OSPI PHY Configuration and Training

Contributed by Nagarajan, Vignesh *Rev 1 – October 31, 2022*

Introduction

This application note describes the Octal SPI (OSPI) PHY mode and provides programing and board design guidelines. It also describes PHY calibration and the usage of the ADI SSLD PHY driver. This note applies to the ADSP-2159x and ADSP-SC59x processors families. It is an addendum to the processor data sheets[1], [2] and the hardware reference manuals[3], [4].

OSPI PHY Mode

The OSPI PHY mode extends the architecture of the OSPI flash controller to allow the interfacing of high-speed flash devices and ensure the reliable data transfers at high speed. The PHY is a physical layer interface of the OSPI controller that is responsible for driving and sampling data/clocks from the OSPI interface pins at high speed. By default, PHY mode is not enabled in the OSPI controller and all transaction bypass the PHY layer. Operations are limited to REFCLK/4 in SDR mode and REFCLK/8 in DDR mode. The clock is limited to 62.5 MHz. Emerging flash devices can operate at higher frequencies close approximating 200 MHz. The implementation of the PHY module that is dedicated to work with the OSPI flash controller enables performing DDR/SDR transfers at these higher frequencies. With PHY mode enabled, the OSPI module can operate at up to 125 MHz.

PHY mode supports a DQS (data strobe) feature which allows some flash devices to reliably operate at higher speeds. Because the flash drives DQS with the data of flash read operations, the skew between the data and DQS is much smaller and more predictable than a similar operation between data from flash and a clock from the OSPI controller. This difference permits reliable sampling of data at higher frequencies. When DQS is enabled, the OSPI in PHY mode can operate at up to 125 MHz.

When a flash device does not support DQS, the OSPI PHY can still operate with an internal OSPI reference clock or a loopback clock. In this case, the frequency of operation is limited to 80 MHz. The PHY interface has a mechanism to adjust the sampling and driving of data through internal DLLs. The mechanism can fine tune the timing delays for the data capture. These DLLs must be configured to appropriate values for correct sampling before driving or sampling data. The sampling depends on the flash device, board traces, and operating conditions. Before performing any data accesses, these delays must be configured to appropriate values. The values must be determined through a PHY calibration training sequence for both DQS and non-DQS modes.

The ADI-supplied device driver (SSLD) for the OSPI module provides an API to train and optimally configure the PHY before performing data accesses to flash device.

OSPI PHY Operating Modes

The OSPI PHY has two operating modes: DLL controller mode and DLL bypass mode. The modes are controlled by a master DLL and a slave DLL delay line. The mode depends on whether the master DLL is bypassed.

DLL Controller Mode

In DLL controller mode, the master delay line determines how many delay elements constitute a complete cycle. This operation occurs when the DLL goes through the locking process to secure the one clock cycle of the interface clock. The lock count is used, along with the programmable fractional delay settings, to determine the actual number of delay elements to program into the slave delay lines. This approach allows the controller to observe a clock and then delay other signals at a fixed percentage of that clock. Therefore, in DLL controller mode, when the DLL is locked to a full or half cycle, programming a value of 0x1F (1/4th of full value of 0x7F) in the RXDLL delays the sampling of data by ¼ th of the clock period. Similarly, programming 0x3F in the RXDLL delays the sampling by a half period.

DLL Bypass Mode

In DLL bypass mode, the master DLL is not used. The delay programmed in the slave DLL delay line indicates the absolute delay for the other signals with respect to the clock. The delay achieved is equal to the absolute delay introduced by each delay element in slave delay line multiplied by the number of delay elements programmed for the delay line.

PHY Clocking

The default reference clock for OSPI is SYSCLK. However, in PHY mode, the reference clock is directly used to drive the interface clock (because no scaling of the clock using BAUD divider is supported in PHY mode). Therefore, SYSCLK clock may not be a suitable choice in PHY mode because it is typically programmed to higher frequency values.

To enable support for PHY mode, the reference clock is selected from one of the options from the CDU output clock (CDU CLK10). (See the CDU chapter of the hardware reference). The reference clock is connected to SYSCLK (which is suitable for non-PHY mode of operation). For PHY mode, the reference clock is either SCLK0_0 or SCLK1_1. Since the OSPI interface clock cannot exceed 125 MHz, SYSCLK cannot be always used. SYSCLK may be much higher than 125 MHz. In such cases, ensure that the CDU module is configured to select an appropriate option to clock the OSPI reference clock within 125 MHz. The clock in PHY mode cannot be divided using BAUD setting in the OSPI module; it is ineffective in PHY mode and the interface clock is always equal to reference clock.

