ADPD7000 SDK User Guide

TABLE OF CONTENTS

REVISION HISTORY

07/29/2022—Rev 0.0.1	Initial Document.
10/20/2022—Rev 0.0.2	.

OVERVIEW

This document provides information about the Software Development Kit (SDK) software developed by ADI for the Multimodal Sensor Front End (ADPD7000). This document outlines the overall architecture, folder structure, and methods for using SDK software on any platform. This document does not explain the SDK library functions.

SOFTWARE ARCHITECTURE

The ADPD7000 SDK library is a collection of APIs that provide a consistent interface to the Multimodal Sensor Front End Devices. The APIs are designed so that there is a consistent interface to the devices.

The library is a software layer that sits between the application and the ADI device. The library is intended to serve two purposes:

1. Provide the application with a set of APIs that can be used to configure ADI device without the need for low-level register access. This makes the application portable across different revisions of the hardware and even across different hardware modules.
2. Provide basic services to aid the application in controlling the components of the device module, such as Clock, ECG, PPG , BioZ configuration.

The driver does not, in any shape or form, alter the configuration or state of device on its own. It is the responsibility of the application to configure the part according to the required mode of operation, poll for status, etc... The library acts only as an abstraction layer between the application and the hardware.

As an example, the application is responsible for the following:

- Write/read register
- Configuring the ECG
- Configuring the PPG
- Configuring the BioZ
- Setting Clock configuration, software reset…
- Reading data from FIFO

The application should access the device only through the exported APIs. It is not recommended for the application to access the hardware device directly using direct SPI access. If the application chooses to directly access the ADPD7000 hardware this should be done in a very limited scope, such as for debug purposes and it should be understood that this practice may affect the reliability of the SDK functions.

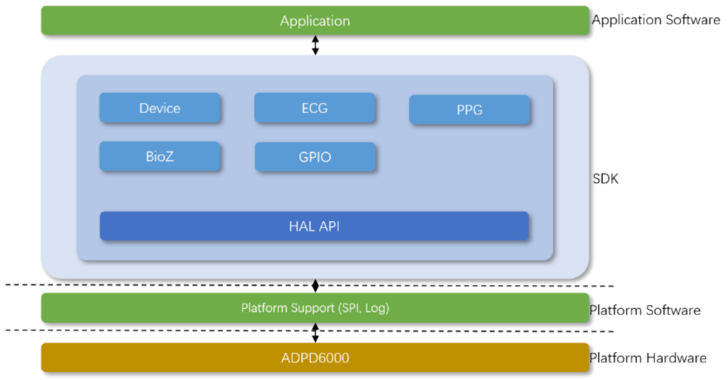

Figure 1 ADPD7000 SDK Architecture

FOLDER STRUCTURE

The collective files of the ADPD7000 library are structure as depicted in Figure 2. Each branch is explained in the following sections. The library is supplied in source format. All source files are in standard C99 to simplify porting to any platform.

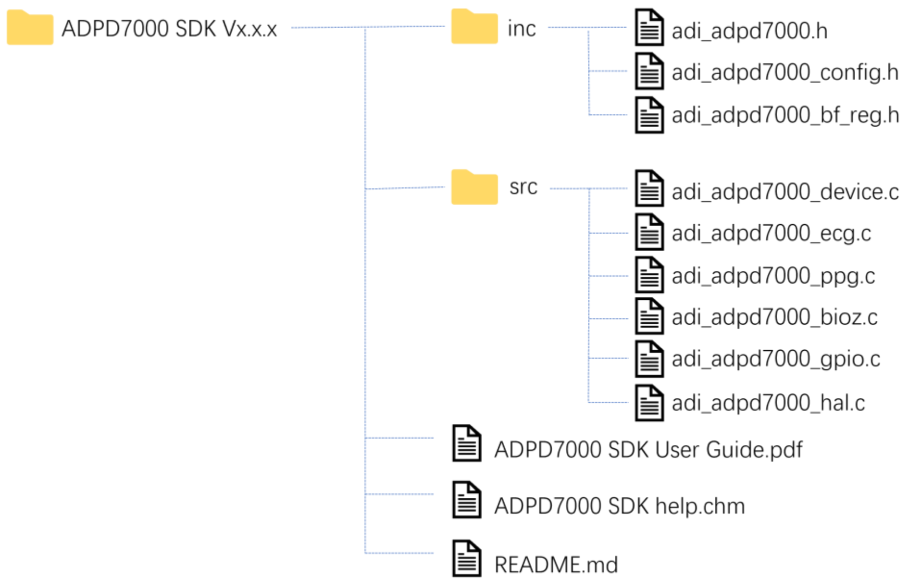

Figure 2. ADPD7000 Source Code Folder Structure

ADPD7000 SDK Vx.x.x/inc

This folder contains all the many private header files used by the APIs.

