**eSPI PRU Firmware**

Applies to Product Release: 01.00.00.00

Publication Date: December 4, 2018

**Document License**

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

**Contributors to this document**

Copyright (C) 2018 Texas Instruments Incorporated - http://www.ti.com/



Texas Instruments, Incorporated  
20450 Century Boulevard  
Germantown, MD 20874 USA

Texas Instruments, Incorporated  
20450 Century Boulevard  
Germantown, MD 20874 USA

**This document is intended for users who are interested in getting more detailed understanding of the firmware design and the interfacing driver design. It mentions the memory maps, structures and design flow of the firmware, as well as the design flow of the host driver.**

**Note: Those who just want to use ICSS eSPI firmware may not need to go through this document.**

**TABLE OF CONTENTS**

LIST OF FIGURES 6

LIST OF TABLES 7

1 Introduction 8

1. Architecture Design Overview 9

1.1. Design Layers 9

1.2. PRU-BB 9

1.2.1. PRU-BB Memory 9

1.2.2. GPI/GPO 9

1.3. PRU-CC 9

1.3.1. PRU-CC Memory 10

1.4. Scratchpad (SPD) Registers 10

1.5. PRU-ICSS Shared Memory 10

1.5.1. ESPI Slave Registers 10

1.5.2. RX Buffers 10

1.5.3. TX Buffers 10

1.6. Host Core 10

1.6.1. Application 10

1.6.2. Driver 10

1.6.3. GPIO 11

1.7. HW Interface 11

2. Design Theory 11

2.1. ESPI Bit-Banging 11

2.1.1. Reset 11

2.1.2. Ready 11

2.1.3. Alert 11

2.1.4. Read Byte 12

2.1.5. Write Byte 12

2.2. ESPI Control 12

2.2.1. Reset 12

2.2.2. Ready 12

2.2.3. Read CMD 12

2.2.4. Handle CMD 12

2.3. Reset 13

2.4. Shared Registers 13

2.4.1. Data 13

2.4.2. Count 13

2.4.3. State 13

2.4.4. Error 13

2.5. Errors 13

2.5.1. NO\_RESPONSE 14

2.5.2. Invalid CS# Deassert 14

2.5.3. FATAL\_ERROR 14

2.5.4. NON\_FATAL\_ERROR 14

3. ESPI Protocol 14

3.1. Overview 14

3.2. Command Phase 14

3.3. TAR Phase 15

3.4. Response Phase 15

3.5. Alert Phase 16

3.6. Wait State 16

4. CRC Calculation 16

5. Slave Registers 16

5.1. Device Identification 16

5.2. General Capabilities and Configurations 16

5.3. Channel 0 Capabilities and Configurations 17

5.4. Channel 1 Capabilities and Configurations 18

5.5. Channel 2 Capabilities and Configurations 18

5.6. Channel 3 Capabilities and Configurations 19

6. Commands 19

6.1. Supported Commands 19

6.2. GET\_STATUS 19

6.3. GET\_CONFIGURATION 19

6.4. SET\_CONFIGURATION 19

6.5. RESET 19

6.6. Posted Transaction 19

6.7. Non-Posted Transaction 20

6.8. Transaction Ordering 20

**6.8.1.** **RX/TX In-Out Dependency** 20

7. PRU/Host Communication 20

7.1. Memory Map 20

7.2. Save Status Register 20

7.3. Internal Status Register 20

7.4. IPC PRU to Host 21

7.4.1. RX Data Available 21

7.4.2. IPC Commands 21

7.5. RESERVED 22

7.6. IPC Host to PRU 22

7.7. RX/TX Buffers 22

7.7.1. Peripheral Channel 22

7.7.2. Virtual Wires Channel 22

7.7.3. OOB Message Channel 22

7.7.4. Flash Access Channel 22

8. RTOS Driver 22

8.1. ESPI\_init 22

8.2. ESPI\_open 22

8.2.1. IRQ Handler 22

8.3. ESPI\_transfer 23

8.3.1. SPI\_Transfer(read) 24

8.3.2. SPI\_Transfer(write) 25

8.4. ESPI\_control 25

8.5. ESPI\_close 25

8.6. Reset 25

8.7. Application Layer 25

9. EVM Support 26

9.1. bbbAM335x 26

9.2. idkAM437x 26

10. External Board 27

10.1. General Design 27

10.2. bbbAM335x 28

10.3. idkAM437x 28

11. Testing 29

11.1. Total Phase Promira Serial Platform 29

12. Limitations 29

12.1. Limited Frequency Support 29

12.2. Limited Channel Support 29

12.3. Slow Clock Not Supported 29

12.4. Slave-Initiated Transactions 29

12.5. Response Modifiers 29

12.6. No Shared Alert# Pin 29

12.7. Queue Depth 29

13. References 29

LIST OF FIGURES

[Figure 1: Architecture Overview 9](#_Toc531696895)

[Figure 2. Bit-bang State Diagram 11](#_Toc531696896)

[Figure 3. Control State Diagram 12](#_Toc531696897)

[Figure 4. Command Phase Timing 14](#_Toc531696898)

[Figure 5. TAR Phase Timing 15](#_Toc531696899)

[Figure 6. Response Phase Timing 16](#_Toc531696900)

[Figure 7. Driver IRQ Handler Flow 23](#_Toc531696901)

[Figure 8. SPI\_Transfer RX Flow 24](#_Toc531696902)

