Software SPI Master

# Introduction

This application note describes the usage of the Software SPI Master (SSM) on Talaria TWO. SSM enables Talaria TWO to act as SPI master using the available GPIOs by generating and processing SPI signals in software. SSM supports a variety of operating modes and the number of SPI interfaces is limited only by the number of available GPIOs. This provides flexibility for Talaria TWO to interface with peripherals in diverse applications.

# Supported Modes and Configuration Options

This section describes the operating modes and options supported by SSM. These options are set in an instance of SPI options structure, spiopts\_t.

## Data Length

SSM can be configured to transfer between 1 and 16 bits (inclusive) per call to spi\_xfer().

## Shift Direction

Transmit data can be shifted out with MSB first or LSB first. Additionally, SSM can be configured to interpret received data as either MSB first or LSB first. Shift direction can be configured independently for transmit and receive directions.

## Clock Phase

SSM supports configuration of clock phase as 0 or 1. This setting is independent of clock polarity. Figure 1 illustrates how this option affects the clock generated by the SSM.

## Clock Polarity

SSM supports configuration of clock polarity as 0 or 1. This setting is independent of clock phase. Figure 1 illustrates how this option affects the clock generated by the SSM.

## 3-Pin / 4-Pin Modes

SSM supports both 3-pin and 4-pin modes of operation. In 3-pin mode, no chip select (CS) is used.

## GPIOs

SSM can be configured to use any GPIO for each of the SPI signals: CLK, MOSI, MISO, and CS (optional).

## Timing

SSM supports configuration of the timing parameters as shown in Figure 1.

# Timing Diagram

Figure 1 illustrates SPI signals generated by the Software SPI Master for each possible combination of clock polarity and phase. The diagram illustrates a 5-bit transfer in 4-wire mode. Timing parameters are also shown.

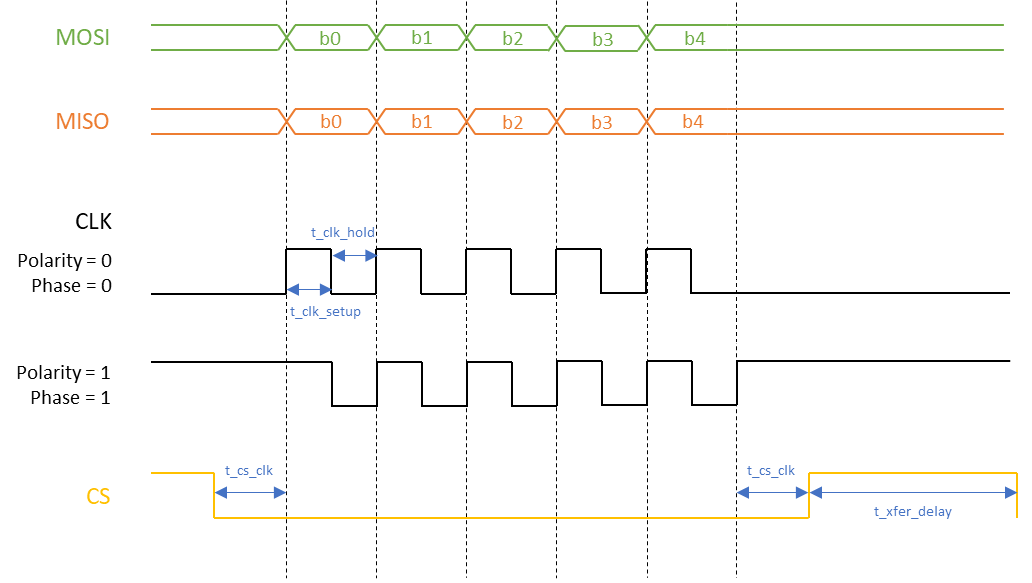


Figure : Timing diagram illustrating SSM operation, 5-bit transfer shown

Data is set up on the MOSI and MISO lines at the times indicated by vertical dotted lines in the diagram (except the last one at the end of the transfer). Data is sampled on the clock edges between the dotted lines.

# IIS2DLPC 3-axis Accelerometer Sample

Software SPI Master is accompanied by a code sample that communicates with an IIS2DLPC 3-axis accelerometer over SPI. This sample is provided in the IIS2DLPC folder alongside the SSM code. The sample puts the IIS2DLPC in a particular operational mode and reads and displays acceleration values from the device. Communication with the IIS2DLPC is accomplished via the SSM.

