RL78 Family

Renesas Sensor Control Modules Software Integration System

Introduction

This application note explains the sensor control modules for HS300x and HS400x (Renesas high performance relative humidity and temperature sensor), FS2012, FS3000 and FS1015 (Renesas High Performance Flow Sensor Module), ZMOD4410 and ZMOD4510 (Digital Gas Sensors), OB1203 (Heart Rate, Blood Oxygen Concentration, Pulse Oximetry, Proximity, Light and Color Sensor) and I2C communication middleware for Renesas sensors using Software Integration System (SIS).

These control modules acquire the sensor data using the IIC Communication component (IIC Communication (Master mode) component). And calculate relative humidity value [%RH] and temperature value [°C] for HS300x and HS400x, flow value [SLPM (standard litter per minute) or [SLPM (standard cubic centimeter per minute)] for FS2012, air velocity value [m/sec] for FS3000/1015, environmental gas value for ZMOD4410, ZMOD4450 and ZMOD4510 and light/proximity/PPG value for OB1203.

Hereinafter, the modules described in this application note is abbreviated as following,

- The sensor control module for HS300x: HS300x SIS module
- The sensor control module for HS400x: HS400x SIS module
- The sensor control module for FS2012: FS2012 SIS module
- The sensor control module for FS3000: FS3000 SIS module
- The sensor control module for FS1015: FS1015 SIS module
- The sensor control module for ZMOD4410, ZMOD4450 and ZMOD4510: ZMOD4XXX SIS module
- The sensor control module for OB1203 SIS module
- The I2C communication middleware module: COMMS SIS module

Target Device

- **Sensors:**
  - Renesas Electronics HS300x and HS400x High Performance Relative Humidity and Temperature Sensor (HS300x sensor and HS400x sensor)
  - Renesas Electronics FS2012, FS3000 and FS1015 Renesas High Performance Flow Sensor Module (FS2012 sensor, FS3000 sensor and FS1015 sensor)
  - Renesas Electronics Digital Gas Sensers ZMOD4410 (ZMOD4410 Indoor Air Quality Platform), ZMOD4450 (ZMOD4450 Refrigeration Air Quality Sensor Platform) and ZMOD4510 (ZMOD4510 Outdoor Air Quality Platform)
  - Renesas Electronics Heart Rate, Blood Oxygen Concentration, Pulse Oximetry, Proximity, Light and Color Sensor (OB1203 sensor)

- **RL78 Family MCUs:**
  MCUs supported the following IIC Communication (Master mode) component
  - Serial Interface IICA
  - Simplified I2C using Serial Array Unit (SAU)

- **Operation confirmed MCU:**
  - RL78/G23 (IIC Communication (Master mode) component)

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Target Compiler

- Renesas Electronics C/C++ Compiler Package for RL78 Family

Reference Documents

- Renesas Electronics HS300x Datasheet (August 8, 2021) (R36DS0010EU0701)
- Renesas Electronics HS400x Datasheet (June 22,2022) (R36DS0022EU0102)
- Renesas Electronics FS2012 Series Datasheet (August 24, 2018)
- Renesas Electronics FS3000 Series Datasheet (May 31, 2022)
- Renesas Electronics FS1015 Series Datasheet (June 2, 2022)
- Renesas Electronics ZMOD4410 Datasheet (December 17, 2021)
- Renesas Electronics ZMOD4510 Datasheet (June 30, 2021)
- Renesas Electronics ZMOD4450 Datasheet (June 30, 2021)
- Renesas Electronics OB1203 Datasheet (January 12, 2021)
- Smart Configurator User’s Manual : RL78 API Reference (R20UT4852)
- RL78/G23 User’s Manual: The latest version can be downloaded from the Renesas Electronics website.
- Technical Update/Technical News
  The latest information can be downloaded from the Renesas Electronics website.
- RL78 Family Compiler CC-RL User's Manual (R20UT3123)
  The latest versions can be downloaded from the Renesas Electronics website.

Contents

1. Overview of Renesas Sensor Control Modules

The Renesas sensor control modules described in this application note is a hardware abstraction layer of Renesas sensors. This hardware abstraction layer includes sensor API and communication middleware for various Renesas sensors. The software architecture of Renesas sensor hardware abstraction layer is shown below “Figure 1-1 Renesas sensor software architecture”.

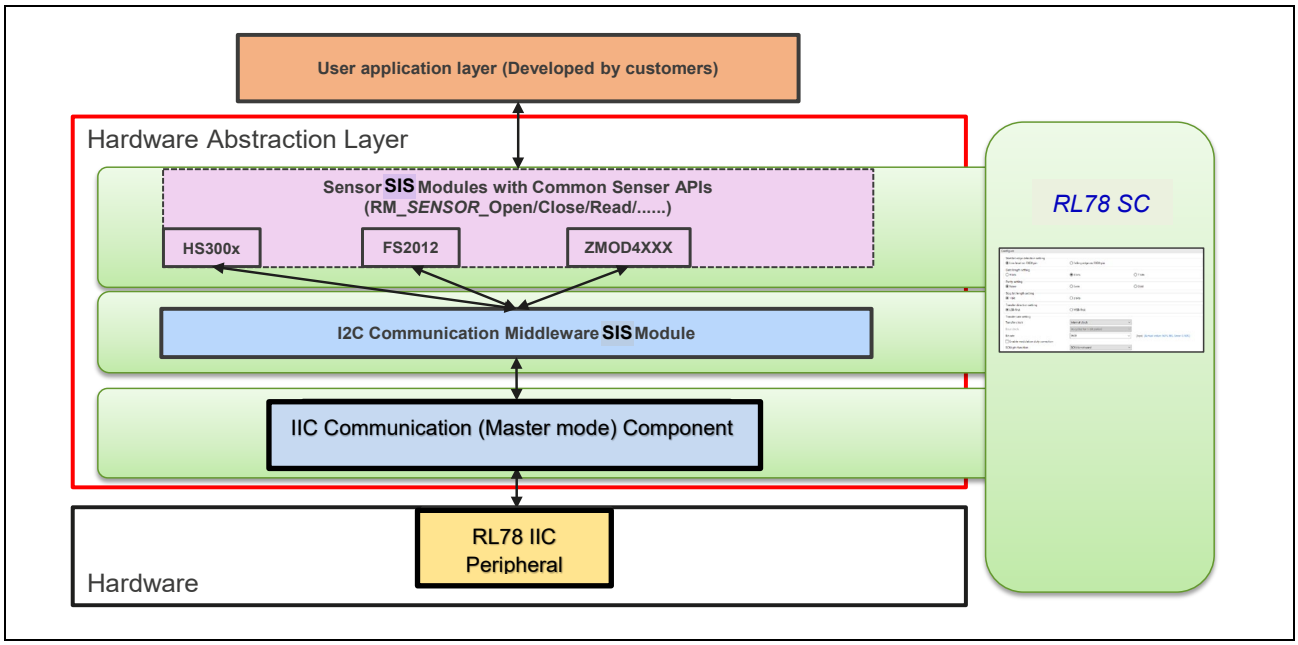


Figure 1-1 Renesas sensor software architecture

The hardware abstraction layer has three layers, “Sensor API”, “I2C communication middleware” and “IIC Communication (Muster mode) component.

The sensor APIs of HS300x and HS400x sensors, FS2012, FS3000 and FS1015 sensors, ZMOD4410, 4450 and 4510 sensors and OB1203 sensor are provided as “HS300x SIS module”, “HS400x SIS module”, “FS2012 SIS module”, “FS3000 SIS module”, “FS1015 SIS module”, “ZMOD4XXX SIS module”, “OB1203 SIS module” and the I2C communication middleware are provided as “I2C communication middleware SIS module”.

The “HS300x SIS module”, “HS400x SIS module”, “FS2012 SIS module”, “FS3000 SIS module”, “FS1015 SIS module”, “ZMOD4XXX SIS module” and “OB1203 SIS module” provide a method to receive sensor data of the HS300x, HS400x, FS2012, FS3000, FS1015, ZMOD4410&ZMOD4450&4510 and OB1203 sensors connected to the I2C bus of RL78 family MCUs via “I2C communication middleware SIS module”.

Table 1-1 shows the available sensors. Table 1-2 shows the available IIC SIS modules.

Table 1-1 Available Sensors

Available Sensors	Reference Datasheet
HS300x High Performance Relative Humidity and Temperature Sensor	HS300x Datasheet (August 9, 2021) (R36DS0010EU0701)
HS400x High Performance Relative Humidity and Temperature Sensor	HS400x Datasheet (June 6, 2022) (R36DS0022EU0102)
FS2012 High Performance Flow Sensor Module	FS2012 Series Datasheet (August 24, 2018)
FS3000 Air Velocity Sensor Module	FS3000 Series Datasheet (May 31, 2022)
FS1015 Air Velocity Sensor Module	FS1015 Series Datasheet (June 2, 2022)
ZMOD4410 Digital Gas Senser (ZMOD4410 Indoor Air Quality Platform)	ZMOD4410 Datasheet (June 30, 2021)
ZMOD4450 Digital Gas Sensor (ZMOD4450 Refrigeration Air Quality Sensor Platform)	ZMOD4450 Datasheet (June 30, 2021)
ZMOD4510 Digital Gas Sensor (ZMOD4510 Outdoor Air Quality Platform)	ZMOD4510 Datasheet (June 30, 2021)
OB1203 Heart Rate, Blood Oxygen Concentration, Pulse oximetry, Proximity, Light and Color Sensor	OB1203 Datasheet (January 12, 2021)

Table 1-2 Available IIC Communication (Master mode) components

Available IIC Communication (Master mode) components	Reference User's Manual
Serial Interface IICA	Smart Configurator User's Manual : RL78 API Reference
Simplified I2C using Serial Array Unit (SAU)	

1.1 Outline of HS300x SIS Module

"Table 1-3 HS300x SIS module API Functions" lists the HS300x SIS module API functions.

Table 1-3 HS300x SIS module API Functions

Function	Description
RM_HS300X_Open ()	This function opens and configures the HS300x SIS module.
RM_HS300X_Close ()	This function disables specified HS300x control block.
RM_HS300X_MeasurementStart ()	This function starts a measurement.
RM_HS300X_Read ()	This function reads ADC data from HS300x sensor.
RM_HS300X_DataCalculate ()	This function calculates humidity [%RH] and temperature [Celsius] from ADC data.
RM_HS300X_ProgrammingModeEnter ()	This function places the HS300x into programming mode.
RM_HS300X_ResolutionChange ()	This function changes the HS300x resolution.
RM_HS300X_SensorIdGet ()	This function obtains the sensor ID of HS300x.
RM_HS300X_ProgrammingModeEixt ()	This function exits the HS300x programming mode.
rm_hs300x_callback ()	This function is callback function for HS300x SIS module.

1.2 Outline of HS400x SIS Module

“HS400x control module API Functions” lists the HS400x SIS module API functions.

Table 1-4 HS400x control module API Functions

Function	Description
RM_HS400X_Open ()	This function opens and configures the HS400x control module.
RM_HS400X_Close ()	This function disables specified HS400x control block.
RM_HS400X_MeasurementStart ()	This function starts a measurement.
RM_HS400X_MeasurementStop ()	This function stops a periodic measurement.
RM_HS400X_Read ()	This function reads ADC data from HS4000x sensor.
RM_HS400X_DataCalculate ()	This function calculates humidity [%RH] and temperature [Celsius] from ADC data.
rm_hs400x_callback ()	This function is callback function for HS400x control module.

1.3 Outline of FS2012 SIS Module

“Table 1-5 FS2012 SIS module API Functions” lists the FS2012 SIS module API functions.

Table 1-5 FS2012 SIS module API Functions

Function	Description
RM_FS2012_Open ()	This function opens and configures the FS2012 Middle module.
RM_FS2012_Close ()	This function disables specified FS2012 control block.
RM_FS2012_Read ()	This reads ADC data from FS2012.
RM_FS2012_DataCalculate ()	This function calculates flow value [SLPM or SCCM] from ADC data.
rm_FS2012_callback ()	This function is callback function for FS2012 SIS module.

1.4 Outline of FS3000 SIS Module

“Table 1-6 FS3000 control module API Functions” lists the FS3000 control module API functions.

Table 1-6 FS3000 control module API Functions

Function	Description
RM_FS3000_Open ()	This function opens and configures the FS3000 control module.
RM_FS3000_Close ()	This function disables specified FS3000 control block.
RM_FS3000_Read ()	This reads ADC data from FS3000.
RM_FS3000_DataCalculate ()	This function calculates air velocity value [m/sec] from ADC data.
rm_fs3000_callback ()	This function is callback function for FS3000 control module.

1.5 Outline of FS1015 SIS Module

“Table 1-7 FS1015 control module API Functions” lists the FS1015 control module API functions.

Table 1-7 FS1015 control module API Functions

Function	Description
RM_FS1015_Open ()	This function opens and configures the FS1015 control module.
RM_FS1015_Close ()	This function disables specified FS1015 control block.
RM_FS1015_Read ()	This reads ADC data from FS1015.
RM_FS1015_DataCalculate ()	This function calculates air velocity value [m/sec] from ADC data.
rm_fs1015_callback ()	This function is callback function for FS1015 control module.

1.6 Outline of ZMOD4XXX SIS Module

"Table 1-8 ZMOD4XXX SIS module API Functions" lists the ZMOD4XXX SIS module API functions.

Table 1-8 ZMOD4XXX SIS module API Functions

Function	Description
RM_ZMOD4XXX_Open ()	This function opens and configures the ZMOD4XXX SIS module.
RM_ZMOD4XXX_Close ()	This function disables specified ZMOD4XXX control block.
RM_ZMOD4XXX_MeasurementStart ()	This function starts a measurement.
RM_ZMOD4XXX_MeasurementStop ()	This function stops a measurement.
RM_ZMOD4XXX_StatusCheck ()	This function read status of ZMOD4410 or ZMOD4510 sensor.
RM_ZMOD4XXX_Read ()	This function reads ADC data from ZMOD4410 or ZMOD4510 sensor.
RM_ZMOD4XXX_ Iaq1stGenDataCalculate ()	This function calculates IAQ (Indoor Air Quality) 1st Gen. values from ADC data.
RM_ZMOD4XXX_ Iaq2ndGenDataCalculate ()	This function calculates IAQ (Indoor Air Quality) 2nd Gen. values from ADC data.
RM_ZMOD4XXX_OdorDataCalculate ()	This function calculates Odor values from ADC data.
RM_ZMOD4XXX_ SulfurOdorDataCalculate ()	This function calculates Sulfur Odor values from ADC data.
RM_ZMOD4XXX_ Oaq1stGenDataCalculate ()	This function calculates OAQ 1st Gen. values from ADC data.
RM_ZMOD4XXX_ Oaq2ndGenDataCalculate ()	This function calculates OAQ 2nd Gen. values from ADC data.
RM_ZMOD4XXX_ RaqDataCalculate ()	This function calculates RAQ values from ADC data.
RM_ZMOD4XXX_ TemperatureAndHumiditySet ()	This function sets temperature and humidity to ZMOD4410 or ZMOD4510 sensor.
rm_zmod4xxx_comms_i2c_callback ()	This function is i2c callback function for ZMOD4XXX SIS module.
rm_zmod4xxx_irq_callback()	This function is irq callback function for ZMOD4XXX SIS module.

1.7 Outline of OB1203 SIS module

“Table 1-9 OB1203 SIS module API Functions” lists the OB1203 SIS module API functions.

Table 1-9 OB1203 SIS module API Functions

Function	Description
RM_OB1203_Open ()	This function opens and configures the ZMOD4XXX SIS module.
RM_OB1203_Close ()	This function disables specified ZMOD4XXX control block.
RM_OB1203_MeasurementStart ()	This function starts a measurement.
RM_OB1203_MeasurementStop ()	This function stops a measurement.
RM_OB1203_LightRead ()	This function reads Light ADC data from OB1203 device.
RM_OB1203_LightDataCalculate ()	This function calculates light data from raw data.
RM_OB1203_ProxRead ()	This function reads Proximity ADC data from OB1203 device.
RM_OB1203_ProxDataCalculate ()	This function calculates proximity data from raw data.
RM_OB1203_PpgRead ()	This function reads PPG ADC data from OB1203 device.
RM_OB1203_PpgDataCalculate ()	This function calculates PPG data from raw data.
RM_OB1203_DeviceStatusGet ()	This function gets device status from OB1203 device.
RM_OB1203_DeviceInterruptCfgSet ()	This function sets device interrupt configurations.
RM_OB1203_GainSet ()	This function sets gain.
RM_OB1203_LedCurrentSet ()	This function sets led currents.
RM_OB1203_FifoinfoGet ()	This function gets FIFO information from OB1203 device.
rm_ob1203_comms_i2c_callback()	This function is callback function for OB1203 SIS module.

1.8 Outline of COMMS (I2C communication middleware) SIS Module

“Table 1-10 Senser communication middleware SIS module API Functions” lists the API functions.

Table 1-10 Senser communication middleware SIS module API Functions

Function	Description
RM_COMMS_I2C_Open ()	The function opens and configures the COMMS SIS module.
RM_COMMS_I2C_Close ()	This function disables specified COMMS SIS module.
RM_COMMS_I2C_Read ()	The function performs a read from I2C device.
RM_COMMS_I2C_Write ()	The function performs a write from the I2C device.
RM_COMMS_I2C_WriteRead ()	The function performs a write to, then a read from the I2C device.
rm_comms_i2c_callback ()	This function is callback function for COMMS SIS module called in I2C driver callback function.

1.9 How to combine sensor control modules and RL78 IIC Components

HS300x SIS module, HS400x SIS module, FS2012 SIS module, FS3000 SIS module, FS1015 SIS module, ZMOD4XXX SIS module, OB1203 SIS module and COMMS SIS module can control simultaneously multiple sensors on any channel of any I2C bus.

However, the sensors using same slave address cannot be connected to a same channel of I2C bus. Therefore, only one HS300x sensor or one HS400x sensor or one FS2012 sensor or one FS3000 sensor or FS1015 sensor or one ZMOD4410 sensor or one ZMOD4510 sensor or one OB1203 sensor can be connected to a same channel of the I2C bus.

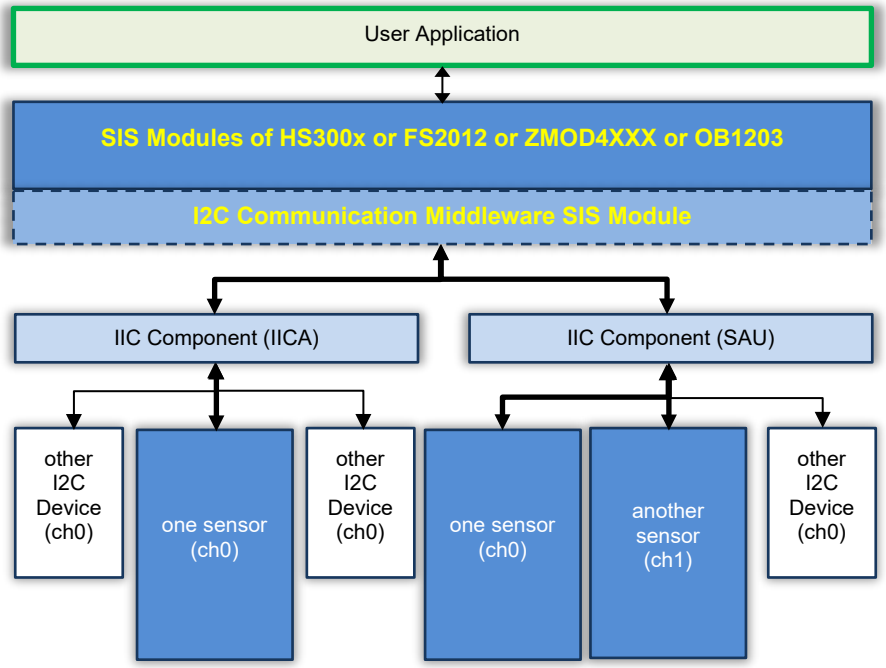
Figure 1-2 shows the relationship of HS300x SIS module, HS400x SIS module, FS2012 SIS module, FS3000 SIS module, FS1015 SIS module, ZMOD4XXX SIS module, OB1203 SIS module and COMMS SIS module, IIC Communication (Master mode) components and the I2C devices.

The I2C communication middleware SIS module is a driver interface function layer to absorb the difference between the HS300x/HS400x/FS2012/FS3000/FS1015/ZMOD4XXX/OB1203 SIS modules and RL78 IIC components.

The initialization processing of these SIS modules opens the module and sets control structure values according to configurations set by user. The initialization of I2C bus is done automatically in system initialize sequence (R_Systeminit), so there is no need to initialize it in the user application.

For the configuration related to this SIS module, refer to "Configuration Overview"

.



Since each I2C bus/channel is configured for each sensor, multiple sensors can be controlled simultaneously.

The IIC component can be controlled simultaneously.

However, since only a slave address is used for each sensor, only one sensor can be controlled on a same channel of the I2C bus.

Figure 1-2 Example of Combination of Sensor (HS300x or FS2012 or ZMOD4410 or ZMOD4450 or ZMOD4510 or OB1203) SIS Modules and IIC Components

1.10 Terminology/Abbreviation

Table 1-11 Terminology/Abbreviation Lists

Terminology/Abbreviation	Description
HS300x Sensor	Indicates HS300x Relative Humidity and Temperature Sensor.
HS400x Sensor	Indicates HS400x Relative Humidity and Temperature Sensor.
FS2012 Sensor	Indicates FS2012 High Performance Flow Sensor Module.
FS3000 Sensor	Indicates FS3000 Air Velocity Sensor Module.
FS1015 Sensor	Indicates FS1015 Air Velocity Sensor Module.
ZMOD4410 Sensor	Indicates Digital Gas Senser ZMOD4410 (Indoor Air Quality Platform)
ZMOD4450 Sensor	Indicates Digital Gas Senser ZMOD4450 (Refrigeration Air Quality Platform)
ZMOD4510 Sensor	Indicates Digital Gas Senser ZMOD4510 (Outdoor Air Quality Platform)
OB1203 Senser	Indicates Pulse Oximetry, Proximity, Light and Color Sensor OB1203.
HS300x SIS Module	Indicates HS300x Relative Humidity and Temperature Sensor control module.
HS400x SIS Module	Indicates HS400x Relative Humidity and Temperature Senser control module.
FS2012 SIS Module	Indicates High Performance Flow Sensor control module.
FS3000 SIS Module	Indicates Air Velocity Sensor control module.
FS1015 SIS Module	Indicates Air Velocity Sensor control module.
ZMOD4XXX SIS Module	Indicates ZMOD4410, ZMOD4450 and ZMOD 4510 Digital Gas Sensor control module.
OB1203 SIS Module	Indicates OB1203 Pulse Oximetry, Proximity, Light and Color Sensor control module.
I2C communication middleware (COMMS) SIS Module	Indicates communication driver interface function layer module.
IIC Communication (Master mode) Component	Indicates IIC Communication (Master mode) Component for Serial Interface IICA or/and Simplified I2C using Serial Array Unit (SAU).
ReST	Repeated Start Condition
SP	Stop Condition
ST	Start Condition

1.11 Operating Test Environment

This section describes for detailed the operating test environments of these SIS modules.

Table 1-12 Operation Test Environment

Item	Contents
Integrated Development Environment	Renesas Electronics e2 studio 2023-01
C Compiler	Renesas Electronics C/C++ compiler for RL78 family V.1.11.00 Compiler options: The integrated development environment default settings are used, with the following option added. -lang = c99
Endian Order	Little-endian
Component Version	IIC Communication (Master mode) Ver.1.1.1
Board Used	RL78/G23 Fast Prototyping Board (RTK7RLG230CSN00ABJ) Relative Humidity Sensor Pmod™ Board (US082-HS3001EVZ) Relative Humidity Sensor Pmod™ Board (QCIOT-HS4001POCZ) Gas Mass Flow Sensor Pmod™ Board (US082-FS2012EVZ) Gas Mass Flow Sensor Pmod™ Board (US082-FS3000EVZ) Gas Mass Flow Sensor Pmod™ Board (US082-FS1015EVZ) TVOC and Indoor Air Quality Sensor Pmod™ Board (US082-ZMOD4410EVZ) Refrigeration Air Quality Sensor Pmod™ Board (US082-ZMOD4450EVZ) Outdoor Air Quality Sensor Pmod™ Board (US082-ZMOD4510EVZ) Pulse Oximetry, Proximity, Light and Color Sensor Pmod™ Board (US082-OB1203EVZ)

1.12 Notes/Restrictions

- The operation by single master control has been confirmed. The operation by multi-master control is unconfirmed. When using it in multi-master control, evaluate it sufficiently.
- Operation has been confirmed only when the data endian is little endian.
- For the notes and restrictions of the IIC Communication (Master mode) component, refer to Smart Configurator User's Manual : RL78 API Reference.

2. API Information

2.1 Hardware Requirements

The MCU used must support one or both of the following functions.

- Serial Interface IICA
- Serial Array Unit (SAU)：Simplified I2C mode

2.2 Software Requirements

The SIS modules are dependent upon the following packages:

- Board Support Package Module (r_bsp) Ver.1.41 or higher
- IIC Communication (Master mode) Component Ver.1.11 or higher

2.3 Supported Toolchains

The SIS modules are tested and work with the following toolchain:

- Renesas RL78 Toolchain v.1.10.00 or higher

2.4 Usage of Interrupt Vector

The SIS modules do not use interrupts. However, the IIC Communication (Master mode) component to be used use interrupts. Refer to Smart Configurator User's Manual : RL78 API Reference for detail information.

2.5 Header Files

All API calls and their supporting interface definitions are located as following.

