Future Technology Devices International Ltd.

# Software Application Development

# FT31xD Android Programmers Guide

**Document Reference No.: FT_000532**

**Version 1.1**

**Issue Date: 2013-05-21**

Android programmers guide describes User APIs for Android open accessory development. The Android application is divided into two layer, User Layer and FT311D/FT312D layer. User layer is not aware of USB communication and calls the APIs exposed by the FT311D/FT312D layer.

**Future Technology Devices International Limited (FTDI)**

Unit 1, 2 Seaward Place, Centurion Business Park, Glasgow G41 1HH United Kingdom
Tel.: +44 (0) 141 429 2777    Fax: + 44 (0) 141 429 2758
E-Mail (Support): support1@ftdichip.com  Web: http://www.ftdichip.com

**Table of Contents**

# 1 Preface

The FT31xD interface is a proprietary interface for Android Open Accessory development. The FT311D IC provides UART/GPIO/PWM/I2C Master/SPI Slave/SPI Master interfaces for Android devices. The FT312D IC provides an enhanced UART interface for Android devices. Android applications can use these interfaces to communicate with their accessories.
Any software code examples given in this document are for information only. The examples are not guaranteed and are not supported by FTDI.

## 1.1 Acronyms and Abbreviations

| Terms | Description |
|---|---|
| USB | Universal Serial Bus |
| FT311D | FTDIChip's interface chip for Android Open Accessory development. |
| FT312D | FTDIChip's enhanced UART interface chip for Android Open Accessory development. |

**Table 1.1 : Acronyms and Abbreviations**

## 1.2 References

FT311D datasheet
FT312D datasheet
Android Develpers, http://developer.android.com/index.html

## 2 Introduction

FTDI provides the FT311D and FT312D interface chips for USB Android Open Accessory development. The FT311D provides UART/GPIO/PWM/I2C Master/SPI slave /SPI Master interfaces for Android devices to communicate with their accessories. The FT312D provides a UART interface for Android devices to communicate with their accessories.

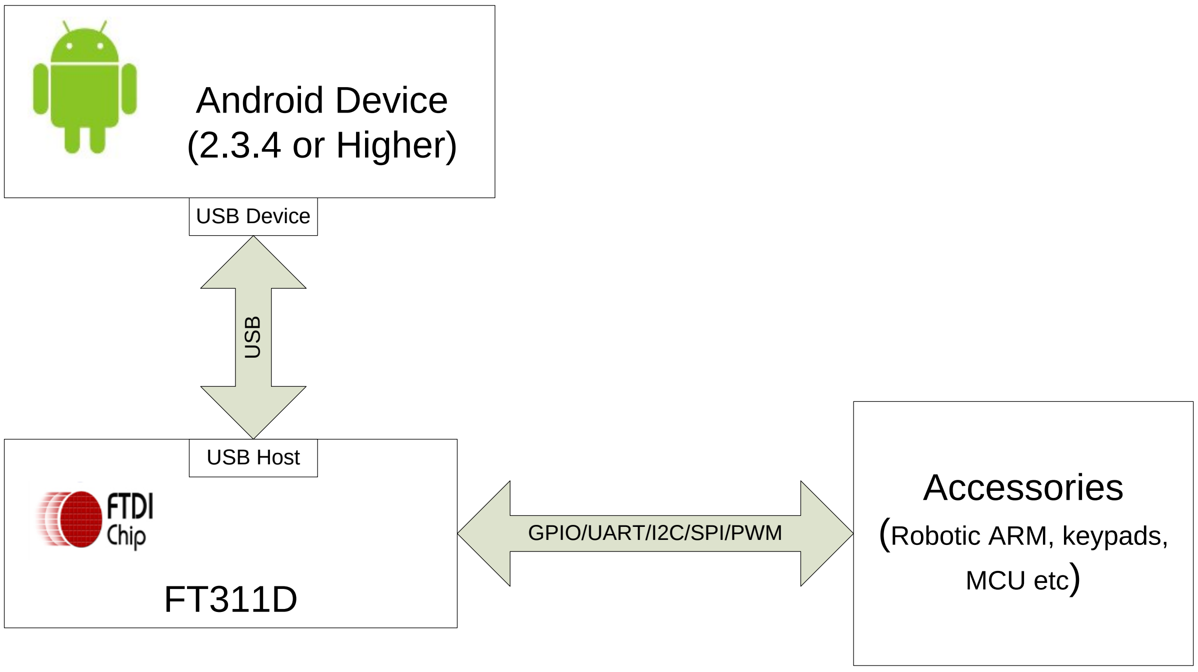

**Figure 2.1 : Android Open Accessory Module with FT311D**

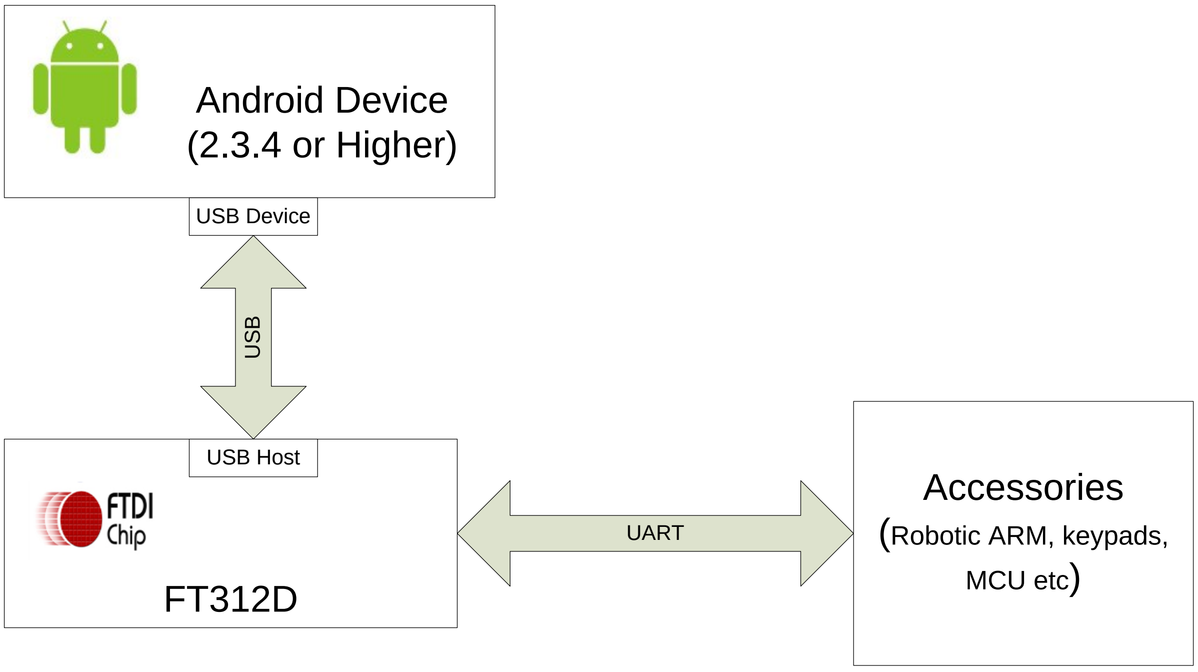

**Figure 2.2 : Android Open Accessory Module with FT312D**

Android Open Accessory development requires an Android Device with Android 2.3.4 or higher and FTDI Chip's interface chip, FT311D/FT312D. Users can choose any one of the interfaces provided by FT311D to connect their accessories.
This document will specify commands and data protocol for communication between Android Devices and the FT311D/FT312D chip. The Interface between FT311D/FT312D and Accessory follow the standard protocol of the selected Interface.

As both FT311D/FT312D are interface chips, most of the application control and behavior is implemented in Android device and accessory.

## 2.1 Android Device

Android devices with Android 3.1 or higher version. This document and examples applies to Android 3.1 and above (the Android API changed from version 2.3.4 to 3.1). The android application is launched based on the parameters like manufacturer, model and version defined in the AndroidManifest.xml file.