ADPD7000 SDK Vx.x.x/src

This folder includes the main API implementation code for the ADPD7000 device APIs. ADI maintains this code as intellectual property and all changes are at their sole discretion.

Document

This SDK contains the two documentations for the ADPD7000 APIs, including user guide and SDK help document.

SOFTWARE INTEGRATION

As ADI provides the full source code, users are required to integrate the Hal layer with their platform-specific code base. This is readily accomplished because the ADPD7000 SDK was developed in C99. The C99 standard was followed to ensure agnostic processor and operating system integration with the SDK code. For integration flow, please refer to figure 3.

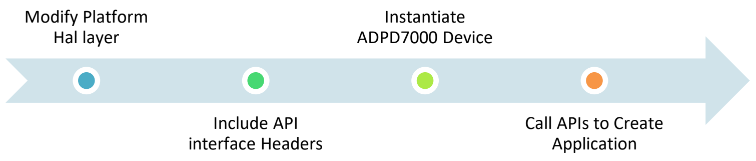
Modify Platform Hal layer
Include API interface Headers
Instantiate ADPD7000 Device
Call APIs to Create Application

Figure 3. ADPD7000 API Integration Flow

MODIFY PLATFORM HAL LAYER

Users will develop on their own hardware-dependent platforms. Therefore, they will use different drivers for the peripherals such as the SPI and GPIO what is included in the ADPD7000 SDK library. Customers can use their own drivers for these peripherals, or they may use standard drivers if they utilize an operating system.

The ADPD7000 SDK was developed such that developers can use any driver of their choosing for their platform requirements. However, there are a few platform dependent functions in the SDK HAL layer. Do NOT modify these layers because a specific function prototype was used for these functions. Instead, modify the functions located in the header file “adi_adpd7000.h” under the /inc directory for specific platform requirements. The required functions in the header file which a developer can substitute their hardware-specific information are contained in Table 1:

Table 1. Required functions for integration

Function	**Description**
*typedef int32_t (*adi_adpd7000_read)(void* user_data, uint8_t *rd_buf, uint32_t rd_len, uint8_t *wr_buf, uint32_t wr_len);*	A function to implement SPI read to the ADPD7000 device on the target hardware.
*typedef int32_t (*adi_adpd7000_write)(void* user_data, uint8_t *wr_buf, uint32_t len);*	A function to implement SPI read to the ADPD7000 device on the target hardware.
*typedef int32_t(*adi_adpd7000_log_write)(void *user_data, int32_t log_type, const char *message, va_list argp);*	A function to implement a log message write operation.

INCLUDE THE ADPD7000 SDK INTERFACE HEADERS

The following header files define the interface to the ADPD7000 SDK and should be included in the application.

inc/adi_adpd7000.h

INSTANTIATE ADPD7000 DEVICE

For each ADPD7000 device, the application must instantiate an ADPD7000 device reference. For each device instantiated by the application, all the required members of the ADPD7000 device must be initialized prior to calling any APIs with that device as parameter. As per the ADPD7000 SDK specification, the following members are required for correct operation of the ADPD7000 APIs.

```
/* APPD7000 device structure, the structure is defined in adpd7000 driver */
 typedef struct
{
    void*                   user_data;
    adi_adpd7000_read       read;          /*!< Function Pointer to HAL SPI read function */
    adi_adpd7000_write      write;         /*!< Function Pointer to HAL SPI write function */
    adi_adpd7000_log_write  log_write;     /*!< Function Pointer to HAL log write function */
} adi_adpd7000_device_t;

/* User should assign SPI data transfer function for adpd7000 device member */
adi_adpd7000_device_t device = {0};
```

```
adpd7000_dev.write = adpd7000_spi_write;
adpd7000_dev.read = adpd7000_spi_read;
```

SPI WORKING MODE

The SPI write operation is illustrated Figure 2, showing a single register write. The first 2 bytes contain the 15 bit register address and also specifies that a write is requested. The remaining two bytes are the 16 data bits to write to the register. The register write only occurs if all 16 bits are shifted in prior to de-assertion of the CSB signal.

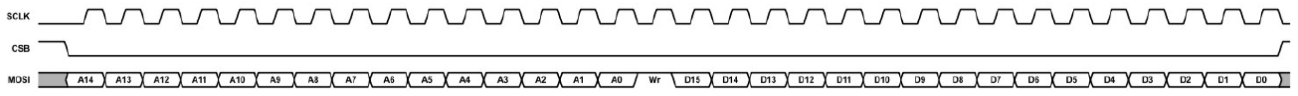
SCLK
CSB
MOSI
A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0 Wr D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

Figure 2 SPI Write operation

Multiple registers can be written if additional 16 bit data is shifted in before deassertion of the CSB signal. The register address automatically increments to the next register after each 16 bits of data. The SPI read operation is illustrated in Figure 3, showing a single register read. The first 2 bytes contain the 15 bit register address and also specifies that a read is requested. Register bits are shifted out starting with the msb.