[Figure 9. SPI\_Transfer TX Flow 25](#_Toc531696903)

[Figure 10. Basic Application Flow 26](#_Toc531696904)

[Figure 11. External Hardware Design 27](#_Toc531696905)

[Figure 12. bbbAM335x Test Hardware 28](#_Toc531696906)

[Figure 13. idkAM437X Test Hardware 28](#_Toc531696907)

LIST OF TABLES

[Table 1. ICSS Shared Memory Allocation 10](#_Toc531696908)

[Table 2. RX Buffer Allocation 10](#_Toc531696909)

[Table 3. TX Buffer Allocation 10](#_Toc531696910)

[Table 4. Shared Register Allocation 13](#_Toc531696911)

[Table 5. PRU-BB Error Codes 13](#_Toc531696912)

[Table 6. General Capabilities and Configurations Slave Register 17](#_Toc531696913)

[Table 7. Channel 0 Capabilities and Configurations 18](#_Toc531696914)

[Table 8. Memory Map 20](#_Toc531696915)

[Table 9. Internal Status Register 21](#_Toc531696916)

[Table 10. IPC Commands 22](#_Toc531696917)

[Table 11. bbbAM335x Pin Use 26](#_Toc531696918)

[Table 12. idkAM437x Pin Use 27](#_Toc531696919)

# Introduction

eSPI is a bi-directional serial protocol that provides a communication link between integrated circuits (ICs), effectively an optimized version of LPC. TI does not currently have hardware IP support for eSPI, and will be utilizing the ICSS to implement support in firmware.

Firmware is designed to run on PRU cores. Each ICSS has 2 PRUs, and the protocol operation will be split between each PRU. One PRU core will use its GPI/GPO pins to send and receive eSPI bytes on the wire. The other PRU core will control the protocol and is in charge of processing any bytes read by the first PRU and passing on any bytes which need to be written out on the wire to the first PRU. PRU programming is done using assembly instructions. Host cores in SOC will configure PRUs for receiving and sending data on the eSPI bus.

This Firmware design is not compatible with any of Ethernet based ICSS firmware design. It means one cannot load I2C firmware in one PRU for example PRU0 and Dual\_emac Firmware in other PRU for example PRU1.

For full documentation of the eSPI protocol, refer to [1]

# Architecture Design Overview

## Design Layers

Application

Driver

PRU-X

(ESPI Bitbang)

GPI/GPO

PRU-Y

(ESPI Control)

Host Core

PRU-ICSS

GPIO

PRU-ICSS Shared Memory

ESPI Slave

ESPI Master

**RESET**

**SCL**

**I/O[0:N]**

**Alert#**

**CS#**

**SCL**

**I/O[0:N]**

**Alert#**

**CS#**

**EN**

**RESET**

**SPD**

PRU-X Memory

PRU-Y Memory

HW

Interface

Figure 1: Architecture Overview

## PRU-BB

PRU-BB is in charge of bit-banging the communication with the master. PRU-BB can be either PRU0 or PRU1 depending on GPI/GPO pin availability on a specific board.

### PRU-BB Memory

PRU-BB has 8kB dedicated DRAM which will be used to store its state jump table.

### GPI/GPO

Bit-banging duties require access to external pins, this will be accomplished using the PRU-BB GPI/GPO. PRU does not support bi-directional GPIO support and will require external hardware to support bi-directional communication with the ESPI master (see: Section 11)

## PRU-CC

PRU-CC is the second PRU subsystem in the PRU-ICSS (i.e. if PRU-BB is determined to be PRU0, PRU-CC is PRU1 and vice versa). PRU-CC is in charge of real-time control over the ESPI firmware such as determining PRU-BB states, decoding and handling incoming ESPI packets from the master, and communicating RX/TX data with the Host.

### PRU-CC Memory

PRU-CC has 8kB dedicated DRAM which will be used for storing the ESPI Command jump table and the CRC calculation table.

## Scratchpad (SPD) Registers

There are 3 banks of scratch registers on the PRU-ICSS, each of which provides single-cycle storing of PRU registers accessible by each PRU cores. This therefore allows for fast communication between PRU cores and is necessary

## PRU-ICSS Shared Memory

Each PRU-ICSS has 12kB shared DRAM which will be allocated as follows:

|  |  |
| --- | --- |
| 0x0000 - 0x001F | ESPI Slave Registers |
| 0x0020 - 0x17FF | RX Buffers |
| 0x1800 - 0x2FFF | TX Buffers |

Table 1. ICSS Shared Memory Allocation

### ESPI Slave Registers

The ESPI Slave Registers are used to track configuration information on the ESPI slave. Most configurations are set by the ESPI Master, and a handful are set at compile-time reflecting system capabilities (see: Section 5).

### RX Buffers

The RX buffers are where the slave PRU will store incoming packets from the master for the driver to handle (see: Section 7.7).

|  |  |
| --- | --- |
| 0x0020 - 0x061F | Ch0 RX Buffer(s) |
| 0x0620 - 0x17FF | RESERVED |

Table 2. RX Buffer Allocation

### TX Buffers

The TX buffers are where the slave Host will store outgoing packets to send to the master (see: Section 7.7).

|  |  |
| --- | --- |
| 0x1800 - 0x1DFF | Ch0 TX Buffer(s) |
| 0x1E00 - 0x2FFF | RESERVED |

Table 3. TX Buffer Allocation

## Host Core

### Application

The application will ultimately drive the ESPI slave communication and will be in charge of handling ESPI packets (see: Section 9).