For application development using FT311D/FT312D, Android applications are divided into two parts:

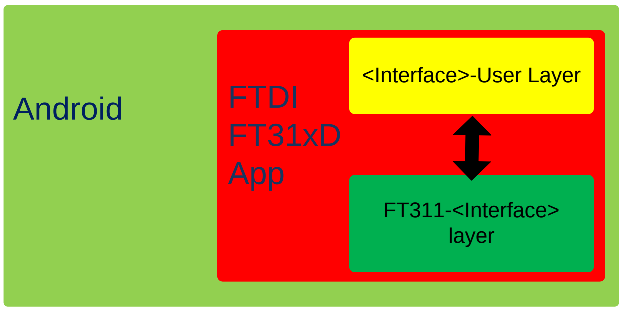

**Figure 2.3 : Android FTDI FT31xD Application**

<Interface>-User layer: this layer is not aware of USB interface between Android Device and FT311D/FT312D. The FT311D provides GPIO, UART, PWM, I2C master, SPI slave and SPI Master interfaces. The FT312D provides UART interface. The Android developer will implement their application in this layer and call the functions implemented by FTDI in FT311-<Interface> layer, defined for each supported interface. For reference code , see the Annex for the respective interface.
FT311-<Interface> layer: This layer implememts FT311<Interface>Interface class. This layer also implements the USB communication between Android device and FT311D/FT312D. The FT311<Interface>Interface class implements function for <Interface>-User layer to use. Using these functions <Interface>-User layer can talk to FT311D/FT312D. Here Interface could be one of GPIO/UART/PWM/I2C Master/SPI Slave/SPI Master. For communication protocol and data packet format see the sections below for each interface in FT311-<Interface> layer.
Note:
<Interface>-User layer and FT311-<Interface> layer are 2 java files. The 2 java files have to be compiled together to form the android package file.
The different interfaces are selected based on the FT311D/FT312D device connected to the androidAndroid device. During USB device enumeration the manufacturer, model and version strings are received from the FT311D/FT312D host device. The androidAndroid device matches these strings with the AndroidManifest.xml file to launch the correct application.
Each application will support only one interface. The application in the androidAndroid device communicatecommunicates to the FT311D/FT312D device using command packets.

## 2.2 FT311D

FT311D acts as USB Host for Android Device. FT311D provides GPIO/PWM/I2C Master/UART/SPI Slave/SPI Master interfaces to connect to accessories.

## 2.3 FT312D

FT312D acts as USB Host for Android Device. FT312D provides a UART interface to connect to accessories.

## 2.4 Accessories

Accessories connects on one of FT311D’s GPIO/UART/I2C Master/PWM/SPI Slave/SPI Master interfaces. Accessories can connect on FT312D's UART interface. Accessories could be robotic controllers, keypads, touchpads MCUs etc.

# 3 FT311D/FT312D UART interface

FT311D and FT312D provide a UART interface, with baud rates from 300 to 921600. The FT311D and FT312D UART transmits data in NRZ data format.

FTDI Chip provides FT311UARTInterface class with SetConfig, ReadData, WriteData routines for UART operations and ResumeAccessory, DestroyAccessory for support functions to resume and destroy the accessory operations. For the use of the functions, please see Annex A.

The method to add FT311UARTInterface.java into the project in Eclipse environment is:
Copy the file into the src directory, e.g src\com\<package name>.
Write click on the project name in eclipse package explore, then select new->file->src->com->"package name"->advanced->link to the file.

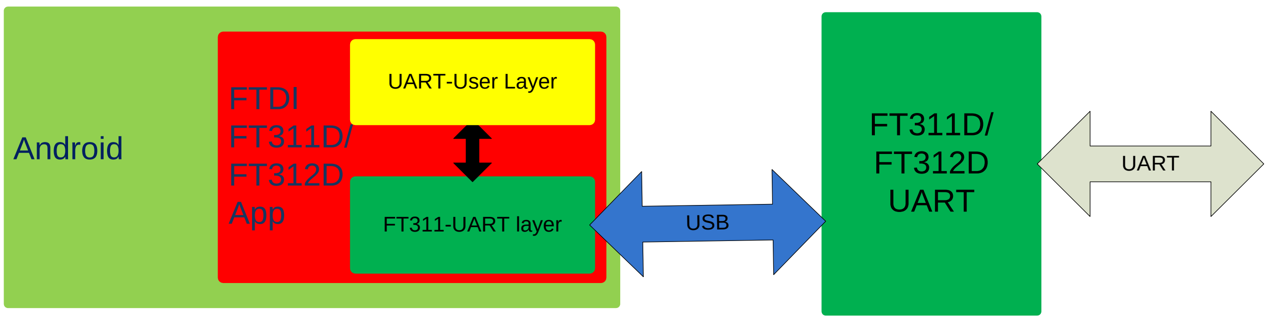

**Figure 3.1 : Android FTDI FT31xD UART Application**

## 3.1 FT311D/FT312D UART-User Layer

This section describes User APIs of FT311UARTInterface class.

### 3.1.1 SetConfig

Android developers use SetConfig(int baudRate, byte dataBits, byte stopBits, byte parity,byte flowControl) function of FT311UARTInterface class to set baud rate, data bits, stop bits, parity and flow control of FT311D/FT312D UART interface.
**Note:** The Android application must send this configuration before sending any application data.

Public void SetConfig(int baudRate, byte dataBits, byte stopBits, byte parity,byte flowControl)
baudRate: baud rate, min 300, max 921600, default set to 9600.
dataBits: data bits, 7: 7-bit databits, 8: 8-data bits, default 8-data bits.
stopBits: stop bits, 1: 1-stop bits, 2: 2-stop bits, default is set to 1-stop bits.
Parity: parity 0: none, 1:odd, 2:even,3:mark and 4:space. default is set to none.
flowControl: flow control, 0: none, 1-cts/rts, default is set to none.

### 3.1.2 SendData

Android developers use SendData(int numBytes, byte[] buffer) function of FT311UARTInterface class to send the data to FT311D/FT312D UART interface.
Public void SendData(int numBytes, byte[] buffer).
numBytes: number of bytes to transmit, maximum 256 per transfer.
Buffer: pointer to data buffer.

### 3.1.3 ReadData

Android developers use ReadData(int numBytes, byte[] buffer, int []actualNumBytes) function of FT311UARTInterface class to receive UART data.

Public void ReadData(int numBytes, byte[] buffer, int []actualNumBytes).
numBytes: number of bytes to read, MAX 256 per transfer.
Buffer: pointer to buffer pointer.
actualNumBytes: the actual number of bytes received.

## 3.2 FT311D/FT312D FT311-UART Layer

FT311-UART layer implements FT311UARTInterface class. The Android user uses the functions of this class to control,configure UART interfaces. All functions with FT311D and FT312D are the same.

The communication between Android device and FT311D/FT312D is done using the maximum 256 bytes long array of 8-bit data.

| |
|---|
| uartData |

**Table 3.1: UART command format**

| Name | Length(byte) | Description |
|---|---|---|
| uartData | N | array of N bytes to transfer. The data value. |

**Table 3.2: UART command packet parameters**

### 3.2.1 SetConfig

Public void SetConfig(int baudRate, byte dataBits, byte stopBits, byte parity, byte flowControl)

The FT311-UART layer sends the 8-bytes of below packet for this command. The remaining bytes of uartData are reserved for this command.

The command packet values for this command are:

| Name | Length | Descriptions |
|---|---|---|
| uartData[0..3] | 4 | First 4 bytes holds the baud rate, in little–endian format. |
| uartData[4] | 1 | Data bits.<br>7: for 7-bit data bits.<br>8: for 8-bit data bits. |
| uartData[5] | 1 | Number of stop bits.<br>1: 1- stop bits<br>2: 2- stop bits. |
| uartData[6] | 1 | Parity.<br>0: none.<br>1: odd<br>2: even.<br>3: mark<br>4: space |
| uartData[7] | 1 | Flow control.<br>0: none<br>1: hardware (CTS/RTS) |

**Table 3.3: UART SetConfig command parameters**

### 3.2.2 SendData

Public void SendData(int numBytes, byte [] buffer)
Android FT311-UART layer uses this command packet to send data using FT311D/FT312D UART interface.
Command details are given below:

| Name | Length(byte) | Description |
|---|---|---|
| uartData | N | Data to be send.<br>Array of N bytes to be transmitted, maximum 256 bytes. |

**Table 3.4: UART SendData command parameters**

### 3.2.3 ReadData

Pu~~blic void ReadData(int numBytes, byte[] buffer, int []actualNumBytes)~~

FT311D/FT312D sends the received data on its UART interface to Android FT311-UART layer in the below packet format.

Read Data details are given below:

| Name | Length(byte) | Values |
|---|---|---|
| uartData | N | UART data received on the UART interface , maximum 256 bytes. |

**Table 3.5: UART ReadData command parameters**

# 4 FT311D GPIO Interface

FT311D provides 7 GPIOs, that can be configured as input or output. The GPIOs are internally pulled up. By default the IOs are configured as Inputs.

FTDI Chip provides FT311GPIOInterface class with ReadPort, WritePort, ConfigPort, ResetPort methods for port operations and ResumeAccessory, DestroyAccessory for support functions to resume and destroy the accessory operations. For the use of the functions, see Annex B.

The Android application needs to include FT311GPIOInterface.java file into the project and call the above mentioned class functions to configure, read port, write port and reset port.

The method to add FT311GPIOInterface.java into the project in Eclipse environment is:
Copy the file into the src directory, e.g src\com\<package name>.
Write click on the project name in eclipse package explore, then select new->file->src->com->”package name”->advanced->link to the file.