- HS300x SIS Module
  r_hs300x_if.h
  rm_hs300x_api.h
  rm_hs300x.h
- HS400x SIS Module
  r_hs400x_if.h
  rm_hs400x_api.h
  rm_hs400x.h
- FS2012 SIS Module
  r_fs2012_if.h
  rm_fsxxxx_api.h
  rm_fs2012.h
- FS3000 SIS Module
  r_fs3000_if.h
  rm_fsxxxx_api.h
  rm_fs3000.h
- FS1015 SIS Module
  r_fs1015_if.h
  rm_fsxxxx_api.h
  rm_fs1015.h
- ZMOD4XXX SIS Module
  r_zmod4xxx_if.h
  rm_zmod4xxx_api.h
  rm_zmod4xxx.h
- OB1203 SIS Module
  r_ob1203_if.h
  rm_ob1203_api.h
  rm_ob1203.h
- I2C communication middleware SIS Module
  r_comms_i2c_if.h
  rm_comms_api.h
  rm_comms_i2c.h

2.6 Integer Types

The projects for these SIS modules use ANSI C99. These types are defined in stdint.h.

2.7 Configuration Overview

The configuration options in these SIS modules are specified in
r_hs300x_rl_config.h and rm_hs300x_instance.c for HS300x SIS module,
r_hs400x_rl_config.h and rm_hs400x_instance.c for HS400x SIS module,
r_fs2012_rl_config.h and rm_fs2012_instance.c for FS2012 SIS module,
r_fs3000_rl_config.h and rm_fs3000_instance.c for FS3000 SIS module,
r_fs1015_rl_config.h and rm_fs1015_instance.c for FS1015 SIS module,
r_zmod4xxx_rl_config.h and rm_zmod4xxx_instance.c for ZMOD4XXX SIS Module,
r_ob1203_rl_config.h and rm_ob1203_instance.c for OB1203 SIS module,
r_comms_i2c_rl_config.h and rm_comms_i2c_rl_instance.c.

It is also necessary to set the IIC Communication (Master mode) component to be used. Refer to Smart Configurator User's Manual : RL78 API Reference for detail information.

2.7.1 HS300x SIS module configuration (r_hs3000_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration options	Description (Smart Configurator display)
RM_HS300X_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_HS300X_CFG_DEVICE_NUM_MAX	Specify maximum numbers of HS300x sensors. Selection: 1 - 2 Default: 1
RM_HS300X_CFG_DATA_BOTH_HUMIDITY_TEMPERATURE	Specify HS300x sensor data type. Selection: Humidity only Both humidity and temperature Default: Both humidity and temperature
RM_HS300X_CFG_PROGRAMMING_MODE	Specify programming mode on or off. Selection: Disabled (0) Enabled (1) Default: Disabled (0)
RM_HS300X_CFG_DEVICE0_COMMS_INSTANCE	Specify using communication line instance for device0. (Note 1) Selection: Comms0 - Comms4 Default: Comms0 (g_comms_i2c_device0)
RM_HS300X_CFG_DEVICE0_CALLBACK	Specify user callback function name. Selection: None (Need user to input.) Default: hs300x_user_callback0
RM_HS300X_CFG_DEVICE1_COMMS_INSTANCE	Specify using communication line instance for device1. (Note 1) Selection: Comms0 - Comms4 Default: Comms1 (g_comms_i2c_device1)
RM_HS300X_CFG_DEVICE1_CALLBACK	Specify user callback function name. Selection: None (Need user to input.) Default: hs300x_user_callback1

Note 1: Do not set same "Comms(x)" number for sensor device 0 and sensor device 1.

2.7.2 HS400x Control Module Configuration (r_hs4000_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration options	Description (Smart Configurator display)
RM_HS400X_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_HS400X_CFG_DEVICE_NUM_MAX	Specify maximum numbers of HS300x sensors. Selection: 1 - 2 Default: 1
RM_HS400X_CFG_MEASUREMENT_TYPE	Specify HS400x sensor measurement type. Selection: Hold Measurement No-Hold Measurement Periodic Measurement Default: No-Hold Measurement
RM_HS400X_CFG_DATA_BOTH_HUMIDITY_TEMPERATURE	Specify HS300x sensor data type. Selection: Humidity only Both humidity and temperature Default: Both humidity and temperature
RM_HS400X_CFG_DEVICE0_COMMS_INSTANCE	Specify using I2C communication device instance for device0. (Note 1) Selection: I2C Communication Device0 - I2C Communication Device 4 Default: I2C Communication Device0 (g_comms_i2c_device0)
RM_HS400X_CFG_DEVICE0_TEMPERATURE_RESOLUTION	Specify HS400x sensor temperature resolution for device0. Selection: 8-bit 10-bit 12-bit 14-bit Default: 14-bit
RM_HS400X_CFG_DEVICE0_HUMIDITY_RESOLUTION	Specify HS400x sensor humidity resolution for device0. Selection: 8-bit 10-bit 12-bit 14-bit Default: 14-bit
RM_HS400X_CFG_DEVICE0_PERIODIC_MEASUREMENT_FREQUENCY	Specify HS400x senor frequency for periodic measurement for device0. Selection: 0.4Hz 1Hz 2Hz Default: 1Hz
RM_HS400X_CFG_DEVICE0_CALLBACK	Specify user callback function name. Selection: None (Need user to input.) Default: hs400x_user_callback0
RM_HS400X_CFG_DEVICE1_COMMS_INSTANCE	Specify using I2C communication device instance for device1. (Note 1) Selection: I2C Communication Device0 - I2C Communication Device 4 Default: I2C Communication Device1 (g_comms_i2c_device1)
RM_HS400X_CFG_DEVICE1_TEMPERATURE_RESOLUTION	Specify HS400x sensor temperature resolution for device1. Selection: 8-bit 10-bit 12-bit 14-bit Default: 14-bit
RM_HS400X_CFG_DEVICE1_HUMIDITY_RESOLUTION	Specify HS400x sensor humidity resolution for device1. Selection: 8-bit 10-bit 12-bit

	14-bit Default: 14-bit
RM_HS400X_CFG_DEVICE1_PERIODIC_MEASUREMENT_FREQUENCY	Specify HS400x senor frequency for periodic measurement for device1. Selection: 0.4Hz 1Hz 2Hz Default: 1Hz
RM_HS400X_CFG_DEVICE1_CALLBACK	Specify user callback function name. Selection: None (Need user to input.) Default: hs400x_user_callback1

Note 1: Do not set same “Comms(x)” number for sensor device 0 and sensor device 1.

2.7.3 FS2012 SIS module configuration (r_fs2012_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration options	Description (Smart Configurator display)
RM_FS2012_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_FS2012_CFG_DEVICE_NUM_MAX	Specify maximum numbers of FS2012 sensors. Selection: 1 - 2 Default: 1
RM_FS2012_CFG_DEVICE_TYPE	Specify device type of FS2012 Sensor. (Note 2) Selection: FS2012-1020-NG FS2012-1100-NG Default: FS2012-1020-NG
RM_FS2012_CFG_DEVICE0_COMMS_INSTANCE	Specify using communication line instance for device0 (Note 1) Selection: Comms0 - Comms4 Default: Comms0 (g_comms_i2c_device0)
RM_FS2012_CFG_DEVICE0_CALLBACK	Specify user callback function name. Selection: None (Need user to input) Default: fs2012_user_callback0
RM_FS2012_CFG_DEVICE1_COMMS_INSTANCE	Specify using communication line instance for device1 (Note 1) Selection: Comms0 - Comms4 Default: Comms1 (g_comms_i2c_device1)
RM_FS2012_CFG_DEVICE1_CALLBACK	Specify user callback function name. Selection: None (Need user to input) Default: fs2012_user_callback1

Note 1: Do not set same “Comms(x)” number for sensor device 0 and sensor device 1. The “x” = 0-4.

Note 2: FS2012-1020-NG is 0 to 2 SLPM (Standard liter er minute) calibrated gas flow sensor mounted on a circuit board with a flow housing, FS2012-1100-NG is 0 to 10 SLPM (Standard liter er minute) calibrated gas flow sensor mounted on a circuit board with a flow housing. This SIS module only supports FS2012-1020-NG and FS2012-1100-NG currently.

2.7.4 FS3000 Control Module Configuration (r_fs3000_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration options	Description (Smart Configurator display)
RM_FS3000_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_FS3000_CFG_DEVICE_NUM_MAX	Specify maximum numbers of FS3000 sensors. Selection: 1 - 2 Default: 1
RM_FS3000_CFG_DEVICE_TYPE	Specify device type of FS3000 Sensor. (Note 2) Selection: FS3000-1005 Default: FS3000-1005
RM_FS3000_CFG_DEVICE0_COMMS_INSTANCE	Specify using I2C communication device instance for device0 (Note 1) Selection: I2C Communication Device0 - I2C Communication Device4 Default: I2C Communication Device0 (g_comms_i2c_device0)
RM_FS3000_CFG_DEVICE0_CALLBACK	Specify user callback function name. Selection: None (Need user to input) Default: fs3000_user_callback0
RM_FS3000_CFG_DEVICE1_COMMS_INSTANCE	Specify using I2C communication device instance for device1 (Note 1) Selection: I2C Communication Device0 - I2C Communication Device4 Default: I2C Communication Device1 (g_comms_i2c_device1)
RM_FS3000_CFG_DEVICE1_CALLBACK	Specify user callback function name. Selection: None (Need user to input) Default: fs3000_user_callback1

Note 1: Do not set same “Comms(x)” number for sensor device 0 and sensor device 1. The “x” = 0-4.

Note 2: FS3000-1005 is a 0 to 7.23 m/sec air velocity range device, FS3000-1015 is a 0 to 15 m/sec air velocity range device. Refer to FS3000 datasheet for detail information. This control module only supports FS3000-1005 currently.

2.7.5 FS1015 Control Module Configuration (r_fs1015_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration options	Description (Smart Configurator display)
RM_FS1015_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_FS1015_CFG_DEVICE_NUM_MAX	Specify maximum numbers of FS1015 sensors. Selection: 1 - 2 Default: 1
RM_FS1015_CFG_DEVICE_TYPE	Specify device type of FS1015 Sensor. (Note 2) Selection: FS1015-1005 Default: FS1015-1005
RM_FS10152_CFG_DEVICE0_COMMS_INSTANCE	Specify using I2C communication device instance for device0 (Note 1) Selection: I2C Communication Device0 - I2C Communication Device4 Default: I2C Communication Device0 (g_comms_i2c_device0)
RM_FS1015_CFG_DEVICE0_CALLBACK	Specify user callback function name. Selection: None (Need user to input) Default: fs1015_user_callback0
RM_FS1015_CFG_DEVICE1_COMMS_INSTANCE	Specify using I2C communication device instance for device1 (Note 1) Selection: I2C Communication Device0 - I2C Communication Device4 Default: I2C Communication Device1 (g_comms_i2c_device1)
RM_FS1015_CFG_DEVICE1_CALLBACK	Specify user callback function name. Selection: None (Need user to input) Default: fs1015_user_callback1

Note 1: Do not set same “Comms(x)” number for sensor device 0 and sensor device 1. The “x” = 0-4.

Note 2: FS1015-1005 is a 0 to 7.23 m/sec air velocity range device, FS1015-1015 is a 0 to 15 m/sec air velocity range device. Refer to FS1015 datasheet for detail information. This control module only supports FS1015-1005 currently.

2.7.6 ZMOD4xxx SIS module configuration (r_zmod4xxx_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

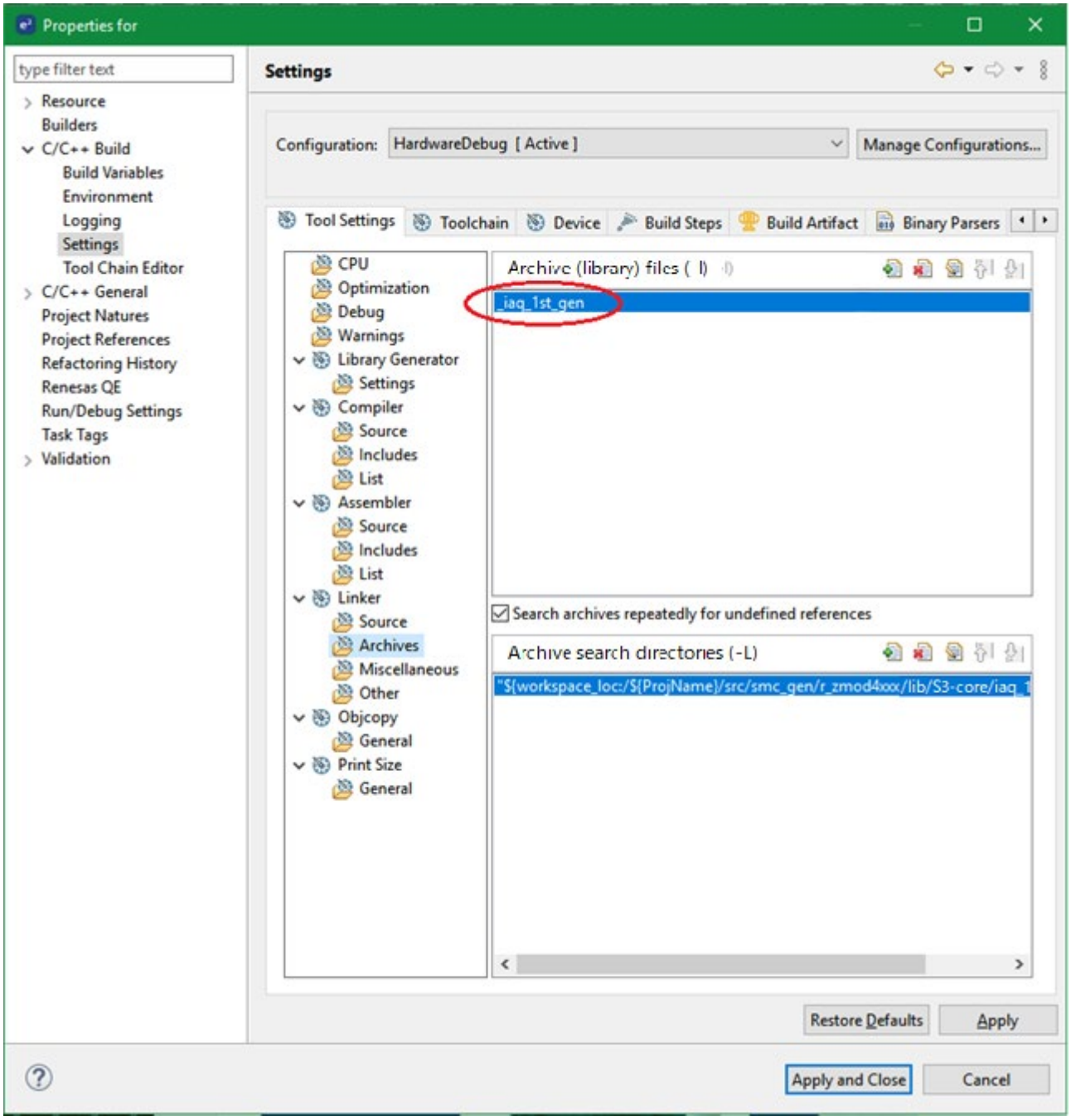
Configuration options	Description (Smart Configurator display)
RM_ZMOD4XXX_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_ZMOD4XXX_CFG_DEVICE_NUM_MAX	Specify maximum numbers of ZMOD4XXX sensors. Selection: 1-2 Default: 1
RM_ZMOD4XXX_CFG_DEVICE0_OPERATION_MODE	Specify operation mode of ZMOD4410, ZMOD4450 and ZMOD4510 sensors. (Note 2, 3) Selection: Not selected IAQ 1st Gen. (Continuous) IAQ 1st Gen. (Low Power) IAQ 2nd Gen. IAQ 2nd Gen. (Ultra Low Power) Odor Sulfur-based Odor OAQ 1st Gen. OAQ 2nd Gen. RAQ Default: Not selected
RM_ZMOD4XXX_CFG_DEVICE0_COMMS_INSTANCE	Specify used communication line number for ZMOD4xxx sensor device0. (Note 1) Selection: Comms0 - 4 Default: Comms0 (g_comms_i2c_device0)
RM_ZMOD4XXX_CFG_DEVICE0_COMMS_I2C_CALLBACK	Specify I2C callback function for ZMOD4xxx sensor device0. Selection: None Default: zmod4xxx_user_i2c_callback0 (Need user to input.)
RM_ZMOD4XXX_CFG_DEVICE0_IRQ_ENABLE	Enable INTC from ZMOD4xxx sensor device0. Selection: Enabled Disabled Default: Disabled
RM_ZMOD4XXX_CFG_DEVICE0_IRQ_CALLBACK	Specify INTC Callback function for ZMOD4xxx sensor device0. Selection: None Default: zmod4xxx_user_irq_callback0 (Need user to input.)
RM_ZMOD4XXX_CFG_DEVICE0_IRQ_NUMBER	Specify INTC number for ZMOD4xxx sensor device0 Selection: INTP0 – INTP15 Default: INTP0
RM_ZMOD4XXX_CFG_DEVICE1_OPERATION_MODE	Specify operation mode of ZMOD4xxx sensors. (Note 2) Selection: Not selected IAQ 1st Gen. (Continuous) IAQ 1st Gen. (Low Power) IAQ 2nd Gen. IAQ 2nd Gen. (Ultra Low Power) Odor Sulfur-based Odor OAQ 1st Gen. OAQ 2nd Gen. RAQ Default: Not selected

RM_ZMOD4XXX_CFG_DEVICE1_COMMS_INSTANCE	Specify used communication line number for ZMOD4xxx sensor device1. (Note 1) Selection: Comms0 - 4 Default: Comms0 (g_comms_i2c_device0)
RM_ZMOD4XXX_CFG_DEVICE1_COMMS_I2C_CALLBACK	Specify I2C callback function for ZMOD4xxx sensor device1. Selection: None Default: zmod4xxx_user_i2c_callback0 (Need user to input.)
RM_ZMOD4XXX_CFG_DEVICE1_IRQ_ENABLE	Enable INTC from ZMOD4xxx sensor device1. Selection: Enabled Disabled Default: Disabled
RM_ZMOD4XXX_CFG_DEVICE1_IRQ_CALLBACK	Specify INTC Callback function for ZMOD4xxx sensor device1. Selection: None Default: zmod4xxx_user_irq_callback1 (Need user to input.)
RM_ZMOD4XXX_CFG_DEVICE1_IRQ_NUMBER	Specify INTC number for ZMOD4xxx sensor device1 Selection: INTP0 – INTP15 Default: INTP0

Note 1: Be sure to specify a valid communication line number.

Note 2: When creating a project using “LLVM for Renesas RL78” toolchain with the “Make the double data type 64-bits wide” of “Additional CPU Option” is enabled, the library files for this option are needed to set by user itself. The library files are attached in sub folders under “..\r_zmod4xxx_rx\lib\” in ZMOD4XXX SIS module. “_64bits” is added in the name of these library files. Replace the library file name with “*_64bits” file name in following figure of “Settings” of “C/C++ Build” in properties of the project after generating the code.

Note 3: In the LLVM project, when changing operation mode, after code generation, the old library name ま may remains in the archive (library) files for linker settings. If the old library name remains, please manually remove it.


Properties for
type filter text
Resource
Builders
C/C++ Build
Build Variables
Environment
Logging
Settings
Tool Chain Editor
C/C++ General
Project Natures
Project References
Refactoring History
Renesas QE
Run/Debug Settings
Task Tags
Validation
Settings
Configuration: HardwareDebug [Active]
Manage Configurations...
Tool Settings
Toolchain
Device
Build Steps
Build Artifact
Binary Parsers
CPU
Optimization
Debug
Warnings
Library Generator
Settings
Compiler
Source
Includes
List
Assembler
Source
Includes
List
Linker
Source
Archives
Miscellaneous
Other
Objcopy
General
Print Size
General
Archive (library) files (l)
_iaq_1st_gen
Search archives repeatedly for undefined references
Archive search directories (-L)
"${workspace_loc:/${ProjName}/src/smc_gen/r_zmod4xxx/lib/S3-core/iaq_1
Restore Defaults
Apply
Apply and Close
Cancel

2.7.7 OB1203 SIS Module Configuration (r_ob1203_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration	Description (Smart Configurator display)
RM_OB1203_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
RM_OB1203_CFG_DEVICE_NUM_MAX	Set the numbers (max.) of OB1203 devices. Selection: 1-2 Default: 1
RM_OB1203_CFG_DEVICE(x)_SENSOR_MODE (“x” = 0-1)	Specify the operation mode of OB1203 device. Selection: Light Sensor mode Proximity Sensor mode Light and Proximity Sensor mode PPG Sensor mode Default: Light Sensor mode
RM_OB1203_CFG_DEVICE(x)_COMMS_INSTANCE (“x” = 0-1)	Specify the instance of the IIC bus. Selection: Comms0 Comms1 Comms2 Comms3 Comms4 Default: Comms0
RM_OB1203_CFG_DEVICE(x)_COMMS_I2C_CALLBACK (“x” = 0-1)	Specify I2C Callback function for OB1203 sensor device. Selection: None Default: ob1203_user_i2c_callback(x) (Need user to input.)
RM_OB1203_CFG_DEVICE(x)_IRQ_ENABLE (“x” = 0-1)	Enable INTC from OB1203 sensor device. Selection: Enabled Disabled Default: Disabled
RM_OB1203_CFG_DEVICE(x)_IRQ_NUMBER (“x” = 0-1)	Specify INTC number for OB1203 sensor device. Selection: INTP0 – INTP15 Default: INTP0
RM_OB1203_CFG_DEVICE(x)_IRQ_CALLBACK (“x” = 0-1)	Specify INTC Callback function for OB1203 sensor device. Selection: None Default: ob1203_user_irq_callback(x) (Need user to input.)
RM_OB1203_CFG_DEVICE(x)_DEVICE_INTERRUPT (“x” = 0-1)	Specify the enable device interrupt for OB1203 sensor device. Selection: Disabled Enable (Light mode) Enable (Proximity mode) Enable (PPG mode) Default: Disabled
RM_OB1203_CFG_DEVICE(x)_PPG_PROX_GAIN (“x” = 0-1)	Specify the gain of ADC output and noise of OB1203 sensor device Selection: 1, 1.5, 2, 3 Default: 1
RM_OB1203_CFG_DEVICE(x)_LED_ORDER (“x” = 0-1)	Specify the LED order of OB1203 sensor device Selection: IR LED first, Red LED second Red LED first, IR LED second Default: IR LED first, Red LED second
RM_OB1203_CFG_DEVICE(x)_LIGHT_SENSOR_MODE (“x” = 0-1)	Specify the operation mode for OB1203 sensor device Selection: LS mode CS mode Default: LS mode
RM_OB1203_CFG_DEVICE(x)_LIGHT_INTERRUPT_TYPE (“x” = 0-1)	Specify the Interrupt type of OB1203 sensor device Selection: Threshold Variation Default: Threshold

RM_OB1203_CFG_DEVICE(x)_LIGHT_INTERRUPT_SOURCE (“x” = 0-1)	Specify the interrupt source of OB1203 device Selection: Clear channel Green channel Red channel Blue channel Default: Clear channel
RM_OB1203_CFG_DEVICE(x)_LIGHT_INTERRUPT_SOURCE (“x” = 0-1)	Specify the number of similar consecutive interrupt event of OB1203 sensor device Selection: None Default: 0x02 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_LIGHT_SLEEP (“x” = 0-1)	Specify the sleep after interrupt of OB1203 sensor device Selection: Disabled Enabled Default: Disabled
RM_OB1203_CFG_DEVICE(x)_LIGHT_GAIN (“x” = 0-1)	Specify the gain for detection range of OB1203 sensor device Selection: 1, 3, 6 Default: 3
RM_OB1203_CFG_DEVICE(x)_LIGHT_UPPER_THRESHOLD (“x” = 0-1)	Specify the upper threshold of OB1203 sensor device Selection: None Default: 0x00CCC (Need user to input)
RM_OB1203_CFG_DEVICE(x)_LIGHT_LOWER_THRESHOLD (“x” = 0-1)	Specify the lower threshold of OB1203 sensor device Selection: None Default: 0x00000 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_LIGHT_VARIANCE_THRESHOLD (“x” = 0-1)	Specify the variance threshold of OB1203 sensor device Selection: +/- 8 counts +/- 16 counts +/- 32 counts +/- 64 counts +/- 128 counts +/- 256 counts +/- 512 counts +/- 1024 counts Default: +/- 128counts
RM_OB1203_CFG_DEVICE(x)_LIGHT_RESOLUTION_PERIOD (“x” = 0-1)	Specify the resolution and measurement period of OB1203 sensor device Selection: Resolution:13 bits. Measurement Period:25ms Resolution:13 bits. Measurement Period:50ms Resolution:13 bits. Measurement Period:100ms Resolution:13 bits. Measurement Period:200ms Resolution:13 bits. Measurement Period:500ms Resolution:13 bits. Measurement Period:1000ms Resolution:13 bits. Measurement Period:2000ms Resolution:16 bits. Measurement Period:25ms Resolution:16 bits. Measurement Period:50ms Resolution:16 bits. Measurement Period:100ms Resolution:16 bits. Measurement Period:200ms Resolution:16 bits. Measurement Period:500ms Resolution:16 bits. Measurement Period:1000ms Resolution:16 bits. Measurement Period:2000ms Resolution:17 bits. Measurement Period:50ms Resolution:17 bits. Measurement Period:100ms Resolution:17 bits. Measurement Period:200ms Resolution:17 bits. Measurement Period:500ms Resolution:17 bits. Measurement Period:1000ms Resolution:17 bits. Measurement Period:2000ms Resolution:18 bits. Measurement Period:100ms Resolution:18 bits. Measurement Period:200ms Resolution:18 bits. Measurement Period:500ms Resolution:18 bits. Measurement Period:1000ms Resolution:18 bits. Measurement Period:2000ms Resolution:19 bits. Measurement Period:200ms Resolution:19 bits. Measurement Period:500ms Resolution:19 bits. Measurement Period:1000ms Resolution:19 bits. Measurement Period:2000ms Resolution:20 bits. Measurement Period:500ms Resolution:20 bits. Measurement Period:1000ms Resolution:20 bits. Measurement Period:2000ms Default: Resolution:18 bits. Measurement Period:100ms