## Running the Application

Before booting Talaria TWO with the sample application, the IIS2DPLC must be connected using 4 GPIOs for the SPI signals (CLK, MOSI, MISO, CS) as well as Power and Ground. By default, the sample application uses the GPIO to SPI signal mapping shown in Table 1. However, any available GPIOs can be used.

**Note**: With this mapping, the JTAG jumper must be removed from the baseboard so that GPIO18 is routed to the peripheral connector instead of being used for JTAG.

|  |  |
| --- | --- |
| **GPIO** | **SPI Signal** |
| GPIO3 | CLK |
| GPIO4 | MOSI |
| GPIO14 | MISO |
| GPIO18 | CS |

Table : Sample GPIO - SPI signal mapping

With this mapping, the connection between the peripheral connecter on Talaria TWO EVK and the IIS2DLPC looks as shown in Figure 2.

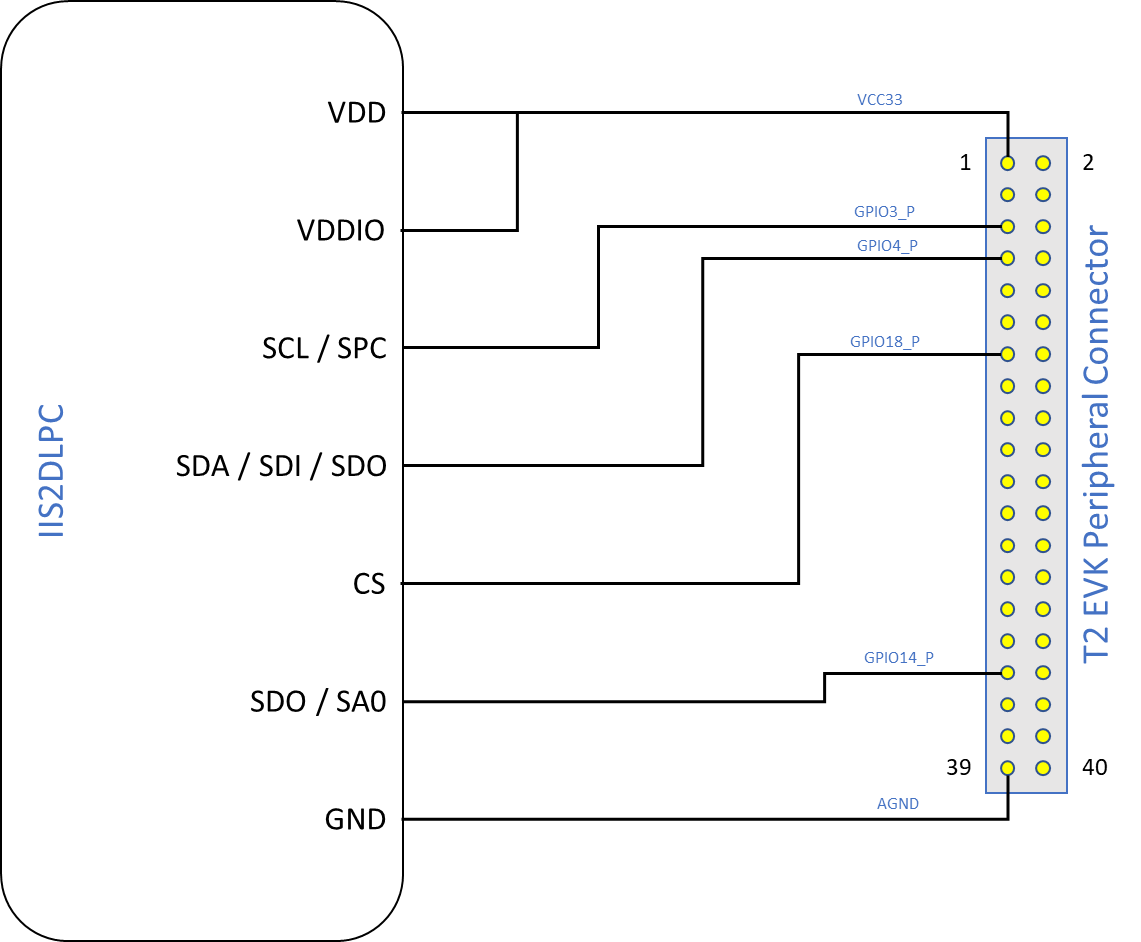


Figure : Sample connection of IIS2DLPC to Talaria TWO EVK

Once the IIS2DPLC has been connected to Talaria TWO, flash the sample app spi\_sensor.elf using the Download Tool.