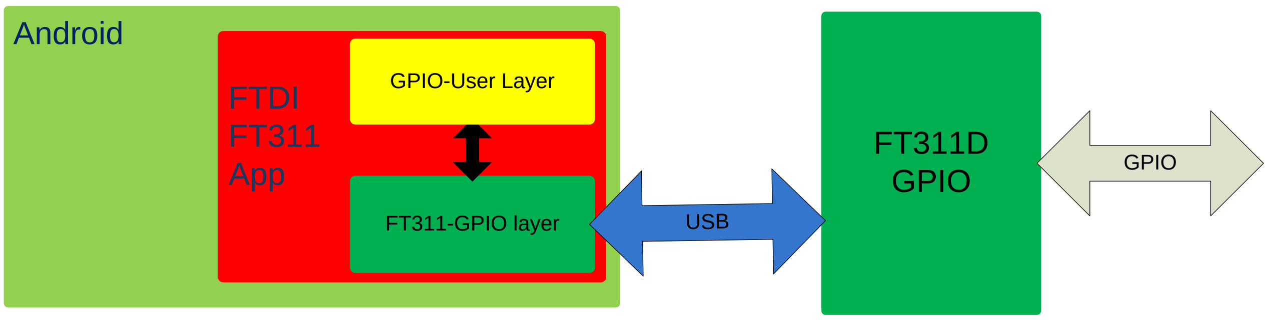

**Figure 4.1 : Android FTDI FT311D GPIO Application**

## 4.1 FT311D GPIO-User Layer

This section describes the User Interface APIs available for GPIO Interface to control, read, write GPIOs. The below listed functions are implemented in FT311-GPIO Interface layer.

### 4.1.1 ConfigPort

Android developer calls the ConfigPort(byte configOutmap, byte configInMap) to configure the FT311D GPIO pins as input or output.

Public void ConfigPort(byte configOutMap, byte configInMap).
configOutMap: the bits set in this bitmap will be configured as output, bit 0 configuring GPIO0 and so on.
configInMap: the bits in this map will be configured as Input, bit 0 configuring GPIO0 and so on.
If there is an overlap of input and output bitmap, input bitmap takes precedance.

### 4.1.2 ReadPort

Android developer uses ReadPort() routine of FT311GPIOInterface class to read the FT311D input ports.
Public byte ReadPort().
Return value is current level on input signals.

### 4.1.3 WritePort

Android developer uses WritePort(byte outData) function of FT311GPIOInterface class to write FT311D output ports. The output configured ports will only be driven.
Public void WritePort(byte outData).
outData: the bitmap of output port data, bit 0 corrosponds to GPIO0.

### 4.1.4 ResetPort

Android developer uses ResetPort() function of FT311GPIOInterface class to reset GPIO interface.  This command will set all the IOs into input mode.

Public void ResetPort().

## 4.2 FT311D FT311-GPIO Layer

This layer implements the FT311GPIOInterface class. This class implements User Interface APIs and communicates with the FT311D using USB.
FT311-GPIO Interface converts read, write, reset and config port commands into 4 bytes packet format to communicate with FT311D. The first byte, gpioCmd, in the table should be transferred first, and so on.

| | | | |
|---|---|---|---|
| gpioCmd | rdwrData | outMap | inMap |

**Table 4.1: GPIO command packet format**

| Name | Length (bytes) | Description |
|---|---|---|
| gpioCmd | 1 | GPIO command(configure/ read/write) |
| rdwrData | 1 | Read/write data |
| outMap | 1 | OUTPUT bitmap of port bits, the bits set in this bitmap will be configured as output. |
| inMap | 1 | INPUT bitmap of port bits, the bits set in this bitmap will be configure as input. |

**Table 4.2: GPIO command packet parameters**

### 4.2.1 ConfigPort

Public void ConfigPort(byte outMap, byte inMap).

Android FT311-GPIO layer sends the below packet to configure the GPIOs. By default the GPIOs are configured as inputs with internally pulled up.

The packet for ConfgPort is as below:

| Name | Length (bytes) | Values |
|---|---|---|
| gpioCmd | 1 | 0x11 |
| rdwrData | 1 | 0x00, reserved, not used for this command. |
| outMap | 1 | Bitmap of output port IOs, set bits will be configured as output. |
| inMap | 1 | Bitmap of input port IOs, set bits will be configured as INPUT. |

**Table 4.3: GPIO ConfigPort command parameters**

### 4.2.2 ReadPort

Public byte ReadPort().

FT311D sends the port data to Android layer, whenever there is a change in the signal levels on input ports.
Note: this data packet is from FT311D to Android, not from FT311-GPIO layer to FT311D.

The packet format for this command is as below:

| Name | Length (bytes) | Values |
|---|---|---|
| gpioCmd | 1 | 0x12 |
| rdwrData | 1 | GPIO status |
| outMap | 1 | 0x00 reserved |
| inMap | 1 | 0x00, reserved |

**Table 4.4: GPIO ReadPort command parameters**

Note: Only input pins are read.

### 4.2.3 WritePort

Public void WritePort(byte outData).

Android FT311-GPIO Layer sends the below packet for a WritePort command.

The Android GPIO Write command packet fields are as follows:

| Name | Length (bytes) | Values |
|---|---|---|
| gpioCmd | 1 | 0x13 |
| rdwrData | 1 | GPIO data |

| | | |
|---|---|---|
| outMap | 1 | 0x00, reserved for this command |
| inMap | 1 | 0x00, reserved for this command. |

**Table 4.5: GPIO WritePort command parameters**

Note: Only output pins are written with the value in rdwrData.

### 4.2.4 ResetPort

Public void ResetPort().

Android FT311-GPIO layer sends the below 4 byte packet to reset the GPIO interface of the FT311D GPIO. This command resets the GPIO module, and configures all GPIOs as inputs.

| Name | Length(bytes) | Values |
|---|---|---|
| gpioCmd | 1 | 0x14 |
| rdwrData | 1 | 0x00, reserved for this command. |
| inMap | 1 | 0x00, reserved for this command. |
| outMap | 1 | 0x00, reserved for this command. |

**Table 4.6: GPIO ResetPort command parameters**

# 5 FT311D PWM Interface

FT311D provides 4 PWM channels, pwm0..pwm3. All PWM channels have the same frequency (period).
Android applications can specify the frequency of all 4 channels with different duty cycles.
FTDI Chip provides FT311PWMInterface class with SetPeriod, SetDutyCycle and Reset methods for PWM operations and ResumeAccessory, DestroyAccessory for support functions to resume and destroy the accessory operations. For the use of the functions, please see Annex C.
The Android application needs to include FT311PWMInterface.java file into the project and call the above mentioned class functions to set period, duty cycle and reset.
The method to add FT311PWMInterface.java into the project in Eclipse environment is:
Copy the file into the src directory, e.g src\com\<package name>.
Write click on the project name in eclipse package explore, then select
new->file->src->com->"package name"->advanced->link to the file.

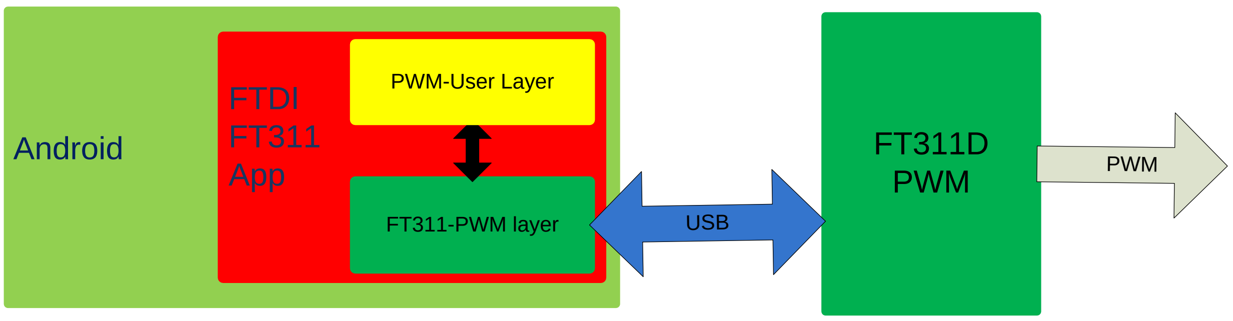

**Figure 5.1 : Android FTDI FT311D PWM Application**

## 5.1 FT311D PWM-User Layer

This section describes the methods of FT311PWMInterface class to be used in User Application layer.

### 5.1.1 SetPeriod

Android developer uses SetPeriod(int period) function of FT311PWMInterface class to set the period of all PWM channels.
Public void SetPeriod(int period);
Function to set the period of PWM channels.
period: period in milliseconds. 1..250, MAX 250 msecs, Minimum 1 msec.