RM_OB1203_CFG_DEVICE(x)_PROX_INTERRUPT_TYPE (“x” = 0-1)	Specify the interrupt type of OB1203 sensor device Selection: Normal Logic Default: Normal
RM_OB1203_CFG_DEVICE(x)_PROX_INTERRUPT_PERSIST (“x” = 0-1)	Specify the number of similar consecutive interrupt events of OB1203 sensor device Selection: None Default: 0x02 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_PROX_SLEEP (“x” = 0-1)	Specify the sleep after interrupt of OB1203 sensor device Selection: Disabled Enabled Default Disabled
RM_OB1203_CFG_DEVICE(x)_PROX_LED_CURRENT (“x” = 0-1)	Specify the led current of OB1203 sensor device Selection: None Default:
RM_OB1203_CFG_DEVICE(x)_PROX_LED_PULSES (“x” = 0-1)	Specify the number of LED pulses of OB1203 sensor device Selection: 1 pulse 2 pulses 4 pulses 8 pulses 16 pulses 32 pulses Default: 8 pulses
RM_OB1203_CFG_DEVICE(x)_PROX_UPPER_THRESHOLD (“x” = 0-1)	Specify the upper threshold of OB1203 sensor device Selection: None Default: 0x00600 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_PROX_LOWER_THRESHOLD (“x” = 0-1)	Specify the lower threshold of OB1203 sensor device Selection: None Default: 0x00000 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_PROX_WIDTH_PERIOD (“x” = 0-1)	Specify the pulse width and measurement period of OB1203 sensor device Selection: Pulse width:26us. Measurement Period:3.125ms Pulse width:26us. Measurement Period:6.25ms Pulse width:26us. Measurement Period:12.5ms Pulse width:26us. Measurement Period:25ms Pulse width:26us. Measurement Period:50ms Pulse width:26us. Measurement Period:100ms Pulse width:26us. Measurement Period:200ms Pulse width:26us. Measurement Period:400ms Pulse width:42us. Measurement Period:3.125ms Pulse width:42us. Measurement Period:6.25ms Pulse width:42us. Measurement Period:12.5ms Pulse width:42us. Measurement Period:25ms Pulse width:42us. Measurement Period:50ms Pulse width:42us. Measurement Period:100ms Pulse width:42us. Measurement Period:200ms Pulse width:42us. Measurement Period:400ms Pulse width:71us. Measurement Period:3.125ms Pulse width:71us. Measurement Period:6.25ms Pulse width:71us. Measurement Period:12.5ms Pulse width:71us. Measurement Period:25ms Pulse width:71us. Measurement Period:50ms Pulse width:71us. Measurement Period:100ms Pulse width:71us. Measurement Period:200ms Pulse width:71us. Measurement Period:400ms Default: Pulse width:42us. Measurement Period:100ms
RM_OB1203_CFG_DEVICE(x)_PROX_MOVING_AVERAGE (“x” = 0-1)	Specify the moving average of OB1203 sensor device Selection: Disabled Enabled Default: Disabled
RM_OB1203_CFG_DEVICE(x)_PROX_HYSTERESIS (“x” = 0-1)	Specify the hysteresis of OB1203 sensor device Selection: None Default: 0x00 (Need user to intput)

RM_OB1203_CFG_DEVICE(x)_PPG_SENSOR_MODE ("x" = 0-1)	Specify the operation mode of OB1203 sensor device. Selection: PPG1 PPG2 Default: PPG2
RM_OB1203_CFG_DEVICE(x)_PPG_INTERRUPT_TYPE ("x" = 0-1)	Specify the interrupt type of OB1203 sensor device Selection: Data FIFO almost Full Default: Data
RM_OB1203_CFG_DEVICE(x)_PPG_IR_LED_CURRENT ("x" = 0-1)	Specify the IR LED current of OB1203 sensor device Selection: None Default: 0x366 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_PPG_RED_LED_CURRENT ("x" = 0-1)	Specify the Rd LED current of OB1203 sensor device Selection: None Default: 0x1B3 (Need user to input)
RM_OB1203_CFG_DEVICE(x)_PPG_POWER_SAVE_MODE ("x" = 0-1)	Specify the power save mode of OB1203 sensor device Selection: Disabled Enabled Default: Disabled
RM_OB1203_CFG_DEVICE(x)_PPG_IR_LED_ANA_CAN ("x" = 0-1)	Specify the IR LED analog cancellation of OB1203 sensor device Selection: Disabled Enabled Default: Disabled
RM_OB1203_CFG_DEVICE(x)_PPG_RED_LED_ANA_CAN ("x" = 0-1)	Specify the Red LED analog cancellation of OB1203 sensor device Selection: Disabled Enabled Default: Disabled
RM_OB1203_CFG_DEVICE(x)_PPG_NUM_AVERAGED_SAMPLE ("x" = 0-1)	Specify the number of averaged PPG samples of OB1203 sensor device Selection: 1 (No averaging) 2 consecutives samples are averaged 4 consecutives samples are averaged 8 consecutives samples are averaged 16 consecutives samples are averaged 32 consecutives samples are averaged Default: 8 consecutives samples are averaged
RM_OB1203_CFG_DEVICE(x)_PPG_WIDTH_PERIOD ("x" = 0-1)	Specify the pulse width and measurement period of OB1203 sensor device Selection: Pulse width:130us. Measurement Period:0.3125ms Pulse width:130us. Measurement Period:0.625ms Pulse width:130us. Measurement Period:1ms Pulse width:130us. Measurement Period:1.25ms Pulse width:130us. Measurement Period:2.50ms Pulse width:130us. Measurement Period:5ms Pulse width:130us. Measurement Period:10ms Pulse width:130us. Measurement Period:20ms Pulse width:247us. Measurement Period:0.625ms Pulse width:247us. Measurement Period:1ms Pulse width:247us. Measurement Period:1.25ms Pulse width:247us. Measurement Period:2.5ms Pulse width:247us. Measurement Period:5ms Pulse width:247us. Measurement Period:10ms Pulse width:247us. Measurement Period:20ms Pulse width:481us. Measurement Period:1ms Pulse width:481us. Measurement Period:1.25ms Pulse width:481us. Measurement Period:2.5ms Pulse width:481us. Measurement Period:5ms Pulse width:481us. Measurement Period:10ms Pulse width:481us. Measurement Period:20ms Pulse width:949us. Measurement Period:2.5ms Pulse width:949us. Measurement Period:5ms Pulse width:949us. Measurement Period:10ms Pulse width:949us. Measurement Period:20ms Default: Pulse width:130us. Measurement Period:1.25ms

RM_OB1203_CFG_DEVICE(x)_PPG_FIFO_ROLLOVER ("x" = 0-1)	Specify the FIFO rollover of OB1203 sensor device Selection: Disabled Enabled (50% offset of the full-scale values) Default: Disabled
RM_OB1203_CFG_DEVICE(x)_PPG_FIFO_EMPTY_NUM ("x" = 0-1)	Specify the FIFO almost full values of OB1203 sensor device Selection: None Default: 0xC

2.7.8 I2C communication middleware SIS Module Configuration (r_comms_i2c_rl_config.h)

The following explains the option names and setting values of this SIS module. The configuration settings shown in following table are set on Smart Configurator.

Configuration	Description (Smart Configurator display)
COMMS_I2C_CFG_PARAM_CHECKING_ENABLE	Specify whether to include code for API parameter checking. Selection: BSP Enabled Disabled Default: BSP
COMMS_I2C_CFG_DEVICE_NUM_MAX	Set the numbers (max.) of I2C devices. Selection: Unused, 1-5 Default: 1
COMMS_I2C_CFG_BUS(x)_DRIVER_TYPE ("x" = 0-4)	Specify the driver type of IIC bus. Selection: Not selected IICA SAU IIC Default: Not selected
COMMS_I2C_CFG_BUS(x)_DRIVER_CH ("x" = 0-4)	Specify the channel number of the IIC bus. Selection: None Default: 0 (Need user to input)
COMMS_I2C_CFG_DEVICE(x)_BUS_CH ("x" = 0-4)	Specify the bus configuration instance. Default: g_comms_i2c_bus0_extended_cfg. (Need user to input)
COMMS_I2C_CFG_DEVICE(x)_SLAVE_ADDR ("x" = 0-4)	Specify the slave address of the IIC bus. Selection: None Default: 0x00 (Need user to input)
COMMS_I2C_CFG_BUS(x)_CALLBACK ("x" = 0-4)	Specify Callback function of the IIC bus. Selection: None Default: comms_i2c_user_callback0 (Need user to input)

2.8 Code Size

Typical code sizes associated with this SIS module are listed below.

The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options described in”2.7Configuration Overview ”. The table lists reference values when the C compiler’s compile options are set to their default values, as described in “2.3 Supported Toolchains”.
The compiler option default values.

- optimization level: 2,
- optimization type: for size
- data endianness: little-endian

The code size varies depending on the C compiler version and compile options.
The values in the table below are confirmed under the following conditions.

- Component Version: IIC Communication (Master mode) Ver.1.10
- Compiler Version:
  Renesas Electronics C/C++ Compiler Package for RL78 Family V1.11.00
  (The option of “-lang = c99” is added to the default settings of the integrated development environment.)
- Configuration Options: Default settings

OS supporting	MCU	SIS Module	Category	Numbers	Condition
Non	RL78/G23	HS300x	ROM	544 bytes	Programming mode disabled
			RAM	22 bytes	
		HS400x	ROM	732 bytes	No-Hold Measurement is selected. The code size is different depended on the selected measurement type.
			RAM	40 bytes	
		FS2012	ROM	406 bytes	
			RAM	15 bytes	
		FS3000	ROM	398 bytes	
			RAM	14 bytes	
		FS1015	ROM	802 bytes	
			RAM	14 bytes	
		ZMOD4XXX	ROM	6,917 bytes	ZMOD4410 IAQ 2nd Gen. The code size is different depended on the selected operation mode.
			RAM	368 bytes	
		OB1203	ROM	2,569 byte	OB1203 Light mode and PPG mode. The code size is different depended on the selected operation mode.
			RAM	84 byte	
		COMMS	ROM	868 bytes	Maximum values when COMMS is used combined with each of above three SIS modules
			RAM	78 bytes	

2.9 Parameters

The API function arguments are shown below.

The structures of “configuration structure” and “control structure” are used as parameters type. These structures are described along with the API function prototype declaration.

The configuration structure is used for the initial configuration of HS300x SIS module, HS400x SIS module, FS2012 SIS module, FS3000 SIS module, FS1015 SIS module, ZMOD4XXX SIS module, OB1203 SIS module and COMMS SIS module during the module open API call. The configuration structure is used purely as an input into each module.

The control structure is used as a unique identifier for each module instance of HS300x SIS module, HS400x SIS module, FS2012 SIS module, FS3000 SIS module, FS1015 SIS module, ZMOD4XXX SIS module, OB1203 SIS module and COMMS SIS module. It contains memory required by the module. Elements in the control structure are owned by the associated module and must not be modified by the application. The user allocates storage for a control structure, often as a global variable, then sends a pointer to it into the module open API call for a module.

2.9.1 Configuration Structure and Control Structure of HS300x SIS Module

(1) Configuration Struct rm_hs300x_cfg_t

This structure is located in “rm_hs300x_api.h” file.

```
/** HS300X Configuration */
typedef struct st_rm_hs300x_cfg
{
    rm_comms_instance_t const * p_instance;  ///< Pointer to Communications Middleware instance.
    void const           * p_context;                ///< Pointer to the user-provided context.
    void const           * p_extend;             ///< Pointer to extended configuration by instance of interface.
    void (* p_callback)(rm_hs300x_callback_args_t * p_args);      ///< Pointer to callback function.
} rm_hs300x_cfg_t;
```

(2) Control Struct rm_hs300x_ctrl_t

This is HS300x SIS module control block and allocates an instance specific control block to pass into the HS300x API calls. This structure is implemented as “rm_hs300x_instance_ctrl_t” located in “rm_hs300x.h” file.

```
/** HS300x Control Block */
typedef struct rm_hs300x_instance_ctrl
{
    uint32_t                        open;             ///< Open flag
    rm_hs300x_cfg_t const     * p_cfg;              ///< Pointer to HS300X Configuration
    rm_comms_instance_t const * p_comms_i2c_instance;    ///< Pointer of I2C Communications
Middleware instance structure
    void const                    * p_context;        ///< Pointer to the user-provided context
    rm_hs300x_programmnig_mode_params_t programming_mode;   ///< Programming mode flag
    uint8_t buf[3];                                   ///< Buffer for I2c communications

    /* Pointer to callback and optional working memory */
    void (* p_callback)(rm_hs300x_callback_args_t * p_args);
} rm_hs300x_instance_ctrl_t;
```

2.9.2 Configuration Structure and Control Structure of HS400x SIS Module

(1) Configuration Struct rm_hs400x_cfg_t

This structure is located in “rm_hs400x_api.h” file.

```
/** HS400X Configuration */
typedef struct st_rm_hs400x_cfg
{
   rm_hs400x_temperature_resolution_t const temperature_resolution;     ///< Resolution for temperature
   rm_hs400x_humidity_resolusion_t const humidity_resolusion; ///< Resolution for humidity
   rm_hs400x_periodic_measurement_frequency_t const frequency; ///< Frequency for periodic measurement
   rm_comms_instance_t const * p_instance;  ///< Pointer to Communications Middleware instance.
   void const               * p_context;               ///< Pointer to the user-provided context.
   void const               * p_extend;          ///< Pointer to extended configuration by instance of interface.
   void (* p_callback)(rm_hs400x_callback_args_t * p_args);       ///< Pointer to callback function.
} rm_hs400x_cfg_t;
```

(2) Control Struct rm_hs400x_ctrl_t

This is HS400x control module control block and allocates an instance specific control block to pass into the HS400x API calls. This structure is implemented as “rm_hs400x_instance_ctrl_t” located in “rm_hs400x.h” file.

```
/** HS400x Control Block */
typedef struct rm_hs400x_instance_ctrl
{
   uint32_t                      open;     ///< Open flag
   rm_hs400x_cfg_t const    * p_cfg;  ///< Pointer to HS300X Configuration
   rm_comms_instance_t const * p_comms_i2c_instance;    ///< Pointer of I2C Communications
Middleware instance structure
   void const                    * p_context;               ///< Pointer to the user-provided context
   rm_hs400x_init_process_param_t init_process_params;  ///<For the initialization process.
   uint8_t resolution_register;      ///< Register for temperature and humidity measurement resolution
settings
   uint8_t periodic_measurement_register[2];  ///< Register for periodic measurement settings
   volatile bool periodic_measurement_stop;   ///< Flag for stop of periodic measurement
   volatile bool no_hold_measurement_read;   ///< Flag for data read of No-Hold measurement
   uint8_t write_buf[18];///< Buffer for data write

   /* Pointer to callback and optional working memory */
   void (* p_callback)(rm_hs400x_callback_args_t * p_args);
} rm_hs400x_instance_ctrl_t;
```

2.9.3 Configuration Structure and Control Structure of FS2012 SIS Module

(1) Configuration Struct rm_fsxxxx_cfg_t

This structure is located in “rm_fsxxxx_api.h” file.

```
/** FSXXXX Configuration */
typedef struct st_rm_fsxxxx_cfg
{
    rm_comms_instance_t const * p_instance;  ///< Pointer to Communications Middleware instance.
    Void const               * p_context;          ///< Pointer to the user-provided context.
    Void const               * p_extend;       ///< Pointer to extended configuration by instance of interface.
    Void (* p_callback)(rm_fsxxxx_callback_args_t * p_args);       ///< Pointer to callback function.
} rm_fsxxxx_cfg_t;
```

(2) Control Struct rm_fsxxxx_ctrl_t

This is FS2012 SIS module control block and allocates an instance specific control block to pass into the FS2012 API calls. This structure is implemented as “rm_fs2012_instance_ctrl_t” located in “rm_fs2012.h” file.

```
/** FS2012 Control Block */
typedef struct rm_fs2012_instance_ctrl
{
    uint32_t                    open;           ///< Open flag
    rm_fsxxxx_cfg_t const     * p_cfg;          ///< Pointer to FS2012 Configuration
    rm_comms_instance_t const * p_comms_i2c_instance;    ///< Pointer of I2C Communications
Middleware instance structure
    void const                * p_context;      ///< Pointer to the user-provided context

    /* Pointer to callback and optional working memory */
    void (* p_callback)(rm_fsxxxx_callback_args_t * p_args);
} rm_fs2012_instance_ctrl_t;
```

2.9.4 Configuration Structure and Control Structure of FS3000 SIS Module

(1) Configuration Struct rm_fsxxxx_cfg_t

This structure is located in “rm_fsxxxx_api.h” file.

```
/** FSXXXX Configuration */
typedef struct st_rm_fsxxxx_cfg
{
    rm_comms_instance_t const * p_instance;  ///< Pointer to Communications Middleware instance.
    void const               * p_context;            ///< Pointer to the user-provided context.
    void const               * p_extend;         ///< Pointer to extended configuration by instance of interface.
    void (* p_callback)(rm_fsxxxx_callback_args_t * p_args);       ///< Pointer to callback function.
} rm_fsxxxx_cfg_t;
```

(2) Control Struct rm_fsxxxx_ctrl_t

This is FS3000 control module control block and allocates an instance specific control block to pass into the FS3000 API calls. This structure is implemented as “rm_fs3000_instance_ctrl_t” located in “rm_fs3000.h” file.

```
/** FS3000 Control Block */
typedef struct rm_fs3000_instance_ctrl
{
    uint32_t                    open;         ///< Open flag
    rm_fsxxxx_cfg_t const    * p_cfg;        ///< Pointer to FS3000 Configuration
    rm_comms_instance_t const * p_comms_i2c_instance; ///< Pointer of I2C Communications Middleware
instance structure
    void const                  * p_context;  ///< Pointer to the user-provided context

    /* Pointer to callback and optional working memory */
    void (* p_callback)(rm_fsxxxx_callback_args_t * p_args);
} rm_fs3000_instance_ctrl_t;
```

2.9.5 Configuration Structure and Control Structure of FS1015 SIS Module

(1) Configuration Struct rm_fsxxxx_cfg_t

This structure is located in “rm_fsxxxx_api.h” file.

```
/** FSXXXX Configuration */
typedef struct st_rm_fsxxxx_cfg
{
    rm_comms_instance_t const * p_instance;  ///< Pointer to Communications Middleware instance.
    void const               * p_context;            ///< Pointer to the user-provided context.
    void const               * p_extend;         ///< Pointer to extended configuration by instance of interface.
    void (* p_callback)(rm_fsxxxx_callback_args_t * p_args);       ///< Pointer to callback function.
} rm_fsxxxx_cfg_t;
```

(2) Control Struct rm_fsxxxx_ctrl_t

This is FS1015 control module control block and allocates an instance specific control block to pass into the FS1015 API calls. This structure is implemented as “rm_fs1015_instance_ctrl_t” located in “rm_fs1015.h” file.

```
/** FS1015 Control Block */
typedef struct rm_fs2012_instance_ctrl
{
    uint32_t                    open;         ///< Open flag
    rm_fsxxxx_cfg_t const     * p_cfg;        ///< Pointer to FS1015 Configuration
    rm_comms_instance_t const * p_comms_i2c_instance; ///< Pointer of I2C Communications Middleware
instance structure
    void const                * p_context;  ///< Pointer to the user-provided context

    /* Pointer to callback and optional working memory */
    void (* p_callback)(rm_fsxxxx_callback_args_t * p_args);
} rm_fs1015_instance_ctrl_t;
```

2.9.6 Configuration Structure and Control Structure of ZMOD4xxx SIS Module

(1) Configuration Struct rm_zmod4xxx_cfg_t

This structure is located in “rm_zmod4xxx_api.h” file.

```
/** ZMOD4XXX configuration block */
typedef struct st_rm_zmod4xxx_cfg
{
   Rm_comms_instance_t const * p_comms_instance;        ///< Pointer to Communications Middleware
instance.   Void const              * p_irq_instance;              ///< Pointer to IRQ(INTP) instance.
   void const             * p_context;                          ///< Pointer to the user-provided context.
   void const             * p_extend;          ///< Pointer to extended configuration by instance of interface.
   void (* p_comms_callback)(rm_zmod4xxx_callback_args_t * p_args);///< I2C Communications callback
   void (* p_irq_callback)(rm_zmod4xxx_callback_args_t * p_args);       ///< IRQ callback
} rm_zmod4xxx_cfg_t;
```

(2) Control Struct rm_zmod4xxx_ctrl_t

This is ZMOD4XXX SIS module control block and allocates an instance specific control block to pass into the ZMOD4XXX API calls. This structure is implemented as “rm_zmod4xxx_instance_ctrl_t” located in “rm_zmod4xxx.h” file.

```
/** ZMOD4XXX control block */
typedef struct st_rm_zmod4xxx_instance_ctrl
{
   uint32_t open;                                           ///< Open flag
   uint8_t  buf[RM_ZMOD4XXX_MAX_I2C_BUF_SIZE];      ///< Buffer for I2C communications
   uint8_t  register_address;                               ///< Register address to access
   rm_zmod4xxx_status_params_t      status;                 ///< Status parameter
   volatile bool                    dev_err_check;          ///< Flag for checking device error
   volatile rm_zmod4xxx_event_t     event;                  ///< Callback event
   rm_zmod4xxx_init_process_params_t init_process_params;   ///< For the initialization process.
   rm_zmod4xxx_cfg_t const        * p_cfg;                  ///< Pointer of configuration block
   rm_comms_instance_t const      * p_comms_i2c_instance;     ///< Pointer of I2C Communications
Middleware instance structure
   rm_zmod4xxx_lib_extended_cfg_t  * p_zmod4xxx_lib;            ///< Pointer of ZMOD4XXX Lib
extended configuration

   void const * p_irq_instance;                             ///< Pointer to IRQ(INTP) instance.
   void const * p_context;                                  ///< Pointer to the user-provided context

   /* Pointer to callback and optional working memory */
   void (* p_comms_callback)(rm_zmod4xxx_callback_args_t * p_args);///< I2C Communications callback
   void (* p_irq_callback)(rm_zmod4xxx_callback_args_t * p_args);       ///< IRQ(INTP) callback
} rm_zmod4xxx_instance_ctrl_t;
```

2.9.7 Configuration Structure and Control Structure of OB1203 SIS Module

(1) Configuration Struct rm_ob1203_cfg_t

This structure is located in “rm_ob1203x_api.h” file.

```
/** OB1203 configuration block */
typedef struct st_rm_ob1203_cfg
{
   rm_comms_instance_t const * p_comms_instance;  ///< Pointer to Communications Middleware
instance.
   void const            * p_irq_instance;              ///< Pointer to IRQ instance.
   void const            * p_context;                   ///< Pointer to the user-provided context.
   void const            * p_extend;          ///< Pointer to extended configuration by instance of interface.
   void (* p_comms_callback)(rm_ob1203_callback_args_t * p_args);///< I2C Communications callback
   void (* p_irq_callback)(rm_ob1203_callback_args_t * p_args);    ///< IRQ callback
} rm_ob1203_cfg_t;
```

(2) Control Struct rm_ob1203_ctrl_t

This is OB1203 SIS module control block and allocates an instance specific control block to pass into the OB1203 API calls. This structure is implemented as “rm_ob1203_instance_ctrl_t” located in “rm_ob1203.h” file.

```
/** OB1203 control block */
typedef struct st_rm_ob1203_instance_ctrl
{
   uint32_t open;                                          ///< Open flag
   rm_ob1203_cfg_t const * p_cfg;                          ///< Pointer of configuration block
   uint8_t  buf[8];                                        ///< Buffer for I2C communications
   rm_ob1203_init_process_params_t init_process_params;///< For the initialization process.
   uint8_t  register_address;                              ///< Register address to access
   volatile rm_ob1203_device_status_t  *p_device_status;   ///< Pointer to device status
   volatile rm_ob1203_fifo_info_t *p_fifo_info;            ///< Pointer to FIFO information structure
   volatile bool fifo_reset;                               ///< Flag for FIFO reset for PPG mode
   volatile bool prox_gain_update;                         ///< Flag for gain update for Proximity mode
   volatile bool interrupt_bits_clear;                     ///< Flag for clearing interrupt bits
   rm_comms_instance_t const * p_comms_i2c_instance;   ///< Pointer of I2C Communications
Middleware instance structure
   rm_ob1203_mode_extended_cfg_t * p_mode;             ///< Pointer of OB1203 operation mode
extended configuration
   void const * p_irq_instance;                            ///< Pointer to IRQ(INTP) instance.
   void const * p_context;                                 ///< Pointer to the user-provided context

   /* Pointer to callback and optional working memory */
   void (* p_comms_callback)(rm_ob1203_callback_args_t * p_args);///< I2C Communications callback
   void (* p_irq_callback)(rm_ob1203_callback_args_t * p_args);       ///< IRQ(INTP) callback
} rm_ob1203_instance_ctrl_t;
```

2.9.8 Configuration Structure and Control Structure of COMMS SIS Module

(1) Configuration Struct rm_comms_cfg_t

This structure is located in “rm_comms_api.h” file.

```
/** Communications middleware configuration block */
typedef struct st_rm_comms_cfg
{
    uint32_t            semaphore_timeout;      ///< timeout for callback.
    void (* p_callback)(rm_comms_callback_args_t * p_args);     ///< Pointer to callback function, mostly
used if using non-blocking functionality.
    void const          * p_lower_level_cfg;    ///< Pointer to lower level driver configuration structure.
    void const          * p_extend;             ///< Pointer to extended configuration by instance of
interface.
    void const          * p_context;            ///< Pointer to the user-provided context
} rm_comms_cfg_t;
```

(2) Control Struct rm_comms_ctrl_t

This is COMMS SIS module control block and allocates an instance specific control block to pass into the COMMS API calls. This structure is implemented as “rm_comms_i2c_instance_ctrl_t” located in “rm_comms_i2c.h” file.

```
/** Communications middleware control structure. */
typedef struct st_rm_comms_i2c_instance_ctrl
{
    rm_comms_cfg_t const                * p_cfg;                ///< middleware configuration.
    rm_comms_i2c_bus_extended_cfg_t     * p_bus;                ///< Bus using this device;
    void                                * p_lower_level_cfg;    ///< Used to reconfigure I2C driver
    uint32_t                            open;                   ///< Open flag.
    uint32_t                            transfer_data_bytes;    ///< Size of transfer data.
    uint8_t                             * p_transfer_data;      ///< Pointer to transfer data buffer.

    /* Pointer to callback and optional working memory */
    void (* p_callback)(rm_comms_callback_args_t * p_args);

    void const                          * p_context;            ///< Pointer to the user-provided context
} rm_comms_i2c_instance_ctrl_t;
```

2.10 Return Values

The API function return values are shown below.