PHY Clock Arbiter

The PHY clock arbiter module selects the sampling clock to be passed through the RX delay line. It is generated when the read data phase of the SPI transfer is detected (based on information from the SPI control module). The sampling clock can be optionally sourced from REFCLK, loopback input, or DQS. The decision is made based on the software configuration. In PHY mode, by default, the gated REFCLK is forwarded into the DLL.

REFCLK

REFCLK is an internal clock. In PHY mode, REFCLK is generated from CDU CLKO10; it is either SCLK0_0 or SCLK1_1. In non-PHY mode, REFCLK is used to serialize the data and drive the OSPI interface. The external clock is driven on the OSPI_CLK pin, which is synchronous to the REFCLK. In PHY mode, by default, a gated REFCLK is forwarded to the DLL.

Loopback Clock

The loopback clock is used to prolong the read data window to ensure that more valid samples are available for the tap data capturing mechanism. The loopback clock comes from the delay line for the controller that is externally located. PHY internal logic automatically adjusts this delay and provides the right sampling window. If the loopback clock (`Read Data Capture Register bit [0]`)and PHY mode (`OSPI Configuration Register bit [3]`) are both enabled, the loopback clock is driven into the RXDLL instead of the gated REFCLK.

DQS Clock

Some flash devices support DQS mode. The data strobe signal is an output from the flash device that indicates when data is being transferred from the flash to the host. The data is then captured by the controller on both rising/falling edges of the DQS signal. The flash uses this data strobe signal only for read and not write. If DQS (`Read Data Capture Register bit [8]`) and PHY mode (`OSPI Configuration Register bit [3]`) are both enabled, the DQS clock is driven into the RXDLL instead of the gated REFCLK.

PHY Training

PHY training is used to calibrate the delay required, fine tune the timing delays required for the transmit and data capture and, configure the DLL (RXDLL and TXDLL) values, accordingly. The DLL delay parameter varies with operating frequency, transfer mode, PHY clock selection, operating temperature, etc.

The read operation is performed iteratively with all combinations of RXDLL, TXDLL and RDCR. A known pattern of read values can be used to verify the read data. The mid value of any wide range of passing pattern of RXDLL and TXDLL can be chosen to configure the DLL registers. When the operating temperature drifts on either side, selecting the right DLL values from a wide range of passing values provides a sustainable operation of the PHY.

Temperature Impact on Passing Range

Processor die temperature affects the IO line delays and can cause a change in the passing range boundaries. The passing range of TDLL and RXDLL can shrink or extend based on the operating temperature, board design, and other factors.

Figure 1 and Figure 2 show the RXDLL and TXDLL passing range for different temperature ranges. It is recommended to choose the mid value of the passing range for optimal operation when the silicon temperature drifts (even after calibration and DLL configuration).