### 5.1.2 SetDutyCycle

Android developer uses SetDutyCycle(byte pwmChannel, byte dutyCycle) function of FT311PWMInterface Class to set the duty cycle of the pwm channel.
Public void SetDutyCycle(byte pwmChannel, byte dutyCycle): function to set the duty cycle of specified pwm channel.
pwmChannel: channel number, should be between 0 ..3
dutyCycle: the percentage value of the duty cycle,e.g. to set 50% duty cycle, specify this value as 50. Minimum 5% and Maximum 95%.

### 5.1.3 Reset

FT311PWMInterface class provides Reset() function for Android user to Reset the PWM interface. This function brings the PWM interface to its default state. In the default state, the period is 1msec, and the duty cyle of all the channels is 0.
Public void Reset().

## 5.2 FT311D FT311-PWM Layer

This layer implements the User API for PWM-User interface.
FT311PWMInterface class uses the below 4 byte packet format to communicate with FT311D
The data is send in the order listed in the table 4.1. For fields of more than one byte, LSB should be sent first.

| | | |
|---|---|---|
| pwmCmd | pwmNumber | cmdData |

**Table 5.1: PWM command packet format**

| Name | Length(bytes) | Description |
|---|---|---|
| pwmCmd | 1 | PWM command. set period, set duty cycle. |
| pwmNumber | 1 | PWM channel number. |
| cmdData | 2 | Command Data, depends on type of the command. |

**Table 5.2: PWM command packet parameters**

### 5.2.1 SetPeriod

Public void SetPeriod(int period).
Android FT311-PWM layer sends the SetPeriod packet to set the period of all channels. The period will be same for all channels.

For PWM set period, the PWM packet fields are described below:

| Name | Length(bytes) | Description |
|---|---|---|
| pwmCmd | 1 | 0x21 |
| pwmNumber | 1 | 0x00, reserved for this command packet |
| cmdData | 2 | Period in milliseconds, MAX 250 and minimum 1 milliseconds. |

**Table 5.3: PWM SetPeriod command parameters**

### 5.2.2 SetDutyCycle

Public void SetDutyCycle(byte pwmChannel, byte dutyCycle).
Android FT311-PWM layer sends SetDutyCycle packet to set the duty cycle of individual channels. Before this command, period for the PWMs should be set.

For PWM set duty cycle, the PWM packet fields are described below:

| Name | Length(bytes) | Description |
|---|---|---|
| pwmCmd | 1 | 0x22 |
| pwmNumber | 1 | Pwm channel number, should be from 0..3. |
| cmdData | 2 | Duty cycle in numerical percentage(%) number.E.g to set 50% duty, set this value to 50. Minimum 5%, max 95 %. |

**Table 5.4: PWM SetDutyCycle command parameters**

### 5.2.3 Reset

Public void Reset().
Android FT311-PWM layer sends this command to reset the PWM module into default state, i.e. set the period to 1 msecmS and set the duty cycle of all channels to zero.

The command packet for this command is as follows:

| Name | Length(bytes) | Description |
|---|---|---|
| pwmCmd | 1 | 0x23 |
| pwmNumber | 1 | 0x00, reserved for this command. |
| cmdData | 2 | 0x00, reserved for this command. |

**Table 5.5: PWM Reset command parameters**

# 6 FT311D $I^2C$ Master Interface

FT311D provides $I^2C$ master interface, running Max 92 Khz. FT311D does not provide $I^2C$ slave functionality.
Note: FT311D $I^2C$ master does not support clock stretching, or multi-master support.

FTDI Chip provides FT311I2CInterface class with Reset, SetFrequency, ReadData and WriteData routines for $I^2C$ operations and ResumeAccessory, DestroyAccessory for support functions to resume and destroy the accessory operations. For the use of the functions, please see Annex D

The Android application needs to include FT311I2CInterface.java file into the project.
The method to add FT311I2CInterface.java into the project in Eclipse environment is:
Copy the file into their src directory, e.g src\com\<package name>.
Right click on the project name in eclipse package explore, then select new->file->src->com->"package name"->advanced->link to the file.

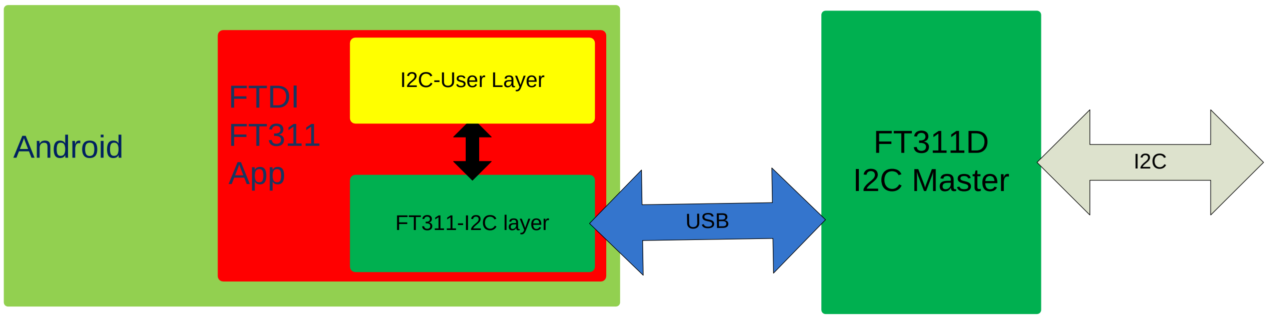

**Figure 6.1 : Android FTDI FT311D $I^2C$ Master Application**

## 6.1 FT311D $I^2C$-User Layer

This section describes the User APIs of FT311I2CInterface class to control, read,and write I²C bus.

### 6.1.1 SetFrequency

FT311D provides a set of values in units of KHz for $I^2C$ frequency. The user has to call SetFrequency(byte freq) function of FT311I2CInterface class to set the frequency of $I^2C$ interface. By default it's set to 92Khz.
Public void SetFrequency(byte freq).
Freq: frequency in units of KHz.
The supported values are: 23,44,60,92.

### 6.1.2 ReadData

Android developer uses ReadData(byte i2cDeviceAddress, byte transferOptions, byte numBytes,byte[] buffer, byte []actualBytes) function to read data.
Public byte ReadData(byte i2cDeviceAddress, byte transferOptions,byte numBytes,byte[] buffer,byte []actualBytes)
I2cDeviceAddress: 7-bits address of $I^2C$ device.
transferOptions: specifies the data transfer options. The bit position defined for each of the options are,
BIT0: if set then a start condition is generated in the $I^2C$ bus before the transfer begins. A bit mask is defined for this option in file I2CDemoActivity.java as bOption.START_BIT
BIT1: if set then a stop condition is generated in the $I^2C$ bus after the transfer ends. A bit mask is defined for this option in file I2CDemoActivity.java as bOption.STOP_BIT
BIT2: some $I^2C$ slaves require the $I^2C$ master to generate a NAK for the last data byte read. Setting this bit enables working with such $I^2C$ slaves. A bit mask is defined for this options in file I2CDemoActivity.java as bOption.NACK_LAST_BYTE
BIT3: the deviceAddress parameter is ignored if this bit is set. This feature may be useful in generating a special $I^2C$ bus conditions that do not require any address to be passed. A bit mask is defined for this options in file I2CDemoActivity.java as bOption.NO_DEVICE_ADDRESS
Bit4-7: Reserved
numBytes: number of bytes to read, maximum 252 bytes per transaction.
buffer: an array of bytes.
actualBytes: the number of actual bytes read.

The return value is a 8-bit bitmap of status, as described below:

| Bit | Value/description |
|---|---|
| 0 | command status.<br>1: error, general error<br>0: command passed. |
| 1 | 1: nack , device  can not accept any more data. |
| 2 | 1: invalid address. |
| 3..7 | Reserved for future use. |

**Table 6.1: I$^2$C Read return status**

This command will complete with one of the following status bits:
Bit 0: set or cleared.
Bit 2: set, invalid device address.

### 6.1.3 WriteData

Android Developers use WriteData(byte i2cDeviceAddress,byte transferOptions,byte numBytes, byte[] buffer, byte [] actualNumBytes) function to write the I$^2$C data to slave.

Public byte WriteData(byte i2cDeviceAddress, byte transferOptions, byte numBytes, byte [] buffer,byte [] actualNumBytes).
I2cDeviceAddress: 7-bit device address
transferOptions: specifies the data transfer options.The bit position defined for each of the options are,
BIT0: if set then a start condition is generated in the I$^2$C bus before the transfer begins. A bit mask is defined for this options in file I2CDemoActivity.java as bOption.START_BIT
BIT1: if set then a stop condition is generated in the I$^2$C bus after the transfer ends. A bit mask is defined for this options in file I2CDemoActivity.java as bOption.STOP_BIT
BIT2: Reserved
BIT3: the deviceAddress parameter is ignored if this bit is set. This feature may be useful in generating a special I$^2$C bus conditions that do not require any address to be passed. A bit mask is defined for this options in file I2CDemoActivity.java as bOption.NO_DEVICE_ADDRESS
Bit4-7: Reserved
numBytes: number of bytes to write,maximum 252 bytes per transfer.
buffer: array of bytes to data.
actualNumBytes: actual number of bytes send.

