave acknowledged all of them and there was no instance when there was no acknowledgment because its counter value is 0. Similarly, by comparing other values, the long-term test ran successfully without any errors.




(x) Variables  Expressions  Breakpoints 			
Name	Type	Value	Location
arg0	unsigned int	0	0x8002B48C
arg1	unsigned int	0	0x8002B488
byte_count	unsigned char	.	0x8002B498
command_code	unsigned char	.	0x8002B499
different_tx_rx_byte_count	int	0	0x8002B4A4
quick_cmd_ACKs	int	46768238	0x8002B508
quick_cmd_NACKs	int	0	0x8002B504
quick_cmd_sent	int	46768238	0x8002B50C
read_block_ACKs	int	46768238	0x8002B4B0
read_block_NACKs	int	0	0x8002B4AC
read_block_sent	int	46768238	0x8002B4B4
read_byte_ACKs	int	46768238	0x8002B4F0
read_byte_NACKs	int	0	0x8002B4EC
read_byte_sent	int	46768238	0x8002B4F4
read_word_ACKs	int	46768238	0x8002B4D0
read_word_NACKs	int	0	0x8002B4CC
read_word_sent	int	46768238	0x8002B4D4
read_write_block_correct_data	int	187072952	0x8002B4A0
read_write_block_incorrect_data	int	0	0x8002B49C
read_write_byte_correct_data	int	46768238	0x8002B4E8
read_write_byte_incorrect_data	int	0	0x8002B4E4
read_write_word_correct_data	int	93536476	0x8002B4C8
read_write_word_incorrect_data	int	0	0x8002B4C4
rx_byte_count	unsigned char	.	0x8002B492
same_tx_rx_byte_count	int	46768238	0x8002B4A8
slave_addr	unsigned char	D	0x8002B49A
smbus_format	enum SMBU...	quick_command	0x8002B49B
status	int	0	0x8002B494
temp_data_read_write_block_byte0	unsigned char	.	0x8002B514
temp_data_read_write_block_byte1	unsigned char	A	0x8002B513
temp_data_read_write_block_byte2	unsigned char	E	0x8002B512
temp_data_read_write_block_byte3	unsigned char	.	0x8002B511
temp_data_read_write_byte	unsigned char	.	0x8002B517
temp_data_read_write_word_byte0	unsigned char	.	0x8002B516
temp_data_read_write_word_byte1	unsigned char	.	0x8002B515
tx_byte_count	unsigned char	.	0x8002B493
write_block_ACKs	int	46768238	0x8002B4BC
write_block_NACKs	int	0	0x8002B4B8
write_block_sent	int	46768238	0x8002B4C0
write_byte_ACKs	int	46768238	0x8002B4FC
write_byte_NACKs	int	0	0x8002B4F8
write_byte_sent	int	46768238	0x8002B500
write_word_ACKs	int	46768238	0x8002B4DC
write_word_NACKs	int	0	0x8002B4D8
write_word_sent	int	46768238	0x8002B4E0

Figure 46. ARM Driver Long-Term Testing Results

5 Design Files

5.1 Schematics

To download the schematics, see the design files at [TIDEP0065](#).

5.2 Bill of Materials

To download the bill of materials (BOM), see the design files at [TIDEP0065](#).

5.3 Gerber Files

To download the Gerber files, see the design files at [TIDEP0065](#).

5.4 Assembly Drawings

To download the assembly drawings, see the design files at [TIDEP0065](#).

6 Software Files

To download the software files, see the design files at [TIDEP0065](#).