Program spi\_sensor.elf (*sdk\_x.y\examples\spi\bin*) using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down
   2. ELF Input: Load the spi\_sensor.elf by clicking on Select ELF File.
   3. Boot arguments: Pass the following boot arguments. if a different GPIO other than the default pins in Table 1 are to be used for the SPI:

|  |
| --- |
| clk\_pin=<gpio pin>, mosi\_pin=<gpio pin>, miso\_pin=<gpio\_pin>, cs\_pin=<gpio\_pin> |

* 1. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y\pc\_tools\Download\_Tool\doc*).

**Note**: x and y refer to the SDK release version. For example: *sdk\_2.4\doc*.

Following is the sample output that is observed on the Download Tool console after flashing the spi\_sensor.elf.

|  |
| --- |
| UART:NWWWWWAEBuild $Id: git-b664be2af $  app=gordon  flash: Gordon ready!  [11.096,578] partitions mounted  UART:NWWWWWAEBuild $Id: git-b664be2af $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWWAEBuild $Id: git-b664be2af $  Using GPIOs: CLK [3] MOSI [4] MISO [14] CS [18]  IIS2DLPC sanity check: [PASSED]  IIS2DLPC device ID: [0x44]  Acceleration (x, y, z):  (7356, -15368, 1972)  (7308, -15324, 1876)  (7328, -15300, 1888)  (7296, -15324, 1900)  (7304, -15328, 1884)  (7336, -15320, 1860)  (7320, -15296, 1908)  (7328, -15324, 1896)  (7328, -15328, 1908)  (7300, -15308, 1880)  (7316, -15316, 1904)  (7312, -15316, 1888)  (5860, -16456, 1848)  (10412, 3676, 11536)  (7232, 9424, 12288)  (5864, 10080, 11340)  (6660, -12080, 13264)  (1652, -18504, 8176)  (-2904, 5252, 13204)  (-8496, 6768, 10912)  (-12576, 7080, 7564)  (-10868, 7332, 10008)  (10276, 2456, 12388)  (15220, 3248, -4752)  (14624, 3052, -5436)  (15284, 3620, -4892)  (16184, 4204, -3188)  (13200, 5372, 10812)  (-12332, 3116, 9852)  (-15476, 2608, 4112)  (-15260, 2604, 3504)  (-15504, 2632, 3668)  (-15516, 2420, 3400)  (-15016, 3412, 5560)  (9332, 6424, 11124)  (16376, -320, -1636)  (16716, -336, -1276)  (16328, -1108, -1748) |

## Code Overview

### Directory structure

Figure : Directory Structure

1. **IISDLPC**
2. accel.c

The accel.c file contains the routines to configure the IIS2DLPC and read the acceleration values from the device.

1. IISDLPC.c

The IISDLPC.c file contains the functions for communicating with an IIS2DLPC 3-axis accelerometer using the software SPI master.

1. IISDLPC.h

This code contains IIS2DLPC register definitions and function prototypes for communicating with an IIS2DLPC 3-axis accelerometer using the software SPI master.

1. **spi**
   1. sw\_spi.c

This code contains the logic to enable software SPI master to measure the readings of the IISDLPC accelerometer and displays the readings from sensor periodically.

* 1. sw\_spi.h

This code contains the software SPI master (SSM) definitions and function prototypes. It provides prototypes for the following functions that initiate, destroy and transfer.

### Using SSM

To use the Software SPI Master, set-up an instance of a spiopts\_t structure to specify SSM options as well as the GPIOs that the SSM will use. Section 5 provides more details on the different operating modes and configuration options supported by the SSM.

|  |
| --- |
| spiopts\_t spiopts = { /\* Set up SSM options here \*/ }; |

Once a spiopts\_t structure has been set up, initialize the SSM by calling the initialization function with a pointer to the options structure:

|  |
| --- |
| spi\_init(&spiopts); |

This will allocate and set up the specified GPIOs for SSM operation, returning true if successful.

After initialization, data is transferred over SPI by calling the transfer function:

|  |
| --- |
| uint16\_t data\_rx, data\_tx;  data\_rx = spi\_xfer(&spiopts, data\_tx); |

This performs a bidirectional transfer of the number of bits specified in the spiopts\_t structure. Transmit data in data\_tx if the data length is configured to be less than 16 bits. Received data in data\_rx.