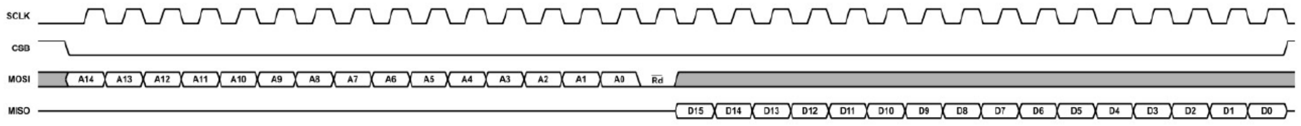
SCLK
CSB
MOSI
A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0 Rd
MISO
D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0

Figure 3 SPI Read operation

In addition, multiple registers can be read if additional 16 bit data is shifted out prior to deassertion of the CSB signal. Reading from the FIFO can be done byte-wise, there is no requirement to read multiples of 16 bits.

The CPOL and CPHA bit of ADPD7000 SPI are all 1, user should use SPI mode 3 on their platform.

Table 2. SPI Mode

SPI Mode	CPOL	CPHA	Clock Polarity	Clock Phase
0	0	0	Logic Low	Data sampled on rising edge and shifted out on the falling edge
1	0	1	Logic Low	Data sampled on the falling edge and shifted out on the rising edge
2	1	0	Logic High	Data sampled on the rising edge and shifted out on the falling edge
3	1	1	Logic High	Data sampled on the falling edge and shifted out on the rising edge

CREATE THE APPLICATION

Use ADPD7000 SDK functions provided in */src/adi_adpd7000.h* to write the application code to initialize, configure and read data from ADPD7000 based on your target application requirements.

STEPS OF CREATING AN APPLICATION

The figure below shows how to create an application to get ECG, PPG or BioZ data form ADPD7000 via calling APIs.

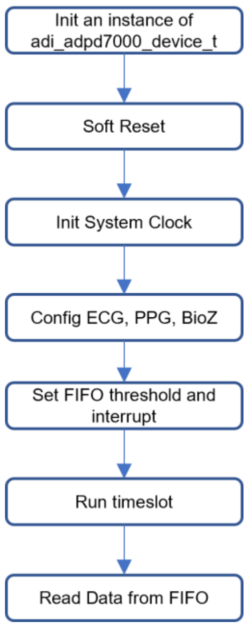

Figure 4 Flow of creating an application

SOFT RESET

User can assert a software reset via calling the function adi_adpd7000_device_sw_reset(), It resets the chip ADPD7000 to its default values and stop all operations, and it does not reset the SPI (or optional I2C) port.

INIT SYSTEM CLOCK

Firstly, user should call adi_adpd7000_device_init() to change trim value, and ADPD7000 provides two options for timeslot clock source, internal low frequency 960KHz and external clocks. User can call adi_adpd7000_device_enable_internal_osc_960k() to enable internal OSC. If use external clock as timeslot clock source, adi_adpd7000_device_config_ext_clock() should be called to configure clock settings.

Operation of ADPD7000 is controlled by an internal configurable controller which generates all timing needed to generate sampling regions and sleep periods. The enabled timeslots are repeated at the timeslot rate which is configured by adi_adpd7000_device_set_slot_freq().

CONFIG ECG, PPG, BIOZ

The ADPD7000 is a multi-modal vital signal monitoring AFE composing of three sophisticated signal chains: optical measurement path, Electrocardiogram (ECG) measurement path, and Body-Index-Analysis (BIA) measurement path. User can call APIs to configure one of those modules or all of them, SDK help document is a good reference to speed up your development.

After modules configuration, it is need to call APIs to enable timeslot for related module, for example, user should call adi_adpd7000_ecg_enable_slot() to enable ECG module.

SET FIFO THRESHOLD

The size of ADPD7000 FIFO is 521 bytes, user can set threshold by calling adi_adpd7000_device_set_fifo_threshold(). It is recommended to set threshold over than the size of data that generated by a sequence. User can call adi_adpd7000_device_get_sequence_fifo_config() to get the size after all module configuration are done.

ADPD7000 supports two separate interrupt output functions, INTX and INTY. Each of these are an option to be driven to the four GPIO pins. These can be used to generate two different interrupt outputs for a host processor if desired. For example, the FIFO threshold interrupt can be routed to INTX and used to drive the host’s DMA channel, and the FIFO overflow, underflow, and TIA saturated interrupts could be routed to INTY and used to drive the host’s additional interrupt pin.