This enumeration is listed in fsp_common_api.h which is included in RL78BSP (Board Support Package Module) Ver.1.30 or higher.

```
typedef enum e_fsp_err
{
   FSP_SUCCESS = 0,

   FSP_ERR_ASSERTION               = 1,      ///< A critical assertion has failed
   FSP_ERR_INVALID_POINTER         = 2,      ///< Pointer points to invalid memory location
   FSP_ERR_INVALID_ARGUMENT        = 3,      ///< Invalid input parameter
   FSP_ERR_INVALID_CHANNEL         = 4,      ///< Selected channel does not exist
   FSP_ERR_INVALID_MODE            = 5,      ///< Unsupported or incorrect mode
   FSP_ERR_UNSUPPORTED             = 6,      ///< Selected mode not supported by this API
   FSP_ERR_NOT_OPEN                = 7,      ///< Requested channel is not configured or API not open
   FSP_ERR_IN_USE                  = 8,      ///< Channel/peripheral is running/busy
   FSP_ERR_OUT_OF_MEMORY           = 9,      ///< Allocate more memory in the driver's cfg.h
   FSP_ERR_HW_LOCKED               = 10,     ///< Hardware is locked
   FSP_ERR_IRQ_BSP_DISABLED        = 11,     ///< IRQ not enabled in BSP
   FSP_ERR_OVERFLOW                = 12,     ///< Hardware overflow
   FSP_ERR_UNDERFLOW               = 13,     ///< Hardware underflow
   FSP_ERR_ALREADY_OPEN            = 14,     ///< Requested channel is already open in a different
configuration
   FSP_ERR_APPROXIMATION           = 15,     ///< Could not set value to exact result
   FSP_ERR_CLAMPED                 = 16,     ///< Value had to be limited for some reason
   FSP_ERR_INVALID_RATE            = 17,     ///< Selected rate could not be met
   FSP_ERR_ABORTED                 = 18,     ///< An operation was aborted
   FSP_ERR_NOT_ENABLED             = 19,     ///< Requested operation is not enabled
   FSP_ERR_TIMEOUT                 = 20,     ///< Timeout error
   FSP_ERR_INVALID_BLOCKS          = 21,     ///< Invalid number of blocks supplied
   FSP_ERR_INVALID_ADDRESS         = 22,     ///< Invalid address supplied
   FSP_ERR_INVALID_SIZE            = 23,     ///< Invalid size/length supplied for operation
   FSP_ERR_WRITE_FAILED            = 24,     ///< Write operation failed
   FSP_ERR_ERASE_FAILED            = 25,     ///< Erase operation failed
   FSP_ERR_INVALID_CALL            = 26,     ///< Invalid function call is made
   FSP_ERR_INVALID_HW_CONDITION    = 27,     ///< Detected hardware is in invalid condition
   FSP_ERR_INVALID_FACTORY_FLASH   = 28,     ///< Factory flash is not available on this MCU
   FSP_ERR_INVALID_STATE           = 30,     ///< API or command not valid in the current state
   FSP_ERR_NOT_ERASED              = 31,     ///< Erase verification failed
   FSP_ERR_SECTOR_RELEASE_FAILED = 32,       ///< Sector release failed
   FSP_ERR_NOT_INITIALIZED         = 33,     ///< Required initialization not complete
   FSP_ERR_NOT_FOUND               = 34,     ///< The requested item could not be found
   FSP_ERR_NO_CALLBACK_MEMORY    = 35,       ///< Non-secure callback memory not provided for non-
secure callback
   FSP_ERR_BUFFER_EMPTY            = 36,     ///< No data available in buffer

   /* Start of RTOS only error codes */
   FSP_ERR_INTERNAL                = 100,    ///< Internal error
   FSP_ERR_WAIT_ABORTED            = 101,     ///< Wait aborted

   /* Start of Sensor specific */
   FSP_ERR_SENSOR_INVALID_DATA,                    ///< Data is invalid.
   FSP_ERR_SENSOR_IN_STABILIZATION,                ///< Sensor is stabilizing.
   FSP_ERR_SENSOR_MEASUREMENT_NOT_FINISHED,        ///< Measurement is not finished.

   /* Start of COMMS specific */
   FSP_ERR_COMMS_BUS_NOT_OPEN,                     ///< Bus is not open.
} fsp_err_t;
```

2.11 Adding the SIS Module to Your Project

This module must be added to each project in which it is used. Renesas recommends using “Smart Configurator” described in (1) or (2). However, “Smart Configurator” only supports some RL78 devices.

(1) Adding the SIS module to your project using “Smart Configurator” in e² studio

By using the “Smart Configurator” in e2 studio, the SIS module is automatically added to your project. Refer to “RL78 Smart Configurator User’s Guide : e² studio(R20AN0579)” for details.

(2) Adding the SIS module to your project using “Smart Configurator” on CS+

By using the “Smart Configurator Standalone version” in CS+, the SIS module is automatically added to your project. Refer to “RL78 Smart Configurator User’s Guide : CS+ (R20AN0580)” for details.

3. HS300x API Functions

3.1 RM_HS300X_Open ()

This function opens and configures the HS300x SIS module. This function must be called before calling any other HS300x API functions.

Format

```
fsp_err_t        RM_HS300X_Open(
        rm_hs300x_ctrl_t * const p_ctrl,
        rm_hs300x_cfg_t const * const p_cfg
);
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.1(1) Configuration Struct rm_hs300x_cfg_t

Return Values

FSP_SUCCESS	HS300x successfully configured.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	Module is already open. This module can only be opened once.

Properties

Prototyped in rm_hs300x.h

Description

This function opens and configures the HS300x SIS module.

This function copies the contents in “p_cfg” structure to the member “p_ctrl->p_cfg” in “p_ctrl” structure.

This function does configurations by setting the members of “p_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets callback and context
- Sets open flag

This function calls open API of COMMS SIS module to open communication middleware after all above initializations are done.

Special Notes

None

3.2 RM_HS300X_Close ()

This function disables specified HS300x control block.

Format

fsp_err_t RM_HS300X_Close (rm_hs300x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_hs300x.h

Description

This function calls close API of COMMS SIS module to close communication middleware.

This function clears open flag after all above are done.

Special Notes

None

3.3 RM_HS300X_MeasurementStart ()

This function starts a measurement.

Format

fsp_err_t RM_HS300X_MeasurementStart (rm_hs300x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_hs300x.h

Description

This function sends the slave address to HS300x sensor and start a measurement.

The function should be called when start a measurement and when measurement data is stale data.

The write API of COMMS SIS module is called in this function to send the slave address to HS300x sensor.

Special Notes

None

3.4 RM_HS300X_Read()

This function reads ADC data from HS300x sensor.

Format

```
fsp_err_t RM_HS300X_Read (
        rm_hs300x_ctrl_t * const p_ctrl,
        rm_hs300x_raw_data_t * const p_raw_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from HS300x sensor.

```
/** HS300X raw data */
typedef struct st_rm_hs300x_raw_data
{
   uint8_t humidity[2];          ///< Upper 2 bits of 0th element are data status
   uint8_t temperature[2];       ///< Lower 2 bits of 1st element are mask
} rm_hs300x_raw_data_t;
```

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options are invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_hs300x.h

Description

This function reads ADC data from HS300x sensor.

The read API of COMMS SIS module is called in this function.

The ADC data read from HS300x sensor is stored in “p_raw_data” structure. The read data length is defined according to GUI configuration setting as 4 bytes (both humidity and temperature) or 2 bytes (humidity only).

Special Notes

None

3.5 RM_HS300X_DataCalculate ()

This function calculates humidity [%RH] and temperature [Celsius] from ADC data.

Format

```
fsp_err_t RM_HS300X_DataCalculate (
        rm_hs300x_ctrl_t * const     p_ctrl,
        rm_hs300x_raw_data_t * const p_raw_data,
        rm_hs300x_data_t * const     p_hs300x_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from HS300x sensor.

```
/** HS300X raw data */
typedef struct st_rm_hs300x_raw_data
{
   uint8_t humidity[2];          ///< Upper 2 bits of 0th element are data status
   uint8_t temperature[2];       ///< Lower 2 bits of 1st element are mask
} rm_hs300x_raw_data_t;
```

p_hs300x_data

Pointer to HS300x sensor measurement results data structure.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_INVALID_DATA	Data is invalid.

Properties

Prototyped in rm_hs300x.h

Description

This function calculates the relative humidity value [%RH] and temperature value in degrees Celsius [°C] from the ADC data stored in “p_raw_data” and stores the calculated results to “p_hs300x_data” structure.

The status of raw data is shown in the upper 2 bits of p_raw_data-> humidity[0]. The raw data is invalid (e.g., stale data) if the status bits do not equal “0b00”. This function checks the status calculating. This function will skip calculation if the raw data is invalid.

The calculation method is based on the following formula given in the HS300x Datasheet. The temperature [°C] range is -40 to +125.

$$Humidity\,[\%RH] = \left(\frac{Humidity\,[13:0]}{2^{14}-1} \right) * 100$$

$$Temperature\,[^{o}C] = \left(\frac{Temperature\,[15:2]}{2^{14}-1} \right) * 165 - 40$$

The “p_hs300x_data” structure is defined as following.

```
/** HS300X sensor data block */
typedef struct st_rm_hs300x_sensor_data
{
    int16_t integer_part;
    int16_t decimal_part;            ///< To two decimal places
} rm_hs300x_sensor_data_t;

/** HS300X data block */
typedef struct st_rm_hs300x_data
{
    rm_hs300x_sensor_data_t humidity;
    rm_hs300x_sensor_data_t temperature;
} rm_hs300x_data_t;
```

Therefore, user application needs to combine the integer_part and decimal_part to a float number for humidity and temperature usage.

Special Notes

None

3.6 RM_HS300X_ProgrammingModeEnter ()

This function sends commands to place the HS300x into programming mode.

Format

fsp_err_t RM_HS300X_ProgrammingModeEnter (rm_hs300x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

Return Values

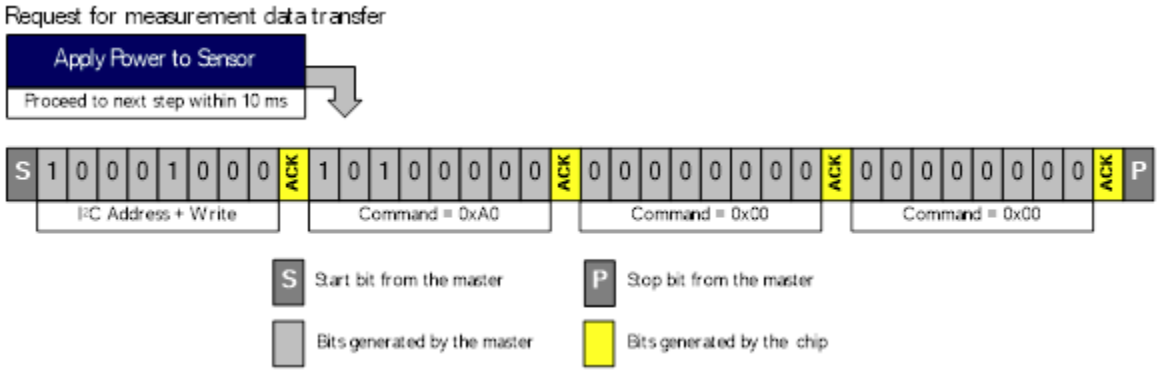
FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_UNSUPPORTED	Programming mode is not supported.

Properties

Prototyped in rm_hs300x.h

Description

This function sends a sequence of commands shown in below figure to place the HS300x into programming mode. This function must be called within 10ms after applying power to the sensor (HS300x).



The sequence of commands is that the master must send the I2C address and a “Write” bit followed by the command 0xA0|0x00|0x00. The detail information is described in “6.8 Accessing the Non-volatile Memory” of HS300x Datasheet Revision April 22, 2020.

Special Notes

This function must be called within 10ms after applying power to the HS300x sensor. This function performs for blocking.

3.7 RM_HS300X_ResolutionChange ()

This function sends commands to change the HS300x resolution.

Format

```
fsp_err_t RM_HS300X_ResolutionChange (
        rm_hs300x_ctrl_t * const p_ctrl,
        rm_hs300x_data_type_t const data_type,
        rm_hs300x_resolution_t const resolution
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

data_type

Data type of HS300x.

```
/** Data type of HS300X */
typedef enum e_rm_hs300x_data_type
{
   RM_HS300X_HUMIDITY_DATA = 0,
   RM_HS300X_TEMPERATURE_DATA,
} rm_hs300x_data_type_t;
```

resolution

Resolution of HS300x.

```
/** Resolution type of HS300X */
typedef enum e_rm_hs300x_resolution
{
   RM_HS300X_RESOLUTION_8BIT = 0,
   RM_HS300X_RESOLUTION_10BIT,
   RM_HS300X_RESOLUTION_12BIT,
   RM_HS300X_RESOLUTION_14BIT,
} rm_hs300x_resolution_t;
```

Return Values

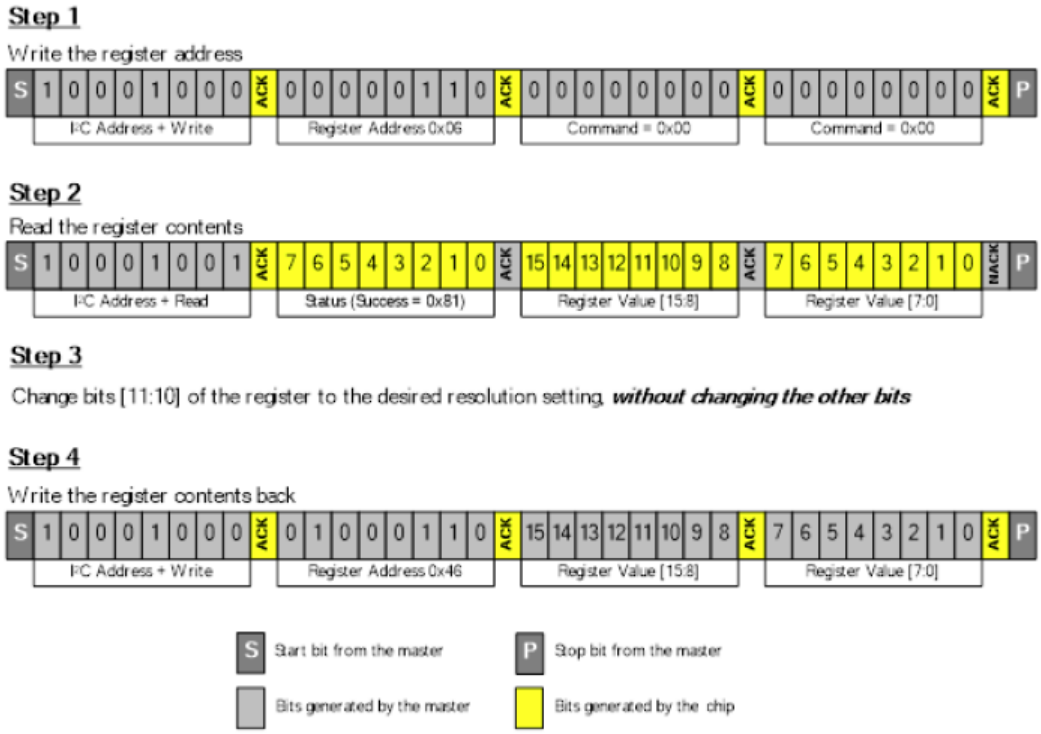
FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_INVALID_MODE	Module is not the programming mode.
FSP_ERR_ABORTED	Communication is aborted.
FSP_ERR_TIMEOUT	Communication is timeout.

Properties

Prototyped in rm_hs300x.h

Description

This function changes measurement resolutions of the HS300x to 8, 10, 12, or 14-bits by writing to the non-volatile memory. The procedure to change or set the resolution is shown in below figure.



The detail information is described in “6.9 Setting the Measurement Resolution” of HS300x Datasheet Revision April 22, 2020.

Special Notes

This function must be called after calling the RM_HS300X_ProgrammingModeEnter function. This function performs for blocking.

3.8 RM_HS300X_SensorIdGet ()

This function obtains the sensor ID of HS300x.

Format

```
fsp_err_t RM_HS300X_SensorIdGet (
        rm_hs300x_ctrl_t * const p_ctrl,
        uint32_t * const p_sensor_id
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

p_sensor_id

Data type of HS300x.

```
/** Data type of HS300X */
typedef enum e_rm_hs300x_data_type
{
   RM_HS300X_HUMIDITY_DATA = 0,
   RM_HS300X_TEMPERATURE_DATA,
} rm_hs300x_data_type_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_INVALID_MODE	Module is not the programming mode.
FSP_ERR_ABORTED	Communication is aborted.
FSP_ERR_TIMEOUT	Communication is timeout.

Properties

Prototyped in rm_hs300x.h

Description

This function writes ID registers address 0x1E and 0x1F then reads the ID numbers.

The detail information is described in “6.10Reading the HS300x ID Number” of HS300x Datasheet Revision April 22, 2020.

Special Notes

This function must be called after calling the RM_HS300X_ProgrammingModeEnter function. This function performs for blocking.

3.9 RM_HS300X_ProgrammingModeEixt ()

This function sends commands to exit the HS300x programming mode.

Format

fsp_err_t RM_HS300X_ProgrammingModeExit (rm_hs300x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.1(2) Control Struct rm_hs300x_ctrl_t.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_INVALID_MODE	Module is not entering the programming mode.
FSP_ERR_UNSUPPORTED	Programming mode is not supported.

Properties

Prototyped in rm_hs300x.h

Description

This function sends the I2C address and a Write bit, followed by the command: 0x80|0x00|0x00 to exit from programming mode, return to normal sensor operation and perform measurements.

The detail information is described in “6.8 Accessing the Non-volatile Memory” of HS300x Datasheet Revision April 22, 2020.

Special Notes

This function must be called within 10ms after applying power to the HS300x sensor. This function performs for blocking.

3.10 rm_hs300x_callback ()

This is callback function for HS300x SIS module.

Format

void rm_hs300x_callback (rm_comms_callback_args_t * p_args)

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
   void const     * p_context;
   rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_hs300x.h

Description

This callback function is called in COMMS SIS module callback function.

The member “event” in “rm_hs300x_callback_args_t” structure which is a member of “rm_hs300x_instance_ctrl_t” structure is set according to COMMS SIS module events status “p_args->event”.

The events of HS300x SIS module are

```
typedef enum e_rm_hs300x_event
{
   RM_HS300X_EVENT_SUCCESS = 0,
   RM_HS300X_EVENT_ERROR,
} rm_hs300x_event_t;
```

And the events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
   RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
   RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_hs300x_callback_args_t” structure is set to “RM_HS300X_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_HS300X_EVENT_ERROR”.

Special Notes

None.

3.11 Usage Example of HS300x SIS Module

```
#include "r_cg_macrodriver.h"
#include "r_hs300x_if.h"
#include "r_comms_i2c_if.h"
#include "Config_TAU0_1.h"

/* Sequence */
typedef enum e_demo_sequence
{
   DEMO_SEQUENCE_1 = (1),
   DEMO_SEQUENCE_2,
   DEMO_SEQUENCE_3,
   DEMO_SEQUENCE_4,
   DEMO_SEQUENCE_5,
   DEMO_SEQUENCE_6,
} demo_sequence_t;

/* Callback status */
typedef enum e_demo_callback_status
{
   DEMO_CALLBACK_STATUS_WAIT = (0),
   DEMO_CALLBACK_STATUS_SUCCESS,
   DEMO_CALLBACK_STATUS_REPEAT,
} demo_callback_status_t;

/* See Developer Assistance in the project */
void g_comms_i2c_bus0_quick_setup(void);
void g_hs300x_sensor0_quick_setup(void);
void timer_callback(void);

void       start_demo(void);
static void demo_err(void);

static volatile demo_callback_status_t  gs_demo_callback_status;
static volatile float                   gs_demo_humidity;
static volatile float                   gs_demo_temperature;
static volatile uint32_t                gs_ms_timer;

void start_demo(void)
{
   fsp_err_t err;
   rm_hs300x_raw_data_t raw_data;
   rm_hs300x_data_t hs300x_data;
   demo_sequence_t sequence = DEMO_SEQUENCE_1;

   R_Config_TAU0_1_Start();

   /* Open the Bus */
   g_comms_i2c_bus0_quick_setup();

   /* Open HS300X */
   g_hs300x_sensor0_quick_setup();
```

```
while (1)
{
   switch(sequence)
   {
      case DEMO_SEQUENCE_1 :
      {
         /* Clear status */
         gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;

         /* Start the measurement */
         err = g_hs300x_sensor0.p_api->measurementStart(g_hs300x_sensor0.p_ctrl);
         if (FSP_SUCCESS == err)
         {
            sequence = DEMO_SEQUENCE_2;
         }
         else
         {
            demo_err();
         }
      }
      break;

      case DEMO_SEQUENCE_2 :
      {
         switch(gs_demo_callback_status)
         {
            case DEMO_CALLBACK_STATUS_WAIT :
               break;
            case DEMO_CALLBACK_STATUS_SUCCESS :
               sequence = DEMO_SEQUENCE_3;
               break;
            case DEMO_CALLBACK_STATUS_REPEAT :
               sequence = DEMO_SEQUENCE_1;
               break;
            default :
               demo_err();
               break;
         }
      }
      break;

      case DEMO_SEQUENCE_3 :
      {
         /* Wait 4 seconds. See table 4 on the page 6 of the datasheet. */
         gs_ms_timer = 4000;
         while (0 < gs_ms_timer);
         sequence = DEMO_SEQUENCE_4;
      }
      break;

      case DEMO_SEQUENCE_4 :
      {
         /* Clear status */
         gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;
```

```
    /* Read data */
    err = g_hs300x_sensor0.p_api->read(g_hs300x_sensor0.p_ctrl, &raw_data);
    if (FSP_SUCCESS == err)
    {
        sequence = DEMO_SEQUENCE_5;
    }
    else
    {
        demo_err();
    }
}
break;

 case DEMO_SEQUENCE_5 :
{
    switch(gs_demo_callback_status)
    {
        case DEMO_CALLBACK_STATUS_WAIT :
            break;
        case DEMO_CALLBACK_STATUS_SUCCESS :
            sequence = DEMO_SEQUENCE_6;
            break;
        case DEMO_CALLBACK_STATUS_REPEAT :
            sequence = DEMO_SEQUENCE_4;
            break;
        default :
            demo_err();
            break;
    }
}
break;

case DEMO_SEQUENCE_6 :
{
    /* Calculate data */
    err = g_hs300x_sensor0.p_api->dataCalculate(g_hs300x_sensor0.p_ctrl, &raw_data, &hs300x_data);
    if (FSP_SUCCESS == err)
    {
        sequence = DEMO_SEQUENCE_1;

        /* Set data */
        gs_demo_humidity     = (float)hs300x_data.humidity.integer_part +
                               (float)hs300x_data.humidity.decimal_part * 0.01F;
        gs_demo_temperature  = (float)hs300x_data.temperature.integer_part +
                               (float)hs300x_data.temperature.decimal_part * 0.01F;
    }
    else if (FSP_ERR_SENSOR_INVALID_DATA == err)
    {
        sequence = DEMO_SEQUENCE_4;
    }
    else
    {
        demo_err();
```

```
            }
        }
        break;

        default :
            demo_err();
            break;
    }
  }
}

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
  /* bus has been opened by startup procees */
}

void hs300x_callback(rm_hs300x_callback_args_t * p_args)
{
  if (RM_HS300X_EVENT_SUCCESS == p_args->event)
  {
    gs_demo_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
  }
  else
  {
    gs_demo_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
  }
}

/* Quick setup for g_hs300x_sensor0. */
void g_hs300x_sensor0_quick_setup(void)
{
  fsp_err_t err;

  /* Open HS300X sensor instance, this must be done before calling any HS300X API */
  err = g_hs300x_sensor0.p_api->open(g_hs300x_sensor0.p_ctrl, g_hs300x_sensor0.p_cfg);
  if (FSP_SUCCESS != err)
  {
    demo_err();
  }
}

/* Timer count down */
void timer_callback(void)
{
  if(0 < gs_ms_timer)
  {
    gs_ms_timer--;
  }
}

static void demo_err(void)
{
  while(1)
```

```
        {
            // nothing
        }
}
```

4. HS400x API Functions

4.1 RM_HS400X_Open ()

This function opens and configures the HS400x SIS module. This function must be called before calling any other HS400x API functions.