### Driver

The ESPI driver communicates with the PRU-ICSS, configuring the PRU firmware and passing data between the PRU and Application (see: Section 8).

### GPIO

PRU-ICSS is not interruptable without polling, so there is no efficient way to check for the RESET signal. Thus, the host driver will handle the RESET signal as an interrupt and reset the PRU firmware accordingly.

## HW Interface

For more information on the Hardware Interface (see: Section 11).

# Design Theory

## ESPI Bit-Banging

Error

Reset

Read Byte

TAR

Write Byte

End

Ready

Alert

CS# Assert

Read new state

1.5 CC’s passed

Read new state

CS# Deassert

Status Reg Changed

Figure 2. Bit-bang State Diagram

### Reset

The Reset state will initialize any local registers and then wait until the Host enables one of the eSPI channels in the slave registers.

### Ready

The Ready state will poll the CS# pin and the Status Register for changes. If CS# asserts, begin reading. If the Status Register value is different than the previous poll, begin the Alert sequence.

### Alert

Discussed in Section 3.5

### Read Byte

Depending on the mode, read either 1, 2, or 4 bits from R31 on the rising edge of SCL# and pass them to PRU-CC via scratchpad. Check for improper CS# de-assertion during each clock cycle. PRU-BB will continue to read while in the Command Phase.

### Write Byte

On each falling edge of SCL#, read a byte from PRU-CC and write either 1, 2, or 4 bits at time and assert the EN bit on R30. EN will be de-asserted in the TAR phase. Check for improper CS# de-assertion during each clock cycle.

## ESPI Control

Reset

Ready

Read CMD

Handle CMD

Error

Jump table

Data avail from PRU-X?

CMD done

Figure 3. Control State Diagram

### Reset

The Reset state will initialize any local registers and then wait until the Host enables one of the eSPI channels in the slave registers.

### Ready

In the Ready state, PRU-CC will poll the shared Data and Count registers. If the count has increased then there is a byte of data to read. Save the byte and increase the internal PRU-CC count register to match the shared count.

### Read CMD

Upon receiving the first byte from PRU-BB, PRU-CC will use a 256-entry jump table to jump to the proper CMD handling function or NO\_RESPONSE.

### Handle CMD

Each CMD will be handled differently. But generally, the packet header is unique to each Command so processing will be hardcoded for each. If there is data associated with the packet, that will be handled in a loop based on parameters in the Header. Once the packet has been completely read, any errors may be determined and handled accordingly. This function will facilitate the TAR and Response phase accordingly for each type of Command.

## Reset

Hard Reset is triggered by the RESET# pin, which will interrupt the host. Upon interruption, the host will disable and clear all data on each PRU, reset each to main(), then enable each PRU and instruct them into the RESET state. Once in the Reset state, both PRUs will wait until the host has set the “Channel Ready” bit in one of the channels.

## Shared Registers

The PRUs will use the scratchpad registers to communicate data between each core

|  |  |  |  |
| --- | --- | --- | --- |
| R0 | RESERVED | Count | Data |
| R1 | RESERVED | State | |
| R2 | RESERVED | Error | |

Table 4. Shared Register Allocation

### Data

Each time a byte is read by PRU-BB it will place it in the Data register for PRU-CC to read and handle. Conversely, every time a byte needs to be written by PRU-CC, it will place it in the Data register for PRU-BB to read and output on the data lines.

### Count

To quickly synchronize each PRU regarding available data, a count will be tracked. The obvious messaging scheme would be to set/clear a flag, but that takes precious extra clock-cycles. The number of bytes written/read will be tracked to synchronize each PRU core. PRU-BB will continually increment the shared count, and PRU-CC will track its own internal count and increment to match the shared count when data is received/written. The actual value of the count is unimportant and will persist between multiple reads/writes.

### State

PRU-CC is in charge of controlling the state of PRU-BB, and this state will be communicated via the Scratchpad.

### Error

If PRU-BB encounters any errors, it will put information about the specific error in the Error register and push to Scratchpad. The list of supported error codes is as follows:

|  |  |
| --- | --- |
| **Error Code** | **Description** |
| 0x00 | No error |
| 0x01 | Invalid CS# de-assert |
| 0x02-0xFFFF | RESERVED |

Table 5. PRU-BB Error Codes

## Errors

For detailed information on eSPI error handling, see [1]-9.2.

### NO\_RESPONSE

If there is an invalid CMD, CRC, or Cycle type, just wait until CS# deasserts and then head to Ready state.

### Invalid CS# Deassert

If CS# is improperly deasserted, the hardware will automatically tri-state meeting the tSHQZ timing requiremt. Immediately jump to Ready state.

### FATAL\_ERROR

Finish reading the packet, and then respond with FATAL\_ERROR after the TAR.

### NON\_FATAL\_ERROR

Finish reading the packet, and then respond with NON\_FATAL\_ERROR after the TAR.

# ESPI Protocol

## Overview

At its core, an ESPI transaction consists of a **Command Phase**, **TAR Phase**, and **Response Phase**. The master always initiates the transaction with the Command Phase, however the slave can signal that it wishes to communicate during the **Alert Phase**.

## Command Phase

The following is an example Command Phase showing a generic transaction. For more information see section [1]-4.2.