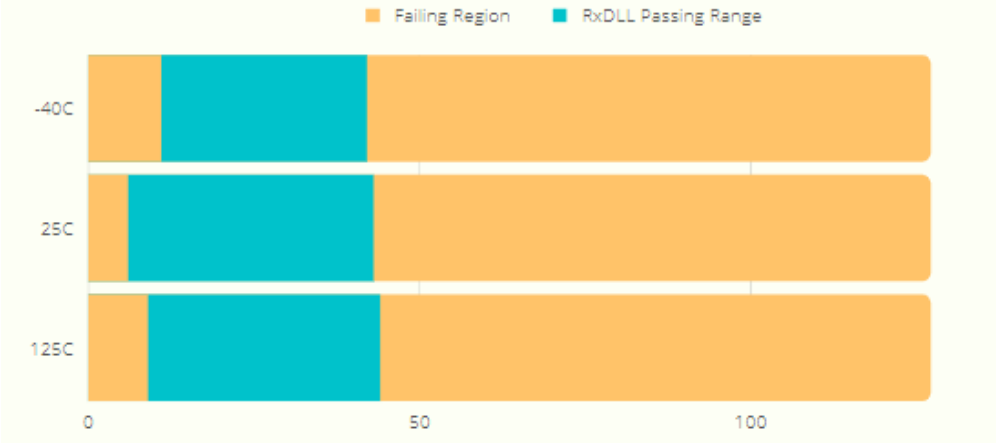

Figure 1: RxDLL Passing Range vs. Temperature

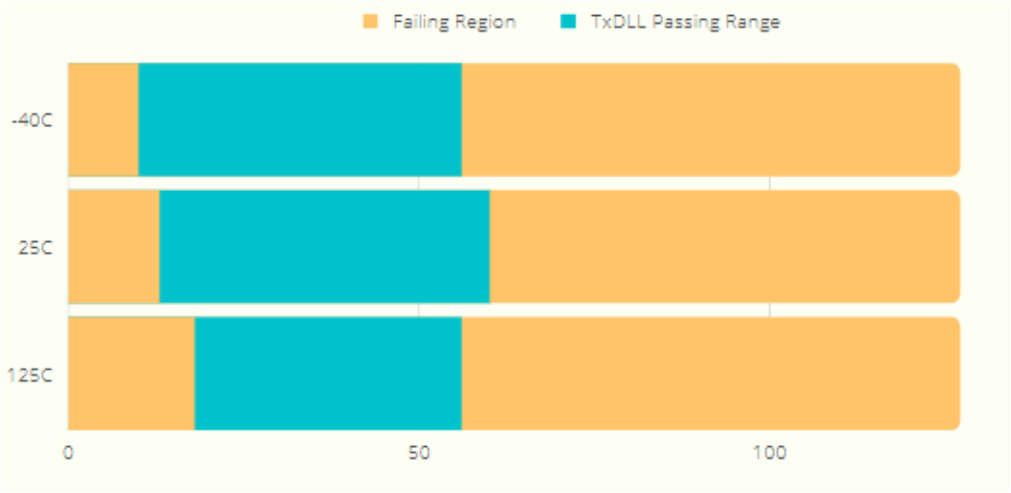

Figure 2: TxDLL Passing Range vs. Temperature

In Figure 1 and Figure 2, passing values were determined by randomly selecting sample values during multiple in-house experiments. Passing ranges and values vary with respect to operating frequency, clock source, and other factors.

SSLD PHY Training Algorithm

The ADI SSLD PHY driver is enabled with a PHY training sequence to calibrate and choose the optimal RXDLL and TXDLL values for the selected clock configurations (REFCLK, LB and DQS). PHY training supports all OSPI transfer modes (single, DPI, dual IO, QPI, quad IO, OPI and octal IO) and both SDR and DDR modes.

The training algorithm configures the PHY mode based on user settings. It reads a known pattern from OSPI memory and iterates using multiple RXDLL, TXDLL and RDCR values. By considering different parameters, the algorithm chooses the optimal value of RXDLL and TXDLL from the larger passing range of values. This approach assists in operating the OSPI PHY continuously, even when the temperature has drifted after training.

Optimized and Non-Optimized PHY Training Mode

Non-optimized mode is the most robust method as it uses a brute force method in which all combinations of RXDLL, TXDLL and RDCR are iterated (128 RX DLL x 128 Tx DLL x 16 RDCR). The algorithm iterates through these different parameters as follows:

1. Program RDCR with values from 0x0 to 0xF.
2. For each RDCR value programmed, iterate through all values of TXDLL from 0x0 to 0x7F before programming the next RDCR value.
3. For each TXDLL value programmed, iterate through all values of RXDLL from 0x0 to 0x7F before programming the next TXDLL value.

The algorithm chooses the first passing value of RDCR. In this passing RDCR value, the algorithm scans through the TXDLL values and identifies the middle TXDLL value in the passing range. (The first passing TXDLL value is the one where the reads start passing for at least two RXDLL values; the last TXDLL is the one after which the following TXDLL has read passing for none or less than two passing RXDLL values). Once the middle TXDLL value is identified, the algorithm searches for the passing RXDLL range for this TXDLL value, and then selects the mid value of RXDLL.

In optimized mode, which is enabled by default in the SSLD, the calibration is iterated within a reduced range of RXDLL and TXDLL values. This mode minimizes the calibration time while maintaining the reliability of the calibration as much as possible. Rather than sweeping through each value of the RDCR, RXDLL and TXDLL (which takes up much more time than might be acceptable in the user application), some optimizations are performed to reduce the overall time.