After the SSM Tx and Rx Operations, Call the destroy function to clean up the SSM and release the previously allocated GPIOs:

|  |
| --- |
| spi\_destroy(&spiopts); |

A sample application that makes use of the SSM is described in section 7 of this document.

### sw\_spi.c

In software SPI master, SPI can be virtualized using software to simulate the physical SPI port. Its Initialization is based on user configuration. spi\_init() initializes the GPIOs for software SPI master operation. Verify that the same pin is not assigned to more than one SPI function.

|  |
| --- |
| if( spiopts->clk\_pin == spiopts->mosi\_pin ||  spiopts->clk\_pin == spiopts->miso\_pin ||  spiopts->mosi\_pin == spiopts->miso\_pin ||  (spiopts->cs\_en && (spiopts->clk\_pin == spiopts->cs\_pin ||  spiopts->mosi\_pin == spiopts->cs\_pin ||  spiopts->miso\_pin == spiopts->cs\_pin )))  {  os\_printf("[SPI] ERROR: the same pin cannot be assigned to more than one SPI signal\n");  return false;  } |

os\_gpio\_request()configures the selected pin as GPIOs for SSM.

|  |
| --- |
| if(!os\_gpio\_request(pin2gpio(spiopts->clk\_pin)))  {  os\_printf("[SPI] ERROR: Could not configure CLK pin as GPIO\n");  return false;  }  if(!os\_gpio\_request(pin2gpio(spiopts->mosi\_pin)))  {  os\_printf("[SPI] ERROR: Could not configure MOSI pin as GIPO\n");    os\_gpio\_free(pin2gpio(spiopts->clk\_pin));  return false;  }  if(!os\_gpio\_request(pin2gpio(spiopts->miso\_pin)))  {  os\_printf("[SPI] ERROR: Could not configure MISO pin as GPIO\n");    os\_gpio\_free(pin2gpio(spiopts->clk\_pin) | pin2gpio(spiopts->mosi\_pin));  return false;  }  if(spiopts->cs\_en && !os\_gpio\_request(pin2gpio(spiopts->cs\_pin)))  {  os\_printf("[SPI] ERROR: Could not configure CS pin as GPIO\n");  os\_gpio\_free(pin2gpio(spiopts->clk\_pin) | pin2gpio(spiopts->mosi\_pin) | pin2gpio(spiopts->miso\_pin));  return false;  } |

The GPIO output state is set before configuring as output value using os\_gpio\_set\_pin().

|  |
| --- |
| gpio\_write(pin2gpio(spiopts->clk\_pin), spiopts->clk\_pol);  os\_gpio\_clr\_pin(pin2gpio(spiopts->mosi\_pin));  if(spiopts->cs\_en)  os\_gpio\_set\_pin(pin2gpio(spiopts->cs\_pin)); |

Configure GPIOs as input or output as appropriate:

|  |
| --- |
| os\_gpio\_set\_output(  pin2gpio(spiopts->clk\_pin) |  pin2gpio(spiopts->mosi\_pin) |  (spiopts->cs\_en ? pin2gpio(spiopts->cs\_pin) : 0) );  os\_gpio\_set\_input(pin2gpio(spiopts->miso\_pin)); |

spi\_destroy() frees GPIOs previously set up for software SPI master operation.

|  |
| --- |
| void spi\_destroy(const spiopts\_t \* spiopts)  {  os\_gpio\_free(  pin2gpio(spiopts->clk\_pin) |  pin2gpio(spiopts->mosi\_pin) |  pin2gpio(spiopts->miso\_pin) |  (spiopts->cs\_en ? pin2gpio(spiopts->cs\_pin) : 0) );  } |

spi\_xfer()performs a bidirectional data transfer using SSM. spi\_init() must first be called with the SPI opts structure before this function is called. This function returns the received data.

|  |
| --- |
| bool clk = (spiopts->clk\_phase == spiopts->clk\_pol); |

This is initialized to the clock level that will be set when the first bit is written out on MOSI.

|  |
| --- |
| uint16\_t mask\_tx = spiopts->shiftdir\_tx == MSB\_FIRST ? (1 << (spiopts->datalen-1)) : 1; |

This is initialized to select the first bit of data\_tx to transmit.