…

CS# Assert

Begin Reading

CMD Byte

Read CMD Byte/Count++

Packet byte 1

Read Packet byte 1/Count++

Packet byte N

Read Packet byte N

Write CRC Byte

Read CRC Byte

Copy CMD

Handle Packet byte 1

Jump to CMD label

…

…

…

Handle Packet byte N

Handle CRC Byte

Update Bitbang State: TAR

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

Check Next State

Check Next State

Check Next State

Check Next State

Jump Next State

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

time

HW Pins

PRU0

PRU1

Scratchpad

D

C

S

E

Data

Count

State

Error

Unread

Just read

Figure 4. Command Phase Timing

## TAR Phase

After 2 clock cycles, the slave must respond to the master. Below is the general TAR case. The next state and first byte to be written must be available by the last SCL Low, which would give us ~15 PRU clock cycles to Handle the CMD. To add extra time, we will send one WAIT\_STATE which guarantees 2 eSPI clock cycles or 20 PRU clock cycles (for more see Section 3.6). Check for CS# deassertion each clock cycle. For more information see section [1]-4.3.

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

SCL High

Handle CMD

SCL Low

SCL High

Read Next State

SCL Low

Update Bitbang State: Write

Prepare Write Byte 0

Jump Next State

D

C

S

E

time

HW Pins

PRU0

Scratchpad

PRU1

SCL Low

D

C

S

E

Clear Alert#/EN

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

SCL Low

Write WAIT\_STATE

(worst case 2 eSPI cycles)

SCL High

D

C

S

E

Data

Count

State

Error

Unread

Just read

Figure 5. TAR Phase Timing

## Response Phase

Immediately after the TAR phase, the slave must respond after the falling edge of SCL and before the next rising edge and all data must be clocked out on the falling edge. The following is a generic Response transaction. For more information see section [1]-4.4.

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

D

C

S

E

Copy Byte 0

Check Count

time

HW Pins

PRU0

Scratchpad

PRU1

Prepare Write Byte 1

Count++

Byte 0

Copy Byte 1/Check Next State

Count++

Byte 1

…

Check Count

Write Byte 0

Write Byte 0

Prepare CRC Byte

Copy CRC Byte/Check State

…

…

…

D

C

S

E

Update Bitbang State: End

D

C

S

E

Write CRC Byte

Jump Next State

D

C

S

E

D

C

S

E

CRC Byte

D

C

S

E

Data

Count

State

Error

Unread

Just read

Figure 6. Response Phase Timing

## Alert Phase

If the status register changes, raise the Alert# pin, update the previous STS register and return to the Ready state to wait for a CS#. Handle the read as normal, and lower the Alert# pin during the TAR or ERROR (whichever happens first).

## Wait State

The master can configure the maximum number WAIT\_STATES allowed before a response must be returned. We are guaranteed at least one WAIT\_STATE, so to simplify implementation we will always send exactly one WAIT\_STATE.

# CRC Calculation

To quickly calculate CRC on each byte, PRU-CC memory holds a lookup table pre-computed based on the values in [1]-6.2. Refer to [2]-4.4 for details on pre-computing the lookup table and using the lookup table for CRC calculation.

# Slave Registers

## Device Identification

There is only one field in this register, which is the ESPI Version ID. (Red fields indicate unsupported features that break spec)

## General Capabilities and Configurations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Bit #** | **R/W?** | **Init Val** | **Init Description** | **Register Name** | **Notes** |
| 31 | RW | 0x0 | Disabled | CRC Checking Enable |  |
| 30 | RW | 0x0 | Disabled | Response Modifier Enable | Not supported |
| 29 | RO | 0x0 |  | RESERVED |  |
| 28 | RW | 0x1 | Alert# Pin | Alert Mode | Pin Only |
| 27 | RW | 0x00 | Single I/O | I/O Mode Select |  |
| 26 |
| 25 | RO | 0x11 (AM335x) 0x00 (AM437x) | Quad I/O (AM335x) Single I/O (AM437x) | I/O Mode Support |  |
| 24 |
| 23 | RW | 0x0 | Driven Alert# | Open Drain Alert# Select |  |
| 22 | RW | 0x000 | 20 MHz | Operating Frequency |  |
| 21 |
| 20 |
| 19 | RO | 0x0 | No open-drain | Open Drain Alert# Supported |  |
| 18 | RO | 0x000 | 20MHz | Maximum Frequency Supported |  |
| 17 |
| 16 |
| 15 | RW | 0x0 | 16 Wait States | Maximum WAIT\_STATE Allowed |  |
| 14 |
| 13 |
| 12 |
| 11 | RO | 0x0 |  | RESERVED |  |
| 10 |
| 9 |
| 8 |
| 7 | RO | 0x0000 0001 | Peripheral Only | Channel Supported |  |
| 6 |
| 5 |
| 4 |
| 3 |
| 2 |
| 1 |
| 0 |