To reduce calibration time, the algorithm:

1. Limits the number of passing RXDLL settings to 16 or 32, as defined by the `RXDLL_PASSLIMIT` macro in the driver. For each configured TXDLL, the algorithm stops when it finds `RXDLL_PASSLIMIT` number of settings.
2. Limits the number of passing TXDLL settings to 16 or 32, as defined by the `TXDLL_PASSLIMIT` macro in the driver. For each configured RDCR, the algorithm stops when it finds `TXDLL_PASSLIMIT` number of settings.
3. If number of passing settings for RXDLL or TXDLL is less than `RXDLL_PASSLIMIT` or `TXDLL_PASSLIMIT`, the algorithm stops sweeping through RXDLL or TXDLL at the first failed setting (for example, the first failed setting after a sequence of passing settings indicating the end of passing). It is not necessary to sweep through the range until a max value algorithm; the likelihood of find any passing setting after this point is very low. This method assumes that there is only one passing band in the entire range of RXDLL values for a given TXDLL setting and only one passing range of TXDLL values for a given RDCR setting.
4. Once a passing RDCR value is identified, no further RDCR values are checked.

Figure 3 shows the optimized mode algorithm flow used to calibrate and configure the RDCR, TXDLL and RXDLL. It's recommended to run this algorithm whenever a user changes an OSPI setting.

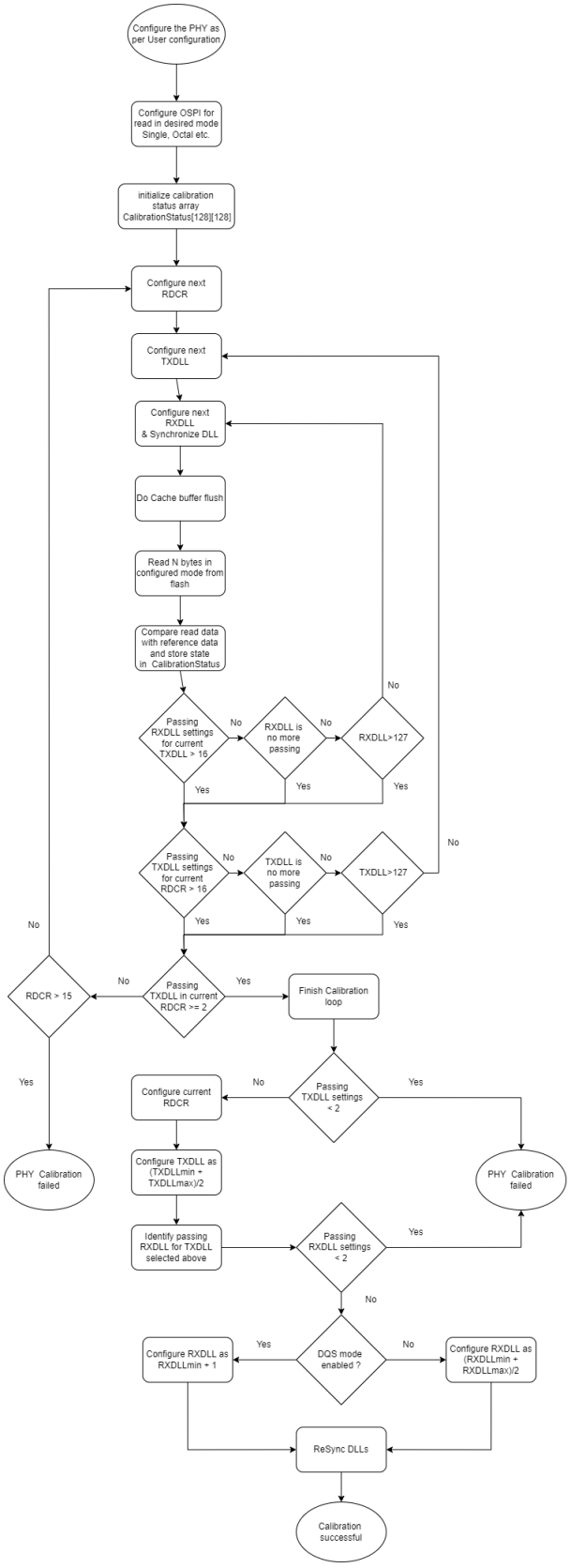
Configure the PHY as per User configuration
Configure OSPI for read in desired mode Single, Octal etc.
initialize calibration status array CalibrationStatus[128][128]
Configure next RDCR
Configure next TXDLL
Configure next RXDLL & Synchronize DLL
Do Cache buffer flush
Read N bytes in configured mode from flash
Compare read data with reference data and store state in CalibrationStatus
Passing RXDLL settings for current TXDLL > 16
RXDLL is no more passing
RXDLL>127
Passing TXDLL settings for current RDCR > 16
TXDLL is no more passing
TXDLL>127
RDCR > 15
Passing TXDLL in current RDCR >= 2
Finish Calibration loop
Passing TXDLL settings < 2
Configure current RDCR
Configure TXDLL as (TXDLLmin + TXDLLmax)/2
PHY Calibration failed
PHY Calibration failed
Identify passing RXDLL for TXDLL selected above
Passing RXDLL settings < 2
DQS mode enabled ?
Configure RXDLL as RXDLLmin + 1
Configure RXDLL as (RXDLLmin + RXDLLmax)/2
ReSync DLLs
Calibration successful
Yes
No