Returns value is a bitmap of 8-bits as below:

| Bit | Value/description |
|---|---|
| 0 | command status.<br>1: error, general error<br>0: command passed. |
| 1 | 1: nack , device  can not accept any more data. |
| 2 | 1: inavlid address. |
| 3..7 | Reserved for future use. |

**Table 6.2: I$^2$C Write return status**

The command can complete with:
Bit 0: set or clear.
Bit 1: set or clear, if set, check the actual length parameter to check the written bytes.
Bit 2: Invalid address  bit 2 set.

### 6.1.4 Reset

Android developers use Reset() function of FT311I2CInterface class to reset the interface. This function will set the I$^2$C interface to default state, 92 KHZ frequency.
Public void Reset().

## 6.2 FT311D FT311-I2C Layer

This layer implements the I²C-User layer functions in FT311I2CInterface class.
FT311I2CInterface class uses 5 bytes command packet to communicate with I$^2$C slave.
The packet bytes are sent in the order listed in table 5.3, first column byte goes first and so on.

| i2cCmd | i2cCmdData0 | I2cCmdData1 | i2cDataLength | i2cData |
|---|---|---|---|---|

**Table 6.3: I²C command packet format**

| Name | Length(bytes) | Description |
|---|---|---|
| i2cCmd | 1 | I²C command; read, write, set frequency, reset. |
| i2cCmdData0 | 1 | I²C command specific data0. |
| i2cCmdData1 | 1 | I²C command specific data1. |
| i2cDataLength | 1 | The number of read/write bytes, maximum per command packet 252 bytes. |
| i2cData | N | Array of data bytes, maximum N is 252 bytes per transfer. |

**Table 6.4: I²C command packet parameters**

### 6.2.1 SetFrequency

Public void SetFrequency(byte freq).

Android FT311-I2C interface class, sends  the below packet format for this command:

| Name | Length(bytes) | Values |
|---|---|---|
| i2cCmd | 1 | 0x31 |
| i2cCmdData0 | 1 | Frequency value, in KHz units of freq. Possible values are 23,44, 60 and 92. |
| I2cCmdData1 | 1 | 0x00, reserved for this command. |
| i2cDataLength | 1 | 0x00, reserved not used for this command. |
| i2cData | 1 | 0x00, reserved not used for this command. |

**Table 6.5: I²C SetFrequency command parameters**

### 6.2.2 WriteData

Public byte WriteData(byte i2cDeviceAddress, byte address, byte numBytes, byte [] buffer,byte []actualNumBytes).
Android FT311-I2C layer sends the below packet for this command.

The command values are :

| Name | Length(bytes) | Values |
|---|---|---|
| i2cCmd | 1 | 0x32 |
| i2cCmdData0 | 1 | I²C device address, 7-bit. |
| I2cCmdData1 | 1 | Data transfer options. |
| i2cDataLength | 1 | n,Length of the data to be transferred, Max value of n per transfer is 252 bytes. |
| i2cCmdData | N | Array of n bytes that Android wants to write.Max data length is 252 bytes. |

**Table 6.6: I²C WriteData command parameters**

FT311D sends to Android FT311-I2C layer the below response after completion of this command.

| Name | Length(bytes) | Values |
|---|---|---|
| i2cCmd | 1 | 0x32 |
| i2cCmdData0 | 1 | Status. |
| I2cCmdData1 | 1 | 0x00, reserved |
| i2cDataLength | 1 | Actual data bytes written, max 252 bytes. |

**Table 6.7: I²C WriteData response parameters**

Status byte for the WriteData command is:

| Bit | Value/description |
|---|---|
| 0 | This bit is only effective if Bit 0 is cleared.<br>1: error, general error<br>0: command passed. |
| 1 | 1:nack , device can not accept any more data. |
| 2 | 1:inavlid address. |
| 3..7 | Reserved for future use. |

**Table 6.8: I²C WriteData return status**

### 6.2.3 ReadData

Public byte ReadData(byte i2cDeviceAddress, byte address, byte numBytes, byte [] buffer,byte []actualNumBytes).

Android FT311-I²C layer sends the below packet to FT311D for this command.
The command values are:

| Name | Length(bytes) | Values |
|---|---|---|
| i2cCmd | 1 | 0x33 |
| i2cCmdData0 | 1 | I²C device address |
| i2cCmdData1 | 1 | Data transfer options. |
| i2cDataLength | 1 | n, Length of the data to be read, maximum 252 bytes per transfer. |

**Table 6.9: I²C ReadData command parameters**

FT311D sends read data back to Android device FT311-I²C layer in the below format.

| Name | Length(byte) | Values |
|---|---|---|
| i2cCmd | 1 | 0x33 |
| I2cCmdData0 | 1 | Status. |
| I2cCmdData1 | 1 | 0x00, reserved |
| i2cDataLength | 1 | Length of the read data, max 252 bytes |
| I2cData | N | Array of N bytes, N max is 252 |

**Table 6.10: I²C ReadData response parameters**

Status byte for ReadData command is:

| Bit | Value/description |
|---|---|
| 0 | command status:<br>1: error, general error<br>0: command passed. |
| 1 | 1: nack , device can not accept any more data. |
| 2 | 1: invalid address. |
| 3..7 | Reserved for future use. |

**Table 6.11: I²C ReadData return status**

### 6.2.4 Reset

Public void Reset()
Android FT311-I²C layer creates the below packet for this command.

| Name | Length(byte) | Values |
|---|---|---|
| I2cCmd | 1 | 0x34 |
| I2cCmdData0 | 1 | 0x00, reserved |
| I2cCmdData1 | 1 | 0x00, reserved |
| I2cDataLength | 1 | 0x00, reserved |
| I2cdata | 1 | 0x00, reserved. |

**Table 6.12: I²C Reset command parameters**

# 7 FT311D SPI Slave Interface

FT311D provides a SPI slave interface with supported clock rates of upto 24MHz. SPI slave will transfer data in MSBit/LSBit of byte based on the configuration.

FTDI Chip provides FT311SPISlaveInterface class with SetConfig, SendData, ReadData, Reset routines for SPI Slave operations and ResumeAccessory, DestroyAccessory for support functions to resume and destroy the accessory operations. For the use of the functions, please see Annex E.

The Android application needs to include FT311SPISlaveInterface.java file into the project and call the FT311SPISlaveInterface functions.

The method to add FT311SPISlaveInterface.java into the project in Eclipse environment is:
Copy the file into the src directory, e.g src\com\<package name>.
Write click on the project name in eclipse package explore, then select new->file->src->com->"package name"->advanced->link to the file.

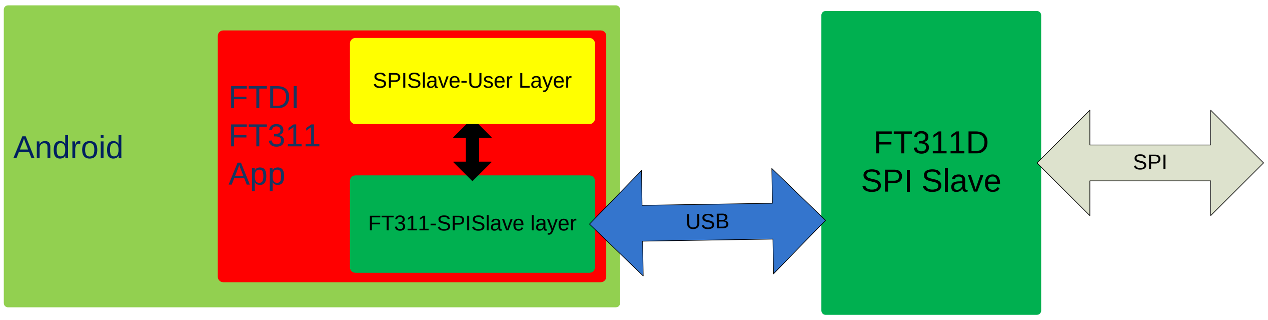

**Figure 7.1 : Android FTDI FT311D SPI Slave Application**

## 7.1 FT311D SPI Slave-User Layer

This layer uses the User functions of FT311-SPISlave layer to configure, read, and write SPI slave interface.