Table 6. General Capabilities and Configurations Slave Register

## Channel 0 Capabilities and Configurations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Bit #** | **R/W?** | **Init Val** | **Init Description** | **Register Name** | **Notes** |
| 31 | RO | 0x0 |  | RESERVED |  |
| 30 |
| 29 |
| 28 |
| 27 |
| 26 |
| 25 |
| 24 |
| 23 |
| 22 |
| 21 |
| 20 |
| 19 |
| 18 |
| 17 |
| 16 |
| 15 |
| 14 | RW | 0x001 | 64 Bytes | Peripheral Channel Maximum Read Request Size |  |
| 13 |
| 12 |
| 11 | RO | 0x0 |  | RESERVED |  |
| 10 | RW | 0x001 | 64 Bytes | Peripheral Channel Maximum Payload Size Selected |  |
| 9 |
| 8 |
| 7 | RO | 0x0 |  | RESERVED |  |
| 6 | RO | 0x001 | 64 Bytes | Peripheral Channel Maximum Payload Size Supported |  |
| 5 |
| 4 |
| 3 | RO | 0x0 |  | RESERVED |  |
| 2 | RW | 0x0 | Disabled | Bus Master Enable |  |
| 1 | RO | 0x0 | Disabled | Peripheral Channel Ready |  |
| 0 | RW | 0x1 | Enabled | Peripheral Channel Enable |  |

Table 7. Channel 0 Capabilities and Configurations

## Channel 1 Capabilities and Configurations

Not supported.

## Channel 2 Capabilities and Configurations

Not supported.

## Channel 3 Capabilities and Configurations

Not supported.

# Commands

For a full list of eSPI commands see Table 3 in [1]-4.2. Currently we are supporting commands required for basic eSPI operation as well as commands which mimic LPC functionality (i.e. the SHORT commands). All other commands are not currently supported.

## Supported Commands

* GET\_STATUS
* GET\_CONFIGURATION
* SET\_CONFIGURATION
* RESET
* PUT\_IORD\_SHORT (non-posted)
* PUT\_IOWR\_SHORT (non-posted)
* PUT\_MEMRD32\_SHORT (non-posted)
* PUT\_MEMWR32\_SHORT (posted)

## GET\_STATUS

Simply read the STS register and send it to the master.

## GET\_CONFIGURATION

If requested address is out of bounds (> 0x3000) respond with FATAL\_ERROR. If address is Reserved, respond with 0x0000. There are only 6 valid addresses, so a jump table will be used to determine Reserved addresses.

## SET\_CONFIGURATION

If requested address is out of bounds, respond with FATAL\_ERROR. If address is Reserved, don’t update the register. There are only 6 valid addresses, so a jump table will be used to determine Reserved addresses.

## RESET

Write default values into the slave registers and head back to Ready state.

## Posted Transaction

Posted transactions will read each byte of the packet into the next open queue slot for the appropriate channel (if available). If not available, immediately respond with FATAL ERROR. The header size of each packet type is known and once the variable data length is known, the firmware will read all data bytes in a loop, then update the bitbang state to TAR, and read/check the CRC byte. Check that the address is valid, if not immediately respond FATAL ERROR. At this point, the INTC can be used to send an interrupt to the Host signaling that there is data to handle. Finally, respond with ACCEPT.

## Non-Posted Transaction

The first part of a Non-Posted transaction is handled the same as a Posted transaction. But once data is available to the Host with no errors, we will respond with DEFER. Once the Host has finished processing the request, it will update the Status Register triggering an Alert# so the Master knows new data is available.

## Transaction Ordering

Posted transactions must be allowed to complete before Non-Posted transactions. In the driver, each queue will be managed by its own Task The driver needs to maintain ordering and must ensure that the NP\_AVAIL bit isn’t set if there are pending Posted writes ahead of any Non-Posted reads ([1]-5.3). Currently only a single-item “queue” is supported so this will be handled by checking the PC\_AVAIL bit when a new NP request arrives.

### **RX/TX In-Out Dependency**

“An eSPI agent cannot make freeing up of the RX queue for a channel dependent on the forward progress of the corresponding TX queue.” ([1]-5.4) Currently this will be handled by responding with a NON\_FATAL\_ERROR when the TX queue is full (communicated via the Internal Status Register). The TXQ-full bit must be set before the RXQ-full bit is cleared. When the slave writes out a TX packet to the master, it will clear the TXQ-full bit.

# PRU/Host Communication

## Memory Map

|  |  |
| --- | --- |
| 0x00 | Slave Status Register |
| 0x04 | Internal Status Register |
| 0x08 | IPC PRU to Host |
| 0x0C-0x1F | RESERVED |

Table 8. Memory Map

## Save Status Register

The Status Register will be held in shared DRAM and polled in the Ready state so the Alert# pin can be quickly raised when the host changes the register. This register will also be used to track the status data structures and IPC.

## Internal Status Register

The host and PRU will use an internal status register to keep track of anything not covered by the Slave Status Register.

|  |  |
| --- | --- |
| **Bit** | **Description** |
| 31 | RESERVED |
| 30 |
| 29 |
| 28 |
| 27 |
| 26 |
| 25 |
| 24 |
| 23 |
| 22 |
| 21 |
| 20 |
| 19 |
| 18 |
| 17 |
| 16 |
| 15 |
| 14 |
| 13 |
| 12 |
| 11 |
| 10 |
| 9 |
| 8 |
| 7 |
| 6 |
| 5 |
| 4 |
| 3 |
| 2 |
| 1 | Ch0 PC TXQ Full |
| 0 | Ch0 NP TXQ Full |

Table 9. Internal Status Register

## IPC PRU to Host

There are 8 INTC channels which the PRU can use to interrupt the Host. eSPI will use 2 of those interrupt channels, one to signal that RX Data is available, and another to signal Error/Reset.