Figure 3: Optimized Mode Algorithm Flow Chart

SSLD PHY Configuration Structure

The OSPI SSLD driver provides a configuration structure that defines the settings for PHY initialization.

Listing 1 shows the configuration settings. This structure must be configured before calling `adi_ospi_PhyInit API`. Table 1 describes the parameters used in the SSLD_PHY structure.

```
typedef struct
{
  bool bPhyPipelineModeEn;
  ADI_OSPI_PHYBYPASSMODE ePhyBypassMode;
  ADI_OSPI_TRANSFERMODE eTransferMode;
  ADI_OSPI_DAC_CMD * TransferCommand;
  uint8_t* pDataPattern;
  bool bAutoCalibEn;
  bool bDqsEnable;
}ADI_OSPI_PHYCONFIGURATION;
```

Listing 1: ADI_OSPI_PHYCONFIGURATION Structure

Parameter	Description
`bPhyPipelineModeEn`	Pipeline PHY mode enable /disable

`ePhyBypassMode`	PHY Master DLL / Bypass Mode
	`ADI_OSPI_PHYBYPASS_DIS = 0,`
	`ADI_OSPI_PHYBYPASS_EN = 1`

`eTransferMode`	SDR/DTR – Selection
	`ADI_OSPI_TRANSFERMODE_STR,`
	`ADI_OSPI_TRANSFERMODE_DTR,`
	`ADI_OSPI_TRANSFERMODE_DTR_P`

`TransferCommand`	DAC read command – Specifies flash read command, bus mode, transfer mode, DMA/core mode etc. Refer to the `ADI_OSPI_DAC_CMD` structure.
`pDataPattern`	Known pattern data to compare against read data during PHY calibration
`bAutoCalibEn`	PHY auto calibration enable/disable, while initializing PHY mode by calling `adi_ospi_PhyInit` API
`bDqsEnable`	DQS mode clock enable /disable

Table 1: SSLD PHY Configuration Structure

SSLD PHY Calibration Optimization

PHY calibration optimization is configured using macros.

- `#define ADI_OSPI_PHYCALIB_OPTIMIZE 1U` – By default, optimization is enabled to reduce the calibration time.

- `#define ADI_OSPI_PHYCALIB_OPTIMIZE 0U` – Configure this macro zero to disable optimization, in which all combination of TXDLL, RXDLL and RDCR values are iterated. This consumes more time.

SSLD PHY API Functions

Table 2 describes the API functions used to configure and control the OSPI PHY.