### 7.1.1 SetConfig

Android developers use SetConfig(byte clockPhase, byte dataOrder) function of FT311SPISlaveInterface class to config clock phase and data order of the SPI slave.
Public void SetConfig(byte clockPhase, byte dataOrder).
clockPhase: Clock phase variable values for different modes, default is set to mode 1.
  0: CPOL=0, CPHA=0.
  1: CPOL=0, CPHA=1.(default)
  2: CPOL=1, CPHA=0.
  3: CPOL=1, CPHA=1.
dataOrder: order of data on the SPI bus
  0: MSB
  1: LSB

### 7.1.2 SendData

Android developers use SendData(byte numBytes, byte[] buffer, byte[] actualNumBytes) of FT311SPISlaveInterface class to send data to the SPI master.
Public byte SendData(byte numBytes, byte[] buffer, byte []actualNumBytes).
numByte: number of bytes to transfer, maximum 255 per transfer.
buffer: array of buffer.
actualNumBytes : actual number of bytes send

Note: The SPI transactions are host initiated, so once this command is initiated by the SPI slave, the data is queued for the SPI host to read. If the SPI host did not issue read/write, the data will stay in queue and the SPI slave can not queue anymore data.

### 7.1.3 ReadData

Android developers use ReadData(byte numBytes, byte [] buffer, byte []actualNumBytes) function of FT311SPISlaveInterface class to Read data.
Public byte ReadData(byte numBytes, byte [] buffer, byte []actualNumBytes).
numBytes: number of bytes to read.
buffer:  array of length of numBytes.
actualNumBytes: actual number of bytes read, max 255 bytes.

### 7.1.4 Reset

Android developers use Reset() function of FT311SPISlaveInterface to reset the SPI slave interface of the FT311D. The default setting sets the clock phase and polarity to mode 1.
Public void Reset().

## 7.2 FT311D FT311-SPISlave Layer

Android FT311-SPISlave layer implements FT311SPISlaveInterface class. The function implemented in this class are used by SPISlave-User layer to communicate with FT311D The communication between Android device and FT311D is done using the below defined packet format.

| spiCmd | spiData |
|---|---|

Table 7.1: SPI Slave command format

| Name | Lentgh (bytes) | Description |
|---|---|---|
| spiCmd | 1 | FT311D communication protocol command. |
| spiData[N] | N | An array of 8-bits to hold read/write SPI data.Value of N should be less than or equal 255 bytes |

Table 7.2: SPI Slave command packet parameters

### 7.2.1 SetConfig

Public void SetConfig(byte clockPhase, byte dataOrder).
The Android FT311-SPISlave layer uses this command to set  clock polarity and phase of FT311D SPI slave Interface.

The command packet values are provided below:

| Name | Length(bytes) | Values |
|---|---|---|
| spiCmd | 1 | 0x51 |
| spiData | 1 | Clock phase.<br>0: CPOL=0, CPHA=0.<br>1: CPOL=0, CPHA=1. default<br>2: CPOL=1, CPHA=0.<br>3: CPOL=1, CPHA=1. |
| spiDataOrder | 1 | Data order on the SPI bus<br>0: MSB<br>1: LSB |

Table 7.3: SPI Slave SetConfig command parameters

### 7.2.2 SendData

Public byte SendData(byte numBytes, byte[] buffer, byte []actualNumBytes).
The Android device can schedule data to be sent to the SPI Master, eventually the SPI Master has to initiate a read command to read this data.

Android FT311-SPISlave layer creates the below packet to transmit the SPI data.

| Name | Length(byte) | Values |
|---|---|---|
| spiCmd | 1 | 0x52 |
| spiData[N] | N | Array of N bytes, specified by numBytes. Maximum length is 255 bytes. |

Table 7.4: SPI Slave SendData command parameters

FT311D sends the below command back to Android FT311-SPISlave layer in response to this command.

| Name | Length(byte) | Values |
|---|---|---|
| spiCmd | 1 | 0x52 |
| spiDataLength | 1 | Length of send data bytes |

**Table 7.5: SPI Slave response parameters**

### 7.2.3 ReadData

FT311D will send the received data from SPI Master to Android application's FT311-SPISlave layer. SPISlave-User layer can read this data with ReadData command as described in section 7.1.3.

FT311D sends the data to Android FT311-SPISlave layer in below mentioned format:

| Name | Length(byte) | Value |
|---|---|---|
| spiCmd | 1 | 0x53 |
| spiData[N] | N | Array of N bytes send by SPI Master, maximum length is 255. |

**Table 7.6: SPI Slave ReadData command parameters**

### 7.2.4 Reset

Android FT311-SPISlave class uses this function to reset the SPI Slave into default state.

The one byte packet for this command is as below:

| Name | Length(byte) | Values |
|---|---|---|
| spiCmd | 1 | 0x54 |

**Table 7.7: SPI Slave Reset command parameters**

# 8 FT311D SPI Master Interface

FT311D provides a SPI master supporting max clock rate of 24 Mhz. FT311D's SPI master will transfer data in MSBit/LSBit of byte based on the configuration.
FTDIChip provides FT311SPIMasterInterface class with SetConfig, SendData, ReadData and Reset routines for SPI Master operations and ResumeAccessory, DestroyAccessory for Android support. For the use of the functions, please see Annex F.

The Android Application needs to include FT311SPIMasterInterface.java file into the project and call the FT311SPIMasterInterface functions.

The method to add FT311SPIMasterInterface.java into the project in Eclipse environment is:
Copy the file into the src directory, e.g src\com\<package name>.
Write click on the project name in eclipse package explore, then select new->file->src->com->"package name"->advanced->link to the file.

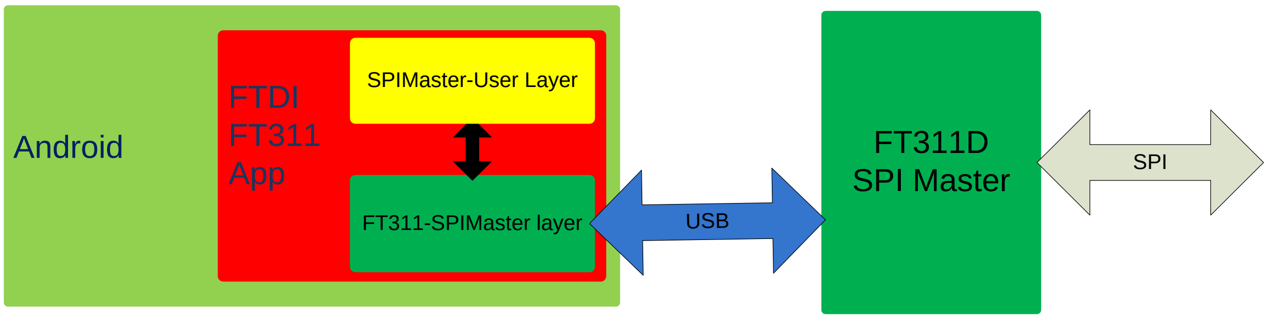

**Figure 8.1 : Android FTDI FT311D SPI Master Application**

## 8.1 FT311D SPIMaster-User layer

This layer uses the functions of FT311-SPIMaster layer, implemented in FT311SPIMasterInterface.java file, to configure SPI interface, read, write SPI data.

### 8.1.1 SetConfig

Android developers use SetConfig(byte clockPhase, byte dataOrder, int clockSpeed) function of FT311SPIMasterInterface class to configure clock phase, data order and clock speed of SPI Master interface.
Public void SetConfig(byte clockPhase,byte dataOrder, int clockSpeed).
clockPhase: Clock phase and polarity of SPI master interface, default is set to mode 1.
0: CPOL=0, CPHA=0 (mode 0).
1: CPOL=0, CPHA=1 (mode 1).
2: CPOL=1, CPHA=0 (mode 2).
3: CPOL=1, CPHA=1 (mode 3).
dataOrder: data oder on the SPI bus
0: MSB
1:LSB

clockSpeed:frequency of SPI interface. Defaut is set to 3 MHz and maximum is 24Mhz.

### 8.1.2 SendData

Android developers use SendData(byte numBytes, byte[] buffer, byte []numBytesSend) function of FT311SPIMasterInterface class to send the data to SPI slave. SPI master sends data in MSBit/LSBit first format.

public byte SendData(byte numBytes, byte [] buffer, byte [] numBytesSend).
numBytes: number of bytes to send. maximum is **255**.
buffer: pointer to data to transmit.
numBytesSend: the actual bytes send.

### 8.1.3 ReadData

Android developers use ReadData(byte numBytes, byte[] buffer, byte [] numBytesRead) function of FT311SPIMasterInterface class to read SPI data. The buffer values are set to 0xff for SPI Master Read operation.

Public byte ReadData(byte numBytes, byte[] buffer, byte [] numBytesRead).

numBytes: number of bytes to read. Maximum value is **255**.
Buffer: pointer to buffer to read data into.
numBytesRead: actual number of bytes read.