|  |
| --- |
| uint16\_t data\_rx = 0; |

Extra bits are padded with 0s. This sets CS low. The CS line is normally held high, which disconnects the peripheral from the SPI bus. Just before data is sent to the peripheral, the line is brought low, which activates the peripheral.

|  |
| --- |
| if(spiopts->cs\_en)  {  os\_gpio\_clr\_pin(pin2gpio(spiopts->cs\_pin));  os\_wait\_usec(spiopts->t\_cs\_clk);  }  /\* Shift bits \*/  for(uint8\_t i = 0; i < spiopts->datalen; i++)  {  bool bit\_tx = data\_tx & mask\_tx;  bool bit\_rx = 0; |

Data is set at the same time as the clock is updated.

|  |
| --- |
| os\_gpio\_set\_value( (clk ? pin2gpio(spiopts->clk\_pin) : 0) | (bit\_tx ? pin2gpio(spiopts->mosi\_pin) : 0),  (!clk ? pin2gpio(spiopts->clk\_pin) : 0) | (!bit\_tx ? pin2gpio(spiopts->mosi\_pin) : 0) );  os\_wait\_usec(spiopts->t\_clk\_setup); |

The following code snippet samples the sensor rx data. It holds the clock for a second when it starts preparing for the next bit.

|  |
| --- |
| clk = !clk;  gpio\_write(pin2gpio(spiopts->clk\_pin), clk);  bit\_rx = os\_gpio\_get\_value(pin2gpio(spiopts->miso\_pin));  if(spiopts->shiftdir\_rx == MSB\_FIRST)  data\_rx = (data\_rx << 1) | bit\_rx;  else  data\_rx = (data\_rx >> 1) | ((uint16\_t)bit\_rx << (spiopts->datalen-1));  /\* Hold \*/  os\_wait\_usec(spiopts->t\_clk\_hold);  /\* Prepare for next bit \*/  clk = !clk;  if(spiopts->shiftdir\_tx == MSB\_FIRST)  mask\_tx >>= 1;  else  mask\_tx <<= 1;  } |

The following gpio\_write() function sets the clock back to its polarity value:

|  |
| --- |
| gpio\_write(pin2gpio(spiopts->clk\_pin), spiopts->clk\_pol); |

The following code block sets the ‘cs’ to high and master disables the communication through the SPI protocol with the slave:

|  |
| --- |
| if(spiopts->cs\_en)  { os\_wait\_usec(spiopts->t\_cs\_clk);  os\_gpio\_set\_pin(pin2gpio(spiopts->cs\_pin)); } |

### sw\_spi.h

The sw\_spi.h file contains the software SPI master (SSM) definitions and function prototypes. It provides prototypes for the following functions that initiate, destroy and transfer.

### accel.c

The accel.c file contains the routines to configure the IIS2DLPC and reads acceleration values from the device by reading and writing from/to the IIS2DLPC registers. For communication with the IIS2DLPC, this code relies on functions and register definitions provided in IIS2DLPC.h and IIS2DLPC.c.

The get\_bootarg\_pins() gets the GPIO pins for SPI signals via boot arguments.

|  |
| --- |
| if(!get\_bootarg\_pins(&clk\_pin, &mosi\_pin, &miso\_pin, &cs\_pin))  {  print\_usage();  return 1;  }  os\_printf("Using GPIOs: CLK [%" PRIu8 "] MOSI [%" PRIu8 "] MISO [%" PRIu8 "] CS [%" PRIu8 "]\n",  clk\_pin, mosi\_pin, miso\_pin, cs\_pin); |

IIS2DLPC\_init initializes SW SPI master for communication with IIS2DLPC.

|  |
| --- |
| if(!IIS2DLPC\_init(&spiopts, clk\_pin, mosi\_pin, miso\_pin, cs\_pin))  {  os\_printf("Could not initialize software SPI master for IIS2DLPC communication; aborting\n");  return 2;  } |

IIS2DLPC\_sanity\_check runs a sanity check. This delay is only to make it easier to examine SPI signals with a scope and is not needed for proper operation. It gives time to examine the initial state of signals after reset and before the first transfer.

|  |
| --- |
| if(IIS2DLPC\_sanity\_check(&spiopts))  {  os\_printf("IIS2DLPC sanity check: [PASSED]\n");  }  else  {  os\_printf("IIS2DLPC sanity check: [FAILED]\n");  os\_printf("Aborting\n");  return 3;  }  os\_printf("IIS2DLPC device ID: [0x%" PRIX8 "]\n", IIS2DLPC\_read\_id(&spiopts)); |