API	Description
`adi_ospi_PhyInit`	This function configures the PHY controller for operating the OSPI PHY with user configured modes. Optional PHY auto calibration is included.
`Adi_ospi_PhyDeinit`	This function uninitializes the OSPI PHY controller
`adi_ospi_DllResynchronize`	This function performs re-synchronization of delay lines. This API needs to be called whenever the user changes the DLL settings.
`Adi_ospi_PhySetTxDelay`	This function configures the Tx delay, which determines the number of delay elements to insert on the data path between the reference clock and the OSPI clock. Valid new delay value must be passed as a parameter. Note: `adi_ospi_DllResynchronize` must be called after setting a new Tx delay value
`adi_ospi_PhySetRxDelay`	This function configures the Rx delay, which determines the number of delay elements to insert on the data path between the reference clock and the sampling clock. A valid new delay value must be passed as a parameter. Note: `adi_ospi_DllResynchronize` must be called after setting a new Rx delay value.
`Adi_ospi_PhySetRxBypass`	This function controls whether the DLL operated in master DLL mode or bypass mode
`adi_ospi_PhyGetDllStat`	This function is used to retrieve the following PHY DLL status information: * DLL lock status * DLL locked mode * DLL unlock counter * DLL lock value * DLL loopback done * DLL cumulative lock decremental steps * DLL cumulative lock incremental steps * DLL Rx delay value * DLL Tx delay value

Table 2: SSLD PHY API

SSLD PHY Calibration Example

Listing 2 shows a code snippet of PHY calibration using the OSPI SSLD driver.

The complete source code for PHY calibration with a project is available in the board support package. For example, `AnalogDevices/EV-SC59x_EZ-KIT-Rel2.0.0/EV-SC59x_EZ-KIT/Examples/drivers/ospi/ospi_phymodecalibration`. The latest ADSP-SC59x/2159x EZ-KIT Board Support Package can be downloaded from the Analog Devices website.

```
/* ********************************************************************************
 *
 * Copyright © 2022 Analog Devices Inc.  All rights reserved.
 *
 * ********************************************************************************/

ADI_OSPI_DAC_CMD DAC_Command = {
  0x0B,                         /* command opcode */
  0x00,                         /* 2nd command byte opcode (Optional) */
  ADI_OSPI_OPI,                 /* Bus mode */
  ADI_OSPI_TRANSFERMODE_DTR_P, /* Transfer mode */
  OSPI_ADI_COREMODE,            /* DMA mode or Core mode */
  ADI_OSPI_CMD_ADDR_SIZE_4,     /* Number of Address Bytes */
  16,                           /* Number of dummy bytes */
  0,                            /* Whether Mode data will be sent */
  0,                            /* Mode byte data */
  (uint8_t *)FLASHADDRESS,      /* Source Pointer to the data buffer in Core mode */
  (uint8_t *)ReadBuff,          /* Dest Pointer to the data buffer in Core mode */
  4,                            /* Size of data buffer in Core mode */
};

int main (int argc, char *argv [])
{
  ADI_OSPI_RESULT eResult = ADI_OSPI_SUCCESS;
  ADI_OSPI_PHYDLLSTATUS PhyDllStatus;
  ADI_OSPI_PHYCONFIGURATION *pPhyConfiguration;
  ADI_OSPI_PHYCONFIGURATION PhyConfiguration;
  pPhyConfiguration = &PhyConfiguration;

  /* Default Pinumux and Interrupt Inits */
  /* SPU Init and Config */

  /* OSPI Driver open */
  /* OSPI Call back register */ (Optional)
  /* Any board specific setting */ (Optional)

  /* Configure OSPI pins Drive strength to 2 for high speed */
  *pREG_PADS0_PORTA0_DS =    (OSPI_DS << 0) | (OSPI_DS << 3) | (OSPI_DS << 6) |
                             (OSPI_DS << 9) | (OSPI_DS << 12) | (OSPI_DS << 15) |
                             (OSPI_DS << 18) | (OSPI_DS << 21);

  /* Initialize PHY data structure */
```

```
  /* Enable Pipeline Mode */
  pPhyConfiguration->bPhyPipelineModeEn = true;

  /* Enable Bypass mode */
  pPhyConfiguration->ePhyBypassMode      = ADI_OSPI_PHYBYPASS_EN;

  /* Configure the Transfer mode */
  pPhyConfiguration->eTransferMode = DAC_CMD.eTransferMode;

  /* Link the read command structure */
  pPhyConfiguration->TransferCommand     = &DAC_CMD;

  /* Link the buffer to compare data's, during calibration */
  pPhyConfiguration->pDataPattern  = WriteBuff;

  /* Enable Auto Calibration */
  pPhyConfiguration->bAutoCalibEn  = true;

  /* Disable DQS mode */
  pPhyConfiguration->bDqsEnable          = false;

  /* PHY mode init and calibration */
  eResult = adi_ospi_PhyInit(*phDevice, pPhyConfiguration);
  if(eResult != ADI_OSPI_SUCCESS)
  {
    prin"f("PHY initialization failed: Error %d"\n", eResult);
  }

  /* Read back DLL value set by Auto Calibration */
  adi_ospi_PhyGetDllStat(*phDevice, &PhyDllStatus);
  prin"f("\n------------------------------------------------------------"\n");
  prin"f("Auto Calibration succ"ss");
  prin"f("\nSelected Rx Delay ="%d", PhyDllStatus.dllRxOutput);
  prin"f("\nSelected Tx Delay = %d"\n", PhyDllStatus.dllTxOutput);

  /* Perform the read operation in the configured mode of OSPI and flash */
  eResult = adi_ospi_DirectRead( *phDevice, &DAC_Command_Read );
  if(eResult != ADI_OSPI_SUCCESS)
  {
    prin"f("Read back failed, Error code returned %d"\n", eResult);
  }

  /* Close the OPSI driver */

  return 0;
}
```