Format

```
fsp_err_t       RM_HS400X_Open(
        rm_hs400x_ctrl_t * const p_ctrl,
        rm_hs400x_cfg_t const * const p_cfg
);
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.2(2) Configuration Struct rm_hs400x_cfg_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.2(1) Control Struct rm_hs400x_ctrl_t

Return Values

FSP_SUCCESS HS400x successfully configured.
FSP_ERR_ASSERTIONNull pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN Module is already open. This module can only be opened once.
FSP_ERR_TIMEOUTcommunication is timeout.FSP_ERR_ABORTEDcommunication is aborted.

Properties

Prototyped in rm_hs400x.h

Description

This function opens and configures the HS400x SISl module.
This function copies the contents in “p_cfg” structure to the member “p_ctrl->p_cfg” in “p_ctrl” structure.
This function does configurations by setting the members of “p_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets callback and context
- Sets open flag

This function calls open API of COMMS SIS module to open communication middleware after all above initializations are done.

Special Notes

None

4.2 RM_HS400X_Close ()

This function disables specified HS400x control block.

Format

fsp_err_t RM_HS400X_Close (rm_hs400x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.2(2) Configuration Struct rm_hs400x_cfg_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_hs400x.h

Description

This function calls close API of COMMS SIS module to close communication middleware.
This function clears open flag after all above are done.

Special Notes

None

4.3 RM_HS400X_MeasurementStart ()

This function starts a measurement.

Format

fsp_err_t RM_HS400X_MeasurementStart (rm_hs400x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.2(2) Configuration Struct rm_hs400x_cfg_t.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.
FSP_ERR_UNSUPPORTED	Hold measurement are unsupported.

Properties

Prototyped in rm_hs400x.h

Description

This function should be called when start a measurement.
Sends the command of measurement to HS400X and start a measurement.
This function supports No-Hold measurement and Periodic measurement only.
If Hold measurement is enabled, please call RM_HS400X_Read() without calling this function.
In Periodic measurement, if the periodic measurement has already run, RM_HS400X_EVENT_ERROR is received in callback because HS400x device replies with NACK.

Special Notes

None

4.4 RM_HS400X_MeasurementStop ()

This function stops a periodic measurement.

Format

fsp_err_t RM_HS400X_MeasurementStop (rm_hs400x_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.2(2) Configuration Struct rm_hs400x_cfg_t.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.
FSP_ERR_UNSUPPORTED	Hold and No-Hold measurement are unsupported.

Properties

Prototyped in rm_hs400x.h

Description

Stop a periodic measurement.
Sends the command of stopping periodic measurement to HS400X.
This function supports periodic measurement only.
If a periodic measurement is not running, RM_HS400X_EVENT_ERROR is received in callback because HS400x device replies with NACK.

Special Notes

None

4.5 RM_HS400X_Read()

This function reads ADC data from HS400x sensor.

Format

```
fsp_err_t RM_HS400X_Read (
        rm_hs400x_ctrl_t * const p_ctrl,
        rm_hs400x_raw_data_t * const p_raw_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.2(2) Configuration Struct rm_hs400x_cfg_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from HS300x sensor.

```
/** HS400X raw data */
typedef struct st_rm_hs400x_raw_data
{
   uint8_t humidity[2];          ///< Upper 2 bits of 0th element are mask
   uint8_t temperature[2];       ///< Upper 2 bits of 0th element are mask
} rm_hs400x_raw_data_t;
```

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options are invalid.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.

Properties

Prototyped in rm_hs400x.h

Description

This function reads ADC data from HS400x sensor.
The read API of COMMS SIS module is called in this function.
The ADC data read from HS400x sensor is stored in “p_raw_data” structure. The read data length is defined according to GUI configuration setting as 4 bytes (both humidity and temperature) or 2 bytes (temperature only).

Special Notes

None

4.6 RM_HS400X_DataCalculate ()

This function calculates humidity [%RH] and temperature [Celsius] from ADC data.

Format

```
fsp_err_t RM_HS400X_DataCalculate (
        rm_hs400x_ctrl_t * const     p_ctrl,
        rm_hs400x_raw_data_t * const p_raw_data,
        rm_hs400x_data_t * const     p_hs400x_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.2(2) Configuration Struct rm_hs400x_cfg_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from HS300x sensor.

```
/** HS400X raw data */
typedef struct st_rm_hs400x_raw_data
{
   uint8_t humidity[2];          ///< Upper 2 bits of 0th element are mask
   uint8_t temperature[2];       ///< Upper 2 bits of 0th element are mask
} rm_hs400x_raw_data_t;
```

p_hs400x_data

Pointer to HS400x sensor measurement results data structure.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_INVALID_DATA	Data is invalid.

Properties

Prototyped in rm_hs400x.h

Description

This function calculates the relative humidity value [%RH] and temperature value in degrees Celsius [°C] from the ADC data stored in “p_raw_data” and stores the calculated results to “p_hs400x_data” structure.
The calculation method is based on the following formula given in the HS400x Datasheet. The temperature [°C] range is -40 to +125.

$$Humidity\,[\%RH] = \left(\frac{Humidity\,[13:0]}{2^{14}-1} \right) * 100$$

$$Temperature\,[^{o}C] = \left(\frac{Temperature\,[15:2]}{2^{14}-1} \right) * 165 - 40$$

The “p_hs400x_data” structure is defined as following.

```
/** HS400X sensor data block */
typedef struct st_rm_hs400x_sensor_data
{
    int16_t integer_part;
    int16_t decimal_part;          ///< To two decimal places
} rm_hs400x_sensor_data_t;

/** HS400X data block */
typedef struct st_rm_hs400x_data
{
    rm_hs400x_sensor_data_t humidity;
    rm_hs400x_sensor_data_t temperature;
} rm_hs400x_data_t;
```

Therefore, user application needs to combine the integer_part and decimal_part to a float number for humidity and temperature usage.

Special Notes

None

4.7 rm_hs400x_callback ()

This is callback function for HS400x control module.

Format

```
void rm_hs400x_callback (rm_comms_callback_args_t * p_args)
```

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
    void const     * p_context;
    rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_hs400x.h

Description

This callback function is called in COMMS SIS module callback function.
The member “event” in “rm_hs400x_callback_args_t” structure which is a member of “rm_hs400x_instance_ctrl_t” structure is set according to COMMS SIS module events status “p_args->event”.
The events of HS400x SIS module are

```
typedef enum e_rm_hs400x_event
{
   RM_HS400X_EVENT_SUCCESS = 0,
   RM_HS400X_EVENT_MEASUREMENT_NOT_COMPLETE,
   RM_HS400X_EVENT_MEASUREMENT_NOT_RUNNING,
   RM_HS400X_EVENT_ALERT_TRIGGERED,
   RM_HS400X_EVENT_ERROR,
} rm_hs400x_event_t;
```

And the events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
    RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
    RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_hs400x_callback_args_t” structure is set to “RM_HS400X_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_HS400X_EVENT_MEASUREMENT_NOT_COMPLETE” and “RM_HS400X_EVENT_ERROR”.
“RM_HS400X_EVENT_MEASUREMENT_NOT_COMPLETE” is set when a measurement is not completed in No-Hold measurement.

Special Notes

None.

4.8 Usage Example of HS400x SIS Module

```
#include "r_cg_macrodriver.h"
#include "r_cg_serial.h"
#include "r_hs400x_if.h"
#include "r_comms_i2c_if.h"
#include "r_bsp_common.h"

#define DEMO_HOLD_MEASUREMENT      (1)
#define DEMO_NO_HOLD_MEASUREMENT   (2)
#define DEMO_PERIODIC_MEASUREMENT  (3)


/* Sequence */
typedef enum e_demo_sequence
{
   DEMO_SEQUENCE_1 = (1),
   DEMO_SEQUENCE_2,
   DEMO_SEQUENCE_3,
   DEMO_SEQUENCE_4,
   DEMO_SEQUENCE_5,
   DEMO_SEQUENCE_6,
} demo_sequence_t;

/* Callback status */
typedef enum e_demo_callback_status
{
   DEMO_CALLBACK_STATUS_WAIT = (0),
   DEMO_CALLBACK_STATUS_SUCCESS,
   DEMO_CALLBACK_STATUS_REPEAT,
} demo_callback_status_t;

/* See Developer Assistance in the project */
void g_comms_i2c_bus0_quick_setup(void);
void g_hs400x_sensor0_quick_setup(void);

void       start_demo(void);
static void demo_err(void);

static volatile demo_callback_status_t  gs_demo_callback_status;
static volatile rm_hs400x_data_t        gs_hs400x_data;

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
   /* bus has been opened by startup procees */
}

void hs400x_user_i2c_callback(rm_hs400x_callback_args_t * p_args)
{
   if (RM_HS400X_EVENT_SUCCESS == p_args->event)
   {
      gs_demo_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
   }
   else if (RM_HS400X_EVENT_MEASUREMENT_NOT_COMPLETE == p_args->event)
   {
      /* No-Hold measurement only. */
      gs_demo_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
   }
   else
   {
      gs_demo_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
   }
}

/* Quick setup for g_hs400x_sensor0. */
void g_hs400x_sensor0_quick_setup(void)
{
```

```
    fsp_err_t err;

    /* Open HS400X sensor instance, this must be done before calling any HS400X API */
    err = RM_HS400X_Open(g_hs400x_sensor0.p_ctrl, g_hs400x_sensor0.p_cfg);
    assert(FSP_SUCCESS == err);
}

void start_demo(void)
{
    fsp_err_t             err;
    rm_hs400x_raw_data_t    raw_data;
#if RM_HS400X_CFG_MEASUREMENT_TYPE == DEMO_PERIODIC_MEASUREMENT
    rm_hs400x_periodic_measurement_frequency_t frequency = g_hs400x_sensor0.p_cfg->frequency;
#endif

#if RM_HS400X_CFG_MEASUREMENT_TYPE == DEMO_HOLD_MEASUREMENT
    demo_sequence_t        sequence = DEMO_SEQUENCE_3;
#else
    demo_sequence_t        sequence = DEMO_SEQUENCE_1;
#endif

    /* Open the Bus */
    g_comms_i2c_bus0_quick_setup();

    /* Open HS400X */
    g_hs400x_sensor0_quick_setup();

    while (1)
    {
        switch(sequence)
        {
            case DEMO_SEQUENCE_1 :
            {
                /* Clear status */
                gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;

                /* Start the measurement */
                err = RM_HS400X_MeasurementStart(g_hs400x_sensor0.p_ctrl);
                if (FSP_SUCCESS == err)
                {
                    sequence = DEMO_SEQUENCE_2;
                }
                else
                {
                    demo_err();
                }
            }
            break;

            case DEMO_SEQUENCE_2 :
            {
                switch(gs_demo_callback_status)
                {
                    case DEMO_CALLBACK_STATUS_WAIT :
                        break;
                    case DEMO_CALLBACK_STATUS_SUCCESS :
                        sequence = DEMO_SEQUENCE_3;
                        break;
                    case DEMO_CALLBACK_STATUS_REPEAT :
                        sequence = DEMO_SEQUENCE_1;
                        break;
                    default :
                        demo_err();
                        break;
                }
            }
            break;
```

```
        case DEMO_SEQUENCE_3 :
        {
#if RM_HS400X_CFG_MEASUREMENT_TYPE == DEMO_PERIODIC_MEASUREMENT
                /* Wait until measurement is complete. */
                switch (frequency)
                {
                   case RM_HS400X_PERIODIC_MEASUREMENT_FREQUENCY_2HZ :
                   {
                      R_BSP_SoftwareDelay(500, BSP_DELAY_MILLISECS);
                   }
                   break;

                   case RM_HS400X_PERIODIC_MEASUREMENT_FREQUENCY_1HZ :
                   {
                      R_BSP_SoftwareDelay(1000, BSP_DELAY_MILLISECS);
                   }
                   break;

                   case RM_HS400X_PERIODIC_MEASUREMENT_FREQUENCY_0P4HZ :
                   {
                      R_BSP_SoftwareDelay(2500, BSP_DELAY_MILLISECS);
                   }
                   break;

                   default :
                      demo_err();
                      break;
                }
#endif
            /* Clear status */
            gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;

            /* Read data */
            err = RM_HS400X_Read(g_hs400x_sensor0.p_ctrl, &raw_data);
            if (FSP_SUCCESS == err)
            {
               sequence = DEMO_SEQUENCE_4;
            }
            else
            {
               demo_err();
            }
        }
        break;

        case DEMO_SEQUENCE_4 :
        {
           switch(gs_demo_callback_status)
           {
              case DEMO_CALLBACK_STATUS_WAIT :
                 break;
              case DEMO_CALLBACK_STATUS_SUCCESS :
                 sequence = DEMO_SEQUENCE_5;
                 break;
              case DEMO_CALLBACK_STATUS_REPEAT :
                 sequence = DEMO_SEQUENCE_3;
                 break;
              default :
                 demo_err();
                 break;
           }
        }
        break;

        case DEMO_SEQUENCE_5 :
        {
```

```
            /* Calculate data */
            err = RM_HS400X_DataCalculate(g_hs400x_sensor0.p_ctrl,
                                          &raw_data,
                                          (rm_hs400x_data_t *)&gs_hs400x_data);
            if (FSP_SUCCESS == err)
            {
                /* Sensor data is valid. Describe the process by referring to the calculated sensor data. */
#if RM_HS400X_CFG_MEASUREMENT_TYPE == DEMO_NO_HOLD_MEASUREMENT
                sequence = DEMO_SEQUENCE_1;
#else
                sequence = DEMO_SEQUENCE_3;
#endif
            }
            else if (FSP_ERR_SENSOR_INVALID_DATA == err)
            {
                /* Sensor data is invalid. */
                sequence = DEMO_SEQUENCE_3;
            }
            else
            {
                demo_err();
            }
        }
        break;

        default :
            demo_err();
            break;
    }
  }
}

static void demo_err(void)
{
  while(1)
  {
    // nothing
  }
}
```

5. FS2012 API Functions

5.1 RM_FS2012_Open ()

This function opens and configures the FS2012 SIS module. This function must be called before calling any other FS2012 API functions.

Format

```
fsp_err_t RM_FS2012_Open (
        rm_fsxxxx_ctrl_t * const p_ctrl,
        rm_fsxxxx_cfg_t const * const p_cfg
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.3(2)Control Struct rm_fsxxxx_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.3(1)Configuration Struct rm_fsxxxx_cfg_t.

Return Values

FSP_SUCCESS	FS2012 successfully configured.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	Module is already open. This module can only be opened once.

Properties

Prototyped in rm_fs2012.h

Description

This function opens and configures the FS2012 SIS module.

This function copies the contents in “p_cfg” structure to the member “p_ctrl->p_cfg” in “p_ctrl” structure.

This function does configurations by setting the members of “p_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets callback and context
- Sets open flag

This function calls open API of COMMS SIS module to open communication middleware after all above initializations are done.

Special Notes

None

5.2 RM_FS2012_Close()

This function disables specified FS2012 control block.

Format

fsp_err_t RM_FS2012_Close (rm_fsxxxx_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.3(2)Control Struct rm_fsxxxx_ctrl_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs2012.h

Description

This function calls close API of COMMS SIS module to close communication middleware.

This function clears open flag after all above are done.

Special Notes

None

5.3 RM_FS2012_Read()

This function reads ADC data from FS2012 sensor.

Format

```
fsp_err_t RM_FS2012_Read (
        rm_fsxxxx_ctrl_t * const p_ctrl,
        rm_fsxxxx_raw_data_t * const p_raw_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.3(2)Control Struct rm_fsxxxx_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from FS2012 sensor.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs2012.h

Description

This function reads ADC data from FS2012 sensor.

The read API of COMMS SIS module is called in this function.

The ADC data read from FS2012 sensor is stored in “p_raw_data” structure. The read data length is 2 bytes according to FS2012 datasheet.

The detail information is described in “7. I2C Sensor Interface” of FS2012 Series Datasheet Revision August 24, 2018.

Special Notes

None

5.4 RM_FS2012_DataCalculate ()

This function calculates flow value [SLPM or SCCM] from ADC data.

Format

```
fsp_err_t RM_FS2012_DataCalculate (
        rm_fsxxxx_ctrl_t * const     p_ctrl,
        rm_fsxxxx_raw_data_t * const p_raw_data,
        rm_fsxxxx_data_t * const     p_fs2012_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.3(2)Control Struct rm_fsxxxx_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from FS2012 sensor.

p_fs2012_data

Pointer to FS2012 sensor measurement results data structure.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs2012.h

Description

This function calculates the flow value [SLPM or SCCM] from the ADC data stored in “rm_fsxxxx_raw_data_t p_raw_data” and stores the calculated results to “rm_fsxxxx_data_t p_fs2012_data” structure.

The “rm_fsxxxx_raw_data_t” and “rm_fsxxxx_data_t” structures are defined as following.

```
/** FSXXXX raw data */
typedef struct st_rm_fsxxxx_raw_data
{
uint8_t adc_data[5];
} rm_fsxxxx_raw_data_t;

** FSXXXX data block */
typedef struct st_rm_fsxxxx_data
{
   rm_fsxxxx_sensor_data_t flow;
   uint32_t               count;
} rm_fsxxxx_data_t;

/** FSXXXX sensor data block */
typedef struct st_rm_fsxxxx_sensor_data
{
   int16_t integer_part;
   int16_t decimal_part;            ///< To two decimal places
} rm_fsxxxx_sensor_data_t;
```

This function calculates the flow value [SLPM or SCCM] from the count value according to the following.

The entire output of the FS2012 is 2 bytes. The flow rate for gas and liquid parts is calculated as follows:

Output Data

- Number of bytes to read out: 2
- First returned byte: MSB
- Second returned byte: LSB

Gas Part Configurations (FS2012-1020-NG and FS2012-1100-NG)

- Conversion to SLPM (Standard liter er minute)
- Flow in SLPM = [(MSB << 8) + LSB] / 1000

The detail information is described in “8. Calculating Flow Sensor Output” of FS2012 Series Datasheet Revision August 24, 2018.

Special Notes
None

5.5 rm_FS2012_callback ()

This is callback function for FS2012 SIS module.

Format

void rm_fs2012_callback (rm_comms_callback_args_t * p_args)

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
   void const     * p_context;
   rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_fs2012.h

Description

This callback function is called in COMMS SIS module callback function.
The member “event” in “rm_fsxxxx_callback_args_t” structure which is a member of “rm_fs2012_instance_ctrl_t ” structure is set according to COMMS SIS module events status “p_args->event”.
The events of FS2012 SIS module are

```
typedef enum e_rm_fsxxxx_event
{
   RM_FSXXXX_EVENT_SUCCESS = 0,
   RM_FSXXXX_EVENT_ERROR,
} rm_fsxxxx_event_t;
```

And the events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
   RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
   RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_fsxxxx_callback_args_t” structure is set to “RM_FSXXXX_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_FSXXXX_EVENT_ERROR”.

Special Notes

None

5.6 Usage Example of FS2012 SIS Module

```
#include "r_smc_entry.h"
#include "r_fs2012_if.h"
#include "r_comms_i2c_if.h"

/* Sequence */
typedef enum e_demo_sequence
{
   DEMO_SEQUENCE_1 = (1),
   DEMO_SEQUENCE_2,
   DEMO_SEQUENCE_3,
   DEMO_SEQUENCE_4,
   DEMO_SEQUENCE_5,
   DEMO_SEQUENCE_6,
} demo_sequence_t;

/* Callback status */
typedef enum e_demo_callback_status
{
   DEMO_CALLBACK_STATUS_WAIT = (0),
   DEMO_CALLBACK_STATUS_SUCCESS,
   DEMO_CALLBACK_STATUS_REPEAT,
} demo_callback_status_t;

/* See Developer Assistance in the project */
void g_comms_i2c_bus0_quick_setup(void);
void g_fs2012_sensor0_quick_setup(void);
void timer_callback(void);

void       start_demo(void);
static void demo_err(void);

static volatile demo_callback_status_t  gs_demo_callback_status;
static volatile float                   gs_demo_flow;
static volatile uint16_t                gs_ms_timer;

void start_demo(void)
{
   fsp_err_t err;
   rm_fsxxxx_raw_data_t raw_data;
   rm_fsxxxx_data_t fs2012_data;
   demo_sequence_t sequence = DEMO_SEQUENCE_1;

   /* Initializing Timer Peripheral */
   R_Config_TAU0_1_Start();

   /* Open the Bus */
   g_comms_i2c_bus0_quick_setup();

   /* Open FS2012 */
   g_fs2012_sensor0_quick_setup();

   while (1)
```

```
{
    switch (sequence)
    {
        case DEMO_SEQUENCE_1 :
        {
            /* Clear status */
            gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;

            /* Read FS2012 ADC Data */
            err = g_fs2012_sensor0.p_api->read(g_fs2012_sensor0.p_ctrl, &raw_data);
            if (FSP_SUCCESS == err)
            {
                sequence = DEMO_SEQUENCE_2;
            }
            else
            {
                demo_err();
            }
        }
        break;

        case DEMO_SEQUENCE_2 :
        {
            switch (gs_demo_callback_status)
            {
                case DEMO_CALLBACK_STATUS_WAIT :
                    break;
                case DEMO_CALLBACK_STATUS_SUCCESS :
                    sequence = DEMO_SEQUENCE_3;
                    break;
                case DEMO_CALLBACK_STATUS_REPEAT :
                    sequence = DEMO_SEQUENCE_1;
                    break;
                default :
                    demo_err();
                    break;
            }
        }
        break;

        case DEMO_SEQUENCE_3 :
        {
            /* Calculate data from ADC data */
            err = g_fs2012_sensor0.p_api->dataCalculate(g_fs2012_sensor0.p_ctrl, &raw_data, &fs2012_data);
            if (FSP_SUCCESS == err)
            {
                gs_demo_flow = (float)fs2012_data.flow.integer_part + (float)fs2012_data.flow.decimal_part * 0.01F;
                sequence = DEMO_SEQUENCE_4;
            }
            else if (FSP_ERR_SENSOR_INVALID_DATA == err)
            {
                sequence = DEMO_SEQUENCE_1;
            }
            else
```

```
            {
                demo_err();
            }
        }
        break;

        case DEMO_SEQUENCE_4 :
        {
            /* FS2012 sample rate. See table 4 on the page 5 of the datasheet. */
            /* Gas : 409.6ms, Liquid : 716.8ms */
            gs_ms_timer = 40960;
            while (0 < gs_ms_timer)
            {
            }
            sequence = DEMO_SEQUENCE_1;
        }
        break;

        default :
            demo_err();
            break;
    }
  }
}

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
  /* bus has been opened by startup procees */
}

void fs2012_callback(rm_fsxxxx_callback_args_t * p_args)
{
  if (RM_FSXXXX_EVENT_SUCCESS == p_args->event)
  {
    gs_demo_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
  }
  else
  {
    gs_demo_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
  }
}

/* Quick setup for g_fs2012_sensor0. */
void g_fs2012_sensor0_quick_setup(void)
{
  fsp_err_t err;

  /* Open FS2012 sensor instance, this must be done before calling any FSXXXX API */
  err = g_fs2012_sensor0.p_api->open(g_fs2012_sensor0.p_ctrl, g_fs2012_sensor0.p_cfg);
  if (FSP_SUCCESS != err)
  {
    demo_err();
  }
```

```
}

/* Timer count down */
void timer_callback(void)
{
    if(0 < gs_ms_timer)
    {
        gs_ms_timer--;
    }
}

static void demo_err(void)
{
    while(1)
    {
        // nothing
    }
}
```

6. FS3000 API Functions

6.1 RM_FS3000_Open ()

This function opens and configures the FS3000 SIS module. This function must be called before calling any other FS3000 API functions.

Format

```
fsp_err_t RM_FS3000_Open (
        rm_fsxxxx_ctrl_t * const p_ctrl,
        rm_fsxxxx_cfg_t const * const p_cfg
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.4(2) Control Struct rm_fs3000_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.4(1) Configuration Struct rm_fsxxxx_cfg_t.

Return Values

FSP_SUCCESS	FS3000 successfully configured.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	Module is already open. This module can only be opened once.

Properties

Prototyped in rm_fs3000.h

Description

This function opens and configures the FS3000 SIS module.
This function copies the contents in “p_cfg” structure to the member “p_ctrl->p_cfg” in “p_ctrl” structure.
This function does configurations by setting the members of “p_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets callback and context
- Sets open flag

This function calls open API of COMMS SIS module to open communication middleware after all above initializations are done.

Special Notes

None

6.2 RM_FS3000_Close()

This function disables specified FS3000 control block.

Format

fsp_err_t RM_FS3000_Close (rm_fsxxxx_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.4(2) Control Struct rm_fs3000_ctrl_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs3000.h

Description

This function calls close API of COMMS SIS module to close communication middleware.
This function clears open flag after all above are done.

Special Notes

None

6.3 RM_FS3000_Read()

This function reads ADC data from FS3000 sensor.

Format

```
fsp_err_t RM_FS3000_Read (
    rm_fsxxxx_ctrl_t * const p_ctrl,
    rm_fsxxxx_raw_data_t * const p_raw_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.4(2) Control Struct rm_fs3000_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from FS3000 sensor.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs3000.h

Description

This function reads ADC data from FS3000 sensor.
The read API of COMMS SIS module is called in this function.
The ADC data read from FS1015 sensor is stored in “p_raw_data” structure. The read data length is 5 bytes according to FS3000 datasheet.

The detail information is described in “5.2. Digital Output Measurements” of FS3000 Series Datasheet.

Special Notes
None

6.4 RM_FS3000_DataCalculate ()

This function calculates air velocity value [m/sec] from ADC data.

Format

```
fsp_err_t RM_FS3000_DataCalculate (
        rm_fsxxxx_ctrl_t * const     p_ctrl,
        rm_fsxxxx_raw_data_t * const p_raw_data,
        rm_fsxxxx_data_t * const     p_fs3000_data
)
```

Parameters

p_ctrl
Pointer to control structure.
The members of this structure are shown in 2.9.4(2) Control Struct rm_fs3000_ctrl_t.

p_raw_data
Pointer to raw data structure for storing the read ADC data from FS3000 sensor.

p_fs3000_data
Pointer to FS3000 sensor measurement results data structure.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties
Prototyped in rm_fs3000.h

Description
This function calculates the air velocity value [m/sec] from the ADC data stored in “rm_fsxxxx_raw_data_t p_raw_data” and stores the calculated results to “rm_fsxxxx_data_t p_fs3000_data” structure.
The “rm_fsxxxx_raw_data_t” and “rm_fsxxxx_data_t” structures are defined as following.

```
/** FSXXXX raw data */
typedef struct st_rm_fsxxxx_raw_data
{
uint8_t adc_data[5];
} rm_fsxxxx_raw_data_t;

** FSXXXX data block */
typedef struct st_rm_fsxxxx_data
{
   rm_fsxxxx_sensor_data_t flow;
   uint32_t                count;
} rm_fsxxxx_data_t;

/** FSXXXX sensor data block */
typedef struct st_rm_fsxxxx_sensor_data
{
   int16_t integer_part;
   int16_t decimal_part;          ///< To two decimal places
```

} rm_fsxxxx_sensor_data_t;

改ページ

This function calculates the air velocity value [m/sec] from the count value.
The relationships between Air velocity and Count value is as follows.