7 References

1. Texas Instruments, *Download CCS*, Code Composer Studio TI Wiki (http://processors.wiki.ti.com/index.php/Download_CCS)
2. Texas Instruments, *AM437x Sitara™ Processors*, AM4379 Datasheet ([SPRS851](#))
3. Texas Instruments, *AM437x ARM® Cortex™-A9 Processors*, AM4379 Technical Reference Manual ([SPRUHL7](#))

8 Terminology

CCS— Code Composer Studio

ICSS— Industrial communication subsystem

PLC— Programmable logic controller

PRU— Programmable real-time unit

9 About the Authors

MUHAMMAD HAISEM KHAN is a master student at University of Stuttgart, Germany. He is pursuing specialization in embedded systems under the INFOTECH program at his university. He has significant knowledge in the fields of industrial automation, embedded systems, real-time systems and real-time programming. As per his curriculum and interests, he was a master intern in the Factory Automation and Control Team in Texas Instruments Freising, Germany. He was responsible for implementation of PRU firmware and ARM driver in SMBus project. Haisem acquired his bachelor's degree in electrical (telecommunication) engineering from National University of Sciences & Technology (NUST) in Islamabad, Pakistan.

PHANINDRA SHYLENDRA is a master student at Hochschule Darmstadt University Of Applied Sciences, Germany in the field of embedded systems and microelectronics. As part of his curriculum, he was a master intern in the Factory Automation and Control Team in Texas Instruments in Freising, Germany. He was responsible for the reference design for developing SMBus interface. Phanindra gained his bachelor's degree in electronics and communications from Saphthagiri College of Engineering (VTU), Bangalore, India.

THOMAS MAUER is a system engineer in the Factory Automation and Control Team at Texas Instruments Freising, where he is responsible for developing reference design solutions for the industrial segment. Thomas brings to this role his extensive experience in industrial communications like industrial Ethernet and fieldbuses and industrial applications. Thomas earned his electrical engineering degree (Dipl. Ing. (FH)) at the University of Applied Sciences in Wiesbaden, Germany.

Revision A History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (July 2016) to A Revision	Page
• Changed from preview page.....	1

IMPORTANT NOTICE FOR TI REFERENCE DESIGNS

Texas Instruments Incorporated ("TI") reference designs are solely intended to assist designers ("Designer(s)") who are developing systems that incorporate TI products. TI has not conducted any testing other than that specifically described in the published documentation for a particular reference design.

TI's provision of reference designs and any other technical, applications or design advice, quality characterization, reliability data or other information or services does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such reference designs or other items.

TI reserves the right to make corrections, enhancements, improvements and other changes to its reference designs and other items.

Designer understands and agrees that Designer remains responsible for using its independent analysis, evaluation and judgment in designing Designer's systems and products, and has full and exclusive responsibility to assure the safety of its products and compliance of its products (and of all TI products used in or for such Designer's products) with all applicable regulations, laws and other applicable requirements. Designer represents that, with respect to its applications, it has all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. Designer agrees that prior to using or distributing any systems that include TI products, Designer will thoroughly test such systems and the functionality of such TI products as used in such systems. Designer may not use any TI products in life-critical medical equipment unless authorized officers of the parties have executed a special contract specifically governing such use. Life-critical medical equipment is medical equipment where failure of such equipment would cause serious bodily injury or death (e.g., life support, pacemakers, defibrillators, heart pumps, neurostimulators, and implantables). Such equipment includes, without limitation, all medical devices identified by the U.S. Food and Drug Administration as Class III devices and equivalent classifications outside the U.S.

Designers are authorized to use, copy and modify any individual TI reference design only in connection with the development of end products that include the TI product(s) identified in that reference design. HOWEVER, NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of the reference design or other items described above may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI REFERENCE DESIGNS AND OTHER ITEMS DESCRIBED ABOVE ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY DESIGNERS AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS AS DESCRIBED IN A TI REFERENCE DESIGN OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TI's standard terms of sale for semiconductor products (<http://www.ti.com/sc/docs/stdterms.htm>) apply to the sale of packaged integrated circuit products. Additional terms may apply to the use or sale of other types of TI products and services.

Designer will fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of Designer's non-compliance with the terms and provisions of this Notice.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2016, Texas Instruments Incorporated