### RX Data Available

The PRU will raise this interrupt every time data has been placed into the RX queue.

### IPC Commands

The following commands are to be written into the IPC Command register when the Error/Reset interrupt is asserted:

|  |  |
| --- | --- |
| **Message #** | **Description** |
| 0x0000 | RESERVED |
| 0x0001 | In-band Reset |
| 0x0002 | ERROR |
| 0x0003 – 0xFFFF | RESERVED |

Table 10. IPC Commands

## RESERVED

20 bytes are reserved to add any new registers in the future. For example, it may be necessary to add an IPC Host to PRU register.

## IPC Host to PRU

The host will primarily communicate with the PRU by changing the “channel enable” statuses or by changing the Status Register. The host may also disable and reset the PRU if necessary.

## RX/TX Buffers

For now, each buffer will be a fixed size and broken up into fixed number of packets. Each channel tracks its own buffers, so the packet size (and thus number of packets) may be different for each channel. **Currently only a single-depth queue is supported.**

### Peripheral Channel

There are separate RX/TX queues for Posted and Non-Posted transactions. Currently each queue is only one 256-byte-slot deep. The largest Peripheral packet header is 11 bytes, therefore we can support 128 byte packages.

### Virtual Wires Channel

Not currently supported.

### OOB Message Channel

Not currently supported.

### Flash Access Channel

Not currently supported.

# RTOS Driver

## ESPI\_init

Create a semaphore for each PRU core and initialize all object fields to NULL.

## ESPI\_open

Initialize the PRUs, load code to each PRU, setup Pinmuxing, enable interrupts, setup semaphores, and finally enable active eSPI channels. Start IRQ handler Task.

### IRQ Handler

Each time an RX available interrupt is fired by the PRU, the Host must determine which channel the interrupt is associated with and Post a semaphore, opening up operation for any pending tasks on that channel.



Figure 7. Driver IRQ Handler Flow

## ESPI\_transfer

SPI\_Transfer(read) will check RX queues in order of priority (Posted first), if there are packets forward them to the application for processing. Else enable interrupts and Pend until interrupted when new data enters the queue. SPI\_Transfer(write) will write the outgoing packet into the TX queue and update the Status Register accordingly. For now there has to be coupling between RX and TX processing where the RX operation will actually be setting the TXQ-full bit if a Non-Posted message comes in.

### SPI\_Transfer(read)



Figure 8. SPI\_Transfer RX Flow

### SPI\_Transfer(write)



Figure 9. SPI\_Transfer TX Flow

## ESPI\_control

At the moment ESPI\_control doesn’t do anything, all running configuration is done by the master.

## ESPI\_close

Clear all PRU memories, disable all interrupts, destroy all semaphores. Stop IRQ Handler Task.

## Reset

The driver will handle Reset as a hardware interrupt. Cancel the transfer, alert the application that a Reset has just taken place, disable both PRUs, set all PRU memory back to default state, reset both PRUs to main, and run anew.

## Application Layer

Each channel will have its own Task, the general call flow within that Task is as follows:



Figure 10. Basic Application Flow

# EVM Support

## bbbAM335x

AM335x has only one ICSS thus we will be using ICSS1. The BeagleBone Black has limited output pins connected to the PRUs, but PRU1 has enough pins to run full quad-mode so PRU1 will be in charge of bitbanging, and PRU0 will be in charge of control.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Core** | **PRU** | **Pin Name** | **GPIO Pin** | **PRU PIN #** | **EVM Port** | **EVM Pin** |
| ICSS1 | PRU1 | IN0 | LCD\_DATA0 | R31:0 | P8 | 45 |
| IN1 | LCD\_DATA1 | R31:1 | P8 | 46 |
| IN2 | LCD\_DATA2 | R31:2 | P8 | 43 |
| IN3 | LCD\_DATA3 | R31:3 | P8 | 44 |
| OUT0 | LCD\_DATA0 | R30:4 | P8 | 41 |
| OUT1 | LCD\_DATA1 | R30:5 | P8 | 42 |
| OUT2 | LCD\_DATA2 | R30:6 | P8 | 39 |
| OUT3 | LCD\_DATA3 | R30:7 | P8 | 40 |
| SCL | LCD\_VSYNC | R31:8 | P8 | 27 |
| ALERT | LCD\_HSYNC | R30:9 | P8 | 29 |
| EN | LCD\_PCLK | R30:10 | P8 | 28 |
| CS | LCD\_AC\_BIAS\_EN | R31:11 | P8 | 30 |
| A8 | n/a | RESET | GPIO3\_19 | n/a | P9 | 26 |

Table 11. bbbAM335x Pin Use

## idkAM437x

AM437x has two ICSS systems, but ICSS0 has no scratchpad, limited IEP access, and limited memory therefore ICSS1 must be used. IDKAM437x has a very limited external pin count connected to PRUs, but PRU0 has the most available and can support single-mode only, thus PRU0 will be in charge of bitbanging, and PRU1 will be in charge of control.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Core** | **PRU** | **Pin Name** | **GPIO Pin** | **EVM Port** | **EVM Pin** |
| ICSS1 | PRU0 | IN0 | GPMC\_AD14 | P16 | 42 |
| OUT0 | LCD\_DATA6 | P16 | 49 |
| SCL | GPMC\_AD12 | P16 | 46 |
| ALERT | GPMC\_AD13 | P16 | 48 |
| EN | GPMC\_CSN1 | J3 | 6 |
| CS | GPMC\_CSN2 | J3 | 8 |
| A9 | n/a | RESET | GPIO5\_9 | J16 | 32 |

Table 12. idkAM437x Pin Use

# External Board

In order to meet timing requirements, voltage requirements, and to simulate GPIO, external hardware is needed.