- FS3000-1005

Air Velocity (m/sec)	Output (Count)
0	409
1.07	915
2.01	1522
3.00	2066
3.97	2523
4.96	2908
5.98	3256
6.99	3572
7.23	3686

The detail information is described in “4. Typical Flow Graphs” of FS3000 Series Datasheet Revision May 31, 2022.

Special Notes
None

6.5 rm_fs3000_callback ()

This is callback function for FS3000 SIS module.

Format

```
void rm_fs3000_callback (rm_comms_callback_args_t * p_args)
```

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
    void const     * p_context;
    rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_fs3000.h

Description

This callback function is called in COMMS SIS module callback function.
The member “event” in “rm_fsxxxx_callback_args_t” structure which is a member of “rm_fs3000_instance_ctrl_t ” structure is set according to COMMS SIS module events status “p_args->event”.
The events of FS3000 SIS module are

```
typedef enum e_rm_fsxxxx_event
{
    RM_FSXXXX_EVENT_SUCCESS = 0,
    RM_FSXXXX_EVENT_ERROR,
} rm_fsxxxx_event_t;
```

And the events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
    RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
    RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_fsxxxx_callback_args_t” structure is set to “RM_FSXXXX_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_FSXXXX_EVENT_ERROR”.

Special Notes

None

6.6 Usage Example of FS3000 SIS Module

```
#include "r_smc_entry.h"
#include "r_fs3000_if.h"
#include "r_comms_i2c_if.h"
#if COMMS_I2C_CFG_DRIVER_I2C
#include "r_riic_rx_if.h"
#endif
#if COMMS_I2C_CFG_DRIVER_SCI_I2C
#include "r_sci_iic_rx_if.h"
#endif

typedef enum e_demo_sequence
{
    DEMO_SEQUENCE_1 = (1),
    DEMO_SEQUENCE_2,
    DEMO_SEQUENCE_3,
    DEMO_SEQUENCE_4,
    DEMO_SEQUENCE_5,
    DEMO_SEQUENCE_6,
} demo_sequence_t;

typedef enum e_demo_callback_status
{
    DEMO_CALLBACK_STATUS_WAIT = (0),
    DEMO_CALLBACK_STATUS_SUCCESS,
    DEMO_CALLBACK_STATUS_REPEAT,
} demo_callback_status_t;

void        g_comms_i2c_bus0_quick_setup(void);
void        g_fs3000_sensor0_quick_setup(void);
void        start_demo(void);
static void demo_err(void);

static volatile demo_callback_status_t  gs_demo_callback_status;
static volatile rm_fsxxxx_data_t        gs_fs3000_data;

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
    i2c_master_instance_t * p_driver_instance = (i2c_master_instance_t *)
g_comms_i2c_bus0_extended_cfg.p_driver_instance;

    /* Open i2c driver */
    if(COMMS_DRIVER_I2C == p_driver_instance->driver_type)
    {
#if COMMS_I2C_CFG_DRIVER_I2C
        riic_return_t ret;
        riic_info_t * p_i2c_info = (riic_info_t *)p_driver_instance->p_info;

        p_i2c_info->ch_no = (uint8_t) p_driver_instance->driver_channel;
        ret = R_RIIC_Open(p_i2c_info);
        if (RIIC_SUCCESS != ret)
        {
            demo_err();
        }
#endif
    }
    else if(COMMS_DRIVER_SCI_I2C == p_driver_instance->driver_type)
    {
#if COMMS_I2C_CFG_DRIVER_SCI_I2C
        sci_iic_return_t ret;
        sci_iic_info_t * p_i2c_info = (sci_iic_info_t *) p_driver_instance->p_info;

        p_i2c_info->ch_no = (uint8_t) p_driver_instance->driver_channel;
        ret = R_SCI_IIC_Open(p_i2c_info);
        if (SCI_IIC_SUCCESS != ret)
        {
```

```
            demo_err();
        }
#endif
    }
}

void fs3000_user_callback0(rm_fsxxxx_callback_args_t * p_args)
{
    if (RM_FSXXXX_EVENT_SUCCESS == p_args->event)
    {
        gs_demo_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
    }
    else
    {
        gs_demo_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
    }
}

/* Quick setup for g_fs3000_sensor0. */
void g_fs3000_sensor0_quick_setup(void)
{
    fsp_err_t err;

    /* Open FS3000 sensor instance, this must be done before calling any FSXXXX API */
    err = RM_FS3000_Open(g_fs3000_sensor0.p_ctrl, g_fs3000_sensor0.p_cfg);
    if (FSP_SUCCESS != err)
    {
        demo_err();
    }
}

void start_demo(void);
void start_demo(void)
{
    fsp_err_t err;
    rm_fsxxxx_raw_data_t raw_data;
    demo_sequence_t sequence = DEMO_SEQUENCE_1;

    /* Open the Bus */
    g_comms_i2c_bus0_quick_setup();

    /* Open FS3000 */
    g_fs3000_sensor0_quick_setup();

    while(1)
    {
        switch(sequence)
        {
            case DEMO_SEQUENCE_1 :
            {
                /* Clear status */
                gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;

                /* Read FS3000 ADC Data */
                err = RM_FS3000_Read(g_fs3000_sensor0.p_ctrl, &raw_data);
                if (FSP_SUCCESS == err)
                {
                    sequence = DEMO_SEQUENCE_2;
                }
                else
                {
                    demo_err();
                }
            }
            break;

            case DEMO_SEQUENCE_2 :
```

```
        {
            switch (gs_demo_callback_status)
            {
                case DEMO_CALLBACK_STATUS_WAIT :
                    break;
                case DEMO_CALLBACK_STATUS_SUCCESS :
                    sequence = DEMO_SEQUENCE_3;
                    break;
                case DEMO_CALLBACK_STATUS_REPEAT :
                    sequence = DEMO_SEQUENCE_1;
                    break;
                default :
                    demo_err();
                    break;
            }
        }
        break;

        case DEMO_SEQUENCE_3 :
        {
            /* Calculate data from ADC data */
            err = RM_FS3000_DataCalculate(g_fs3000_sensor0.p_ctrl,
                                                        &raw_data,
                                                        (rm_fsxxxx_data_t *)&gs_fs3000_data);
            if (FSP_SUCCESS == err)
            {
                sequence = DEMO_SEQUENCE_4;
                /* Sensor data is valid. Describe the process by referring to the calculated sensor data. */
            }
            else if (FSP_ERR_SENSOR_INVALID_DATA == err)
            {
                sequence = DEMO_SEQUENCE_1;
                /* Sensor data is invalid. Checksum error occurs. */
            }
            else
            {
                demo_err();
            }
        }
        break;

        case DEMO_SEQUENCE_4 :
        {
            /* Wait 125 milliseconds. See table 4 on the page 7 of the datasheet. */
            R_BSP_SoftwareDelay(125, BSP_DELAY_MILLISECS);
            sequence = DEMO_SEQUENCE_1;
        }
        break;

        default :
            demo_err();
            break;
    }
  }
}

static void demo_err(void)
{
  while(1)
  {
    // nothing
  }
}
```

7. FS1015 API Functions

7.1 RM_FS1015_Open ()

This function opens and configures the FS1015 SIS module. This function must be called before calling any other FS1015 API functions.

Format

```
fsp_err_t RM_FS1015_Open (
        rm_fsxxxx_ctrl_t * const p_ctrl,
        rm_fsxxxx_cfg_t const * const p_cfg
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.5(2) Control Struct rm_fs1015_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.5(1) Configuration Struct rm_fsxxxx_cfg_t.

Return Values

FSP_SUCCESS	FS1015 successfully configured.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	Module is already open. This module can only be opened once.

Properties

Prototyped in rm_fs1015.h

Description

This function opens and configures the FS1015 SIS module.
This function copies the contents in “p_cfg” structure to the member “p_ctrl->p_cfg” in “p_ctrl” structure.
This function does configurations by setting the members of “p_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets callback and context
- Sets open flag

This function calls open API of COMMS SIS module to open communication middleware after all above initializations are done.

Special Notes

None

7.2 RM_FS1015_Close()

This function disables specified FS1015 control block.

Format

fsp_err_t RM_FS1015_Close (rm_fsxxxx_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.5(2) Control Struct rm_fs1015_ctrl_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs1015.h

Description

This function calls close API of COMMS SIS module to close communication middleware.
This function clears open flag after all above are done.

Special Notes

None

7.3 RM_FS1015_Read()

This function reads ADC data from FS1015 sensor.

Format

```
fsp_err_t RM_FS1015_Read (
        rm_fsxxxx_ctrl_t * const p_ctrl,
        rm_fsxxxx_raw_data_t * const p_raw_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.5(2) Control Struct rm_fs1015_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from FS1015 sensor.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs1015.h

Description

This function reads ADC data from FS1015 sensor.
The read API of COMMS SIS module is called in this function.
The ADC data read from FS1015 sensor is stored in “p_raw_data” structure. The read data length is 3 bytes according to FS1015 datasheet.
The detail information is described in “Digital Output Measurements” of FS1015 Series Datasheet.

Special Notes

None

7.4 RM_FS1015_DataCalculate ()

This function calculates air velocity value [m/sec] from ADC data.

Format

```
fsp_err_t RM_FS1015_DataCalculate (
        rm_fsxxxx_ctrl_t * const     p_ctrl,
        rm_fsxxxx_raw_data_t * const p_raw_data,
        rm_fsxxxx_data_t * const     p_fs1015_data
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.5(2) Control Struct rm_fs1015_ctrl_t.

p_raw_data

Pointer to raw data structure for storing the read ADC data from FS1015 sensor.

p_fs1015_data

Pointer to FS1015 sensor measurement results data structure.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_fs1015.h

Description

This function calculates the air velocity value [m/sec] from the ADC data stored in “rm_fsxxxx_raw_data_t p_raw_data” and stores the calculated results to “rm_fsxxxx_data_t p_fs1015_data” structure.
The “rm_fsxxxx_raw_data_t” and “rm_fsxxxx_data_t” structures are defined as following.

```
/** FSXXXX raw data */
typedef struct st_rm_fsxxxx_raw_data
{
uint8_t adc_data[5];
} rm_fsxxxx_raw_data_t;

** FSXXXX data block */
typedef struct st_rm_fsxxxx_data
{
   rm_fsxxxx_sensor_data_t flow;
   uint32_t               count;
} rm_fsxxxx_data_t;

/** FSXXXX sensor data block */
typedef struct st_rm_fsxxxx_sensor_data
{
   int16_t integer_part;
   int16_t decimal_part;            ///< To two decimal places
} rm_fsxxxx_sensor_data_t;
```

This function calculates the air velocity value [m/sec] from the count value.
The relationships between Air velocity and Count value is as follows.

- FS1015-1005

Air Velocity (meter/sec)	Analog Output (Volt)	Digital Output (Counts)
0	0.5	409
1.07	1.118	915
2.01	1.858	1522
3	2.522	2066
3.97	3.08	2523
4.96	3.55	2908
5.98	3.075	3256
6.99	4.361	3572
7.23	4.5	3686

The detail information is described in “Flow Output Curve” of FS1015 Series Datasheet Revision February 10, 2020.

Special Notes
None

7.5 rm_fs1015_callback ()

This is callback function for FS1015 SIS module.

Format

```
void rm_fs1015_callback (rm_comms_callback_args_t * p_args)
```

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
    void const     * p_context;
    rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_fs1015.h

Description

This callback function is called in COMMS SIS module callback function.
The member “event” in “rm_fsxxxx_callback_args_t” structure which is a member of “rm_fs3000_instance_ctrl_t ” structure is set according to COMMS SIS module events status “p_args->event”.
The events of FS1015 SIS module are

```
typedef enum e_rm_fsxxxx_event
{
    RM_FSXXXX_EVENT_SUCCESS = 0,
    RM_FSXXXX_EVENT_ERROR,
} rm_fsxxxx_event_t;
```

And the events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
    RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
    RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_fsxxxx_callback_args_t” structure is set to “RM_FSXXXX_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_FSXXXX_EVENT_ERROR”.

Special Notes

None

7.6 Usage Example of FS1015 Contrl Module

```
#include "r_smc_entry.h"
#include "r_fs1015_if.h"
#include "r_comms_i2c_if.h"
#if COMMS_I2C_CFG_DRIVER_I2C
#include "r_riic_rx_if.h"
#endif
#if COMMS_I2C_CFG_DRIVER_SCI_I2C
#include "r_sci_iic_rx_if.h"
#endif

typedef enum e_demo_sequence
{
   DEMO_SEQUENCE_1 = (1),
   DEMO_SEQUENCE_2,
   DEMO_SEQUENCE_3,
   DEMO_SEQUENCE_4,
   DEMO_SEQUENCE_5,
   DEMO_SEQUENCE_6,
} demo_sequence_t;

typedef enum e_demo_callback_status
{
   DEMO_CALLBACK_STATUS_WAIT = (0),
   DEMO_CALLBACK_STATUS_SUCCESS,
   DEMO_CALLBACK_STATUS_REPEAT,
} demo_callback_status_t;

void       g_comms_i2c_bus0_quick_setup(void);
void       g_fs1015_sensor0_quick_setup(void);
void       start_demo(void);
static void demo_err(void);

static volatile demo_callback_status_t  gs_demo_callback_status;
static volatile rm_fsxxxx_data_t        gs_fs1015_data;

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
   i2c_master_instance_t * p_driver_instance = (i2c_master_instance_t *)
g_comms_i2c_bus0_extended_cfg.p_driver_instance;

   /* Open i2c driver */
   if(COMMS_DRIVER_I2C == p_driver_instance->driver_type)
   {
#if COMMS_I2C_CFG_DRIVER_I2C
      riic_return_t ret;
      riic_info_t * p_i2c_info = (riic_info_t *)p_driver_instance->p_info;

      p_i2c_info->ch_no = (uint8_t) p_driver_instance->driver_channel;
      ret = R_RIIC_Open(p_i2c_info);
      if (RIIC_SUCCESS != ret)
      {
            demo_err();
      }
#endif
   }
   else if(COMMS_DRIVER_SCI_I2C == p_driver_instance->driver_type)
   {
#if COMMS_I2C_CFG_DRIVER_SCI_I2C
      sci_iic_return_t ret;
      sci_iic_info_t * p_i2c_info = (sci_iic_info_t *) p_driver_instance->p_info;

      p_i2c_info->ch_no = (uint8_t) p_driver_instance->driver_channel;
      ret = R_SCI_IIC_Open(p_i2c_info);
      if (SCI_IIC_SUCCESS != ret)
      {
```

```
            demo_err();
        }
#endif
    }
}

void fs1015_user_callback0(rm_fsxxxx_callback_args_t * p_args)
{
    if (RM_FSXXXX_EVENT_SUCCESS == p_args->event)
    {
        gs_demo_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
    }
    else
    {
        gs_demo_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
    }
}

/* Quick setup for g_fs1015_sensor0. */
void g_fs1015_sensor0_quick_setup(void)
{
    fsp_err_t err;

    /* Open FS1015 sensor instance, this must be done before calling any FSXXXX API */
    err = RM_FS1015_Open(g_fs1015_sensor0.p_ctrl, g_fs1015_sensor0.p_cfg);
    if (FSP_SUCCESS != err)
    {
        demo_err();
    }
}

void start_demo(void);
void start_demo(void)
{
    fsp_err_t err;
    rm_fsxxxx_raw_data_t raw_data;
    demo_sequence_t sequence = DEMO_SEQUENCE_1;

    /* Open the Bus */
    g_comms_i2c_bus0_quick_setup();

    /* Open FS1015 */
    g_fs1015_sensor0_quick_setup();

    while(1)
    {
        switch(sequence)
        {
            case DEMO_SEQUENCE_1 :
            {
                /* Clear status */
                gs_demo_callback_status = DEMO_CALLBACK_STATUS_WAIT;

                /* Read FS1015 ADC Data */
                err = RM_FS1015_Read(g_fs1015_sensor0.p_ctrl, &raw_data);
                if (FSP_SUCCESS == err)
                {
                    sequence = DEMO_SEQUENCE_2;
                }
                else
                {
                    demo_err();
                }
            }
            break;

            case DEMO_SEQUENCE_2 :
```

```
        {
            switch (gs_demo_callback_status)
            {
                case DEMO_CALLBACK_STATUS_WAIT :
                    break;
                case DEMO_CALLBACK_STATUS_SUCCESS :
                    sequence = DEMO_SEQUENCE_3;
                    break;
                case DEMO_CALLBACK_STATUS_REPEAT :
                    sequence = DEMO_SEQUENCE_1;
                    break;
                default :
                    demo_err();
                    break;
            }
        }
        break;

        case DEMO_SEQUENCE_3 :
        {
            /* Calculate data from ADC data */
            err = RM_FS1015_DataCalculate(g_fs1015_sensor0.p_ctrl,
                                          &raw_data,
                                                (rm_fsxxxx_data_t *)&gs_fs1015_data);
            if (FSP_SUCCESS == err)
            {
                sequence = DEMO_SEQUENCE_4;
                /* Sensor data is valid. Describe the process by referring to the calculated sensor data. */
            }
            else if (FSP_ERR_SENSOR_INVALID_DATA == err)
            {
                sequence = DEMO_SEQUENCE_1;
                /* Sensor data is invalid. Checksum error occurs. */
            }
            else
            {
                demo_err();
            }
        }
        break;

        case DEMO_SEQUENCE_4 :
        {
            /* Wait 125 milliseconds. See table 4 on the page 3 of the datasheet. */
            R_BSP_SoftwareDelay(125, BSP_DELAY_MILLISECS);
            sequence = DEMO_SEQUENCE_1;
        }
        break;

        default :
            demo_err();
            break;
    }
  }
}

static void demo_err(void)
{
  while(1)
  {
    // nothing
  }
}
```

8. ZMOD4XXX API Functions

8.1 RM_ZMOD4XXX_Open ()

This function opens and configures the ZMOD4XXX SIS module. This function must be called before calling any other ZMOD4XXX API functions.

Format

```
fsp_err_t RM_ZMOD4XXX_Open (
        rm_zmod4xxx_ctrl_t * const p_api_ctrl,
        rm_zmod4xxx_cfg_t const * const p_cfg
);
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.6(1) Configuration Struct rm_zmod4xxx_cfg_t

Return Values

FSP_SUCCESS	ZMOD4xxx successfully configured.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	Module is already open. This module can only be opened once.
FSP_ERR_UNSUPPORTED	Unsupported product ID.
FSP_ERR_TIMEOUT	communication is timeout.
FSP_ERR_ABORTED	communication is aborted.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function opens and configures the ZMOD4XXX SIS module.

This function copies the contents in “p_cfg” structure to the member “p_api_ctrl->p_cfg” in “p_api_ctrl” structure. This function does configurations by setting the members of “p_api_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets ZMOD4XXX library specification
- Sets parameters of callback and context
- Sets open flag

This function calls following after all above initializations are done.

- Opens API of COMMS SIS module to open communication middlewareOpens IRQ open
- Initializes the sensor device (ZMOD4410 or ZMOD4510)
- Initializes the used sensor library

Special Notes

None

8.2 RM_ZMOD4XXX_Close ()

This function disables specified ZMOD4XXX control block. This function should be called when the sensor is closed.

Format

fsp_err_t RM_ZMOD4XXX_Close (rm_zmod4xxx_ctrl_t * const p_api_ctrl)

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calls closing API of COMMS SIS module to close communication middleware and IRQ close function.

This function clears open flag after all above are done.

Special Notes

None

8.3 RM_ZMOD4XXX_MeasurementStart ()

This function starts a measurement and should be called when a measurement is started.

Format

fsp_err_t RM_ZMOD4XXX_MeasurementStart (rm_zmod4xxx_ctrl_t * const p_api_ctrl)

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	communication is timeout.
FSP_ERR_ABORTED	communication is aborted.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function sends the measurement start to command register of ZMOD4410 or ZMOD4510 sensor and starts a measurement after the “event” in “p_api_ctrl” structure is cleared.

Special Notes

When starting the next measurement after previous measurement is finished, a delay time is needed. The delay time is depended on the selected operation mode. The detail information of delay time value can be found in “case DEMO_SEQUENCE_8 :” in “void start_demo(void)” function described in 8.16 Usage Example of ZMOD4XXX SIS Module.

8.4 RM_ZMOD4XXX_MeasurementStop ()

This function stops a measurement and should be called when a measurement is to be stopped.

Format

fsp_err_t RM_ZMOD4XXX_MeasurementStop (rm_zmod4xxx_ctrl_t * const p_api_ctrl)

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options are invalid.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	communication is timeout.
FSP_ERR_ABORTED	communication is aborted.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function sends the measurement stop to command register of ZMOD4410 or ZMOD4510 sensor and stops a measurement.

Special Notes

None

8.5 RM_ZMOD4XXX_StatusCheck ()

This function reads the status of sensor and should be called when polling is used.

Format

fsp_err_t RM_ZMOD4XXX_StatusCheck (rm_zmod4xxx_ctrl_t * const p_api_ctrl);

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	communication is timeout.
FSP_ERR_ABORTED	communication is aborted.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function reads measurement status of ZMOD4410 and ZMD4510 sensor from sensor register. This function returns either measurement success or 100ms timeout.

Special Notes

None

8.6 RM_ZMOD4XXX_Read ()

This read ADC data from ZMOD4410 or ZMOD4510 sensor. This function should be called when measurement finished.

Format

```
fsp_err_t RM_ZMOD4XXX_Read (
        rm_zmod4xxx_ctrl_t * const p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const p_raw_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure for storing ADC data read from sensor. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.
FSP_ERR_SENSOR_MEASUREMENT_NOT_FINISHED	Measurement is not finished.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function checks measurement status by either polling or using busy/interrupt pin. After the measurement status is confirmed as finished, this function reads ADC data and stores data to “p_raw_data” structure.

Special Notes

None

8.7 RM_ZMOD4XXX_Iaq1stGenDataCalculate ()

This function calculates IAQ 1st Gen. values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_Iaq1stGenDataCalculate (
        rm_zmod4xxx_ctrl_t * const              p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const          p_raw_data,
        rm_zmod4xxx_iaq_1st_data_t * const      p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing IAQ 1st Gen. calculation result.
This structure is declared as below.

```
/** ZMOD4XXX IAQ 1st gen data structure */
typedef struct st_rm_zmod4xxx_iaq_1st_data
{
   float rmox;          ///< MOx resistance.
   float rcda;          ///< CDA resistance.
   float iaq;           ///< IAQ index.
   float tvoc;          ///< TVOC concentration (mg/m^3).
   float etoh;          ///< EtOH concentration (ppm).
   float eco2;          ///< eCO2 concentration (ppm).
} rm_zmod4xxx_iaq_1st_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates IAQ results using ZMOD4410 IAQ 1st Gen. library and stores the results into the “rm_zmod4xxx_iaq_1st_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.8 RM_ZMOD4XXX_Iaq2ndGenDataCalculate ()

This function calculates IAQ 2nd Gen. values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_Iaq2ndGenDataCalculate (
        rm_zmod4xxx_ctrl_t * const              p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const          p_raw_data,
        rm_zmod4xxx_iaq_2nd_data_t * const      p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing IAQ 2nd Gen. calculation result.
This structure is declared as below.

```
/** ZMOD4XXX IAQ 2nd gen data structure */
typedef struct st_rm_zmod4xxx_iaq_2nd_data
{
   float rmox[13];          ///< MOx resistance.
   float log_rcda;          ///< log10 of CDA resistance.
   float iaq;               ///< IAQ index.
   float tvoc;              ///< TVOC concentration (mg/m^3).
   float etoh;              ///< EtOH concentration (ppm).
   float eco2;              ///< eCO2 concentration (ppm).
} rm_zmod4xxx_iaq_2nd_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates IAQ results using ZMOD4410 IAQ 2nd Gen. library and stores the results into the “rm_zmod4xxx_iaq_2nd_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.9 RM_ZMOD4XXX_OdorDataCalculate ()

This function calculates Odor values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_OdorDataCalculate (
        rm_zmod4xxx_ctrl_t * const          p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const      p_raw_data,
        rm_zmod4xxx_odor_data_t * const     p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing Odor calculation result.
This structure is declared as below.

```
/** ZMOD4XXX Odor structure */
typedef struct st_rm_zmod4xxx_odor_data
{
   bool  control_signal;      ///< Control signal input for odor lib.
   float odor;                ///< Concentration ratio for odor lib.
} rm_zmod4xxx_odor_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates Odor results from r_mox and odor parameters using ZMOD4410 Odor library and stores the results into the “rm_zmod4xxx_odor_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.10 RM_ZMOD4XXX_SulfurOdorDataCalculate ()

This function calculates Sulfur Odor values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_SulfurOdorDataCalculate (
        rm_zmod4xxx_ctrl_t * const                  p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const              p_raw_data,
        rm_zmod4xxx_sulfur_odor_data_t * const      p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing Sulfur Odor calculation result.
This structure is declared as below.

```
/** ZMOD4XXX Sulfur-Odor structure */
typedef struct st_rm_zmod4xxx_sulfur_odor_data
{
   float rmox[9];                          ///< MOx resistance.
   float intensity;                        ///< odor intensity rating ranges from 0.0 to 5.0 for sulfur lib
   rm_zmod4xxx_sulfur_odor_t odor;         ///< sulfur_odor classification for lib
} rm_zmod4xxx_sulfur_odor_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates Sulfur Odor results from ADC data using ZMOD4410 Sulfur Odor library and stores the results into the “rm_zmod4xxx_sulfur_odor_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.11 RM_ZMOD4XXX_Oaq1stGenDataCalculate ()

This function calculates OAQ 1st Gen. values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_Oaq1stGenDataCalculate (
        rm_zmod4xxx_ctrl_t * const              p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const          p_raw_data,
        rm_zmod4xxx_oaq_1st_data_t * const      p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing OAQ 1st Gen. calculation result.
This structure is declared as below.