User can call adi_adpd7000_device_enable_fifo_thres_interrupt() to enable FIFO threshold interrupt, and the function adi_adpd7000_gpio_set_output() is used to select GPIO output mode, the code 2 is for INTX, and 3 is for INTY. For more detailed

GPIO descriptions, please refer to datasheet.

If the FIFO is overflow, user should stop sequence immediately, and soft reset chip, then run timeslot again.

RUN TIMESLOT

ADPD7000 enters OFF mode during powerup reset, user can call adi_adpd7000_device_enable_slot_operation_mode_go() to enable GO mode and start timeslot sequence. During GO mode, user can NOT change any registers until calling adi_adpd7000_device_enable_slot_operation_mode_go() to immediately stop sequence.

READ DATA FROM FIFO

The Data in the FIFO is arranged according to the enabled timeslots and followed the same sequence. For example, if ECG Timeslot, 3 PPG Timeslots and 2 BIA timeslots are enabled, then the data in the FIFO is shown as Figure 5.

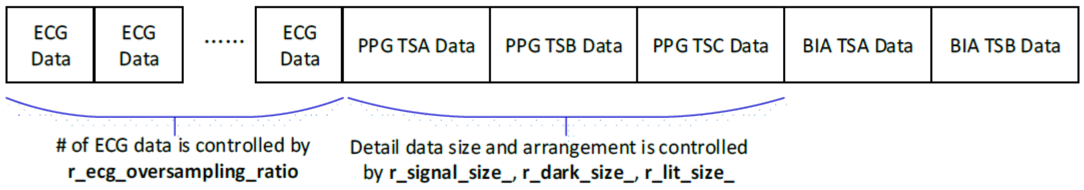
ECG Data
ECG Data
……
ECG Data
PPG TSA Data
PPG TSB Data
PPG TSC Data
BIA TSA Data
BIA TSB Data
# of ECG data is controlled by **r_ecg_oversampling_ratio**
Detail data size and arrangement is controlled by **r_signal_size_**, **r_dark_size_**, **r_lit_size_**

Figure 5 Example of data arrangement in FIFO

In order to read FIFO data simply, three APIs are provided to user to get ECG, PPG and BioZ data from FIFO, the three APIs are adi_adpd7000_ecg_read_fifo(), adi_adpd7000_ppg_read_fifo(),adi_adpd7000_bioz_read_fifo(). Those functions have a parameter with the structure adi_adpd7000_fifo_config_t, user should call adi_adpd7000_device_get_sequence_fifo_config() to get the parameter after all modules configuration are done.

EXAMPLE HAL SPI IMPLEMENTATION BASED ON STM32L4

```
int32_t adpd7000_spi_write(void* user_data, uint8_t *tx_data, uint32_t len)
{
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_7, GPIO_PIN_RESET);
    HAL_SPI_Transmit(&hspi3, tx_data, len, 100);
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_7, GPIO_PIN_SET);
    return 0;
}

int32_t adpd7000_spi_read(void* user_data, uint8_t *rx_data, uint32_t rx_len, uint8_t *tx_data, uint32_t tx_len)
{
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_7, GPIO_PIN_RESET);
    HAL_SPI_Transmit(&hspi3, tx_data, tx_len, 100);
    HAL_SPI_Receive(&hspi3, rx_data, rx_len, 100);
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_7, GPIO_PIN_SET);
    return 0;
}
void MX_SPI3_Init(void)
{
    /* SPI3 parameter configuration*/
    hspi3.Instance = SPI3;
    hspi3.Init.Mode = SPI_MODE_MASTER;
    hspi3.Init.Direction = SPI_DIRECTION_2LINES;
    hspi3.Init.DataSize = SPI_DATASIZE_8BIT;
    hspi3.Init.CLKPolarity = SPI_POLARITY_HIGH;
    hspi3.Init.CLKPhase = SPI_PHASE_2EDGE;
    hspi3.Init.NSS = SPI_NSS_SOFT;
    hspi3.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_16;
    hspi3.Init.FirstBit = SPI_FIRSTBIT_MSB;
    hspi3.Init.TIMode = SPI_TIMODE_DISABLE;
    hspi3.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    hspi3.Init.CRCPolynomial = 7;
    hspi3.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
    hspi3.Init.NSSPMode = SPI_NSS_PULSE_ENABLE;
    if (HAL_SPI_Init(&hspi3) != HAL_OK)
    {
      Error_Handler();
    }
}
int32_t adpd7000_port_init( void )
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    HAL_GPIO_WritePin(GPIOI, GPIO_PIN_7, GPIO_PIN_SET);
    GPIO_InitStruct.Pin = GPIO_PIN_7;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOI, &GPIO_InitStruct);
    MX_SPI3_Init();
    return 0;
}
```

Figure 4. Example Implementation of SPI HAL function