IISDLPC\_set\_mode sets the IIS2DLPC operating mode.

|  |
| --- |
| IIS2DLPC\_set\_mode(&spiopts); |

This code reads the values from the accelerometer through SPI.

|  |
| --- |
| os\_printf("Acceleration (x, y, z):\n");  while(1)  {  accel\_t accel = {0, 0, 0};  char dispbuf[DISPBUF\_LEN] = "";  IIS2DLPC\_read\_accel(&spiopts, &accel);  snprintf(dispbuf, DISPBUF\_LEN, "(%" PRId16 ", %" PRId16 ", %" PRId16 ")", accel.accel\_x, accel.accel\_y, accel.accel\_z);  os\_printf("%-\*s\r", DISPBUF\_LEN-1, dispbuf);  os\_msleep(ACCEL\_READ\_PERIOD\_MS);  } |

The get\_bootarg\_pins() retrieves GPIO pin numbers for SPI signals from boot arguments. If all pin numbers are valid, the pin numbers are stored in the locations pointed to by the pin\_out arguments and the function returns true. Otherwise, the function returns false, and the memory pointed to by the pin\_out arguments remain unchanged. Also, it prints all the pin details in the console. If no boot arguments are provided for the GPIO pins, the default pins are selected for SPI.

|  |
| --- |
| uint8\_t clk\_pin = 0, mosi\_pin = 0, miso\_pin = 0, cs\_pin = 0;  if(!clk\_pin\_out || !mosi\_pin\_out || !miso\_pin\_out || !cs\_pin\_out)  return false;  clk\_pin = os\_get\_boot\_arg\_int("clk\_pin", CLK\_PIN\_DEFAULT);  mosi\_pin = os\_get\_boot\_arg\_int("mosi\_pin", MOSI\_PIN\_DEFAULT);  miso\_pin = os\_get\_boot\_arg\_int("miso\_pin", MISO\_PIN\_DEFAULT);  cs\_pin = os\_get\_boot\_arg\_int("cs\_pin", CS\_PIN\_DEFAULT);  if(!gpio\_pin\_valid(clk\_pin))  {  os\_printf("Invalid GPIO pin number specified for CLK\n");  return false;  }  if(!gpio\_pin\_valid(mosi\_pin))  {  os\_printf("Invalid GPIO pin number specified for MOSI\n");  return false;  }  if(!gpio\_pin\_valid(miso\_pin))  {  os\_printf("Invalid GPIO pin number specified for MISO\n");  return false;  }  if(!gpio\_pin\_valid(cs\_pin))  {  os\_printf("Invalid GPIO pin number specified for CS\n");  return false;  }  \*clk\_pin\_out = clk\_pin;  \*mosi\_pin\_out = mosi\_pin;  \*miso\_pin\_out = miso\_pin;  \*cs\_pin\_out = cs\_pin;  return true;  } |

gpio\_pin\_valid() checks a GPIO pin number against the array of valid GPIO pin numbers. Returns true if the pin number appears in the list, false otherwise.

|  |
| --- |
| for(size\_t i = 0; i < sizeof(VALID\_GPIOS) / sizeof(VALID\_GPIOS[0]); i++)  if(pin == VALID\_GPIOS[i])  return true;  return false; |

The IIS2DLPC functions added in accel.c interact with the IIS2DLPC in a manner that is specific to these functions and the use of the device by this application. The IIS2DLPC\_sanity\_check() runs a sanity check by writing values to an IIS2DLPC R/W register and reading them back.

|  |
| --- |
| uint8\_t data8\_initial = IIS2DLPC\_read8(spiopts, IIS2DLPC\_TAP\_THS\_X);  if(!IIS2DLPC\_reg\_wr\_test(spiopts, IIS2DLPC\_TAP\_THS\_X, 0xAA, 0xAA))  return false;  if(!IIS2DLPC\_reg\_wr\_test(spiopts, IIS2DLPC\_TAP\_THS\_X, 0x55, 0x55))  return false;  if(!IIS2DLPC\_reg\_wr\_test(spiopts, IIS2DLPC\_TAP\_THS\_X, 0x0, 0x0))  return false;  if(!IIS2DLPC\_reg\_wr\_test(spiopts, IIS2DLPC\_TAP\_THS\_X, 0xFF, 0xFF))  return false;  if(!IIS2DLPC\_reg\_wr\_test(spiopts, IIS2DLPC\_TAP\_THS\_X, data8\_initial, data8\_initial))  return false;  return true; |