Listing 2: OSPI PHY Calibration Code Snippet

Hardware and Design Guidelines

For high-speed signals, maintain good signal integrity in the PCB design. The following general layout guidelines are recommended to prevent signal integrity problems.

- To maintain the timing margins, match trace lengths of all data lines within +/- 3 mils relative to the OSPI_DQS signal. The OSPI_DQS signal should be matched with respect to the clock signal.
- Route all data signals and OSPI_DQS on the same signal layer, having the same ground reference. Changing the ground reference plane can change the trace impedance.
- All data signals and strobe should have the same number of vias and layer changes. This design help to ensure that the data signals along with the accompanying strobe will have the same effective delay.
- Avoid routing over a split plane as the return current path is not able to follow the signal trace. If a signal must be routed over two different reference planes, add a stitching capacitor between the reference planes and place the capacitor close to the signal path.
- If the signals change layers and reference planes from one ground plane to another, add ground vias or capacitors s close to the layer change vias to avoid return path discontinuities.
- Use a ground plane as the primary reference or return paths for all signals. Whenever a power layer is used as the reference plane, it is important to ensure that the power layer is low-noise and there is proper stitching at the reference plane transitions to guarantee return path continuity.
- Maintain at least 3x spacing of the trace width for clock and strobe signals from other signal traces to minimize noise coupling.
- Route all the signals with the controlled impedance (typically 50 ohm) to reduce signal reflection.
- Use minimum number of vias in the clock trace. A via causes impedance discontinuity and signal reflections.
- The OSPI_DQS trace should be shorter than all data lines. Route the clock and strobe line (DQS) as short as possible. Try to avoid using serpentine routing and maintain a straight trace as much as possible, to minimize impedance variation. The maximum recommended trace length for the OSPI signals is 4 inches.
- Keep stubs as short as possible to avoid reflections. Keep stub propagation delay to <20% of the signal rise time.
- The OSPI_MISO signal requires a pull-up resistor. To ensure the correct signal level on the data pins during tri-state (Hi-Z), external pull-ups can be used on data pins (D0-D7).
- The pull-up resistor on the slave select signal ensures that the memory is deselected when the pin is in a high-impedance mode such as during reset.

References

[1] *ADSP-SC595/SC596/SC598 SHARC+ Processor Hardware Reference, Rev 0.2, August 2022. Analog Devices, Inc.*

[2] *ADSP-21591/21593/21594/ADSP-SC591/SC592/SC594 SHARC+ Processor Hardware Reference, Rev 0.4, April 2022. Analog Devices, Inc.*

[3] *ADSP-21591/21593/21594/ADSP-SC591/SC592/SC594: SHARC+ Dual-Core DSP Data Sheet, Rev. A , April 2022. Analog Devices, Inc.*

[4] *ADSP-SC595/SC596/SC598: SHARC+ Dual-Core DSP with Arm Cortex-A55 Preliminary Data Sheet, Rev. PrE, July 2022.*

Document History

Revision	Description
Rev 1 – October 26, 2022, *by Nabeel Shah and Vignesh Nagarajan*	Initial Release.