### 8.1.4 Reset

Android developers use Reset() function of FT311SPIMasterInterface class to reset the SPI Master interface. It resets the SPI interface to default state, i.e SPI clock frequency of 3Mhz, clock and phase to mode 1 and data order to MSB.
Public void Reset(void).

## 8.2 FT311D FT311-SPIMaster Layer

This layer implements FT311SPIMasterInterface class. The functions use a Maximum 256 byte long packet format to communicate with FT311D. The format of Packet is described below for each command.
Table 8.1 The packet for SPI master to communicate with FT311D

| | |
|---|---|
| spiCmd | spiData |

**Table 8.1: SPI Master command format**

| Name | Length | Description |
|---|---|---|
| spiCmd | 1 | SPI command identifier. |
| spiData | N | SPI master command data.<br>Maximum length is 255. |

**Table 8.2: SPI Master command packet parameters**

### 8.2.1 SetConfig

Public void SetConfig(byte clockPhase, byte dataOrder, int clockSpeed).
Android FT311-SPIMaster layer sends the 4 bytes packet format for this command.

The command packet for this command is as follows:

| Name | Length | Description |
|---|---|---|
| spiCmd | 1 | 0x61 |
| spiData[0] | 1 | clockPhase, the clock phase and polarity value to configure SPI master.<br>0: CPOL=0, CPHA=0 (mode 0).<br>1: CPOL=0, CPHA=1 (mode 1).<br>2: CPOL=1, CPHA=0 (mode 2).<br>3: CPOL=1, CPHA=1 (mode 3). |
| spiData[1] | 1 | Data order on the SPI bus<br>0: MSB<br>1: LSB |
| spiData[2..5] | 4 | clockSpeed, The baud rate in little endian format. Max is 24Mhz, default is set to 3 Mhz |

**Table 8.3: SPI Master Setconfig command parameters**

### 8.2.2 SendData

Public byte SendData(byte numBytes, byte [] buffer, byte [] numBytesSend).

Android FT311-SPIMaster layer sends the below packet format for this comamnd.

| Name | Length | Description |
|---|---|---|
| spiCmd | 1 | 0x62, command identifier for SPI master Send Data. |
| spiData[N] | numBytes | The pointer to buffer to send. The buffer length can not be more than 255. |

**Table 8.4: SPI Master SendData command parameters**

In response to this command, the FT311D sends the below packet to the Android FT311-SPIMaster layer:

| Name | Lentgh | Description |
|---|---|---|
| spiCmd | 1 | 0x62 |
| spiData[N] | numBytes | The data read while sending the data |

| | | |
|---|---|---|
| | | to SPI slave |

Table 8.5: SPI Master SendData response parameters

### 8.2.3 ReadData

Public byte ReadData(byte numBytes,byte [] buffer, byte []numBytesRead).

Android FT311-SPIMaster layer sends the 2 byte long packet for this command.
The command packet for ReadData is as follows:

| Name | Length | Description |
|---|---|---|
| spiCmd | 1 | 0x63 |
| spiData[N] | numBytes | numBytes long buffer, with values set to 0xff. The max value of numBytes should be less than or equal to 255. |

Table 8.6: SPI Master ReadData command parameters

In response to this command, the FT311D sends the read data to FT311-SPIMaster layer. The FT311D uses the below packet to send the read data to Android:

| Name | Length | Description |
|---|---|---|
| spiCmd | 1 | 0x63 |
| spiData[N] | numBytes | numBytes long buffer pointer with values read from SPI slave. The maximum value of numBytes should be less than or equal to 255. |

Table 8.7: SPI Master ReadData response parameters

### 8.2.4 Reset

Public void Reset().

This command will reset the SPI master interface into default state, i.e clock frequency of 3Mhz, clock phase and polarity to mode 1 and data order to MSB.

The Android FT311-SPIMaster layer sends 1 packet for this command.

| Name | Length | Description |
|---|---|---|
| spiCmd | 1 | 0x64 |

Table 8.8: SPI Master Reset command parameters

# 9 Annex A : UART

```
/* configure the UART */
configButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                configButton.setBackgroundResource(drawable.start);
                uartInterface.SetConfig(baudRate, dataBit, stopBit, parity, flowControl);
        }
});

/* read data */
public void run()
{
        while(true)
        {
                Message msg = mHandler.obtainMessage();
                try
                {
                        Thread.sleep(50);
                }
                 catch(InterruptedException e)
                {
                }
                status = uartInterface.ReadData((byte)64, readBuffer, actualNumBytes);
                mHandler.sendMessage(msg);
        }
}

/*write data*/
writeButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
         {
                // TODO Auto-generated method stub
                writeButton.setBackgroundResource(drawable.start);
                if(writeText.length() != 0x00)
                {
                        numBytes = (byte)writeText.length();
                        for(count=0;count<numBytes;count++)
                        {
                                writeBuffer[count] = writeText.getText().charAt(count);
                        }
                        status = uartInterface.SendData(numBytes, writeBuffer);
                }
        }
});
```

# 10 Annex B : GPIO

```
/*user code to configure port data*/
configbutton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                configbutton.setBackgroundResource(drawable.start);
                togglebutton0.setChecked(false);
                togglebutton1.setChecked(false);
                togglebutton2.setChecked(false);
                togglebutton3.setChecked(false);
                togglebutton4.setChecked(false);
                togglebutton5.setChecked(false);
                togglebutton6.setChecked(false);
                outData &= ~outMap;
                writedata.setText("0x" + Integer.toHexString(outData & 0xff));
                gpiointerface.ConfigPort(outMap, inMap);
        }
});

/*user code to read the accessory data*/
readbutton.setOnClickListener(new View.OnClickListener()
 {
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                readbutton.setBackgroundResource(drawable.start);
                inData = gpiointerface.ReadPort();
                ProcessReadData(inData);
        }
});


/*user code to write port*/
writebutton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                writebutton.setBackgroundResource(drawable.start);
                /*send only the output mapped data*/
                outData &= outMap;
                writedata.setText("0x" + Integer.toHexString(outData & 0xff));
                gpiointerface.WritePort(outData);
        }
});
```

# 11 Annex C : PWM

```
/* set the period */
periodButton.setOnClickListener(new View.OnClickListener()
 {
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                periodButton.setBackgroundResource(drawable.start);
                /*read the period value*/
                if(periodValue.length() == 0x00){
                periodValue.setText("1");
        }
        period = Integer.parseInt(periodValue.getText().toString());
        /*take care of zero*/
        if(period == 0x00){
        period = 1;
        periodValue.setText("1");
        }

        pwmInterface.SetPeriod(period);

        }
});

/* set duty cycle */
public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser)
{
        // TODO Auto-generated method stub
        dutyCycle = (byte)progress;
        dutyCycle1Value.setText(Integer.toString(progress));;
        pwmInterface.SetDutyCycle((byte)1,dutyCycle);
}
```