## General Design

The external board needs to tri-state the I/O lines tSHQZ after CS is deasserted. The PRU output pins are set to 3.3V, and the eSPI spec calls for 1.8V so a voltage translator must be used. Also due to limitations of the PRU there are no shared GPIO pins, but separate GPI and GPO pins. This is accounted for by allowing the PRU to tristate the GPO pins when not in use with the EN pin.

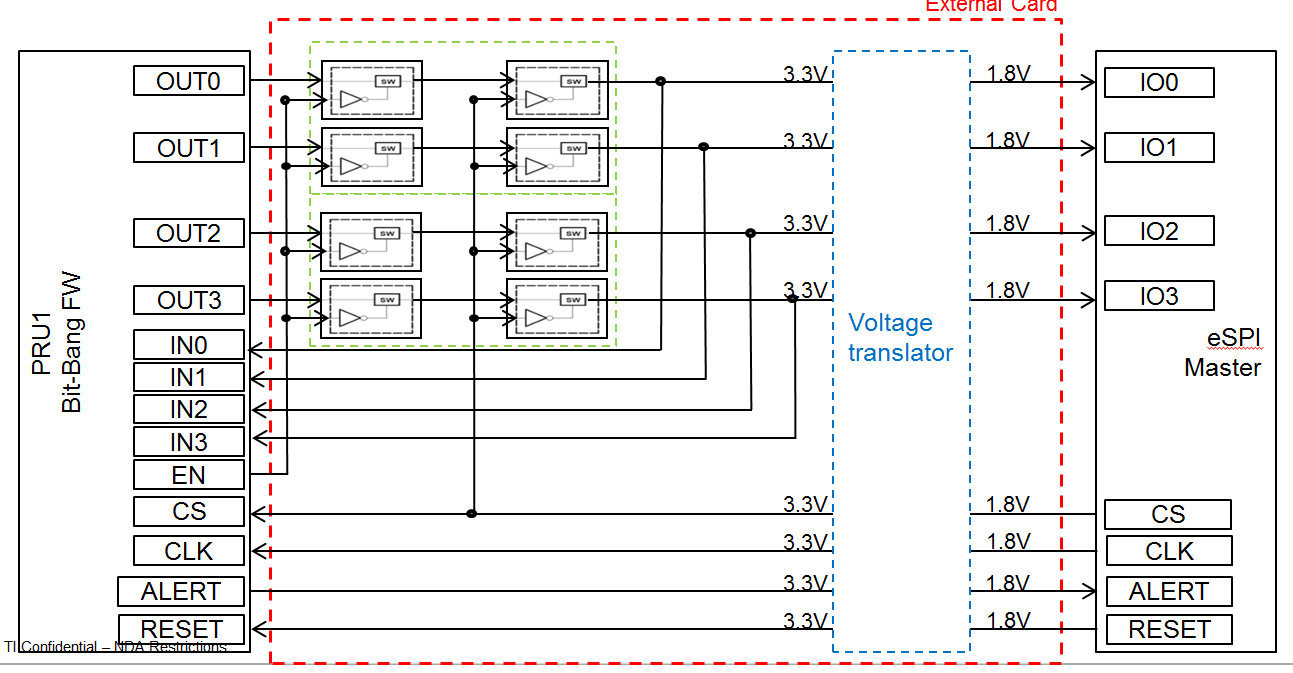


Figure 11. External Hardware Design

## bbbAM335x

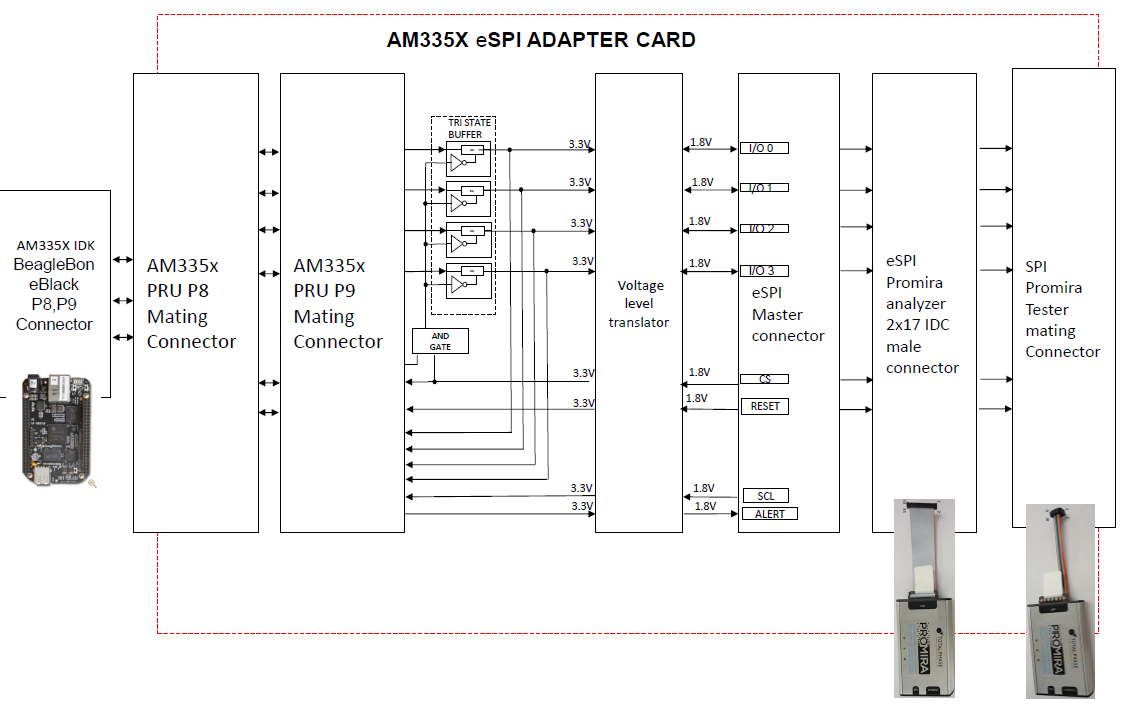


Figure 12. bbbAM335x Test Hardware

## idkAM437x

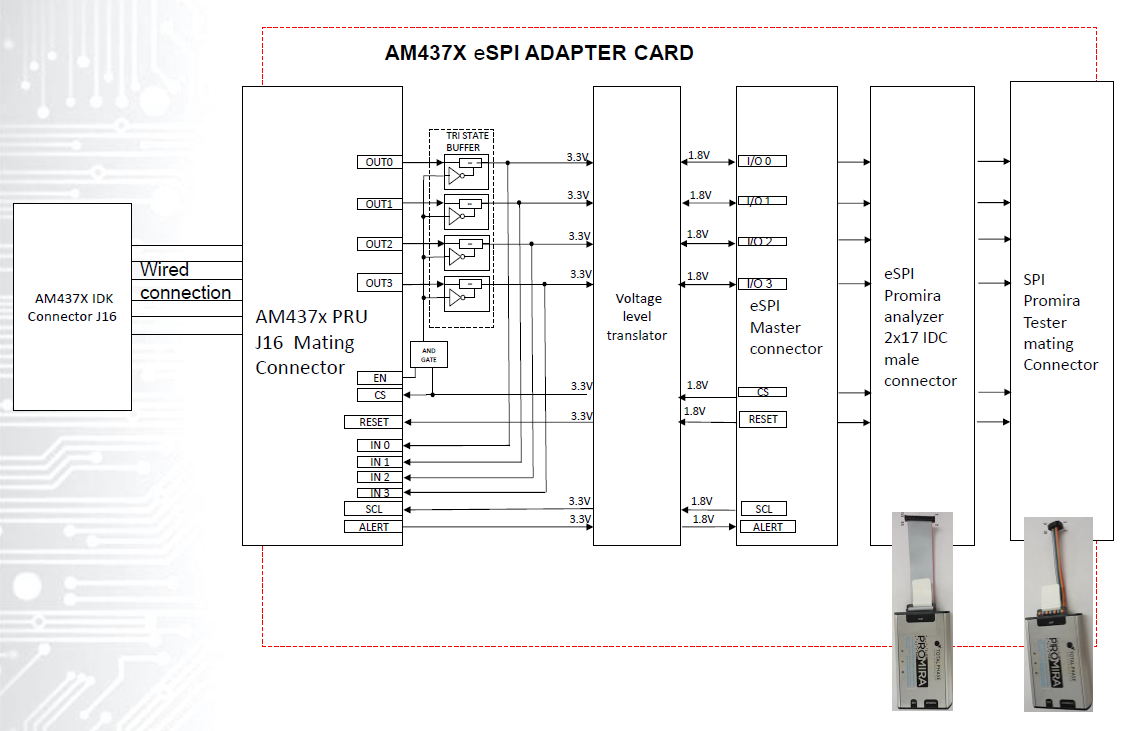


Figure 13. idkAM437X Test Hardware

# Testing

## Total Phase Promira Serial Platform

Total Phase makes a tool called the “Promira Serial Platform” which can act as both eSPI master and as a protocol analyzer. We will employ two for such purposes.

# Limitations

## Limited Frequency Support

Due to the 200MHz PRU-ICSS clock, only 20MHz eSPI is supported

## Limited Channel Support

Only Channel 0 is currently supported, and within Channel 0 only the SHORT transactions are supported.

## Slow Clock Not Supported

If the clock frequency drops below ~18MHz, data written by slave may be read early by the master.

## Slave-Initiated Transactions

Following from 12.1, there are currently no slave-initiated transactions supported (SHORT transactions are master to slave only)

## Response Modifiers

Response modifiers are not currently supported.

## No Shared Alert# Pin

[1] defines an ability to use I/O[1] in place of a dedicated Alert# pin in Single mode. This is not currently supported.

## Queue Depth

For this initial implementation, the simplest design possible is to use message “queues” with only one entry. However, it obviously benefits the protocol to handle multiple messages at a time so deeper queues should be supported in the future.

# References

* [1] Intel Enhanced Serial Peripheral Interface (eSPI) Interface Base Specification Rev1.0
* [2] Understanding and implementing the CRC (Cyclic Redundancy Check) calculation http://www.sunshine2k.de/articles/coding/crc/understanding\_crc.html#ch44