```
/** ZMOD4XXX OAQ 1st gen data structure */
typedef struct st_rm_zmod4xxx_oaq_1st_data
{
   float rmox[15];          ///< MOx resistance
   float aiq;               ///< Air Quality
} rm_zmod4xxx_oaq_1st_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates AQI results from ADC data using ZMOD4510 OAQ 1st Gen. library and stores the results into the “rm_zmod4xxx_oaq_1st_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.12 RM_ZMOD4XXX_Oaq2ndGenDataCalculate ()

This function calculates OAQ 2nd Gen. values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_Oaq2ndGenDataCalculate (
        rm_zmod4xxx_ctrl_t * const              p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const          p_raw_data,
        rm_zmod4xxx_oaq_2nd_data_t * const  p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing OAQ 2nd Gen. calculation result.
This structure is declared as below.

```
/** ZMOD4XXX OAQ 2nd gen data structure */
typedef struct st_rm_zmod4xxx_oaq_2nd_data
{
   float    rmox[8];                ///< MOx resistance.
   float    ozone_concentration;    ///< The ozone concentration in part-per-billion
   uint16_t fast_aqi;               ///< 1-minute average of the Air Quality Index according to the EPA standard based on ozone
   uint16_t epa_aqi;                ///< The Air Quality Index according to the EPA standard based on ozone
} rm_zmod4xxx_oaq_2nd_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates OAQ results from ADC data using ZMOD4510 OAQ 2nd Gen. library and stores the results into the “rm_zmod4xxx_oaq_2nd_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.13 RM_ZMOD4XXX_RaqDataCalculate ()

This function calculates RAQ values from ADC data.

Format

```
fsp_err_t RM_ZMOD4XXX_RaqDataCalculate (
        rm_zmod4xxx_ctrl_t * const          p_api_ctrl,
        rm_zmod4xxx_raw_data_t * const      p_raw_data,
        rm_zmod4xxx_odor_data_t * const     p_zmod4xxx_data
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in. This structure is declared as below.

```
/** ZMOD4XXX raw data structure */
typedef struct st_rm_zmod4xxx_raw_data
{
   uint8_t adc_data[32];
} rm_zmod4xxx_raw_data_t;
```

p_zmod4xxx_data

Pointer to calculation result data structure storing RAQ calculation result.
This structure is declared as below.

```
/** ZMOD4XXX RAQ structure */
typedef struct st_rm_zmod4xxx_raq_data
{
   bool  control_signal;      ///< Control signal input for raq lib.
   float raq;                 ///< Concentration ratio for raq lib.
} rm_zmod4xxx_raq_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_SENSOR_IN_STABILIZATION	Module is stabilizing.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_zmod4xxx.h

Description

This function calculates RAQ results from r_mox and odor parameters using ZMOD4450 RAQ library and stores the results into the “rm_zmod4xxx_raq_data_t *p_zmod4xxx_data) structure.

Special Notes

None

8.14 RM_ZMOD4XXX_TemperatureAndHumiditySet ()

This function sets relative humidity (in %RH) and temperature (in °C) values for IAQ 2nd Gen ULP mode an OAQ 2nd Gen calculation.

Format

```
fsp_err_t RM_ZMOD4XXX_TemperatureAndHumiditySet (
        rm_zmod4xxx_ctrl_t * const      p_api_ctrl,
        float                           temperature,
        float                           humidity
)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.6(2) Control Struct rm_zmod4xxx_ctrl_t.

tmperature

Temperature value (in °C) set to “p_api_ctrl -> temperature”.

humidity

Humidity value (in %RH) set to “p_api_ctrl -> humidity”.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_zmod4xxx.h

Description

In OAQ 2nd Gen operation, an additional temperature and humidity measurement is recommended, and the algorithm has an auto-compensation included. This function sets environmental relative humidity (in %RH) and temperature (in °C) values for OAQ 2nd Gen calculation. This function should be called before RM_ZMOD4XXX_Oaq2ndGenDataCalculate () is called for calculation.

The detail information is described in “5.5 Environmental Temperature and Humidity” of ZMOD4510 Datasheet Revision June 30, 2021.

Special Notes

None

8.15 rm_zmod4xxx_comms_i2c_callback ()

This is callback function for ZMOD4XXX SIS module.

Format

void rm_zmod4xxx_comms_i2c_callback (rm_comms_callback_args_t * p_args)

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
  void const     * p_context;
  rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_zmod4xxx.h

Description

This callback function is called in COMMS SIS module callback function.
The member “event” in “rm_zmod4xxx_callback_args_t” structure which is a member of “rm_zmod4xxx_instance_ctrl_t” structure is set according to COMMS SIS module events status “p_args->event”.
The events of ZMO4XXX SIS module are

```
/** Event in the callback function */
typedef enum e_rm_zmod4xxx_event
{
  RM_ZMOD4XXX_EVENT_SUCCESS = 0,
  RM_ZMOD4XXX_EVENT_MEASUREMENT_COMPLETE,
  RM_ZMOD4XXX_EVENT_MEASUREMENT_NOT_COMPLETE,
  RM_ZMOD4XXX_EVENT_DEV_ERR_POWER_ON_RESET,  ///< Unexpected reset
  RM_ZMOD4XXX_EVENT_DEV_ERR_ACCESS_CONFLICT, ///< Getting invalid results while results readout
  RM_ZMOD4XXX_EVENT_ERROR,
} rm_zmod4xxx_event_t;
```

And the events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
  RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
  RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_zmod4xxx_callback_args_t” structure is set to “RM_ZMOD4XXX_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_ZMOD4XXX_EVENT_ERROR”. After above judgement, the “event” of “rm_zmod4xxx_callback_args_t” structure is changed to “RM_ZMOD4XXX_EVENT_MEASUREMENT_COMPLETE” or “RM_ZMOD4XXX_EVENT_MEASUREMENT_NOT_COMPLETE” or “RM_ZMOD4XXX_EVENT_DEV_ERR_ACCESS_CONFLICT” or “RM_ZMOD4XXX_EVENT_DEV_ERR_POWER_ON_RESET” after checking the “status” and “dev_err_check” of “rm_zmod4xxx_instance_ctrl_t”.

Special Notes

None.

8.16 Usage Example of ZMOD4XXX SIS Module

```
#include "r_smc_entry.h"
#include "r_comms_i2c_if.h"
#include "r_zmod4xxx_if.h"

typedef enum e_demo_sequence
{
   DEMO_SEQUENCE_1 = (1),
   DEMO_SEQUENCE_2,
   DEMO_SEQUENCE_3,
   DEMO_SEQUENCE_4,
   DEMO_SEQUENCE_5,
   DEMO_SEQUENCE_6,
   DEMO_SEQUENCE_7,
   DEMO_SEQUENCE_8,
   DEMO_SEQUENCE_9,
} demo_sequence_t;

typedef enum e_demo_callback_status
{
   DEMO_CALLBACK_STATUS_WAIT = (0),
   DEMO_CALLBACK_STATUS_SUCCESS,
   DEMO_CALLBACK_STATUS_REPEAT,
} demo_callback_status_t;

void g_comms_i2c_bus0_quick_setup(void);
void g_zmod4xxx_sensor0_quick_setup(void);
void start_demo(void);
void demo_err(void);

static volatile demo_callback_status_t  gs_i2c_callback_status = DEMO_CALLBACK_STATUS_WAIT;
#if RM_ZMOD4XXX_CFG_DEVICE0_IRQ_ENABLE
static volatile demo_callback_status_t  gs_irq_callback_status = DEMO_CALLBACK_STATUS_WAIT;
#endif

static volatile rm_zmod4xxx_iaq_1st_data_t      gs_iaq_1st_gen_data;
static volatile rm_zmod4xxx_iaq_2nd_data_t      gs_iaq_2nd_gen_data;
static volatile rm_zmod4xxx_odor_data_t         gs_odor_data;
static volatile rm_zmod4xxx_sulfur_odor_data_t  gs_sulfur_odor_data;

void zmod4xxx_comms_i2c_callback(rm_zmod4xxx_callback_args_t * p_args)
{
   if (RM_ZMOD4XXX_EVENT_ERROR != p_args->event)
   {
      gs_i2c_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
   }
   else
   {
      gs_i2c_callback_status = DEMO_CALLBACK_STATUS_REPEAT;
   }
}

/* TODO: Enable if you want to use a IRQ callback */
```

```
void zmod4xxx_irq_callback(rm_zmod4xxx_callback_args_t * p_args)
{
#if RM_ZMOD4XXX_CFG_DEVICE0_IRQ_ENABLE
   FSP_PARAMETER_NOT_USED(p_args);

   gs_irq_callback_status = DEMO_CALLBACK_STATUS_SUCCESS;
#else
   FSP_PARAMETER_NOT_USED(p_args);
#endif
}

/* Quick setup for g_zmod4xxx_sensor0. */
void g_zmod4xxx_sensor0_quick_setup(void)
{
   fsp_err_t err;

   /* Open ZMOD4XXX sensor instance, this must be done before calling any ZMOD4XXX API */
   err = g_zmod4xxx_sensor0.p_api->open(g_zmod4xxx_sensor0.p_ctrl, g_zmod4xxx_sensor0.p_cfg);
   if (FSP_SUCCESS != err)
   {
      demo_err();
   }
}

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
   /* bus has been opened by startup process */
}

void start_demo(void)
{
   fsp_err_t              err;
   rm_zmod4xxx_raw_data_t  raw_data;
   demo_sequence_t        sequence = DEMO_SEQUENCE_1;
   rm_zmod4xxx_lib_type_t  lib_type = RM_ZMOD4XXX_CFG_DEVICE0_OPERATION_MODE;

   /* Clear status */
   gs_i2c_callback_status = DEMO_CALLBACK_STATUS_WAIT;
#if G_ZMOD4XXX_SENSOR0_IRQ_ENABLE
   gs_irq_callback_status = DEMO_CALLBACK_STATUS_WAIT;
#endif

   /* Open the Bus */
   g_comms_i2c_bus0_quick_setup();

   /* Open ZMOD4XXX */
   g_zmod4xxx_sensor0_quick_setup();

   while(1)
   {
      switch(sequence)
      {
         case DEMO_SEQUENCE_1 :
```

```
            {
                /* Clear status */
                gs_i2c_callback_status = DEMO_CALLBACK_STATUS_WAIT;
#if RM_ZMOD4XXX_CFG_DEVICE0_IRQ_ENABLE
                gs_irq_callback_status = DEMO_CALLBACK_STATUS_WAIT;
#endif

                /* Start measurement */
                err = g_zmod4xxx_sensor0.p_api->measurementStart(g_zmod4xxx_sensor0.p_ctrl);
                if (FSP_SUCCESS == err)
                {
                    sequence = DEMO_SEQUENCE_2;
                }
                else
                {
                    demo_err();
                }
            }
            break;

            case DEMO_SEQUENCE_2 :
            {
                /* Check I2C callback status */
                switch (gs_i2c_callback_status)
                {
                    case DEMO_CALLBACK_STATUS_WAIT :
                        break;
                    case DEMO_CALLBACK_STATUS_SUCCESS :
                        sequence = DEMO_SEQUENCE_3;
                        break;
                    case DEMO_CALLBACK_STATUS_REPEAT :
                        sequence = DEMO_SEQUENCE_1;
                        break;
                    default :
                        demo_err();
                        break;
                }
            }
            break;

#if RM_ZMOD4XXX_CFG_DEVICE0_IRQ_ENABLE
            case DEMO_SEQUENCE_3 :
            {
                /* Check IRQ callback status */
                switch (gs_irq_callback_status)
                {
                    case DEMO_CALLBACK_STATUS_WAIT :
                        break;
                    case DEMO_CALLBACK_STATUS_SUCCESS :
                        gs_irq_callback_status = DEMO_CALLBACK_STATUS_WAIT;
                        sequence = DEMO_SEQUENCE_5;
                        break;
                    default :
                        demo_err();
```

```
                break;
            }
        }
        break;
#else
        case DEMO_SEQUENCE_3 :
        {
            /* Clear status */
            gs_i2c_callback_status = DEMO_CALLBACK_STATUS_WAIT;

            /* Get status */
            err = g_zmod4xxx_sensor0.p_api->statusCheck(g_zmod4xxx_sensor0.p_ctrl);
            if (FSP_SUCCESS == err)
            {
                sequence = DEMO_SEQUENCE_4;
            }
            else
            {
                demo_err();
            }
        }
        break;

        case DEMO_SEQUENCE_4 :
        {
            /* Check I2C callback status */
            switch (gs_i2c_callback_status)
            {
                case DEMO_CALLBACK_STATUS_WAIT :
                    break;
                case DEMO_CALLBACK_STATUS_SUCCESS :
                    sequence = DEMO_SEQUENCE_5;
                    break;
                case DEMO_CALLBACK_STATUS_REPEAT :
                    sequence = DEMO_SEQUENCE_3;
                    break;
                default :
                    demo_err();
                    break;
            }
        }
        break;
#endif
        case DEMO_SEQUENCE_5 :
        {
            /* Clear status */
            gs_i2c_callback_status = DEMO_CALLBACK_STATUS_WAIT;

            /* Read data */
            err = g_zmod4xxx_sensor0.p_api->read(g_zmod4xxx_sensor0.p_ctrl, &raw_data);
            if (FSP_SUCCESS == err)
            {
                sequence = DEMO_SEQUENCE_6;
            }
```

```
    else if (FSP_ERR_SENSOR_MEASUREMENT_NOT_FINISHED == err)
    {
        sequence = DEMO_SEQUENCE_3;

        /* Delay 50ms */
        R_BSP_SoftwareDelay (50, BSP_DELAY_MILLISECS);
    }
    else
    {
        demo_err();
    }
}
break;

case DEMO_SEQUENCE_6 :
{
    /* Check I2C callback status */
    switch (gs_i2c_callback_status)
    {
        case DEMO_CALLBACK_STATUS_WAIT :
            break;
        case DEMO_CALLBACK_STATUS_SUCCESS :
            sequence = DEMO_SEQUENCE_7;
            break;
        case DEMO_CALLBACK_STATUS_REPEAT :
            sequence = DEMO_SEQUENCE_5;
            break;
        default :
            demo_err();
            break;
    }
}
break;

case DEMO_SEQUENCE_7 :
{
    /* Calculate data */
    switch (lib_type)
    {
        case RM_ZMOD4410_LIB_TYPE_IAQ_1ST_GEN_CONTINUOUS :
        case RM_ZMOD4410_LIB_TYPE_IAQ_1ST_GEN_LOW_POWER :
            err = g_zmod4xxx_sensor0.p_api->iaq1stGenDataCalculate(g_zmod4xxx_sensor0.p_ctrl,
                                                   &raw_data,
                                                   (rm_zmod4xxx_iaq_1st_data_t*)&gs_iaq_1st_gen_data);
            break;
        case RM_ZMOD4410_LIB_TYPE_IAQ_2ND_GEN :
            err = g_zmod4xxx_sensor0.p_api->iaq2ndGenDataCalculate(g_zmod4xxx_sensor0.p_ctrl,
                                                   &raw_data,
                                                   (rm_zmod4xxx_iaq_2nd_data_t*)&gs_iaq_2nd_gen_data);
            break;
        case RM_ZMOD4410_LIB_TYPE_ODOR :
            err = g_zmod4xxx_sensor0.p_api->odorDataCalculate(g_zmod4xxx_sensor0.p_ctrl,
                                                  &raw_data,
                                                  (rm_zmod4xxx_odor_data_t*)&gs_odor_data);
```

```
            break;
        case RM_ZMOD4410_LIB_TYPE_SULFUR_ODOR :
            err = g_zmod4xxx_sensor0.p_api->sulfurOdorDataCalculate(g_zmod4xxx_sensor0.p_ctrl,
                                                    &raw_data,
                                                    (rm_zmod4xxx_sulfur_odor_data_t*)&gs_sulfur_odor_data);
            break;
        default :
            demo_err();
            break;
    }

    if (FSP_SUCCESS == err)
    {
        /* Gas data is valid. Describe the process by referring to each calculated gas data. */
    }
    else if (FSP_ERR_SENSOR_IN_STABILIZATION == err)
    {
        /* Gas data is invalid. Sensor is in stabilization. */
    }
    else
    {
        demo_err();
    }

    sequence = DEMO_SEQUENCE_8;
}
break;

case DEMO_SEQUENCE_8 :
{
    switch (lib_type)
    {
        case RM_ZMOD4410_LIB_TYPE_IAQ_1ST_GEN_CONTINUOUS :
        case RM_ZMOD4410_LIB_TYPE_ODOR :
            sequence = DEMO_SEQUENCE_3;
            break;
        case RM_ZMOD4410_LIB_TYPE_IAQ_1ST_GEN_LOW_POWER :
            /* See Table 3 in the ZMOD4410 Programming Manual. */
             R_BSP_SoftwareDelay (5475, BSP_DELAY_MILLISECS);
            sequence = DEMO_SEQUENCE_1;
            break;
        case RM_ZMOD4410_LIB_TYPE_IAQ_2ND_GEN :
        case RM_ZMOD4410_LIB_TYPE_SULFUR_ODOR :
            /* IAQ 2nd Gen : See Table 4 in the ZMOD4410 Programming Manual. */
            /* Sulfur Odor : See Table 6 in the ZMOD4410 Programming Manual. */
              R_BSP_SoftwareDelay1990, BSP_DELAY_MILLISECS);
            sequence = DEMO_SEQUENCE_1;
            break;
        default :
            demo_err();
            break;
    }
}
break;
```

```
            default :
            {
                demo_err();
            }
            break;
        }
    }
}

void demo_err(void)
{
    while(1)
    {
        // nothing
    }
}
```

9. OB1203 API Functions

9.1 RM_OB1203_Open ()

This function opens and configures the OB1203 SIS module. This function must be called before calling any other OB1203 API functions.

Format

```
fsp_err_t RM_OB1203_Open (
        rm_ob1203_ctrl_t * const p_api_ctrl,
        rm_ob1203_cfg_t const * const p_cfg
);
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.7(1) Configuration Struct rm_ob1203_cfg_t

Return Values

FSP_SUCCESS	Successfully configured.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	Module is already open. This module can only be opened once.
FSP_ERR_TIMEOUT	communication is timeout.
FSP_ERR_ABORTED	communication is aborted.

Properties

Prototyped in rm_ob1203.h

Description

This function opens and configures the OB1203 SIS module.

This function copies the contents in “p_cfg” structure to the member “p_api_ctrl->p_cfg” in “p_api_ctrl” structure. This function does configurations by setting the members of “p_api_ctrl” structure as following:

- Sets related instance of COMMS SIS module
- Sets parameters of callback and context
- Sets open flag

This function calls following after all above initializations are done.

- Opens API of COMMS SIS module to open communication middleware
- Initializes the sensor device (OB1203)

Special Notes

None

9.2 RM_OB1203_Close ()

This function disables specified OB1203 control block. This function should be called when the sensor is closed.

Format

fsp_err_t RM_OB1203_Close (rm_ob1203_ctrl_t * const p_api_ctrl)

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

Return Values

FSP_SUCCESS	Successfully closed.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_ob1203.h

Description

This function calls closing API of COMMS control module to close communication middleware function.

This function clears open flag after all above are done.

Special Notes

None

9.3 RM_OB1203_MeasurementStart ()

This function starts a measurement.

Format

fsp_err_t RM_OB1203_MeasurementStart (rm_ob1203_ctrl_t * const p_api_ctrl)

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_ob1203.h

Description

This function sends the measurement start to command register of OB1203 sensor and starts a measurement after the “event” in “p_api_ctrl” structure is cleared.

Special Notes

None.

9.4 RM_OB1203_MeasurementStop ()

This function stops a measurement.

Format

fsp_err_t RM_OB1203_MeasurementStop (rm_ob1203_ctrl_t * const p_api_ctrl)

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

Return Values

FSP_SUCCESS	Successfully data decoded.
FSP_ERR_ASSERTION	Null pointer, or one or more configuration options are invalid.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_ob1203.h

Description

This function sends the measurement stop to command register of OB1203 sensor and stops a measurement.

Special Notes

If device interrupt is enabled, interrupt bits are cleared after measurement stop. If PPG mode, FIFO information is also reset after measurement stop.

9.5 RM_OB1203_DeviceStatusGet ()

This function reads the status of sensor.

Format

```
fsp_err_t RM_OB1203_DeviceStatusGet (
        rm_ob1203_ctrl_t * const              p_api_ctrl,
        rm_ob1203_device_status_t * const     p_status)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_status

Pointer to data structure for status.

```
/** OB1203 device status */
typedef struct st_rm_ob1203_device_status
{
    bool power_on_reset_occur;
    bool light_interrupt_occur;
    bool light_measurement_complete;
    bool ts_measurement_complete;
    bool fifo_afull_interrupt_occur;   ///< FIFO almost full interrupt
    bool ppg_measurement_complete;
    bool object_near;
    bool prox_interrupt_occur;
    bool prox_measurement_complete;
} rm_ob1203_device_status_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.

Properties

Prototyped in rm_ob1203.h

Description

This function gets device status from OB1203 device. Clear all interrupt bits after read.

Special Notes

None

9.6 RM_OB1203_LightRead ()

This reads ADC data of Light from OB1203 device. This function should be called when measurement finished.

Format

```
fsp_err_t RM_OB1203_LightRead (
        rm_ob1203_ctrl_t * const p_api_ctrl,
        rm_ob1203_raw_data_t * const p_raw_data,
        rm_ob1203_light_data_type_t type)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_raw_data

Pointer to raw data structure for storing ADC data read from sensor.

```
/** OB1203 raw data structure */
typedef struct st_rm_ob1203_raw_data
{
   uint8_t adc_data[32];
} rm_ob1203_raw_data_t;
```

Type

Light Data Type.

```
/** Data type of Light */
typedef enum e_rm_ob1203_light_data_type
{
   RM_OB1203_LIGHT_DATA_TYPE_ALL = 0, ///< Common
   RM_OB1203_LIGHT_DATA_TYPE_CLEAR,   ///< Common
   RM_OB1203_LIGHT_DATA_TYPE_GREEN,   ///< Common
   RM_OB1203_LIGHT_DATA_TYPE_BLUE,    ///< CS mode only
   RM_OB1203_LIGHT_DATA_TYPE_RED,     ///< CS mode only
   RM_OB1203_LIGHT_DATA_TYPE_COMP,    ///< Common. Temperature compensation data.
} rm_ob1203_light_data_type_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_ob1203.h

Description

This function reads ADC data selected by rm_ob1203_light_data_type_t and stores data to “p_raw_data” structure.

Special Notes

None

9.7 RM_OB1203_ProxRead ()

This reads ADC data of Proximity from OB1203 device. This function should be called when measurement finished.

Format

```
fsp_err_t RM_OB1203_ProxRead (
        rm_ob1203_ctrl_t * const p_api_ctrl,
        rm_ob1203_raw_data_t * const p_raw_data)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_raw_data

Pointer to raw data structure for storing ADC data read from sensor.

```
/** OB1203 raw data structure */
typedef struct st_rm_ob1203_raw_data
{
   uint8_t adc_data[96];
} rm_ob1203_raw_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.

Properties

Prototyped in rm_ob1203.h

Description

This function reads ADC data and stores data to “p_raw_data” structure.

Special Notes

None

9.8 RM_OB1203_PpgRead ()

This read ADC data of PPG from OB1203 sensor. This function should be called when measurement finished.

Format

```
fsp_err_t RM_OB1203_PpgRead (
        rm_ob1203_ctrl_t * const p_api_ctrl,
        rm_ob1203_raw_data_t * const p_raw_data,
        uint8_t const  number_of_samples)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_raw_data

Pointer to raw data structure for storing ADC data read from sensor.

```
/** OB1203 raw data structure */
typedef struct st_rm_ob1203_raw_data
{
   uint8_t adc_data[32];
} rm_ob1203_raw_data_t;
```

number_of_samples

number of PPG samples. One sample is 3 bytes.

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_TIMEOUT	Communication is timeout.
FSP_ERR_ABORTED	Communication is aborted.

Properties

Prototyped in rm_ob1203.h

Description

This function reads ADC data and stores data to “p_raw_data” structure.

Special Notes

None

9.9 RM_OB1203_LightDataCalculate ()

This calculates light values from ADC data.

Format

```
fsp_err_t RM_OB1203_LightDataCalculate (
        rm_ob1203_ctrl_t * const          p_api_ctrl,
        rm_ob1203_raw_data_t * const      p_raw_data,
        rm_ob1203_light_data_t * const    p_ob1203_data)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t..

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in.

```
/** OB1203 raw data structure */
typedef struct st_rm_ob1203_raw_data
{
   uint8_t adc_data[32];
} rm_ob1203_raw_data_t;
```

p_ob1203_data

Pointer to calculation result data structure storing Light data. calculation result.

```
/** OB1203 light data structure */
typedef struct st_rm_ob1203_light_data
{
   uint32_t clear_data;        ///< Clear channel data (20bits).
   uint32_t green_data;        ///< Green channel data (20bits).
   uint32_t blue_data;         ///< Blue channel data (20bits).
   uint32_t red_data;          ///< Red channel data (20bits).
   uint32_t comp_data;         ///< Temperature compensation (Comp) channel data (20bits).
} rm_ob1203_light_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_ob1203.h

Description

This function calculates Light results and stores the result into the rm_ob1203_light_data_t

Special Notes

None

9.10 RM_OB1203_ProxDataCalculate ()

This function calculates Proximity values from ADC data.

Format

```
fsp_err_t RM_OB1203_ProxDataCalculate (
        rm_ob1203_ctrl_t * const            p_api_ctrl,
        rm_ob1203_raw_data_t * const        p_raw_data,
        rm_ob1203_prox_data_t * const       p_ob1203_data)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t..

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in.