IIS2DLPC\_reg\_wr\_test() writes a value to the IIS2DLPC register, reads the register, and compares the read value with an expected value.

|  |
| --- |
| uint8\_t data8\_read = 0;  IIS2DLPC\_write8(spiopts, addr, data8\_write);  if((data8\_read = IIS2DLPC\_read8(spiopts, addr)) != data8\_expected)  {  os\_printf("Register WR test: unexpected register value\n");  os\_printf("Addr: [0x%" PRIX8 "] Wrote: [0x%" PRIX8 "] Read: [0x%" PRIX8 "] Expected: [0x%" PRIX8 "]\n",  addr, data8\_write, data8\_read, data8\_expected);  return false;  }  return true;  } |

IIS2DLPC\_read\_id() reads the device ID of the IIS2DLPC.

|  |
| --- |
| return IIS2DLPC\_read8(spiopts, IIS2DLPC\_WHO\_AM\_I); |

IIS2DLPC\_set\_mode() sets the IIS2DLPC in the mode that will be used for this application. Here low noise, low power mode 4, 12.5Hz data rate, on demand mode and trigger acceleration reading via register bit are enabled.

|  |
| --- |
| data8 = IIS2DLPC\_read8(spiopts, IIS2DLPC\_CTRL6);  data8 = setbit(data8, IIS2DLPC\_CTRL6\_LOW\_NOISE\_OFFSET);  IIS2DLPC\_write8(spiopts, IIS2DLPC\_CTRL6, data8);    IIS2DLPC\_write8(spiopts, IIS2DLPC\_CTRL1,  (0x2 << IIS2DLPC\_CTRL1\_ODR\_OFFSET) |  (0x2 << IIS2DLPC\_CTRL1\_MODE\_OFFSET) |  (0x3 << IIS2DLPC\_CTRL1\_LP\_MODE\_OFFSET));  data8 = IIS2DLPC\_read8(spiopts, IIS2DLPC\_CTRL3);  data8 = setbit(data8, IIS2DLPC\_CTRL3\_SLP\_MODE\_SEL\_OFFSET);  IIS2DLPC\_write8(spiopts, IIS2DLPC\_CTRL3, data8); |

IIS2DLPC\_read\_accel()triggers an acceleration reading from the IIS2DLPC and waits for the result. IIS2DLPC\_set\_mode must be called before calling this function.

|  |
| --- |
| data8 = IIS2DLPC\_read8(spiopts, IIS2DLPC\_CTRL3);  data8 = setbit(data8, IIS2DLPC\_CTRL3\_SLP\_MODE\_1\_OFFSET);  IIS2DLPC\_write8(spiopts, IIS2DLPC\_CTRL3, data8); |

This reads the acceleration values from sensor:

|  |
| --- |
| accel\_x = IIS2DLPC\_read8(spiopts, IIS2DLPC\_OUT\_X\_L);  accel\_x |= (uint16\_t)IIS2DLPC\_read8(spiopts, IIS2DLPC\_OUT\_X\_H) << 8;  accel\_y = IIS2DLPC\_read8(spiopts, IIS2DLPC\_OUT\_Y\_L);  accel\_y |= (uint16\_t)IIS2DLPC\_read8(spiopts, IIS2DLPC\_OUT\_Y\_H) << 8;  accel\_z = IIS2DLPC\_read8(spiopts, IIS2DLPC\_OUT\_Z\_L);  accel\_z |= (uint16\_t)IIS2DLPC\_read8(spiopts, IIS2DLPC\_OUT\_Z\_H) << 8; |

Acceleration values are stored in 16-bit 2's complement format. This conversion relies on implementation-defined behavior.

|  |
| --- |
| accel->accel\_x = (int16\_t)accel\_x;  accel->accel\_y = (int16\_t)accel\_y;  accel->accel\_z = (int16\_t)accel\_z; |

### IISDLPC.c

The functions in IISDLPC.c makes use of the SSM to communicate with an IIS2DLPC. This contains functions for communicating with an IIS2DLPC 3-axis accelerometer using SSM.

