**Kneron Inc**

Document Name: **Kneron I2C Master Mode Driver API**

# I2C Master Mode Driver API

## Kneron Inc

Engineering Design Document

# Table of Contents

# 1 Introduction

This document describes I2C master mode driver API usage, it is designed in polling way instead of interrupt way, user should be aware of this. At present it supports only 7-bit slave address and bus speed 100 Khz, 400 Khz and 1Mhz; it provides few simplified API to transmit and receive data to or from I2C peripherals. There are four I2C controllers on Mozart, user needs to set up correct pin configuration before using them.

# 2 Reference

Kneron Mozart Design Specification, Rev. 0.5, Faraday, Feb. 2019
FTIIC010_Block_Data_Sheet_v1.23.pdf, Faraday, December. 2018

# 3 Acronyms, Abbreviations, Definitions

I2C Bus – Inter-Integrated Circuit Bus
Polling way – Using CPU to Periodically wait for the particular status of peripherals

# 4 API description

```
kdp_status_e kdp_i2c_init(
        kdp_i2c_ctrl_e ctrl_id,
        kdp_i2c_bus_speed_e bus_speed)

typedef enum{
   KDP_I2C_CTRL_0,
   KDP_I2C_CTRL_1,
   KDP_I2C_CTRL_2,
   KDP_I2C_CTRL_3
}kdp_i2c_ctrl_e;

typedef enum{
   KDP_I2C_SPEED_100K,
   KDP_I2C_SPEED_400K,
   KDP_I2C_SPEED_1M
}kdp_i2c_bus_speed_e;
```

| Parameters | Description |
|---|---|
| ctrl_id | I2C controller ID, please refer to kdp_i2c_ctrl_e |
| bus_speed | Bus speed, please refer to kdp_i2c_bus_speed_e |

| Return | Description |
|---|---|
| KDP_STATUS_OK | Successful to execute the call |
| KDP_STATUS_ERROR_CTRL_NO_EXIST | Invalid I2C controller ID |
| KDP_STATUS_ERROR_I2C_RESET_FAILED | Failed to reset the I2C controller |

The kdp_i2c_init() initializes the specified I2C controller including clock enable, controller reset and bus speed configuration.

**kdp_status_e kdp_i2c_deinit(kdp_i2c_ctrl_e ctrl_id)**

| Parameters | Description |
|---|---|
| ctrl_id | I2C controller ID, please refer to kdp_i2c_ctrl_e |

| Return | Description |
|---|---|
| KDP_STATUS_OK | Successful to execute the call |
| KDP_STATUS_ERROR | Failed to execute the call |

This function disables the corresponding clock for the controller.

**kdp_status_e kdp_i2c_transmit(**
**kdp_i2c_ctrl_e ctrl_id,**
**uint16_t slave_addr,**
**uint8_t *data,**
**uint32_t num,**
**kdp_bool_e with_STOP)**

| Parameters | Description |
|---|---|
| ctrl_id | I2C controller ID, please refer to kdp_i2c_ctrl_e |
| slave_addr | 7-bit slave address of I2C device |
| Data | A buffer address for data transmission |
| Num | Number of bytes to transmit |
| with_STOP | Set STOP condition on the bus after data is all sent out |

| Return | Description |
|---|---|
| KDP_STATUS_OK | Successful to execute the call |
| KDP_STATUS_ERROR_I2C_BUS_BUSY | Bus is busy |
| KDP_STATUS_ERROR_I2C_DEV_NACK | Device does not respond ACK |
| KDP_STATUS_ERROR_I2C_STATUS_TIMEOUT | Completion status timeout |

This function will first set START condition then send slave address for write operations; if 9$^{th}$ bit is NACK, it returns DEV_NACK error, and if it is ACK, controller will continue to send out all data with specified number of bytes, once it is done it will set STOP condition while the 'with_STOP' is KDP_BOOL_TRUE.
For every byte transmission, it returns DEV_NACK error while encountering NACK at 9$^{th}$ bit.

**kdp_status_e kdp_i2c_receive(**
**kdp_i2c_ctrl_e ctrl_id,**
**uint16_t slave_addr,**
**uint8_t *data,**
**uint32_t num,**

**kdp_bool_e with_STOP)**

| Parameters | Description |
|---|---|
| ctrl_id | I2C controller ID, please refer to kdp_i2c_ctrl_e |
| slave_addr | 7-bit slave address of I2C device |
| Data | A buffer address for data transmission |
| Num | Number of bytes to transmit |
| with_STOP | Set STOP condition on the bus after data is all sent out |

| Return | Description |
|---|---|
| KDP_STATUS_OK | Successful to execute the call |
| KDP_STATUS_ERROR_I2C_BUS_BUSY | Bus is busy |
| KDP_STATUS_ERROR_I2C_DEV_NACK | Device does not respond ACK |
| KDP_STATUS_ERROR_I2C_STATUS_TIMEOUT | Completion status timeout |

This function will first set START condition then send slave address for read operations; if 9$^{th}$ bit is NACK, it returns DEV_NACK error, and if it is ACK, controller will continue to read data from slave devices with specified number of bytes, and do ACK at 9$^{th}$ bit for every byte reception, except the last byte reception with ‘with_STOP’ is set; in that case, controller will do NACK and set STOP condition.