```
/** OB1203 raw data structure */
typedef struct st_rm_ob1203_raw_data
{
   uint8_t adc_data[96];
} rm_ob1203_raw_data_t;
```

p_ob1203_data

Pointer to calculation result data structure storing Proximity calculation result.

```
/** OB1203 proximity data structure */
typedef struct st_rm_ob1203_prox_data
{
   uint16_t proximity_data;        ///< Proximity data.
} rm_ob1203_prox_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_ob1203.h

Description

This function calculates Proximity results and stores the results into the rm_ob1203_prox_data_t.

Special Notes

None

9.11 RM_OB1203_PpgDataCalculate ()

This function calculates PPG values from ADC data.

Format

```
fsp_err_t RM_OB1203_PpgDataCalculate (
        rm_ob1203_ctrl_t * const          p_api_ctrl,
        rm_ob1203_raw_data_t * const      p_raw_data,
        rm_ob1203_ppg_data_t * const      p_ob1203_data)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_raw_data

Pointer to raw data structure which ADC data read from sensor is stored in.

```
/** OB1203 raw data structure */
typedef struct st_rm_ob1203_raw_data
{
   uint8_t adc_data[32];
} rm_ob1203_raw_data_t;
```

p_ob1203_data

Pointer to calculation result data structure storing PPG calculation result.

```
/** OB1203 PPG data structure */
typedef struct st_rm_ob1203_ppg_data
{
   uint32_t ppg_data[32];          ///< PPG data (18bits).
} rm_ob1203_ppg_data_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_UNSOPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_ob1203.h

Description

This function calculates PPG results and stores the results into the rm_ob1203_ppg_data_t.

Special Notes

None

9.12 RM_OB1203_DeviceInterruptCfgSet ()

This function configures device interrupt.

Format

```
fsp_err_t RM_OB1203_DeviceInterruptCfgSet (
        rm_ob1203_ctrl_t * const          p_api_ctrl,
        rm_ob1203_device_interrupt_cfg_t  const      interrupt_cfg)
```

Parameters

p_api_ctrl

Pointer to control structure.

The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

Interrupt_cfg

Device interrupt configuration structure for each operation mode.

```
/** OB1203 device interrupt configuration structure */
typedef struct st_rm_ob1203_device_interrupt_cfg
{
   rm_ob1203_operation_mode_t        light_prox_mode; ///< Light Proximity mode only. If Light
                                     mode uses IRQ, set RM_OB1203_OPERATION_MODE_LIGHT. If Proximity
                                     mode uses IRQ, set RM_OB1203_OPERATION_MODE_PROXIMITY.
   rm_ob1203_light_interrupt_type_t  light_type;      ///< Light mode interrupt type.
   rm_ob1203_light_interrupt_source_t light_source;   ///< Light mode interrupt source.
   rm_ob1203_prox_interrupt_type_t   prox_type;       ///< Proximity mode interrupt type.
   uint8_t persist;                                   ///< The number of similar consecutive Light mode or
                                     Proximity interrupt events that must occur before the interrupt is asserted (4bits).
   rm_ob1203_ppg_interrupt_type_t ppg_type;           ///< PPG mode interrupt type.
} rm_ob1203_device_interrupt_cfg_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_ob1203.h

Description

This function configures device interrupts for each operation mode.

Special Notes

None

9.13 RM_OB1203_GainSet ()

This function configures gain value.

Format

```
fsp_err_t RM_OB1203_GainSet (
        rm_ob1203_ctrl_t * const        p_api_ctrl,
        rm_ob1203_gain_t  const         gain)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

gain

Gain configuration structure.

```
/** OB1203 Gain structure */
typedef struct st_rm_ob1203_gain
{
   rm_ob1203_light_gain_t    light;          ///< Gain for Light mode
   rm_ob1203_ppg_prox_gain_t ppg_prox;       ///< Gain for PPG mode and Proximity mode
} rm_ob1203_gain_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.

Properties

Prototyped in rm_ob1203.h

Description

This function configures gain for each operation mode

Special Notes

None

9.14 RM_OB1203_LedCurrentSet ()

This function configures currents for LED.

Format

```
fsp_err_t RM_OB1203_LedCurrentSet (
        rm_ob1203_ctrl_t * const            p_api_ctrl,
        rm_ob1203_led_current_t  const      led_current)
```

Parameters

p_api_ctrl

Pointer to control structure.

The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

led_current

Current configuration for LED.

```
/** OB1203 LED currents structure */
typedef struct st_rm_ob1203_led_current
{
   uint16_t ir_led;                ///< IR LED current.
   uint16_t red_led;               ///< Red LED current.
} rm_ob1203_led_current_t;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERROR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_ob1203.h

Description

This function configures LED currents for each operation mode.

Special Notes

None

9.15 RM_OB1203_FifoInfoGet ()

This function gets FIFO information (write_index, read_index and overflow_counter).

Format

```
fsp_err_t RM_OB1203_FifoInfoGet (
        rm_ob1203_ctrl_t * const        p_api_ctrl,
        rm_ob1203_fifo_info_t * const   p_fifo_info)
```

Parameters

p_api_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.7(2) Control Struct rm_ob1203_ctrl_t.

p_fifo_info

Pointer to FIFO information.

```
/** OB1203 FIFO information structure */
typedef struct st_rm_ob1203_fifo_info
{
    uint8_t write_index;        ///< The FIFO index where the next sample of PPG data will be
                                written in the FIFO.
    uint8_t read_index;         ///< The index of the next sample to be read from the FIFO_DATA
                                register.
    uint8_t overflow_counter;   ///< If the FIFO Rollover Enable bit is set, the FIFO overflow counter
                                counts the number of old samples (up to 15) which are overwritten
                                by new data.
    uint8_t unread_samples; ///< The number of unread samples calculated from the write index and
                                the read index.
} rm_ob1203_fifo_info_t;;
```

Return Values

FSP_SUCCESS	Successfully started.
FSP_ERR_ASSERTION	Null pointer passed as a parameter.
FSP_ERR_NOT_OPEN	Module is not open.
FSP_ERR_UNSUPPORTED	Operation mode is not supported.

Properties

Prototyped in rm_ob1203.h

Description

This function gets FIFO information for PPG mode. Light and Proximity modes are not supported.
- write_index is the FIFO index where the next sample of PPG data will be written in the FIFO.
- read_index is the index of the next sample to be read from the register.
- overflow_counter is the number of old samples (up to 15) which are overwritten by new data. If the FIFO Rollover is enabled, the FIFO overflow counter counts.
- unread_samples is the number of unread FIFO samples, which can be calculated by write index and read index.

Special Notes

None

9.16 rm_ob1203_comms_i2c_callback ()

This is callback function for OB1203 SIS module.

Format

void rm_ob1203_comms_i2c_callback (rm_comms_callback_args_t * p_args)

Parameters

p_args

Pointer to callback parameter definition.

```
/** Communications middleware callback parameter definition */
typedef struct st_rm_comms_callback_args
{
   void const     * p_context;
   rm_comms_event_t event;
} rm_comms_callback_args_t;
```

Return Values

None

Properties

Prototyped in rm_ob1203.h

Description

This callback function is called in COMMS SIS module callback function.
The member “event” in “rm_ob1203_callback_args_t” structure which is a member of “rm_ob1203_instance_ctrl_t” structure is set according to COMMS SIS module events status “p_args->event”.
The events of OB1203 SIS module are

```
/** Event in the callback function */
typedef enum e_rm_ob1203_event
{
   RM_OB1203_EVENT_SUCCESS = 0,
   RM_OB1203_EVENT_ERROR,
} rm_ob1203_event_t;
```

And the events of COMMS control module are

```
typedef enum e_rm_comms_event
{
   RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
   RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_ob1203_callback_args_t” structure is set to “RM_OB1203_EVENT_SUCCESS” when the COMMS SIS module events status is “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_OB1203_EVENT_ERROR”.

Special Notes

None.

9.17 Usage Example of OB1203 SIS Module

```
#include "r_smc_entry.h"
#include "r_comms_i2c_if.h"
#include "r_ob1203_if.h"
#include "ob1203_bio/ob1203_bio.h"

typedef enum e_demo_sequence
{
    DEMO_SEQUENCE_1 = (1),
    DEMO_SEQUENCE_2,
    DEMO_SEQUENCE_3,
    DEMO_SEQUENCE_4,
    DEMO_SEQUENCE_5,
    DEMO_SEQUENCE_6,
    DEMO_SEQUENCE_7,
    DEMO_SEQUENCE_8,
    DEMO_SEQUENCE_9,
    DEMO_SEQUENCE_10,
    DEMO_SEQUENCE_11,
    DEMO_SEQUENCE_12,
} demo_sequence_t;

void g_comms_i2c_bus0_quick_setup(void);
void start_demo(void);
void demo_err(void);

static spo2_t gs_spo2;
static volatile ob1203_bio_data_t gs_ob1203_bio_data;

/* Quick setup for g_comms_i2c_bus0. */
void g_comms_i2c_bus0_quick_setup(void)
{
    /* bus has been opened by startup process */
}

void start_demo(void);
void start_demo(void)
{
    bool result;
    rm_ob1203_raw_data_t raw_data;
    rm_ob1203_ppg_data_t ppg_data;
    ob1203_bio_t ob1203_bio;
    bool change = false;
    bool valid = false;
    bool update = false;
    bool ready = false;
    ob1203_bio_gain_currents_t gain_currents;
    demo_sequence_t sequence = DEMO_SEQUENCE_1;

    /* Set default gain and currents */
    gain_currents.gain.ppg_prox = g_ob1203_sensor1_extended_cfg.ppg_prox_gain;
    gain_currents.currents.ir_led = g_ob1203_sensor1_extended_cfg.ppg_ir_led_current;
    gain_currents.currents.red_led = g_ob1203_sensor1_extended_cfg.ppg_red_led_current;

    /* Open the Bus */
    g_comms_i2c_bus0_quick_setup();

    /* Open OB1203 Bio extension */
    result = ob1203_bio_open(&ob1203_bio,
                             (rm_ob1203_instance_t*)&g_ob1203_sensor0, // Proximity mode
                             (rm_ob1203_instance_t*)&g_ob1203_sensor1,  // PPG mode
                             &gs_spo2);
    if (false == result)
    {
        demo_err();
    }
```

```
while (1)
{
   switch (sequence)
   {
      case DEMO_SEQUENCE_1 :
      {
         /* Initialize an operation mode */
         result = ob1203_bio_operation_mode_init(&ob1203_bio);
         if (false == result)
         {
            demo_err();
         }

         sequence = DEMO_SEQUENCE_2;
      }
      break;

      case DEMO_SEQUENCE_2 :
      {
         /* Start a measurement */
         result = ob1203_bio_measurement_start(&ob1203_bio);
         if (false == result)
         {
            demo_err();
         }

         sequence = DEMO_SEQUENCE_3;
      }
      break;

      case DEMO_SEQUENCE_3:
      {
         /* Wait measurement period */
         result = ob1203_bio_measurement_period_wait(&ob1203_bio);
         if (false == result)
         {
            demo_err();
         }

         sequence = DEMO_SEQUENCE_4;
      }
      break;

      case DEMO_SEQUENCE_4 :
      {
         /* Check if an operation mode needs to be changed */
         result = ob1203_bio_mode_change_check(&ob1203_bio, &change);
         if (false == result)
         {
            demo_err();
         }

         if (false != change)
         {
            /* Stop the measurement */
            result = ob1203_bio_measurement_stop(&ob1203_bio);
            if (false == result)
            {
               demo_err();
            }

            /* Change to another mode */
            sequence = DEMO_SEQUENCE_1;
         }
         else
         {
            /* No change */
```

```
            sequence = DEMO_SEQUENCE_5;
        }
    }
    break;

    case DEMO_SEQUENCE_5:
    {
        /* Read raw data */
        result = ob1203_bio_ppg_raw_data_read(&ob1203_bio, &raw_data);
        if (false == result)
        {
            demo_err();
        }

        sequence = DEMO_SEQUENCE_6;
    }
    break;

    case DEMO_SEQUENCE_6:
    {
        /* Calculate PPG data from raw data */
        result = ob1203_bio_ppg_data_calculate(&ob1203_bio, &raw_data, &ppg_data, &valid);
        if (false == result)
        {
            demo_err();
        }

        if (false != valid)
        {
            /* Valid data */
            sequence = DEMO_SEQUENCE_7;
        }
        else
        {
            /* Check if an operation mode needs to be changed */
            result = ob1203_bio_mode_change_check(&ob1203_bio, &change);
            if (false == result)
            {
                demo_err();
            }

            if (false != change)
            {
                /* Stop the measurement */
                result = ob1203_bio_measurement_stop(&ob1203_bio);
                if (false == result)
                {
                    demo_err();
                }

                /* Change to another mode */
                sequence = DEMO_SEQUENCE_1;
            }
            else
            {
                /* Invalid data */
                sequence = DEMO_SEQUENCE_3;
            }
        }
    }
    break;

    case DEMO_SEQUENCE_7:
    {
        /* Auto gain and currents control */
        result = ob1203_bio_auto_gain_currents_control(&ob1203_bio,
                                                       &ppg_data,
```

```
                                        &gain_currents,
                                        &update);
        if (false == result)
        {
            demo_err();
        }

        if (false != update)
        {
            /* Stop the measurement */
            result = ob1203_bio_measurement_stop(&ob1203_bio);
            if (false == result)
            {
                demo_err();
            }

            /* Reconfigure gain and currents */
            result = ob1203_bio_gain_currents_reconfigure(&ob1203_bio, &gain_currents);
            if (false == result)
            {
                demo_err();
            }

            sequence = DEMO_SEQUENCE_2;
        }
        else
        {
            sequence = DEMO_SEQUENCE_8;
        }
    }
    break;

    case DEMO_SEQUENCE_8:
    {
        /* Check if the preparation for the algorithm is complete */
        result = ob1203_bio_algorithm_preparation_check(&ob1203_bio, &ready);
        if (false == result)
        {
            demo_err();
        }

        if (false == ready)
        {
            /* Stop the measurement */
            result = ob1203_bio_measurement_stop(&ob1203_bio);
            if (false == result)
            {
                demo_err();
            }

            /* Reset the algorithm */
            result = ob1203_bio_algorithm_reset(&ob1203_bio);
            if (false == result)
            {
                demo_err();
            }

            /* Clear PPG samples */
            result = ob1203_bio_samples_clear(&ob1203_bio);
            if (false == result)
            {
                demo_err();
            }

            sequence = DEMO_SEQUENCE_2;
        }
        else
```

```
        {
            sequence = DEMO_SEQUENCE_9;
        }
    }
    break;

    case DEMO_SEQUENCE_9:
    {
        /* Add PPG samples */
        result = ob1203_bio_samples_add(&ob1203_bio, &ppg_data);
        if (false == result)
        {
            demo_err();
        }

        sequence = DEMO_SEQUENCE_10;
    }
    break;

    case DEMO_SEQUENCE_10:
    {
        /* Calculate heart rate and SpO2 values */
        result = ob1203_bio_hr_spo2_calculate(&ob1203_bio, (ob1203_bio_data_t *)&gs_ob1203_bio_data);
        if (false == result)
        {
            demo_err();
        }

        sequence = DEMO_SEQUENCE_11;
    }
    break;

    case DEMO_SEQUENCE_11:
    {
        /* Calculate a respiration rate value */
        result = ob1203_bio_rr_calculate(&ob1203_bio,
                                         (ob1203_bio_data_t *)&gs_ob1203_bio_data);
        if (false == result)
        {
            demo_err();
        }

        sequence = DEMO_SEQUENCE_12;
    }
    break;

    case DEMO_SEQUENCE_12:
    {
        /* Check perfusion index (PI) */
        result = ob1203_bio_perfusion_index_check(&ob1203_bio,
                                                  &valid);
        if (false == result)
        {
            demo_err();
        }

        if (false != valid)
        {
            sequence = DEMO_SEQUENCE_3;
        }
        else
        {
            /* Stop the measurement */
            result = ob1203_bio_measurement_stop(&ob1203_bio);
            if (false == result)
            {
                demo_err();
```

```
                }

                /* Reset the algorithm */
                result = ob1203_bio_algorithm_reset(&ob1203_bio);
                if (false == result)
                {
                    demo_err();
                }

                sequence = DEMO_SEQUENCE_2;
            }
        }
        break;

        default:
        {
            demo_err();
        }
        break;
    }
  }
}

void demo_err(void)
{
  while (1)
  {
    // nothing
  }
}
```

10. COMMS (I2C communication middleware) API Functions

10.1 RM_COMMS_I2C_Open()

This function opens and configures the COMMS (I2C communication middleware) SIS module.

Format

```
fsp_err_t RM_COMMS_I2C_Open (
        rm_comms_ctrl_t * const p_ctrl,
        rm_comms_cfg_t const * const p_cfg
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.8(2) Control Struct rm_comms_ctrl_t.

p_cfg

Pointer to configuration structure.
The members of this structure are shown in 2.9.8(1) Configuration Struct rm_comms_cfg_t.

Return Values

FSP_SUCCESS	: Communications Middle module successfully configured.
FSP_ERR_ASSERTION	: Null pointer, or one or more configuration options is invalid.
FSP_ERR_ALREADY_OPEN	: Module is already open. This module can only be opened once.
FSP_ERR_COMMS_BUS_NOT_OPEN	: I2C driver is not open.

Properties

Prototyped in rm_comms_i2c.h

Description

This function opens and configures the COMMS SIS module.

This function copies the contents in “p_cfg” structure to the member “p_ctrl->p_cfg” in “p_cfg” structure.

This function does configurations by setting the members of “p_ctrl” structure as following:

- Sets bus configuration
- Sets lower-level driver configuration
- Sets callback and context
- Sets open flag

Special Notes

None

10.2 RM_COMMS_I2C_Close()

This function disables specified COMMS SIS module.

Format

fsp_err_t RM_COMMS_I2C_Close (rm_comms_ctrl_t * const p_ctrl)

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.8(2) Control Struct rm_comms_ctrl_t.

Return Values

FSP_SUCCESS	: Communications Middle module successfully configured.
FSP_ERR_ASSERTION	: Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	: Module is not open.

Properties

Prototyped in rm_comms_i2c.h

Description

This function clears current device on bus and open flag.

Special Notes

None

10.3 RM_COMMS_I2C_Read()

This function performs a read from I2C device.

Format

```
fsp_err_t RM_COMMS_I2C_Read (
        rm_comms_ctrl_t * const p_ctrl,
        uint8_t * const p_dest,
        uint32_t const bytes
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.8(2) Control Struct rm_comms_ctrl_t.

p_dest

Pointer to the buffer to store read data.

bytes

Number of bytes to read.

Return Values

FSP_SUCCESS	: Communications Middle module successfully configured.
FSP_ERR_ASSERTION	: Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	: Module is not open.
FSP_ERR_INVALID_CHANNEL	: Invalid channel.
FSP_ERR_INVALID_ARGUMENT	: Invalid argument.
FSP_ERR_IN_USE	: Bus is busy.

Properties

Prototyped in rm_comms_i2c.h

Description

This function calls internal function “rm_comms_i2c_bus_read()” to start read operation from I2C bus which is IICA bus or SAU bus depending on the device (sensor) connection.

The internal function “rm_comms_i2c_bus_read()” does bus re-configuration according to contents in “p_ctrl”. Then it calls “R_Config_IICA0_MasterReceive()” API when the device (sensor) is connected to IICA bus, calls “R_Config_IIC00_MasterReceive()” API when the device (sensor) is connected to SAU bus.

The receive pattern of “R_Config_IICA0_MasterReceive()” and “R_Config_IIC00_MasterReceive()” is set as master reception. In this pattern, the master (RL78 MCU) receives data from the slave.

Please refer to following documents for detail of “R_Config_IICA0_MasterReceive()” API and “R_Config_IIC00_MasterReceive()” API:

- Smart Configurator User’s Manual : RL78 API Reference (R20UT4852)

Special Notes

None

10.4 RM_COMMS_I2C_Write()

This function performs a write from the I2C device.

Format

```
fsp_err_t RM_COMMS_I2C_Write (
        rm_comms_ctrl_t * const p_ctrl,
        uint8_t * const p_src,
        uint32_t const bytes
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.8(2) Control Struct rm_comms_ctrl_t.

p_src

Pointer to the buffer to store writing data.

bytes

Number of bytes to write.

Return Values

FSP_SUCCESS	: Communications Middle module successfully configured.
FSP_ERR_ASSERTION	: Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	: Module is not open.
FSP_ERR_INVALID_CHANNEL	: Invalid channel.
FSP_ERR_INVALID_ARGUMENT	: Invalid argument.
FSP_ERR_IN_USE	: Bus is busy.

Properties

Prototyped in rm_comms_i2c.h

Description

This function calls internal function “rm_comms_i2c_bus_write()” to start write operation to I2C bus which is IICA bus or SAU bus depending on device (sensor) connection.

The internal function “rm_comms_i2c_bus_write()” does bus re-configuration according to contents in “p_ctrl”. Then it calls “R_Config_IICA0_MasterSend()” API when the device (sensor) is connected to IICA bus, calls “R_Config_IIC00_MasterSend()” API when the device (sensor) is connected to SAU bus.

Please refer to following documents for detail of “R_Config_IICA0_MasterSend()” API and “R_Config_IIC00_MasterSend()” API:

- Smart Configurator User’s Manual : RL78 API Reference (R20UT4852)

Special Notes

None

10.5 RM_COMMS_I2C_WriteRead()

This function performs a write to, then a read from the I2C device.

Format

```
fsp_err_t RM_COMMS_I2C_WriteRead (
        rm_comms_ctrl_t * const          p_ctrl,
        rm_comms_write_read_params_t const write_read_params
)
```

Parameters

p_ctrl

Pointer to control structure.
The members of this structure are shown in 2.9.8(2) Control Struct rm_comms_ctrl_t.

write_read_params

Parameters structure for writeRead API.

```
/** Struct to pack params for writeRead */
typedef struct st_rm_comms_write_read_params
{
   uint8_t * p_src;             ///< pointer to buffer for storing write data
   uint8_t * p_dest;            ///< pointer to buffer for storing read data
   uint8_t   src_bytes;         ///< number of write data
   uint8_t   dest_bytes;        ///< number of read data
} rm_comms_write_read_params_t;
```

Return Values

FSP_SUCCESS	: Communications Middle module successfully configured.
FSP_ERR_ASSERTION	: Null pointer, or one or more configuration options is invalid.
FSP_ERR_NOT_OPEN	: Module is not open.
FSP_ERR_INVALID_CHANNEL	: Invalid channel.
FSP_ERR_INVALID_ARGUMENT	: Invalid argument.
FSP_ERR_IN_USE	: Bus is busy.

Properties

Prototyped in rm_comms_i2c.h

Description

This function calls internal function “rm_comms_i2c_bus_write_read ()” to start writing to I2C bus, then reading from I2C bus with re-start. The I2C bus is RIIC bus or SCI bus depending on device (sensor) connection.

The internal function “rm_comms_i2c_bus_write_read ()” does bus re-configuration according to contents in “p_ctrl”. Then it calls “R_Config_IICA0_MasterSend()” API when the device (sensor) is connected to IICA bus, calls “R_Config_IIC00_MasterSend()” API when the device (sensor) is connected to SAU bus.
After, in interrupt processing, it calls “R_Config_IICA0_MasterReceive()” or “R_Config_IIC00_MasterReceive()”.
In this pattern, the master (RX MCU) transmits data to the slave. After the transmission completes, a restart condition is generated, and the master receives data from the slave.

Special Notes

None.

10.6 rm_comms_i2c_callback

This is callback function for COMMS SIS module called in I2C driver callback function.

Format

```
void rm_comms_i2c_callback (rm_comms_ctrl_t const * p_api_ctrl)
```

Parameters

p_ctrl

Pointer to instance control structure.

```
/** Communications middleware control structure. */
typedef struct st_rm_comms_i2c_instance_ctrl
{
    rm_comms_cfg_t const           * p_cfg; ///< middleware configuration.
    rm_comms_i2c_bus_extended_cfg_t * p_bus; ///< Bus using this device;
    void    * p_lower_level_cfg;           ///< Used to reconfigure I2C driver
    uint32_t  open;                        ///< Open flag.
    uint32_t  transfer_data_bytes;         ///< Size of transfer data.
    uint8_t * p_transfer_data;             ///< Pointer to transfer data buffer.

    /* Pointer to callback and optional working memory */
    void (* p_callback)(rm_comms_callback_args_t * p_args);

    void const * p_context;          ///< Pointer to the user-provided context
} rm_comms_i2c_instance_ctrl_t;
```

Return Values

None

Properties

Prototyped in rm_comms_i2c.h

Description

This callback function is common callback function called in I2C driver callback function.
The member “event” in “rm_comms_callback_args_t” structure which is a member of “rm_comms_cfg_t” structure is set by local function “rm_comms_i2c_bus_callbackErrorCheck” according to I2C bus status.
The events of COMMS SIS module are

```
typedef enum e_rm_comms_event
{
    RM_COMMS_EVENT_OPERATION_COMPLETE = 0,
    RM_COMMS_EVENT_ERROR,
} rm_comms_event_t;
```

The “event” of “rm_comms_callback_args_t” structure is set to “RM_COMMS_EVENT_OPERATION_COMPLETE” otherwise set to “RM_COMMS_EVENT_ERROR”.
For RTOS application, local function “rm_comms_i2c_process_in_callback” is used for releasing semaphore and call user callback function.

Special Notes

None.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	December 9, 2021	-	First Release
1.10	March 20, 2022	15, 16 18 20	Supports the IAQ 2nd Gen. (Ultra-Low Power) Supports multiple devices on the same bus. Updates the structure of instance control.
1.20	April 27, 2022	-	Add OB1203 sensor
1.30	June 30, 2020	-	Add FS3000 sensor, FS1015 sensor and HS400x sensor
1.40	August 31, 2022		Added descriptions of ZMOD4450 to ZMOD4XXX SIS modules
1.41	March 3, 2023	-	Deleted R_ZMOD4XXX_SoftwareDelay Updated test environments

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.