IIS2DLPC\_init() initializes SSM for communication with an IIS2DLPC. This function sets up the supplied SPI options structure with values required for communicating with an IIS2DLPC and initializes the SSM with the options structure.

|  |
| --- |
| spiopts->datalen = 16;  spiopts->shiftdir\_tx = MSB\_FIRST;  spiopts->shiftdir\_rx = MSB\_FIRST;  spiopts->clk\_phase = 0;  spiopts->clk\_pol = 1;  spiopts->cs\_en = true;  spiopts->clk\_pin = clk\_pin;  spiopts->mosi\_pin = mosi\_pin;  spiopts->miso\_pin = miso\_pin;  spiopts->cs\_pin = cs\_pin;  spiopts->t\_clk\_setup = 25;  spiopts->t\_clk\_hold = 25;  spiopts->t\_cs\_clk = 25;  spiopts->t\_xfer\_dly = 25;  return spi\_init(spiopts); |

IIS2DLPC\_destroy() cleans the resources (GPIOs) allocated by IIS2DLPC\_init(). IIS2DLPC\_read8() reads a byte from the IIS2DLPC device at a given address.

|  |
| --- |
| uint16\_t data\_rx = 0;  data\_rx = spi\_xfer(spiopts, 0x8000 | (uint16\_t)(addr & 0x7F) << 8);  return data\_rx & 0xFF; |

IIS2DLPC\_write8() writes a byte to the IIS2DLPC device at a given address.

|  |
| --- |
| spi\_xfer(spiopts, (uint16\_t)(addr & 0x7F) << 8 | data); |

### IISDLPC.h

IIS2DLPC\_init() initializes spiopts\_t structure with the options required for communicating with an IIS2DLPC device. It has parameters for GPIO pin numbers to be used for SPI signals, which are registered in the structure. After initializing the structure, this function calls spi\_init() to initialize the SSM.

IIS2DLPC\_read8() reads the value of an IIS2DLPC register at a given address and returns the data. This function must be passed as a pointer to spiopts\_t structure previously initialized with IIS2DLPC\_init().

IIS2DLPC\_write8() writes a value to an IIS2DLPC register at a given address. This function must be passed as a pointer to spiopts\_t structure previously initialized with IIS2DLPC\_init().

IIS2DLPC\_destroy() frees the resources previously allocated by IIS2DLPC\_init() after the communication with the device is complete.

# SPI Throughput

The sample application (spi\_throughput.c) demonstrates the SPI throughput measurement.

This application transfers address and data bytes over the SPI interface. Each iteration transfers 1 R/W bit, 7 address bits, and data bits available in the RX\_BUF\_SZ buffer. After running the throughput test, the application prints the throughput (in kbps) over the console.

**Application flow:**

1. Application writes ‘0’ on the ctrl register 2 (IIS2DLPC\_CTRL2) to select SPI 4 wire interface and this disables auto address increment.
2. Application reads the sensor ID (IIS2DLPC\_WHO\_AM\_I) from the sensor and prints it over the console.
3. Application now starts the throughput test.
4. After 100000 iterations, the application prints throughput over the console.

## Running the Application

Before booting Talaria TWO with the sample application, IIS2DPLC must be connected using 4 GPIOs for the SPI signals (CLK, MOSI, MISO, CS) as well as Power and Ground. By default, the sample application uses the GPIO to SPI signal mapping shown in Table 1. However, any available GPIOs can be used. The connection between the peripheral connecter on Talaria TWO EVK and the IIS2DLPC is as shown in Figure 2.

**Note**: With signal mapping mentioned in Table 1, the JTAG jumper must be removed from the baseboard so that GPIO18 is routed to the peripheral connector instead of being used for JTAG.

Once the IIS2DPLC has been connected to Talaria TWO, flash the sample application spi\_throughput.elf using the Download Tool.

Program spi\_throughput.elf (*sdk\_x.y\examples\spi\bin*) using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down
   2. ELF Input: Load the spi\_throughput.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y\pc\_tools\Download\_Tool\doc*).

**Note**: x and y refer to the SDK release version. For example: *sdk\_2.4\doc*.

Following is a sample output observed on the Download Tool console after flashing spi\_throughput.elf.

|  |
| --- |
| Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWWAEBuild $Id: git-e31bdbe13 $  Chip ID: 0x44  Running throughput test...  100000 iterations  42987509 usec  205600000 bits transferred in 42 seconds (4782 kbps) |