# 12 Annex D : I2C

```
/*Set the frequency*/
freqButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                /*read the bytes from the write box*/
                if(freqText.length() != 0)
                        deviceAddress = (byte) Integer.parseInt(freqText.getText().toString());
                else
                        deviceAddress = 92; /*default*/

                freqText.setText(Integer.toString(deviceAddress));
                i2cInterface.SetFrequency(deviceAddress);
        }
});

/*read data from the I2C slave */
readButton.setOnClickListener(new View.OnClickListener()
 {
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                readButton.setBackgroundResource(drawable.start);

                /*for every read, clear the previous read*/
                readText.setText("");

                status = 0x00;
                /*do the sanity checks*/
                /*device address?*/
                if(deviceAddressText.length() == 0)
                {
                        status = 0x4;
                }

                /*address to read from*/
                if(addressText.length() == 0)
                {
                        status |= 0x04;
                }

                /*if all good, then only proceed to read*/
                if(status == 0x00)
                {
                        deviceAddress = (byte) Integer.parseInt(deviceAddressText.getText().toString());
                        numBytes = (byte)Integer.parseInt(numBytesText.getText().toString());
                        address = (byte)Integer.parseInt(addressText.getText().toString());

                        /*blocking call*/
                        byte transferOptions;
                        transferOptions = bOption.START_BIT;
                        byteCount = 0;
                        writeBuffer[byteCount++] = address;
                        /* write the address first without stop condition */
                        status = i2cInterface.WriteData(deviceAddress, transferOptions, byteCount, writeBuffer,
                        actualNumBytes);
```

```
                    /*read the bytes from the text box*/
                    transferOptions=(bOption.START_BIT|bOption.STOP_BIT| bOption.NACK_LAST_BYTE);
                    status = i2cInterface.ReadData(deviceAddress, transferOptions, numBytes, readBuffer,
                    actualNumBytes);

                    /*for(count = 0;count<(actualNumBytes[0]-1);count++)
                    {
                            readText.append(Integer.toString(readBuffer[count]));
                            readText.append(",");
                    }
                    readText.append(Integer.toString(readBuffer[count]));*/

                    char [] displayReadbuffer;
                    displayReadbuffer = new char[60];
                    int displayActualNumBytes;
                    displayActualNumBytes = actualNumBytes[0];

                    for(count = 0;count<(actualNumBytes[0]);count++)
                    {
                            displayReadbuffer[count] = (char)readBuffer[count];
                    }
                    readText.append(String.copyValueOf(displayReadbuffer,0, displayActualNumBytes));
            }

            statusText.setText("0x"+Integer.toHexString(status));
        }
});

/* write data to I2C slave device */
writeButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
            // TODO Auto-generated method stub
            writeButton.setBackgroundResource(drawable.start);

            /*by default, status is good*/
            status = 0x00;
            /*do the sanity checks for required values*/
            /*device address needed??*/
            if(writeDeviceAddressText.length() == 0)
            {
                    status = 0x4;
            }
            /*the write values have to be specified*/
            if(writeText.length() == 0)
            {
                    status |= 1; /*general error*/

            }
            /*also need the address to write to*/
            if(writeAddressText.length() == 0)
            {
                    status |= 0x04;
            }
            /*if all good??/*
            if(status ==  0x00)
            {
                    /*read the bytes from the write box*/
                    deviceAddress = (byte) Integer.parseInt(writeDeviceAddressText.getText().toString());
                    numBytes = (byte) writeText.length();
                    address = (byte)Integer.parseInt(writeAddressText.getText().toString());
```

```
                /*parse the string*/
                byteCount = 0;
                tempCount = 0;
                /* insert 1 byte address at the start,  add 1 address byte*/
                writeBuffer[byteCount++] = address;
                byteCount += (byte)writeText.length();

                for(count=0;count<numBytes;count++)
                {
                        writeBuffer[count+1] = (byte)writeText.getText().charAt(count);
                }

                /* subtract 1 for the address byte */
                writeNumBytesText.setText(Integer.toString(byteCount-1));
                /*blocking call*/
                byte transferOptions;
                transferOptions = (bOption.START_BIT | bOption.STOP_BIT);
                status = i2cInterface.WriteData(deviceAddress, transferOptions, byteCount, writeBuffer,
                actualNumBytes);
            }
            writeStatusText.setText("0x"+Integer.toHexString(status));
        }
});
```

# 13 Annex E : SPI Slave

```
/*config SPI Slave interface */
configButton.setOnClickListener(new View.OnClickListener()
 {
        @Override
        public void onClick(View v)
        {
                configButton.setBackgroundResource(drawable.start);
                clockPhaseMode = (byte)clockPhaseSpinner.getSelectedItemPosition();
                dataOrderSelected = (byte)dataOrderSpinner.getSelectedItemPosition();
                spisInterface.SetConfig(clockPhaseMode,dataOrderSelected);
        }
});

/* read data */
public void run()
{
        while(true)
        {
                Message msg = mHandler.obtainMessage();
                try
                {
                        Thread.sleep(50);
                }
                 catch(InterruptedException e)
                {
                }

                numReadWritten[0] = 0x00;
                status = spisInterface.ReadData((byte)64, readBuffer, numReadWritten);
                mHandler.sendMessage(msg);
        }
}

/*write data*/
writeButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                writeButton.setBackgroundResource(drawable.start);
                numReadWritten[0] = 0x00;

                /*parse the string*/
                byteCount = 0;
                tempCount = 0;

                /*read the bytes from the write box*/
                if(writeText.length() != 0)
                {
                        byteCount = (byte)writeText.length();
                        for(count=0; count < byteCount; count++)
                        {
                                readBuffer[count] = (byte)writeText.getText().charAt(count);
                        }

                        /*send only when there is something to be send*/
                        if(byteCount != 0x00)
                                status = spisInterface.SendData(byteCount, readBuffer, numReadWritten);
                }
                writeStatusText.setText(Integer.toString(numReadWritten[0]));

        }
});
```

# 14 Annex F : SPI Master

```
/*config the SPI Master Interface*/
configButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                configButton.setBackgroundResource(drawable.start);

                if(clockFreqText.length() == 0x00)
                {
                        clockFreqText.setText(Integer.toString(3000000));
                }

                clockPhaseMode = (byte)clockPhaseSpinner.getSelectedItemPosition();
                dataOrderSelected = (byte)dataOrderSpinner.getSelectedItemPosition();
                clockFreq = Integer.parseInt(clockFreqText.getText().toString());
                spimInterface.SetConfig(clockPhaseMode, dataOrderSelected, clockFreq);
        }
});

/* read data */
readButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                readButton.setBackgroundResource(drawable.start);
                numReadWritten[0] = 0x00;

                if(numBytesText.length() != 0)
                {
                        numBytes = (byte)Integer.parseInt(numBytesText.getText().toString());
                        for(count=0;count<numBytes;count++)
                        {
                                readWriteBuffer[count]=(byte)0xff;
                        }

                        status = spimInterface.ReadData(numBytes, readWriteBuffer, numReadWritten);

                        char [] displayReadbuffer;
                        displayReadbuffer = new char[64];
                        int displayActualNumBytes;
                        displayActualNumBytes = numReadWritten[0];

                        for(count = 0;count<(numReadWritten[0]);count++)
                        {
                                displayReadbuffer[count] = (char)readWriteBuffer[count];
                        }
readText.append(String.copyValueOf(displayReadbuffer, 0, displayActualNumBytes));

                }
                statusText.setText(Integer.toString(numReadWritten[0]));
        }
});

  /*write data*/
writeButton.setOnClickListener(new View.OnClickListener()
{
        @Override
        public void onClick(View v)
        {
                // TODO Auto-generated method stub
                writeButton.setBackgroundResource(drawable.start);
                numReadWritten[0] = 0x00;

                /*parse the string*/
                byteCount = 0;
                tempCount = 0;

                /*read the bytes from the write box*/
                if(writeText.length() != 0)
                {
                        numBytes = (byte) writeText.length();
                        byteCount = (byte)writeText.length();
                        for(count=0; count < byteCount; count++)
                        {
                                readWriteBuffer[count] = (byte)writeText.getText().charAt(count);
                        }

                        /*send only when there is something to be send*/
                        if(byteCount != 0x00)
status = spimInterface.SendData(byteCount, readWriteBuffer, numReadWritten);
                }

                writeStatusText.setText(Integer.toString(numReadWritten[0]));
        }
});
```

# 15 Appendix A – List of Tables & Figures

## 15.1 List of Tables

## 15.2 List of Figures

# 16 APPENDIX B - Contact Information

Head Office – Glasgow, UK

Future Technology Devices International Limited
Unit 1, 2 Seaward Place, Centurion Business Park
Glasgow G41 1HH
United Kingdom
Tel: +44 (0) 141 429 2777
Fax: +44 (0) 141 429 2758

| | |
|---|---|
| E-mail (Sales) | sales1@ftdichip.com |
| E-mail (Support) | support1@ftdichip.com |
| E-mail (General Enquiries) | admin1@ftdichip.com |

Branch Office – Tigard, Oregon, USA

Future Technology Devices International Limited (USA)
7130 SW Fir Loop
Tigard, OR 97223
USA
Tel: +1 (503) 547 0988
Fax: +1 (503) 547 0987

| | |
|---|---|
| E-Mail (Sales) | us.sales@ftdichip.com |
| E-Mail (Support) | us.support@ftdichip.com |
| E-Mail (General Enquiries) | us.admin@ftdichip.com |

Branch Office – Taipei, Taiwan

Future Technology Devices International Limited (Taiwan)
2F, No. 516, Sec. 1, NeiHu Road
Taipei 114
Taiwan, R.O.C.
Tel: +886 (0) 2 8791 3570
Fax: +886 (0) 2 8791 3576

| | |
|---|---|
| E-mail (Sales) | tw.sales1@ftdichip.com |
| E-mail (Support) | tw.support1@ftdichip.com |
| E-mail (General Enquiries) | tw.admin1@ftdichip.com |

Branch Office – Shanghai, China

Room 1103, No. 666 West Huaihai Road,
Shanghai, 200052
China
Tel: +86 21 62351596
Fax: +86 21 62351595

| | |
|---|---|
| E-mail (Sales) | cn.sales@ftdichip.com |
| E-mail (Support) | cn.support@ftdichip.com |
| E-mail (General Enquiries) | cn.admin@ftdichip.com |

# 17 Appendix C - Revision History

| | |
|---|---|
| Document Title: | FT31xD Android Programmers Guide |
| Document Reference No.: | FT_000532 |
| Clearance No.: | FTDI# 307 |
| Product Page: | http://www.ftdichip.com/FTProducts.htm |
| Document Feedback: | Send Feedback |

| | |
|---|---|
| Version 1.0 | Initial Release |
| Version 1.1 | Updated to include FT312D |